STATE VECTORS: A PRELIMINARY SURVEY

by

W. D. Maurer

Memorandum No. ERL-M347

8 August 1972

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

STATE VECTORS: A PRELIMINARY SURVEY

W. D. Maurer

CONTENTS

STATE VECTORS: A PRELIMINARY SURVEY

W. D. Maurer

Introduction

We present here a critical survey of work done over the past fifteen years on mathematical models of computational processes. Central to much of this work is the concept of the state vector, analogous to the instantaneous description of a Turing machine, and the state vector function whose values are integers, truth values, labels, elements of a set of variables, or further state vectors. Any program, statement, expression, term, factor, or condition in any programming language may be represented by a state vector function, which may be said to constitute its meaning.

The study of computational processes has been marred by differences in terminology; state vectors themselves are known by at least six different names. One of the tools in our presentation is the use of glossaries of programming terminology expressed in tabular form. By this means, similarities and differences among various programming concepts may be observed. It is our hope that this survey will provide the research worker with a means of giving proper credit to the initial users of many of these concepts.

Many of the important results cited here may be stated without any reference to state vectors. These include Floyd's methods of proving the correctness of a program, Knuth's discussion of semantic attributes in programming languages, and Manna's work on termination of algorithms. However, in order to apply these results, we must first make a specific model of the program to which the application is made. We shall see that this may be done using state vectors or their generalizations in all cases of interest.

By unifying, in this way, the work of McCarthy, Floyd, Knuth, and Manna, we are able to provide a single theory encompassing the subjects of syntax, semantics, and program verification for arbitrary programming languages and computers. A program is verified by using Floyd's methods and their extensions to prove partial correctness, and Manna's methods and their extensions to prove termination. In order to prove the statements demanded by these methods, we construct the derivation tree of each command in the language, and associate semantic attributes, as introduced by Knuth, with each nonterminal in this tree. These semantic attributes are functions of the current state vector, as studied by McCarthy and others.

The existence of a unified theory by no means implies that the subject of computation science is closed. For most large programs, it is very difficult to apply directly the methods of Floyd and Manna, even in the presently known extended versions. The possibility of errors in proofs of correctness of programs, which are usually even harder to find than errors in the programs themselves, strongly suggests that we need many more computer aids to verification than we have at the present time. The extreme difficulty of proving the correctness of even simple programs involving floating-point numbers, and the fact that many well-known algorithms of this kind are not correct in the mathematical sense, indicates the need for mathematical theory to influence future hardware and software developments with respect to real arithmetic. The proof of correctness of programs admitting data structure is theoretically possible, but practically almost unexplored. In particular, most compilers remain unproved, thus preventing us from being certain about the behavior of even a verified program written in an algebraic language.

Similar questions remain open with respect to language definition. The syntax of FORTRAN is no more cumbersome, when described in BNF, than is the semantics of almost all existing programming languages when described in terms of state vector functions. Considerations of program correctness should be expected to have a far-reaching effect on future language design. Many features of existing languages, having to do with recursion, parameter calling methods, order of evaluation, and so on, are defined informally and then interpreted differently in different implementations. Any formal definition of such a feature implies a choice of interpretation with which not everyone might agree.

What the unified theory does give us is the assurance that work in the area of computational models may ultimately be applied to the solution of real computing problems. With a little ingenuity, state vector models may be built even in the presence of input-output, dynamic types, multiple use of names, or self-modification. In fact, any existing programming language may be described in state vector terms, provided that it may be interpreted in a semantically unambiguous way. Similarly, program verification techniques, although they were first developed for greatly simplified languages, may be applied to any language containing commands, and even to languages in which commands are created dynamically.

The plan of this paper is as follows. In Part 1 we consider models of state vectors and computation sequences, entirely apart from the programs which give rise to them. In Part 2 we treat the association of state vector functions with various component parts of a program. In Part 3 we consider mathematical models for programs themselves. In Part 4 we show how to give relations between the syntax of an arbitrary programming language and its semantics in state vector terms, and compare this approach to semantics with others. Finally, in Part 5, we apply this notion of state vector semantics to the problems of proving correctness, termination, equivalence, and other assertions about programs.

The author is greatly indebted to Ralph London for his exhaustive bibliography on proving the correctness of programs [London 70a], without which this survey would never have been completed.

1 STATE VECTOR FUNDAMENTALS

We shall start by reviewing various published definitions of what are essentially state vectors. These definitions are not all quite the same, and they differ from each other not only terminologically but in other ways, such as the restrictions placed on the number of variables or the number of values. Provided that the restrictions are properly chosen, we may construct state vectors for sequential machines, Turing machines, and the like.

1.1 Definitions of State Vectors

The first appearance of state vectors for an abstract machine with infinitely many random-access registers is in [Kaphengst 59]; the term Maschinenstellungen is used here. However, the first appearance of the state vector concept in connection with a general theory of programs and computers is in [McCarthy 63], and it is from this paper that we take the term "state vector."

