C B [illegible]

THE AXIOMATIC METHOD

PART II.

C.A.R. HOARE. ~~Early 1968~~.

DEC 67

Summary.

This paper illustrates the manner in which the axiomatic method may be applied to the rigorous definition of programming languages. It does not give a full definition of any particular language, but rather explains and discusses a selection of very general axioms, which will apply to broad classes of programming language.

1. Introduction.

There are many purposes for which the description of a programming language is required, and it is usual to construct different types of description for each purpose. For example:

> a teaching manual, for those learning to program;
>
> a reference manual, explaining more advanced details to the initiated programmer;
>
> a programmers' guide, describing the use of a particular implementation;
>
> an implementors' manual, containing suitable instructions and hints;

even the language compiler itself may be regarded as a form of language description - indeed, the only one which is meaningful to a machine.

This paper deals with yet another purpose for which a language description is required, namely the standardisation of the language across many implementations; and a form of description which is suitable for this purpose is rather different from the forms which are familiar from their use in other contexts. A programming language standard, like standards for other artefacts, is intended to lay down a set of necessary properties of a certain class of products (implementations); and these conditions must be satisfied by any product which claims to conform to the standard. Most standards will leave the product designer considerable freedom in specifying its non-essential features, and will even permit variation within predefined limits for the essential design parameters. Similarly, a language description formulated for language standardisation purposes must permit reasonable variation of implementation details, so that the language can be successfully implemented on a variety of computers, using a variety of implementation techniques. However, this variety must be constrained within certain bounds, since otherwise it would not be possible to write programs which could be accepted and run successfully on all implementations.

Most descriptions of a standard are carefully and fairly unambiguously expressed in an ordinary natural language, supplemented by a few formulae, tables, and diagrams; and they use terminology which is familiar and widely understood in the relevant field. However, in the field of programming languages, there has not yet developed any established terminology, as will be revealed be a comparison of the descriptions of existing and proposed language standards. Furthermore, the larger programming languages are probably more complicated in their structure than any other product which has ever been considered as a candidate for standardisation. A need has therefore been expressed (1) for some rigorous or even formal technique for describing a language in a manner suitable for standardisation. Part I of this paper suggests that the axiomatic method, as practiced by geometers, mathematicians, and logicians, may be well suited for this purpose.

In the construction of axiom sets modelling the essential features of programming languages, the following design criteria are relevant:

1. The axioms should not place constraints on language implementors which might be unacceptable for certain hardware designs or configurations of equipment.

2. The axioms should be sufficiently deterministic for the programmer to be able to secure the results which he wants on any implementation which satisfies the axioms.

3. The axioms should be formulated in such a way that many of them will apply unchanged to many programming languages. In this way it should be possible to gain an insight into the genuine similarities and differences between the various languages.

4. The axioms should be reasonably independent of each other, so that the language designer may freely discuss one axiom or a small group of axioms in isolation, without fear of unexpected interactions and alterations in other parts of the language.

5. The primitive (undefined) terms in the axioms should correspond with our pre-existing intuitive understanding of the behaviour of programming languages, and the axioms should recommend themselves as "self-evident" to one who is familiar with programming. This will ease the task of constructing proofs concerning the correctness of programs.

6. The number of primitive concepts and axioms should be kept to a reasonable minimum.

2. Simple types.

The easiest and most obvious application of the axiomatic method is in the definition of the primitive operations on simple values of the various types. Sets of axioms which define the structure and behaviour of a simple type will bear a close resemblance to axiom sets already familiar to mathematical logicians, with changes made necessary only by the finitude of computer representations of the values concerned. A suggestion for using axiom sets for defining the primitive operations of a programming language was independently made by Laski (2).

2.1 The Boolean type.

In the case of the Boolean type of ALGOL 60, the axioms are identical to those which describe Boolean algebra. The axioms are not in any way tautological or redundant, even if at first glance they appear so.

1. true and false are the only truthvalues
2. $\neg$true = false and $\neg$false = true.
3. true $\wedge$ true = true and true $\wedge$ false = false $\wedge$ true = false $\wedge$ false = false.
4. false $\vee$ false = false and true $\vee$ false = false $\vee$ true = true $\vee$ true = true.

2.2 The integer type.

The most familiar axiomatisation of the properties of integers is that given by Peano (3), which is based on the concept of one number x' being the successor of another number x, i.e. x' = x + 1.

The axioms given below are slightly adapted from those of Peano (3) since they have to deal with negative numbers and with the possible finitude of the number range. However, the basic concept is still that of the successor; and the fact that x is the successor of y is expressed ySx.

It is useful here to introduce the concept of an ancestor or a descendant with respect to a given relationship r. If the relationship r is that of being a person's parent, then the r-descendants of a person are his children, and his childrens' children and so on; and his r-ancestors are his ancestors in the normal sense. More rigorously, x is an r-ancestor of y if there is a sequence $x_1, x_2, \dots x_n$ such that $x = x_1$, $x_n = y$, and $x_i\, r\, x_{i+1}$ for $1 \leq i < n$. Putting $n = 1$, we obtain the convention that everything is its own r-ancestor. Similarly x is an r-descendant of y if it is an r^{-1}-descendant of y, where r^{-1} is the relational inverse of r. (eg. offspring rather than parent).

These concepts can be axiomatised as follows:-

1. x is an r-ancestor of x (reflexivity)
2. If x is an r-ancestor of y and z r x then z is an r-ancestor of y (closure).
3. If x has property P and whenever y has property P and zry then z has property P then all r-ancestors of x have property P (induction)
4. $x\, r^{-1} y \equiv y\, r\, x$
5. x is an r-descendant of y $\equiv$ x is an r^{-1}-ancestor of y.

In future sections we shall also need further concepts. A relationship r is said to be <u>convergent</u> if it is a many-one relationship; and is <u>linear</u> if it is a one-one relationship. A relationship r is said to be <u>non-cyclic</u> if there is no set of elements $x_1, x_2, \dots x_n$ such that $x_n\, r\, x_1$ and $x_i\, r\, x_{i+1}$ for $1 \leq i < n$. These concepts may be expressed rigorously:

6. r is convergent $=_{df} \forall x,y,z\ (x\, r\, y$ and $x\, r\, z \supset y = z)$
7. r is linear $=_{df}$ both r and r^{-1} are convergent
8. r is noncyclic $=_{df} \forall x,y.$ (x is an r-ancestor of y and y is an r-ancestor of x $\supset x=y$).