McCarthy's original definition of state vectors is displayed in Table 1, along with a number of others which have appeared since then. [redacted] Conspicuously absent from this list are the "snapshots" of [Naur 66]; programmers may visualize state vectors most easily as "snapshot" dumps, but Naur's snapshots (as opposed to his General Snapshots) are defined too informally and cursorily for inclusion here.

State vectors as functions first appear in [Elgot and Robinson 64], where they are called content functions. A content function k assigns to each variable x its current value k(x). The set K of all content functions applicable to a given computation is the set of all functions from A, the set of all variables, to B, the set of all legal values. McCarthy, since he does not use this functional notation, introduces a function $c(v, \xi)$ which gives the current value of the variable v when the current state vector is ξ. By writing $c_\xi(v)$ for $c(v, \xi)$, we obtain the content function c_ξ corresponding to the state vector ξ. (Elgot and Robinson are concerned mainly with computers rather than programs, and thus they use the term "addresses" for variables and "words" for values.)

McCarthy 63 — "We shall define the state vector ξ of a program at a given time, to be the set of current assignments of values to the variables of a program. In the case of a machine language program, the state vector is the set of current contents of those registers whose contents change during the course of execution of the program."

Elgot and Robinson 64 — "We assign symbols a_0, a_1, a_2, a_3, ... to the individual storage locations...We denote by b_0, b_1, b_2, b_3, ... the units of information that can be stored in the a_j....Let $A = \{a_0, a_1, a_2, \ldots\}$, $B = \{b_0, b_1, b_2, \ldots\}$; then the configuration in the memory at a specified time can be represented by a function $k(x)$ from A into B."

Strachey 66 — "The 'content of the store'....is a primitive concept and we shall always denote it by σ....associated with each command γ in the program there is a $(\sigma \rightarrow \sigma)$ operator θ such that obeying the command γ in a state where the store content is σ produces a new state $\sigma' = \theta\sigma$."

Maurer 66 — "All computers have a memory, which is a finite collection of 'elements'....We may now drop the quotes and speak of the memory of a computer as a finite set M....A particular state (sometimes known as an 'instantaneous description') of the computer is then....a function from M into the set of all integers from 0 to n-1."

Engeler 67 — "Each operation is intended to produce....a map from A^ω to A^ω (where A^ω is the set of all sequences $\langle a_0, a_1, \ldots\rangle$ of elements of A). The following are the types of operations from which the admissible ones are selected: $x_i := c_k$, which transforms $\langle a_0, \ldots, a_i, \ldots\rangle$ into $\langle a_0, \ldots, e_k, \ldots\rangle$;...."

Luckham, Park and Paterson 70 — "σ and $P(\sigma)$ denote, respectively, a possible sequence of instructions of the schema P....and the sequence of vectors of values assigned to $\langle L_1, \ldots, L_n\rangle$ when σ is executed. $\sigma(i)$ is the i-th member of σ, $P(\sigma)(i)$ is the vector after $\sigma(i)$ is executed and $P(\sigma)(i, j)$ is its j-th component."

Scott 70 — "In the first place, what is a store? Physically, we have several remarkable answers, but mathematically it comes down to being simply an assignment (a function) which connects contents to locations. Speaking more precisely, the (current) state of the store, call it σ, is mathematically a function $\sigma: L \rightarrow V$ which assigns to each location $\ell \in L$ (the set of all locations) its (current) contents $\sigma(\ell) \in V$ (the set of all allowable values)."

The use of a separate set of legal values for each variable first appears (in a state vector context) in [Maurer 66], although this idea is also suggested, semi-formally, in [Ershov 60]. Although [Maurer 66] is also basically concerned with computers rather than programs, it is clear that different variables often do have different kinds of values in practical situations. In fact, the set of all legal values of a variable may be identified with the type of that variable, as suggested in the ALGOL report [Naur et al. 60] and elsewhere. It is also pointed out in [Maurer 66] that tape squares may be variables in an infinite set of variables. The functional notation S(x) denotes the value of x, and the state vector S is called a state.

A concept known as the "content of the store" (the informal nature of the definition of which is strongly emphasized) appears in [Strachey 66], but it is clear from the context that it is effectively a state vector. This paper is concerned with data structures and functions acting upon them which may or may not be defined at a given instant of time (because, for example, they may go down a sublist which isn't there). Consequently the content of the store is said to "contain" not only variable values but also the values of all those expressions which are legal at the given instant of time.

Explicit correspondences between ALGOL-like assignment statements and state vector functions (see also section 2.3 of this paper) first appear in [Engeler 67], although they are suggested in both [McCarthy 63] and [Strachey 66]. Engeler is the first to require that the number of variables be infinite; this is also done in [Kaplan 68], which otherwise follows McCarthy. No special name, other than "sequence of elements of A," is given here to a state vector, where A is the underlying set (the set of all values).

From 1967 on, several other definitions were made. The assignments to the registers of a computer language and to the set of bound names of a programming language as introduced in [Narasimhan 67] are effectively state vectors, though informally defined. Machines with infinitely many storage locations, including tape squares, are studied in [Wagner 68]; the values of storage locations range over a finite alphabet, and there are finitely many tape heads, each of which is effectively a variable whose values are tape squares. Composite objects, as defined for PL/I by the Vienna group [Lucas and Walk 69] can be made to look like state vectors of structured variables; they are essentially more complex than state vectors, however, and their study will be deferred to section 4.3. The condition for the correctness of a control path (see section 5.1) is given in state vector terms in [Cooper 69]; here $\{L\}$ is used for the set of values of the (finite number of) registers. The subject of verification is taken up in great detail in [King 69] and [Good 70]; they use the term "state vector" and specify a finite number of variables, each with its own set of values. In [Luckham, Park and Paterson 70], a time variable is explicitly introduced; $P(\sigma)(i)$ is the state vector at time $\underline{i}$ of the program P, and $P(\sigma)(i,j)$ is its j-th component, that is, the value which it imparts to the j-th variable. Strachey's concept of the content of the store is put into explicit functional form in [Scott 70]; finally, Wegner's information structures [Wegner 70] are informally defined but are suggestive of state vectors, presumably in a data-structure setting. A partial glossary of basic terminology used in connection with state vectors is given in Table 2.

Kaphengst 59	Maschinenstellung $\mathfrak{M}$	Fächer	Worte	$\mathfrak{M}(\phi)$ heisst der Inhalt von ϕ
McCarthy 63	State vector ξ	Variables	Values	$c(a, \xi)$ is the a-component of ξ
Elgot and Robinson 64	Content function k (or state (k, a))	Addresses	Words	$k(x)$ is the content of x
Strachey 66	Content of the store, σ	Generalised addresses	Bit patterns	$(C(\alpha))(\sigma)$ is the R-value corresponding to the L-value α
Maurer 66	State S	Memory elements	Elements of the base set	$S(x)$ is the state of x
Engeler 67	Sequence $\langle a_0, a_1, \ldots\rangle$ of elements of A	Variables	Elements of the underlying set	---------
Narasimhan 67	Assignment to a set of registers	Registers	Elements of the range of a register	---------
Cooper 69	Set $\{L\}$ of values of the registers	Registers	Values	L_i is a register whose current value is also denoted by L_i
Luckham, Park and Paterson 70	Vector of values assigned to $\langle L_1, \ldots, L_n\rangle$	Location symbols	Elements of D	$P(\sigma)(i,j)$ is the j-th component of the vector $P(\sigma)(i)$
Scott 70	State of the store, σ	Locations	Values	$\sigma(\ell)$ is the contents of ℓ

TABLE 2

In most of these papers, state vectors are represented either as vectors, or as functions, or as trees, or as elements of a cartesian product. There is a natural mathematical correspondence between these notions. Any function f is completely specified by the set of all pairs (x, f(x)), for all x in its domain. If the domain is known, the first elements of the pairs may be eliminated, and the function specified by a finite or infinite vector of its values. Any vector, in turn, is an element of the cartesian product of those sets to which its various components are constrained to belong. The a-component $f_a(x)$ of the element x of a cartesian product may equally as well be regarded as the value $f_x(a)$ of the function f_x corresponding to x when applied to a. Finally, any function f may be represented by a finite or infinite directed tree in which all links lead from the root to terminal nodes; each link corresponds to an element x of the domain of the function, which is its label, and leads from the root to a terminal node labeled f(x). This representation is advantageous when f(x) is itself a tree, and will be discussed further in section 4.3.

A computation is represented by a finite or infinite sequence of state vectors. If the sequence is infinite, the computation is endless; if it is finite, the computation terminates. There is also an associated sequence of labels (or operators or statement numbers) through which the computation passes in execution order; these may also be thought of as values of a program counter or control register. Such sequences may be required to terminate when transfer is made to an undefined label, or to a designated halt or exit, or when the current operation is undefined, or even (as suggested in [Elgot and Robinson 64] in connection with "passive words") when the current operation is a transfer to itself. A glossary of terminology relating to computation sequences is displayed in Table 3.

	Sequence of state vectors *with* prog-counter	Sequence of state vectors *without* prog-counter	Sequence of operations involved	Criterion for termination of a computation
Yanov 58	Application sequence	Evaluation sequence	Value of a schema	Reaching a terminal operator
Engeler 67	Construction	---------	Trace	Transfer to an undefined label
Scott 67	*Alternating* sequence $(L_0, m_0, L_1, m_1, \ldots, L_n, m_n)$, where the L_i are labels and the m_i are states			Transfer to a halt instruction
Elgot 68	Computation of a diagram	Track	Trace	Transfer to an element of the set of exits
Luckham, Park and Paterson 70	---------	Computation sequence	Execution sequence	Transfer to STOP
Wegner 70	Computation and trace are synonymous; program counter not specified			No function applicable to the current information structure
King 71	Execution sequence	---------	---------	Transfer to halt

TABLE 3

1.2 Variables and Values

[McCarthy 63] uses the terms "variables" and "values" for ALGOL-like programs, and "registers" and "contents" for machine language programs. It is clear, however, that a register may be regarded as a variable, whose contents are its value. Also, on most real computers, "register" refers only to the internal registers and not to the memory, core or otherwise; and indeed those state vector models which are concerned principally with computers rather than programs are divided between the use of the terms "register" and "storage location" (or "address"). We shall continue, in this paper, to refer to variables and values in all cases. A variable, then, is anything to which a state vector assigns a value directly.