Now it is easy to characterise the essential properties of the successor relationship S, and of integers.

1. S is linear
2. x is an integer if and only if x is an S-ancestor or an S-descendant of zero (0)
3. $-1S0$ & $0S1$
4. $x + 0 = x$
5. ySy' & $zSz' \supset (z = x + y \equiv z' = x + y')$
6. $x \times 0 = 0$
7. ysy' & $z' = z + x \supset (z = x \times y \equiv z' = x \times y')$

The remaining arithmetic operations can be similarly described.

The main problem is constructing axioms for computer arithmetic is to avoid making any presuppositions on the range of integers provided by an implementation, and to permit a variety of treatments for integer overflow. For example, many implementations will fail to detect integer overflow, and will use modulo arithmetic throughout. Other more sophisticated implementations will detect integer overflow, and jump to some interrupt program to take alternative action. Finally, one should not rule out the possibility that an implementation places effectively no bound on the size of an integer.

It appears that the axiomatic method not only permits the decision on these points to be made by an implementation; but it also permits the implementor to describe his decision in terms of supplementary axioms, which make the previous general axioms more categorical, at the expense of making them more machine-dependent. For example, the decisions described above can be embodied in a choice of one of the three following axioms:

8. S is cyclic (modulo arithmetic)

9. intmin and intmax are integers but there is no integer y such that ySintmin or intmaxSy (finite range arithmetic)

10. S is non cyclic and for all x there is a y such that xSy and a z such that zSx (infinite range).

2.3 The real type.

Axioms describing the necessary properties of floating point arithmetic can be constructed fairly simply in terms of integer arithmetic, using a technique similar to that proposed by Peano to define the arithmetic of fractions. However, it is essential to recognise that a floating point number is not itself a fraction, but only an approximation to a fraction a/b; in fact, the same floating point number is an approximation to many different fractions. If, for example, an implementation uses a 3-digit decimal mantissa, the number $.123_{10}0$ is an approximation for the fractions:

1226/10000, 1234/10000, 123/1001,

and even 123/1000

If x is a floating point number, we write x = A(a/b) to express the notion that x is an approximation of the fraction a/b, where a and b are integers. The properties of A are described by axioms:

1. A is a many-one function

2. $A(a/b) = A(c \times a/c \times b)$

3. $x = A(a/b)\ \&\ x = A(c/b)\ \&\ a \leq i \leq c \supset x = A(i/b)$

4. If x,y, and z are floating point, and $x = y + z$ then there exist integers a,b,c such that $x = A((a+b)/c)$ and $y = A(a/b)$ & $z = A(b/c)$

5. If x,y, and z are floating point and $x = y \times z$ then there exist integers a,b,c,d, such that $x = A((a \times b)/(c \times d))$ and $y = A(a/c)$ and $z = A(b/d)$

A proposal for the axiomatisation of floating point arithmetic was made by van Wijngaarden (4). This was based on the arithmetic of the real continuum rather than that of integers or fractions.

3. Semantics: static considerations.

The primitive operands and operations of a scientific programming language are intended to behave like the familiar objects of mathematics; and it is not surprising that the axioms which describe their behaviour should have a fairly familiar appearance. However, in the axiomatisation of the semantic and dynamic properties of a programming language, there is not such an established tradition to draw upon, and we must rely more on intuition for the formulation and verification of axiom sets.

In this section, we consider the static properties and structure of a program, excluding consideration of its dynamic execution.

3.1 Program Structure.

In dealing with the structure of programs, it is very inconvenient to regard them merely as linear streams of characters. It is more appropriate to consider them as tree structures, in which the relationships between the meaningful parts and the whole have been made explicit, as is done, for example, by a syntactic analysis. In this way, it is possible to separate a discussion of semantics from the details of syntax and notation, as was suggested by McCarthy (5) in his proposal for an "abstract syntax".

As an example of the correspondence between a program text and its tree structure, consider the phrase:

a + b x sinh(-x)

which is pictured as a tree in figure 1. Each circle represents a meaningful subphrase of this phrase; and each phrase is connected by a line to all its immediately constituent subphrases. There is no need to impose a limit to the number of branches emanating from each node, but in practice, most phrases will have either two constituents, or one constituent, or, of course, none.

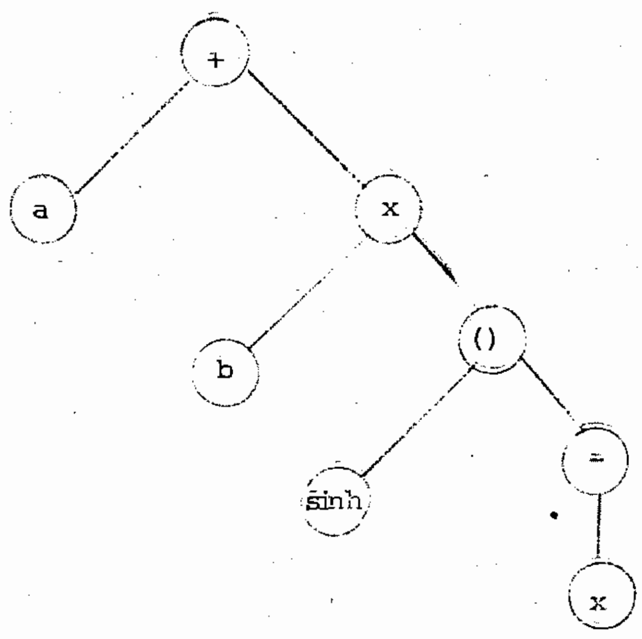

Fig 1.

In order to deal with the structure of phrases, we introduce a primitive relation C which holds between two phrases which are adjacent nodes on some branch of the tree; thus xCy indicates that nodes x and y are connected by a single line leading upward from x to y i.e., x is an (immediate) constituent of y. We also have the concept of one phrase being contained in another; if x is contained in y, this means that there is a series of lines (a path) leading upward from x to y. Obviously, x is contained in y if, and only if, it is a C-ancestor of y.

1. x is contained in y = df x is a C-ancestor of y.

The main property of the C-relationship is that each phrase is a constituent of at most one other phrase; that no phrase is contained in itself. These facts are embodied in the axiom:

2. C is convergent and non-cyclic.

It will be helpful here to introduce a few auxiliary definitions and notations.