What sorts of things can be variables? Clearly, simple variables in algebraic languages, and registers and memory cells in machine languages, are variables. What about subscripted variables, however? If we have an array of n numbers, should this be a variable whose values are sequences of n numbers, or should it be n variables, each of whose values are numbers? Clearly either point of view is possible; we should be able to go back and forth between one viewpoint and the other, as the occasion arises. In the same way, we can regard an n-bit word in a computer as a variable whose values are patterns of n bits, or we can regard it as n variables, each of whose values are 0 or 1.

A general construction which is applicable to this situation is given in [Maurer 66]. If M is the set of all variables, we may let $\mathfrak{D}$ be a decomposition of M, that is, a partition into disjoint sets. If the sets of $\mathfrak{D}$ are now regarded as variables, each of whose set of values is itself a (restricted) state vector, the state vectors over M are effectively the same as the state vectors over $\mathfrak{D}$. This idea is further generalized in the definition of composite objects [Lucas and Walk 69] as trees; each level of such a tree corresponds to a level of decomposition. Thus we may treat arrays whose elements are structures, whose elements are arrays, et cetera, as is allowable in PL/I.

One quantity which certainly varies during the course of a computation is the program counter, or current statement number, as discussed in the preceding section. [McCarthy 66] calls this a "pseudovariable"; [Elgot and Robinson 64] distinguish between *content functions* k and *states* (k, a), where the address a is clearly meant to represent the current address. [King 71], however, treats the program counter as an actual variable, and his state vectors all have program counter components. It will become clear in part 2 of this paper that this should not invariably be the case; we need state vectors without program counter components as the arguments of certain state vector functions. However, there is a certain advantage in not having to treat the program counter as a special case.

Can the squares of a tape, the cards in a deck, the lines on an output sheet, and so on, be regarded as variables? This question is directly related to two more fundamental questions. The first is whether the next state in a computation should be dependent *only* on the current state, and on nothing else. This is certainly not true in automata theory, where tapes are treated on a different basis from internal states, but it becomes true if we substitute "instantaneous description" for "state." Thus either point of view may be taken, depending on whether a state vector is considered as analogous to a state (of a Turing machine, say) or to an instantaneous description. The more restrictive point of view appears in [Park 68]: "if the command is an 'output' command, it has an effect not reflected in the state as envisaged here. On the other hand, the effect of an 'input' command is determined by some 'environment' which again is not specified in the state." In an operator algorithm [Ershov 6a] or in a RASP [Elgot and Robinson 64], on the other hand, the current state clearly depends on the previous state, and on nothing else. The restricted viewpoint may clearly always be expanded to the general one; any quantity which affects the next state may be treated as a variable in an expanded model.

The second question is whether the total number of variables is allowed to be infinite. Most authors who refer to variables as "registers" specify that their total number is to be finite. If a tape is regarded as a single variable whose values are the possible contents of the tape, this causes no difficulty, provided that a variable is allowed to have an infinite number of values. However, if each tape square is a variable, such a model effectively restricts the total length of a tape to be finite. Some authors require the number of variables to be infinite, which is a bit unnatural when dealing with subroutines which perform no input-output operations. On the other hand, [McCarthy 63] and many others leave the total number of variables unspecified; [Elgot and Robinson 64] specifically state that it may even be uncountable; and [Scott 70] actually studies the topological properties of certain uncountable data types.

Whenever the number of variables is infinite, a state vector is apparently not a finite object. This logical difficulty may be overcome, however, if we specify that each operation on state vectors must change only a finite number of variables (this is part of what [Elgot and Robinson 64] call being "finitely determined"). In this case, it will always be true at any stage of a finite or infinite computation that only a finite number of variables have values different from their initial ones, and state vectors may then be formulated to give values only to these variables. This may require, of course, that different sets of variables be used at different stages of a computation; this question will be taken up again in section 2.1.

We pass now from variables to values. The computer languages of [Narasimhan 67] and the computing machines of [Orgass and Fitch 69a] specify that the total number of values of a variable must be finite. Under these conditions, no machine or program involving infinite tapes may be modeled. In all the other work which we have been considering, however, there is no restriction of this kind. A point of greater controversy is whether different variables should be allowed to have different sets of values. It is quite common for this to be forbidden, and, in consequence, for different kinds of variable quantities to be treated on different bases. Thus in [Wagner 68] the tape heads, which have infinitely many possible positions, are distinguished from the storage locations, whose values range over a finite alphabet. A much more common distinction of this kind is between the program counter and the other variables; this may or may not be desirable, but if all variables must have the same set of values, it is apparently required, since the values of the program counter are statements or their labels, while the values of other variables presumably are not.