3. An atom x is a phrase with no path leading down from it. i.e. there is no y such that yCx.

4. A disjoint phrase x is a phrase which is not a part of any other phrase, i.e. there is no y such that xCy.

5. The whole of a phrase x is defined as the disjoint phrase which contains it. i.e. x is contained in the whole of x, and the whole of x is disjoint.

6. A monadic phrase x is one which has only a single constituent y.

7. A dyadic phrase x is one which has exactly two immediate constituents which are known as its left constituent and its right constituent.

We shall often use the notation x = y to indicate that x is monadic, and y is its only constituent; or the notation x = y.z to indicate that x is dyadic and y is its left constituent and z is its right constituent.

It is obvious that the C-relation is a many-one rather than a one-one relation, since a phrase may have many constituents. Functions which act as inverses of the C-operation are known as selectors, i.e., if S is a selector then S(x) Cx for all x.

Examples of selectors are:

1. The function U which maps a monadic phrase onto its unique consituent.

2. The functions H and T, which map a dyadic phrase onto its left and right constituents respectively.

3. The selector "body" which maps a block onto the statement or sequence of statements which form the body of the block.

3.2 Semantic categories.

In the description of a programming language, we are concerned with the behaviour and properties of the individual phrases of each program. Phrases with essentially similar semantic properties may be grouped together in semantic categories, each of which is given some suggestive name, for example "truthvalue", "integer", "assignment", "jump". It is natural to use the $\in$ -notation of set-theory to represent the category membership of a phrase, for example, "x $\in$ integer"; or less formally, one may simply write "x is an integer".

The behaviour and properties of elements of each semantic category will be defined in terms of axioms which are true of all members of that category; examples of axioms applying to the category of integers have already been given. If axioms are to apply to more than one language, it will be useful to define a fairly large number of semantic categories, and then to specify any particular language as not containing any phrases of those categories not represented in the language. Thus a language which does not deal with reals may be specified by stating that this category is empty. This freedom to introduce categories which are empty in a given language will be even more useful when considering intersections and other relationships between categories.

It is a great convenience in describing the properties of phrases to postulate a separate semantic category to cover each aspect of its behaviour. For example, an arithmetic expression involving multiplication shares many properties with all other arithmetic expressions, and even shares some of its properties (e.g. sequentiality) with statements. Such a phrase will therefore belong to many categories; and if these categories have been previously defined, then a particular category can be defined as the intersection of these more embracing categories. It seems harmless to postulate the existence of the intersection of two categories, although in many cases the intersection will be empty, either because the properties of the categories are mutually contradictory, or because the intersection category is simply not represented in a given language. The intersection of two categories may be formally denoted by the traditional set-intersection notation (n); or it may be informally expressed by combination of adjectives and nouns. However, it must not be supposed that the set-union of categories in necessarily a category; in many cases, it would be misleading to do so.

Another useful technique for dealing at one time with several semantic categories sharing certain properties is to postulate certain relationships between categories. For example, let C stand for a category consisting of the values of some simple type, for example, truthvalues or integers. Corresponding to this category there may be a category of variables to which values of the given type may be assigned: these will be known as C-variables. Similarly, there will be a category of expressions, which when evaluated will designate an element of category C: these will be known as C-expressions. Finally, there will be a class of assignment statements which assign a value of category C to a C-variable: these will be known as C-assignments. This technique is very useful in modelling the type-constraints which can and should be checked at compile-time. For example, it is a necessary property of a C-assignment that its right hand side shall be a C-expression, and its left hand side a C-variable-expression. The use of this form of quantification does not imply that in any particular language a non-empty category, or even a meaningful category, will result from an arbitrary substitutuion for the category variable C. Thus, although "procedure" is a non-empty category in many languages, few languages permit "procedure-assignments." This technique of dealing with type-constraints is a close semantic analogue of the syntactic method proposed by van Wijngaarden (6).

3.3 Denotation.

In a natural language, the meaning of a sentence is derived ultimately from the meaning or denotation of the individual words of which the sentence is composed. This is true also in a programming language; it is only the atomic constituents of a phrase which establish a connection between the phrase and the entities which form the subject matter of the phrase. This is because the atoms are capable of denoting some object other than themselves; for example, a constant may denote a certain value of some type. Identifiers in general denote objects which are themselves phrases; for example, an occurrence of a procedure identifier denotes a procedure body together with its heading, and an occurrence of a label identifier denotes the compound tail to which its defining occurrence is prefixed. An identifier for a variable or formal parameter denotes the relevant variable or formal; and it is convenient to regard variables and formal parameters themselves as atomic "phrases", which are constituent parts of the block to which the quantities are local. We write y = Dx (y is the denotation of x) to indicate that x denotes y.

Note that the relationship of denotation is an entirely static one, subsisting between phrases and other phrases or objects. This relationship is permaent, and cannot change during the execution of the program. Thus a variable is denoted by one or more occurrences of a variable identifier, but the relationship which holds between a variable and its current value at any time is not one of denotation.

In constructing a picture of a phrase which includes the denotations of its atomic constituents, an arrow may be drawn from the denoting atom to a circle representing the object which it denotes, for example:

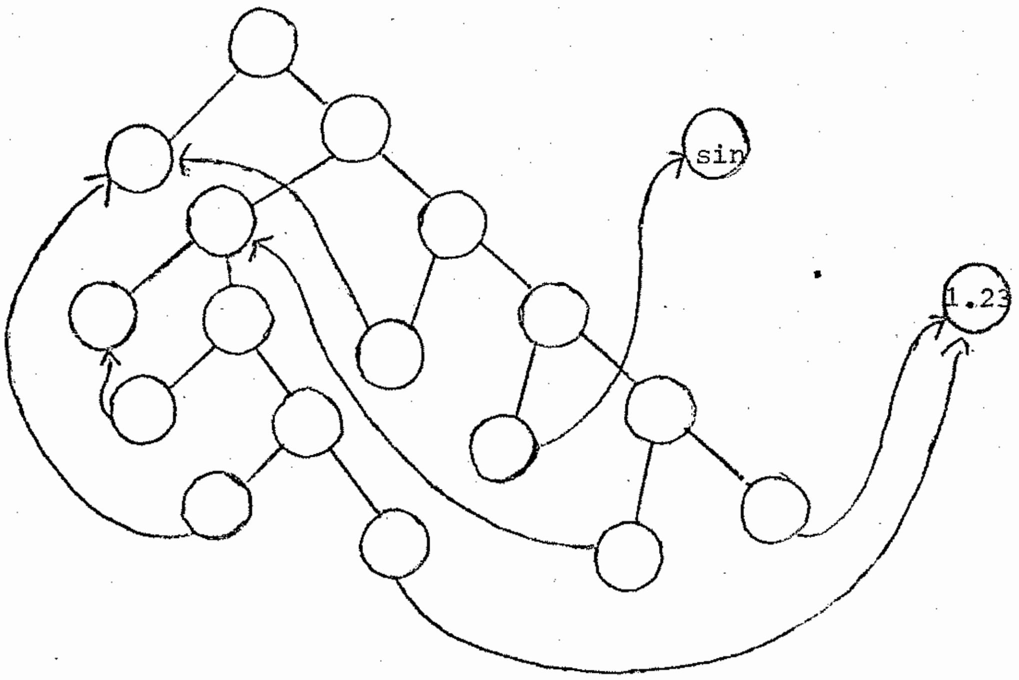

Fig 2.

Alternatively, in the case of larger phrases, a phrase may be labelled by an identifier, and the same identifier written within an atom indicates that the atom denotes the labelled phrase. It is most important to realise that more than one atomic phrase can denote one and the same object. In this aspect, denotation is quite different from structural relationship C which connects the subphrases of a particular phrase. Two disjoint atoms may share the same denotation whereas it is not possible for two disjoint phrases to share the same subphrase.

3.4 Expressions and their results.

We are now in a position to deal with the category of phrases known as expressions. Expressions have the capability of delivering a result on evaluation; we therefore postulate a primitive (undefined) function, which maps an expression x onto its result y (y = result (x)); provided, of course, that this result exists.

Expressions are normally classified in accordance with the type of the result which they deliver, for example, integer-expressions are capable of delivering integer results, but not reals. This kind of type-constraint may be expressed generally by describing the properties of C-expressions, where C stands for an arbitrary semantic category.

An expression in a programming language can usually be converted to a tree-form, in which every phrase is either dyadic, monadic, or atomic. An atomic C-expression is normally known as a C-constant, and is assumed to denote some value of category C. This denotation is taken as the result of the constant. A monadic C-expression has associated with it a monadic operator P. Its result is obtained by applying P to the result delivered by its only constituent. A dyadic C-expression has similarly a dyadic operator P, which is applied to the results of the constituent subphrases to deliver the result of the whole expression. These facts are summarised as follows:

1. If x is a C-expression and has a result, then this result is a C.
2. If x is an atomic C-expression, then result (x) = denotation (x).
3. If a C-expression x = .y is a P-operation then result (x) = P(result(y)).
4. If a C-expression x = y.z is a P-operation then result (x) = P(result(y), result(z)).

Note the equality of the last two axioms is a strong equality. The existence of the results of the constituents together with the applicability of the operator to these results, forms a necessary and sufficient condition for the existence of the result of the whole phrase. It is assumed that the existence and identity of the result of a P-operation can be derived from axioms defining the operator P.

We next deal with conditional C-expressions, in which the result is selected from a pair of alternatives in accordance with the truth or falsity of a condition.

5. If a C-expression x = i. (t.e) is a conditional then
   t and e are C-expressions
   i is a truthvalue-expression
   if result (i) = true then result (x) = result (t)
   if result (i) = false then result (x) = result (e)
   if i has no result, then neither has x.

This formulation leaves open the possibility that the discarded limb of the conditional may also have a result, which is irrelevant to the result of the conditional. In a practical implementation, this result would never be evaluated, since it is certainly never needed, and might not even exist. Nevertheless, there seems to be no harm in permitting a phrase to have a result, which is ignored.

3.5 Procedures.

The determination of the results of expressions is by itself a fairly trivial task, and can be carried out by a desk calculating machine as well as an automatic digital computer. However, it is not an insignificant part of the definition of a high-level programming language. The definition can be readily extended to cover the evaluation of the whole class of general recursive functions, provided that a suitable axiomatisation can be given to the concept of a procedure, a procedure call, and of parameter substitution.

A procedure call or a function designator is defined in ALGOL 60 as an operation of making a new copy of some procedure body, with replacement of formal parameters by the actuals. The resulting copy is then evaluated, and its result (if any) is taken as the result of the procedure call. These facts may be expressed:

1. If x = .y is a C-procedure call then y is a C-procedure-expression result (y) is a C-expression (i.e. the new copy of the procedure body) & result (x) = result (result (y)).

In this axiom, it is assumed that y delivers a result which is a copy of some procedure body. If there is no parameter-subsitution, this is relatively trivial; the procedure identifier is represented by an atom which denotes the procedure body itself, and delivers a copy of it as result. If a single parameter substitution is involved, the procedure-expression will be a dyadic phrase, of which the left constituent delivers a procedure, and the right constituent delivers the actual parameter. The procedure itself is also a dyadic phrase, of which the left constituent is the formal parameter and the right constituent is the body of the procedure. If more than one parameter has to be substituted, this can be achieved by successive substitution of each parameter in the result of previous substitutions; so this case does not involve any new principle. Thus two axioms suffice:

2. If x = .y is a procedure-expression, then result (x) is a copy of the denotation of y.

3. If x - y.z is a procedure-expression, then result (x) is a copy of the right constituent of the result of y, with result (z) substituted for the left constituent of the result of y.

The concept of copying and substitution which underlie the axioms quoted above, are concepts which have a strong intuitive basis; however, it is still desirable to make the intuitive content explicit by means of axioms, as Peano did in the case of counting. We write xMy to indicate that x is a copy of y, with or without parameter substitution; to make the substitution explicit, we write $x\ M^f_a\ y$ to indicate that the actual a has been substituted for the formal f. The first properties of copying are that each copy has at most one original, although one original may have many copies and that no phrase is directly or indirectly a copy of itself, i.e.

4. M is convergent and non-cyclic.

Next we must ensure that a copy has all the same semantic properties as its original, and that it has the same structure.

5. If xMx' and C is a syntactic category then x ∈ C if and only if x' ∈ C.

6. If xMx' and s is a selector then s(x) M s(x').

Treatment of denotation relationships are rather more complicated, since we must distinguish the case where an atom of the original denotes a phrase of the original itself. In this case we want to ensure that the atom within the copy denotes the corresponding phrase in the copy itself, rather than denoting the same phrase within the original. Thus in figure 3, we wish to copy (a) as (b) rather than as (c). The relevant axiom is:

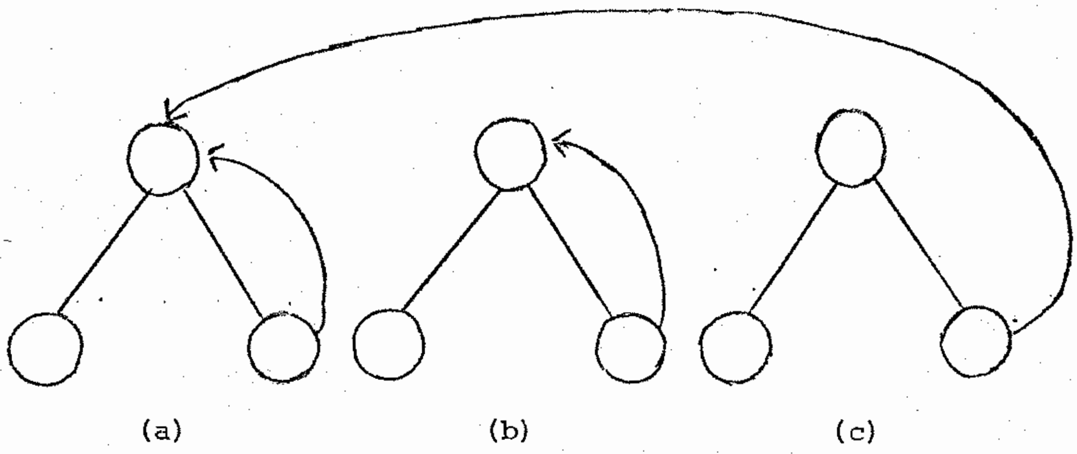

Fig 3.

7. If xMx' and whole (x)My and the denotation of x' is contained in y then the denotation (x) is contained in the whole of x and denotation (x)M denotation (x')

We now consider the case where the original of an atomic phrase is the formal parameter, and the phrase itself is therefore the actual parameter. An occurrence of a formal parameter is represented as an atomic phrase denoting the formal parameter itself and the copy of the formal denotes the actual parameter. This models the old FORTRAN parameter mechanism (call "by denotation"); the ALGOL parameter mechanism could be described by different axioms, or by transforming name parameters into functions, in the manner described by Landin (6)

8. If $xM_a^f x'$ & x' denotes f then x denotes a.

Finally we need to ensure that any atoms of the original which denote constants or global quantities are copied as atoms with identical denotation.

9. If $xM_a^f x'$ & x' is not contained in the original of the whole of x then x and x' have the same denotation.

4. Dynamic Concepts.

Up to the present point, the axioms have related to a purely functional language, which is nevertheless powerful enough to compute all computable functions by general recursive techniques. There has been no need to introduce any concept of "executing" the programs in some time-sequence; although it is obvious that any actual implementation of the language will evaluate functions in some time-sequence, it does not matter what this time-sequence is. A sensible implementation will avoid wasting time on the evaluation of the discarded limbs of conditionals which may often be undefined; but there is no reason at this stage to prohibit the implementation from doing so, provided that it does not go on for ever, and thereby fail to deliver any result at all.

However, there are many applications of computers for which the purely functional approach to programming is impractical, for example, in dealing with large arrays, input/output, and in the simulation of discrete event systems. Languages designed for these purposes will contain assignments, declarations, compound statements, input/output, and even jumps. The meaning of these features can only be explained in terms of a more or less determinate sequence of execution of the individual phrases of the program.

4.1 Events.

One can axiomatise the execution of a program and its subphrases by postulating an abstract set of events which are related by a successor-relationship S. The fact that event x is immediately followed by event y is indicated by writing xSy. In most programming languages, one could postulate that S is a linear relationship, guaranteeing that each event either precedes or follows each other event; but in languages which permit parallel actions, this is no longer true. Most of the axioms quoted in the following sections have been formulated to apply with equal validity to languages with or without parallelism. However, even in a parallel enviroment it seems reasonable to postulate that S is non-cyclic. Each event in the execution of a program is associated with some phrase of the program or with a copy (more precisely an M-ancestor) of a phrase of the program. We distinguish the case of the event associated with the initiation of the execution of the phrase, from the case when it terminates execution of the phrase; if x is a phrase and e is an event, we write e = init (x) in the former case, and e = termin (x) in the latter. We can now define the execution of a program P as the set of events which are S-descendants of init (P). Also, we can define what is meant by the assertion that a program P terminates successfully: it is equivalent to saying that termin (P) is an S-descendant of init (P). We use the letter P to denote the program under execution.

In a procedural language, it is a regular occurence that an expression of the program is executed more than once, and on each case delivers a different result. This would seem to invalidate some of the previous axioms, which assume the uniqueness of the result of each expression. This difficulty can be evaded by ensuring in our model that no phrase is executed more than once, and in every case of "repeated" execution, it is in fact a copy of the phrase which is executed. If this convention is observed, it is possible to state that each phrase in initiated at most once, and terminated at most once. Furthermore, it is obvious that the termination of a phrase (if it occurs) must be an S-descendant of its initiation.

The above general remarks are summarised together with some useful definitions, as follows:-

1. S is non-cyclic.
2. x is an event if and only if it is an S-descendant of init (P)
3. a phrase x is initiated $=_{df}$ init (x) is an event.
4. a phrase x is successfully terminated $=_{df}$ termin (x) is an event.
5. init and termin are one-one mappings between events and phrases.

6. termin (x), if it exists, in an S-descendant of init (x).

7. x precedes y $=_{df}$ y follows x $=_{df}$ x is an S-ancestor of y, where x and y are events.

It is useful to extend this last definition to cases when x or y are phrases rather than events. In this case, the time of execution of the phrase is identified with the event of its termination, for example:-

8. x precedes y = df y follows x = df termin (x) precedes termin (y) where x and y are phrases.

4.2 Sequencing.

We are now in a position to examine the sequence in which "control" is passed between the various phrases of a program. In many cases, when a phrase is initiated, control passes immediately to one of its constituents; and if execution of the constituent terminates, then control passes to another constituent, and so on, until all constituents are terminated. Phrases which have this property of passing control to their constituents are known as <u>normal</u>; they include monadic and dyadic expressions. Note that the "normality" of a phrase does not prohibit parallel execution of its constituents.