If each variable has its own set of values, it is always possible to take the union of all these sets and form a universal set of values. A distinction must then be made between "legal" state vectors and others; explicit allowance for this possibility is made in [Elgot and Robinson 64], [Maurer 66], and [Wagner 68]. State vector functions, in turn, must be partial functions (which they usually are anyway). Whenever an explicit range is specified for content functions, this range effectively becomes the set of all legal values of any variable. On the other hand, from the cartesian product viewpoint, it appears more natural to work with the cartesian product of arbitrary sets (each of which is the set of values of some variable) than specifically with the cartesian product of several copies of the same set. This is reinforced by the fact that most programming languages have type declarations, whose purpose, mathematically, is to specify, for certain variables, the set of all values which they may assume. Similarly, most computers have registers of various sizes, together with input-output devices assuming variables from a character set.

A summary of the properties given to state vectors by various authors appears in Table 4. In this connection we should mention one peculiar property not mentioned in this table: the reluctance on the part of some authors to use the concept of a set. McCarthy, Strachey, and Engeler carefully avoid saying that there is a set of all variables with which a given state vector associates values; Burstall speaks of "sorts," rather than sets. If mathematical logic is taken to underlie set theory [Suppes 57] rather than the reverse [Ehlers 68], it might seem that mathematical logic should directly underlie computation science, without set theory in between. Other workers in mathematical logic, however, use sets of variables and sets of values without hesitation ([Elgot and Robinson 64], [Scott 70]).

	Formal or informal definition?	Total number of variables	Total number of values	Separate values for each variable?
McCarthy 63	Informal	Not specified	Not specified	Not specified
Elgot & Robinson 64	Formal	Unrestricted	Unrestricted	No
Strachey 66	Informal	Not specified	Not specified	Not specified
Maurer 66	Formal	Unrestricted	Unrestricted	Yes
Naur 66	Informal	Not specified	Not specified	Not specified
Engeler 67	Formal	Must be (countably) infinite	Unrestricted	No
Narasimhan 67	Informal	Must be finite*	Must be finite*	Yes
Kaplan 68	Formal	Must be (countably) infinite	Not specified	No
Wagner 68	Formal	Must be (countably) infinite	Must be finite	No
Cooper 69	Formal	Must be finite	Not specified	No
Orgass and Fitch 69a	Formal	Must be finite	Must be finite	No
Luckham, Park and Paterson 70	Formal	Must be finite**	Unrestricted	No
Scott 70	Formal	Not specified	Not specified	No
Good 70	Formal	Must be finite	Unrestricted	Yes
King 71	Formal	Must be finite	Not specified	Yes

TABLE 4

* This is for "computer languages." For "programming languages," the total number of variables must be countably infinite, while the total number of values is not specified.

** According to the definition on page 224 of this paper. An earlier definition (p. 222) seemingly removes this restriction.

1.3 Structured Sets of Values

The set of values of a variable is normally provided with structure of various kinds. The most commonly encountered sets of values are the real numbers and the integers, and their counterparts, the one-word floating point numbers and the one-word integers. These are provided with binary operations of addition, subtraction, multiplication, division, and exponentiation, the unary minus, the ordering relation $<$, and various other functions such as sines and cosines. The set $\{$true, false$\}$, which is the set of values of logical or Boolean variables, is provided with the unary operation NOT and the binary operations AND and OR, and, in ALGOL, with $\rightarrow$ and $\equiv$ as well.

From the mathematical point of view, we may study "ideal" algorithms in which any real number, or any integer, can be the value of a variable, or we may study "actual" algorithms in which the values of variables are constrained in the usual way by the limitations of finite storage space in real computers. The only difference between the mathematical models in the two cases is in the sets of values and the properties of operations on these sets. In particular, addition and multiplication of real numbers are associative, whereas addition and multiplication of floating-point numbers are not, even in the absence of overflow and underflow. This must be taken into account when proving the correctness (see section 5.1) of an algorithm involving floating-point numbers. For integers, the difference between ideal and actual algorithms is purely a matter of overflow and zero division; if an ideal algorithm involving only integer variables is correct (so that, in particular, it involves no zero division), then the corresponding actual algorithm is always correct whenever no overflow occurs.

Other kinds of structure have been proposed for data types, or sets of values. In [Scott 70], a complete lattice structure and a topological structure are proposed for data types. Each data type is assumed to have a countable, computable dense subset, and thus its topology is separable. Mappings between data types are assumed to be continuous and monotonic. All of these conditions are automatically satisfied by finite data types. If D and D' are data types, then the set of all functions from D to D' is also a data type; such types are further studied in [Milner 72].