There is a class of non-normal phrases which pass control to only one of their constituents, and which terminate on termination of this constituent; none of the other constituents are executed. Phrases with this characteristic are known as <u>selections</u>; an example of such a selection is given by the dyadic phrase which forms the right constituent of a conditional.

A third mode of sequencing is exhibited by procedure calls, which transmit control to some copied phrase, and which terminate on termination of this phrase. These modes of sequencing are more formally described in the following axioms.

1. If a phrase x is normal and it is initiated then it has at least one constituent y such that init (x) S init (y) and if any constituent is not initiated, this can only be because some other constituent is initiated but not successfully terminated.

Here we introduce an auxiliary definition.

2. x terminates y =df termin (x) S termin (y) and result (x) = result (y).

3. If a phrase x is a selection and is initiated then it has a constituent y such that init (x) S init (y) and y terminates x and y is the only constituent of x which is initiated.

4. If x is a call of y then there is a unique z such that y = result (z) and termin (z) S init (y) and y terminates x.

5. Expressions, except calls, are normal, and if i(t.e) is a conditional then t.e is a selection and if .y is a call, and is initiated, then .y is a call of result (y).

4.3 Variables and Assignments.

One of the main characteristics of computing machines, and of the languages which control their operation, is that they are capable of storing values in some storage medium, of fetching these values whenever required, and of changing the values by assignment during the course of a computation. It is this last feature which most clearly distinguishes procedure-oriented programming languages from the more traditional branches of mathematics, which have already been successfully formalised.

The ease with which assignment can be modelled is a crucial test of the success of a rigorous method for programming language description.

An assignment involves two items, a value which is to be assigned and a variable to receive the value; and it is laid down that these must be of identical type. If the language permits automatic type-conversion, then it is assumed that all necessary transfer functions are inserted by a preliminary scan of the source text. The concept of a value of any type is defined by the axioms governing that type. A variable, on the other hand, cannot be dealt with on the same static basis, since in many languages a variable comes into existence dyanamically on entry to the block to which the variable is local. Now suppose we model a variable as an atomic phrase which is an actual constituent of the block to which that variable is local, and then arrange to make a fresh copy of the block whenever we enter it. This will automatically ensure that we obtain fresh copies of all the variables on block entry, which is exactly what we want. Let us also represent each occurrence of the variable identifier in the program as an atom which <u>denotes</u> the phrase representing the variable. Now the copying process, as described in section 3.5 (7), will ensure that all variable identifiers in the new copy will correctly denote the <u>new</u> copies of the variables rather than the old.

Thus it is natural to equate a variable with the <u>declaratory</u> occurrence of its identifier; and every non-declaratory occurrence is assumed to denote this. Furthermore, we wish to state categorically that declarations occur only as constituents of a block, and it is not possible to refer to a declared quantity from outside its scope. In addition to declarations, a block normally contains a <u>body</u>, which is executed when the block is entered; we introduce a selector to select the body from a block. The remarks made above are summarised in the following axioms.

1. If x is a C-declaration then x is a C; and x is never initiated; and if x is a constituent of b, then b is a block and if y denotes x then some M-descendant of y is contained in b.

2. If x = .y is a block-generator then result (y) is a copy of y and x is a call of result (y).

3. If b is a block then x = body (b) if and only if x is a constituent of b but is not a declaration.

4. If x is a block and is initiated then init (x) S init (body(x)) and body (x) terminates x.

The body of a block is normally a statement or sequence of statements, known as a compound tail in ALGOL 60 terminology. A sequence of phrases is conveniently represented as a dyadic phrase, the left constituent of which is the first statement of the sequence; the other constituent is the remainder of the sequence, which may itself be a sequence or a single statement (or expression). If a block body is to pass back a result, this is conveniently taken from the evaluation of the <u>last</u> statement or expression of the sequence. These facts are summarised as follows.

5. If x = y.z is a sequence then x is normal, termin (y) S init (z); z terminates x.

We now need to introduce the concept of a current value of a variable at a given time. If e is an event, g is a value, and v is a variable, we write g = current value (v,e) to signify that g is the current value of v at the time of occurrence of event e. We also introduce the category of an assignment statement, which is a normal dyadic phrase, whose left constituent delivers a variable and whose right constituent delivers a value; and assignment has the effect of defining the current value of the relevant variable on termination of the assignment.

6. If a = 1.r is a C-assignment then 1 is a C-variable-expression; r is a C-expression; a is normal; r terminates a; current value (result (1), termin (a)) = result ($\underset{r}{y}$).

Note that an assignment has a result equal to that of its right hand side; this is useful in dealing with languages permitting multiple assignment, and is harmless in those that do not.

It is essential to draw a distinction between an occurrence of a variable identifier as the left constituent of an assignment, and its occurrence as a primary in an expression. The left-hand occurrence can be represented simply as an atom denoting the variable or by a variable-expression in the general case; when this is evaluated, it delivers the variable itself as a result, which is what is wanted. However, when the identifier occurs on the right-hand side, what is required is the <u>current value</u> rather than the variable itself. Such occurrences as primaries are known as <u>variable-evaluators</u>, and are represented as monadic phrases, whose only constituent delivers the variable as a result, and which itself delivers the current value of that variable:

7. If x = .y is a C-variable-evaluator then x is normal; y is a C-variable-expression; if q = currentvalue (result (y), termin (y)) then termin (y) S termin (x) and result (x) = q.

Finally, we need an axiom to state that the current value of a variable does not change between one assignment to that variable and the next. For a language which does not permit parallel assignments to the same variable, one could formulate the rule that the only reason for a change in value is an intervening assignment. However, to cater for the possibility of parallelism, one would need to be more subtle:

8. If all successfully terminated assignments to x either precede e_1 or strictly follow e_2 then currentvalue (x, e_2) = currentvalue (x, e_1).

4.4 <u>Jumps.</u>

In addition to assignment, the jump is another aspect of procedural programming languages which presents problems for elegant formalisation. The treatment given here follows closely that given by Landin in (7).

In ALGOL 60, the destination of a jump is determined as the result of evaluating a designational expression. This result is generally supposed to be a label, which may be identified with (a copy of) that part of the compound tail to which the defining occurrence of the label is prefixed. This section of program will be wholly contained within the block to which the label is local; in fact it will (in general) be that part of the compound tail of the block which occurs between the <u>defining</u> occurrence of the label identifier and the <u>end</u> of the block. Such a section of program will be known as a <u>program-point</u>.