A number of programming languages constructed in the last ten years have used data types of considerable complexity. We have already mentioned the structures of PL/I, which were originally derived from COBOL structures. Similar features are found in ALGOL 68 [van Wijngaarden et al. 69], PASCAL [Wirth 71], and elsewhere. A different kind of data type is encountered in LISP, SNOBOL, EULER, and APL, where the type of a variable may change dynamically. Thus the same variable, within the same job, may be an integer and a list in LISP, a string and a pattern in SNOBOL, an integer and a real number in EULER, or a two-dimensional and a three-dimensional array in APL. In such cases the type of a variable is itself a variable, and has its own component within the current state vector. Alternatively, the value of any variable may be given in the current state vector as a pair (t, v), where v is the actual value and t is the type.

All of these structural considerations may be thought of as slight generalizations of the concept of a state vector. That is, instead of a simple function from one set to another, we have a function whose domain (the set of values) is provided with some further structure. Another kind of generalization of state vectors is suggested by the problems of storage allocation. A programming language with variable array dimensioning may be defined with respect to an infinite sequence x_1, x_2, ..., of cells and a storage allocation mechanism which assigns to each variable, including each element of each array, one of these cells. This assignment changes as the program progresses, and we may therefore speak of the current storage allocation, as well as the current state vector. Thus [Kaplan 68] defines the program state vector of a program π to be the ordered pair (M_π, ξ), where M_π maps program variables into the positive integers and ξ maps the positive integers into values. In a similar way, [Park 68] defines the state of a computation to be a pair $<\mathcal{L}, C>$, where $\mathcal{L}$ maps the currently legal expressions (among which are the variables) into some set of locations, while C maps each location into its current value. In Park's model, the set of currently legal expressions also changes as the program progresses, because an expression may refer to sublists of a list, and is legal only if the sublists are currently present.

1.4 State Vectors of Specific Machines

Any real computer, and any real programming language, has state vectors. In addition, automata of various kinds may be regarded as machines with state vectors. Automata, in the present context, are specific machines with specific rules, rather than being general models for all computers. It is true that we could choose, as the set of states of a sequential or Turing machine, the set of all state vectors (without input-output components) of some digital computer; then, provided that we were willing to live with a restriction to a single input and output tape (for a sequential machine) or a single scratch tape (for a Turing machine), we could model a computer as a sequential machine or a Turing machine. However, as we shall see in part 5 of this paper, the proof of assertions about programs and computers generally requires that computers, and programs, be modelled in other ways.

As an example, let us make a state vector model of a Turing machine [Davis 58]. In doing this, there are two arbitrary choices to make. The first is whether to model the tape as a single variable, whose values are sequences of symbols, or to model each separate square as a variable whose values are symbols. The second way is clearly more natural, but, as we have seen, it is incompatible with the model restrictions of [Cooper 69], [Good 70], and [King 71]. (No method of modelling a Turing machine is compatible with the restrictions of [Orgass and Fitch 69a].)

The other choice is whether to model a moving tape and a fixed head or a moving head and a fixed tape. In the first case, the i-th square of the tape is defined to be that square which is currently i squares to the right of the head; the (-i)-th square is i squares to the left of the head. Each "move left" or "move right" operation sets the value of each such variable to the current value of the variable on its right or left respectively. In the second case, no square changes its value unless it is directly under the head, but the head is also a variable whose values are positions on the tape. (This variable is distinct from the variable whose values are the states.) Although an extra variable is involved, the moving-head model is much easier to manage than the fixed-head model when each square is a variable, since only three variables can possibly change (in the most general form of Turing machine) at each stage. Writing Q for the state variable, P for the variable representing the head, and T_i $(-\infty < i < \infty)$ for the tape squares, passage from the current state vector S to the next state vector S' is given, in this model, by

$$S'(T_i) = S(T_i) \qquad \text{for all } i \neq S(P)$$

$$S'(P) = \begin{cases} S(P) + 1 & \text{for "move right"} \\ S(P) & \text{for "no motion"} \\ S(P) - 1 & \text{for "move left"} \end{cases}$$

$$S'(Q) = f(S(Q), S(T_{S(P)}))$$

for "let the next state be f(x, y), where x is the current state and y is the contents of the current tape square"

$$S'(T_{S(P)}) = g(S(Q), S(T_{S(P)}))$$

for "print g(x, y), where x and y are as above"

Those who are familiar with simple Turing machine programming will be well aware that, as [Minsky 61] points out, the state variable, which we have called Q here, normally plays the rôle of a program counter. This may cause some confusion concerning the term "state vector" for a set of current "states" of variables; it should be clear from the preceding discussion that a state vector is not, as this analogy might suggest, a vector of program counters, but rather a vector of variable values in the ordinary sense.