We may now represent a <u>go to</u> statement as a monadic phrase whose only constituent is a designational expression. We assume that a designational expression will deliver as its result a <u>fresh copy</u> of some program point. This copy is then initiated; and if it terminates, its successor is the termination, <u>not</u> of the <u>go to</u> statement, but of the block to which the program point is local. In fact, the jump itself can never terminate successfully. The block to which a program point is local may be determined as the smallest block containing an M-descendant of the program point.

1. x = locality (p), where p is a program point, = df x is a block, and x contains an M-descendant q or p, and if any other block contains an M-descendant of p, then it also contains an M-descendant of x.

2. If x = .y is a jump then y is a program-point-expression; if y has a result than locality (result (y)) is a call of result (y).

In the case where a label is prefixed to the final end of a block, the relevant program point consists of a null statement. Null statements are also useful in representing the suppressed <u>else</u> part of a conditional statement.

3. If x is a null-statement then x is atomic; init (x) S termin (x); x has no result.

5. <u>Further features.</u>

The language features axiomatised in the preceding sections cover the most basic features of procedure-oriented programming languages. In fact, they match almost exactly the features of Landin's Imperative Applicative Expressions (7) and these have been shown to be sufficient for the semantic definition of the whole of ALGOL 60. However, it does seem desirable to use the axiomatic method to explicate certain more advanced features of modern programming languages, for example dynamic arrays, input/output, record handling, and even parralellism.

5.1 <u>Dynamic Arrays.</u>

An array is a homogeneous collection of elements of the same category. In the case of a single-dimensional array, the elements are simple variables; but the elements of a multi-dimensional array will themselves be arrays of one lower dimension. It seems reasonable to postulate that no element belongs to more than one array, and that no element appears more than once within any given array. It is possible therefore to represent an array as a <u>phrase</u>, whose constituents are identified with the elements of the array.

Two new functions "l" and "u" are introduced, which map each array onto its lower and upper subscript bounds respectively. We also need a function "subscript", which maps an integer within the subscript range of an array onto a <u>selector</u> which will select the corresponding element of that array. These functions are connected by the following axioms, in which A is assumed to be of category C-array.

1. for all i such that l (A) $\leq$ i $\leq$ u (A), subscript (i)(A) exists and is a C

2. l (subscript (i) A)= l(subscript (j)A) and u(subscript (i)A)= u(subscript (j)A)

(i.e. multidimensional arrays are rectangular).

In a language with static array bounds, like FORTRAN, there is no need for any further axioms dealing with the dynamic aspect of arrays. A multidimensional array can be converted (by columns) into the corresponding single-dimensional array, and all references to it can be converted by a static syntactic translator into the appropriate reference to this single-dimensional array. The array declaration (DIMENSION statement) can be similarly expanded into an array with the appropriate number of elements, just as it is when translated into machine code.

However, in a language such as ALGOL 60, which permits arrays to be constructed dynamically, with dynamic bounds, the problem cannot be dealt with by static translation; some semantic, or run-time, mechanism is required as well. We therefore need to axiomatise a concept of an array-generator, which creates new instances of arrays. An array-generator needs three parameters:

1. An indication of the category of the elements, which in the case of multidimensional arrays may be themselves arrays
2. A specification of the lower bound.
3. A specification of the upper bound.

It is convenient to specify the category of the elements by an example, which is copied the appropriate number of times.

4. If x = c. (lb.ub) is an array-generator then x and lb.ub are normal; lb and ub are integer-designators; if x terminates successfully then result (x) is an array

   and l (result (x)) = result (lb)

   and u (result (x)) = result (ub)

   and each constituent of x is a copy of result (c).

It is necessary to insert the proviso that x terminates successfully, since the array generator may fail as a result of inadequacy of storage in any finite implementation.

There is still the requirement in ALGOL 60 to link up all occurrences of the array identifier inside the block with the newly generated array. A suitable technique for doing this is to use the parameter mechanism to set up the linkage. Thus the array identifier is regarded as a formal parameter, and the array generator features as the corresponding actual parameter. This means that the subscript bounds of the array will naturally be evaluated in the enviroment of the surrounding block, thereby evading a slight but tricky problem in the definition, implementation and use of ALGOL 60, at the expense of a fairly severe syntactic transformation of the original source program.

Finally, we need to ensure that whenever an array is generated it is an entirely fresh array, not the same as any other array previously generated. This is done by introducing the syntactic category of "generator", which is assumed to include all array-generators.

5. If x and y are generators and result (x) = result (y) then x = y.
6. An array-generator is a generator

5.2 Input and output.

Input and output in the conventional sense of FORTRAN and ALGOL 60 is concerned with ordered sequences of values. Legible (formatted) input/output can be dealt with on the same basis as communication with backing stores, which normally takes place in non-legible internal form. This is done by regarding characters as a specific type of value. In our axiomatisation, we may regard a file as a sequence of constants, each of which denote a value of some simple type. This sequence can be represented in the same way as the sequence of statements in a compound tail, i.e., as a dyadic phrase, whose left constituent is the "current" element of the file, and whose right constituent is the remainder of the file.

The relevant axiom is

1. If x = a.b is a c-file then a denotes a C and b is either a C-file or an end-of-file indicator.

In any given computation or sub-computation, it is assumed that a file is either an input-C-file or an output-C-file. The only operation on an output-C-file is an output operation, and the only operation on an input-C-file is an input-operation. The properties of these operations are easily described:

2. If x = y.z is a C-output operation, then x is normal; y is a C-expression; z is an output-C-file-expression. If x is successfully terminated and if currentposition (result (z), termin (z)) = a.b then denotation (a) = result (y) and current position (result (z), termin (x)) = b.

3. If x = .y is a C-input-operation then x is normal; y is an input-C-file designator. If x is successfully terminated and currentposition (result (y), termin (y)) = a.b then result (x) = denotation (a). and currentposition (a.b, termin (x)) = b.

In these two axioms, there is no means of proving the successful termination of an input or output operation. This permits an implementation to stop the program, or jump to an error routine, in the event of hardware malfunction. Furthermore, the result of reading beyond the end-of-file indication has been carefully left undefined.

There is still one loose end in the axioms 2 and 3: it is necessary to guarantee that the file remains stationary in between successive input and output operations. This can be done as follows:-

4. x is an operation of f = df (x = y.z is a C-output-operation or x = .z is a C-input-operation) and result (z) = f.

5. If all operations on f either precede e_1 or follow e_2, then currentposition (f, e_2) = currentposition (f, e_1).

Note that this statement is patterned on the corresponding assertion about assignment, and is equally valid in a parallel as in a strictly sequential environment.

5.3 Record handling.

The example of dynamic generation of arrays illustrates a technique which can be used to deal with other forms of dynamic storage allocation, for example, record handling as described in (8) A record class declaration may be regarded as a declaration consisting of a sequence of variables representing the fields. This declaration acts as the original from which all actual records of the class are copied; so that each field is a copy of the corresponding field in the record class declaration. Fields of a particular record may be referred to from within the program by means of field designators. A field designator consists of an expression delivering a record of the appropriate class, and a field identifier, denoting the original of the required field within the record class declaration. The action of a field designator is explained by means of a field-selection function selectfield (f), (where f is a field), which yields a selector capable of accessing that field in any record.

1. If x = .y is a record-generator then x is a generator; y denotes a record class declaration; if x terminates successfully then result (x) is a copy of denotation (y)

2. If f is contained in original (r) then original (selectfield (f)(r)) = f and selectfield (f)(r) is contained in r.

3. If x = y.z is a field-designator then x is normal; result (x) = selectfield (denotation (y))(result (z)), if it exists; otherwise x does not terminate successfully.

The remaining facility required for record handling is the introduction of reference-variables, which may "point to" records of some class, or else take a null value. However, apart from null itself there is no need to introduce a separate concept of a reference value, since we may equate a reference value with the record itself (not, of course, the value of the record).

4. If v is an R-reference-variable and a = 1.r is an assignment, and result (1) = v, then either result (r) = null or original (result(r)) = R.

Note that in 4, R stands for a record class declaration, which is a phrase of the program itself. Thus we have encountered a case of a semantic category generated by a program at run time.

5.4 Parallelism.

In the preceding sections, we have carefully refrained from stating the sequence of execution of the constituents of a phrase; and we have even left open the possibility that they are executed in parallel. The choice of sequencing rules can be expressed by means of axioms. We need to distinguish at least four possibilities.

1. A phrase has a regidly determined sequence of execution. Such phrases are known as <u>strictly</u> sequential.

2. A phrase has no determined sequence of execution, but its constituents must be executed in <u>some</u> sequence. This seems to correspond to the situation with the primaries of an expression in ALGOL 60. Such a phrase may be known as <u>weakly</u> sequential.

3. The execution of the constituents of the phrase is <u>interleaved,</u> in the sense that the events involved in the execution of one constituent may appear mingled amoung the events associated with the execution of another; but all events are linearly ordered. This interleaving is guaranteed when a single machine attempts to simulate parallelism.

4. There is no necessary ordering relationship amoung the events involved in the execution of one constituent and those of another. This is a genuine parallelism, such as might be achieved by multi-processor implementations of the language.

These four possibilities may be more rigorously described:

1. If x.y is strictly sequential, then termin (x) S init (y)
2. If x.y is weakly sequential then either termin (x) S init (y) or termin (y) S init (x)
3. An event e is involved in execution of phrase P = df there is an x such that e = init (x) or e = termin (x) and P contains a call-ancestor of x (using "call" as the name of the relationship of a being a call of b.)
4. If x.y is interleaved then for any event e involved in the execution of x and any event f involved in the execution of y, either e follows f or e precedes f.
5. If x.y is parallel then it is not interleaved, i.e., there is at least one event in the execution of x and one in that of y which have no defined sequence.

A language which specifies parallel or interleaved execution, should also contain some means for the synchronisation of the parallel streams. Several techniques have been proposed: one of the best defined is the "semaphore" concept introduced by E.W. Dijkstra. A semaphore acts as if it were a finite collection of some item which may be "borrowed" or "returned" by the parallel processes. If a process attempts to borrow an item when the collection is empty, it is held up until some other process returns an item. Otherwise, the effect of borrowing and returning is merely to decrement or increment the count of items available.

We thus introduce a new category of variable, the semaphore, and two operations which may be performed upon it.

6. If x = .y is a semaphore-operation then x is normal
   y is a semaphore-expression
   x is either a borrow or a return, if x is successfully terminated, then x is said to be an operation on result (y)

7. If x and y are operations on the same semaphore, then x either precedes or follows y.

8. If x is an operation on semaphore s, then the set of all operations on s that precede x does not contain more borrows than returns.

This gives a very implicit definition of the essential nature of a semaphore, and it does not contain any hint on how it is to be implemented on either a single processor or a multiple processor system. However, it does seem to be powerful enough for a programmer to prove that his programs have the desired properties, and that they will therefore work on any implementation which satifies the axioms.

6. Syntax

In this paper, no attempt has been made to deal with the notations and syntax of a programming language. It is assumed that some method will be available for specifying the correspondence between program texts and their abstract representation. An example of such a method is given by Landin. (7)

7. Conclusion.

This paper has demonstrated and explained certain axioms of the sort which are likely to feature in the axiomatic definition of many general-purpose procedure-oriented programming languages. It is not clear at the present stage how far these axioms satisfy the design criteria laid down in the introduction. Such an appraisal can be made only after a systematic application of the method to several languages, and the attempt to apply the axioms in proofs of the correctness of programs. Such proofs would be likely to be excessively long, until a fairly powerful set of metatheorems are developed from the axioms.

Acknowledgements.

This paper is published by kind permission of the Director of the National Computing Centre.

References.

1.	Criteria to be applied in the standardisation of a programming language	-	ISO/TC97/SC5.
2.	J.G. Laski	-	Algol Bulletin.
3.	Peano	-	Sul cocsetti di numero
4.	A. van Wijngaarden	-	BIT
5.	J. McCarthy.	-	Formal language Definition Languages.
6.	A. van Wijngaarden	-	MR 93
7.	P.J. Landin.	-	A Correspondence between ALGCL 60 and Church's Lambda-notation. Parts I and II. Comm. A.C.M. February/March 1965.
8.	N. Wirth and C.A.R. Hoare.	-	A Contribution to the Development of ALGOL. Comm. A.C.M. June, 1966.