Other specific abstract machines than Turing machines may also be modeled using state vectors. For the complete sequential machines of [Ginsburg 62] (which are adaptations of the finite automata of Moore and Mealy), using an input variable, state variable, output variable, and fixed read-write heads, if the machine is $(K, \Sigma, \Delta, \delta, \lambda)$ and the current state vector is $\{$<u>input</u> $= (i_1, i_2, \ldots)$, <u>state</u> $= k$, <u>output</u> $= (o_1, o_2, \ldots)\}$, where $i_j \in \Sigma$ and $o_j \in \Delta$ for each j, then the next state vector is $\{$<u>input</u> $= (i_2, i_3, \ldots)$, <u>state</u> $= \delta(k, i_1)$, <u>output</u> $= (\lambda(k, i_1), o_1, o_2, \ldots)\}$. The strings upon which a Markov algorithm [Curry 63] operates are effectively its state vectors; likewise, pushdown automata and other special-purpose automata have their state vectors, as noted in [Engeler 67] and also informally in [Scott 67].

Besides automata, a number of other computational models have been constructed. We have already mentioned the <u>Maschinenstellungen</u> of [Kaphengst 59]; Kaphengst's ΠPM, or program-controlled computing machine, is similar to the Q-machine of [Melzak 61], the "infinite abacus" of [Lambek 61], and the URMs, LRMs, and SRMs of [Shepherdson and Sturgis 63], although only Kaphengst gives an explicit description of the state vectors involved. Two other computational models are developed almost, but not quite, to the point of introducing state vectors; these are the program schemata of [Yanov 58] and the operator algorithms of [Ershov 60].

Yanov, whose work is well summarized in [Rutledge 64], introduces <u>evaluations</u>, which are almost, but not quite, state vectors. Each evaluation contains the current values, not of the variables in the computation, but of certain <u>expressions</u> (specifically, Boolean expressions or predicates). There are presumed to be only a finite number of these, so that simply choosing for our predicates all expressions of the form {var=val} will not fit Yanov's model when there are an infinite number of either variables or legal values. Experience with modelling programs in this way suggests that it is not always clear, even for a completely deterministic algorithm, what the values of all predicates will be after a given command, presuming that all their values are known before that command. Yanov therefore allows an operator (= command) to specify, for each incoming evaluation, a <u>set</u> of possible outgoing evaluations. This feature of Yanov's model allows it to handle the nondeterministic case directly, without alteration, whereas in the state vector model a state vector and a command specify the next state vector uniquely if and only if the command is deterministic (as noted in [Burstall 72]). Yanov also introduces the "argument domain of an interpretation" which acts somewhat like a set of all state vectors, although the state vector structure is not spelled out.

The operator algorithms of Ershov are modeled after digital computers, rather than programming languages; the value of a variable may be an operator, as happens, for example, with the instruction words of a computer. His definition is informal, and does not involve state vectors explicitly; he represents variables by non-italicized letters, and the current value of any variable is represented by the corresponding italicized letter. Ershov does point out, however, that different variables should be expected to have different sets of values.

Mention should also be made of the SECD machine [Landin 64] and the sharing machine [Landin 65]. These machines have a construction (the <u>environment</u>) which is much like a state vector, but the rôle it plays in the model is distinctly secondary. The philosophy here is that recursive functions of conditional expressions should be used in computing whenever possible, and manipulation of variables in the ALGOL sense resorted to only when necessary.

2 PROGRAM COMPONENTS AND STATE VECTOR FUNCTIONS

Any computer or programming language is representable either as a context-free language in standard BNF or as a subset of such a language as determined by certain non-BNF restrictions (such as that every variable used must be declared). It is with the nonterminals in such a representation that we now concern ourselves. With certain of these nonterminals (or "metalinguistic variables" [Naur et al. 60]) we may associate functions of the current state vector whose values may be state vectors, real numbers, integers, truth values, members of a set of variables, statements in a program, etc. These functions are generally partial; however, in proving the correctness of a program, conditions are imposed with respect to which the functions are total (see also section 5.1). Such associations have been made by various authors; we now discuss the most common ones.

2.1 General Commands

Passage from one stage of a computation to the next is made by <u>instructions</u> or <u>commands</u> in computer languages and by <u>operations</u> or <u>statements</u> in programming languages. Each of these may have a <u>statement number</u> or a <u>statement name</u> or a <u>label</u> or something similar. Table 5 gives the terminology used by several authors with respect to these concepts.

A command may be associated with a function which takes state vectors into state vectors. This is the easiest of our associations to make, and it has been remarked upon by a great number of authors, as displayed in Table 6. (See also the similar comments by Strachey and by Engeler from Table 1.) If S is the current state vector and we execute the command C, then $f_C(S)$ is the next state vector in the computation. The function f_C corresponding to C has various names; we shall call it the <u>effect</u> of C, following [Park 68] and others. This name includes as a special case the term "side effect" of earlier vintage; side effects are discussed in the next section. In the nondeterministic case, the effect is multi-valued; this is studied in [Burstall 72]. [redacted]

What happens when an instruction or command is modified by a program and then later executed in modified form? The answer, as noted semi-formally in [Ershov 60] and formally in [Elgot and Robinson 64], is that the instruction itself becomes a value of a variable. Thus <u>we may have variables whose values are state vector functions</u>. Or, to simplify things a bit, we can have a distinguished set of "operators," a map from operators into state vector functions, and variables whose values are operators. With such a variable we may always associate a single state vector function which is ap-

Elgot and Robinson 64	instruction	instruction location
McCarthy 66	statement	statement number
Strachey 66	command	label
Engeler 67	operation	numeral
Elgot 68	command	occurrence
Good 70	node operation	node
Luckham, Park and Paterson 70	instruction	prefix
Burstall 70	statement	point
King 71	statement	statement name
Igarashi 71	statement	label symbol

TABLE 5

McCarthy 63	"When a block of program having a single entrance and exit is executed, the effect of this execution on the state vector may be described by a function $\xi' = r(\xi)$ giving the new state vector in terms of the old."
Elgot and Robinson 64	"For any $b \in B$ then, h_b defines a transition from one state (element of Σ_0) into another state.... h_b will be called an (atomic) <u>instruction</u>....We also speak of an arbitrary mapping of Σ_0 into Σ_0 as an <u>instruction</u>."
Maurer 66	"An instruction then, is a method of passing from one state to another state, i. e., a map I: $\mathcal{S} \rightarrow \mathcal{S}$, where $\mathcal{S}$ is the set of all states under consideration."
Park 68	"It is necessary to talk about the <u>effect</u> of a command. The device adopted here is to refer to an instantaneous <u>state</u> of a computation. The state $\langle \mathcal{L}, C \rangle$ consists of two mappings $\mathcal{L}$ and C ....The effect of the given command produces the new state $\langle \mathcal{L}_1, C_1 \rangle$."
Cooper 69	"In a program scheme we have a set of registers $L_1, L_2, \ldots, L_n$....let $\psi_{ij}(\{L\})$ be the final set of values of the registers, $\{L\}$ being the set of values at the start...."
Orgass and Fitch 69a	"As a result of our definition of the state of a computing machine, we are able to assume that an instruction is a partial recursive function from γ-tuples of natural numbers into γ-tuples of natural numbers."
Scott 70	"The (current) state of the store, call it σ, is mathematically a function $\sigma: L \rightarrow V$....Let Σ be the set of all states....a command is a function $\gamma: \Sigma \rightarrow \Sigma$ which transforms (old) states to (new) states."

(Further examples of effects include the function <u>next</u>(s) of [Burstall 70], which produces the new state (= state vector) s' in terms of the current state s, and the function J[A] of [Igarashi 71], which associates with each statement A a map from $|\mathfrak{B}|$ into $|\mathfrak{B}|$, where $|\mathfrak{B}|$ is effectively the set of all state vectors under consideration.)

TABLE 6

plied whenever that variable operator is executed. The value of this function applied to the state vector S is $f_i(S)$ whenever $S(v) = f_i$, where v is the variable operator.

In associating effects with commands, problems arise with domains and ranges. If the command C involves a subscripted variable, the associated function f_C is inapplicable to any state vector which specifies the given subscript as being out of range. If C involves the square root of X, then $f_C(S)$ is not defined if S(X) is negative (unless X is complex). In general, the precise domain of f_C may be difficult to determine. In practice, however, it usually suffices to specify $f_C : \mathcal{S} \rightarrow \mathcal{S}$ as a partial function, that is, to determine some set $\mathcal{S}$ of state vectors which <u>contains</u> the domain of f_C. In proving the correctness of programs, we also determine a subset of $\mathcal{S}$ which is contained in the domain of such a function, as will become clear in section 5.1; the restriction of f_C to this smaller subset is then a total function.

The set $\mathcal{S}$ generally consists of all state vectors of a certain form, and we must also be concerned with <u>their</u> domains and ranges. Each such state vector must obviously have a component for every variable appearing in the command. ([Igarashi 71] uses V[A] for the set of all variables occurring in the command A.) But what about a variable that does not so appear? Such a variable may always be included by specifying that it does not change; that is, if v is the variable and f_C the function, then $S(v) = S'(v)$ where $S' = f_C(S)$. The problem arises when we ask how many variables should be included in this way. At one extreme, we could specify that every conceivable legal variable name ought to be included; this solves the problem of specifying a unique domain and range for f_C, but in a rather unnatural way. At the other extreme, we could require that there be no such extraneous variables; but this is a bit awkward when we seek to find the effect of a succession of commands.

The most reasonable middle ground seems to be to determine all variables occurring in the program which contains the given command. (In some languages, such as SNOBOL, in which variable names may be constructed dynamically, this may not be possible, and we may have to go back to the first alternative.) The effect of a command then involves state vectors having a component for each program variable. This also takes care of the case in which function calls in the command may cause changes in variables which do not appear directly in it.

A related problem concerns whether to allow a command to specify the next command. If the state vectors in the range of the effect of the command have program-counter components, this can be done directly, as in [King 71]. Otherwise, we may introduce a new state vector function (such as the direct transition function of [Elgot 68]) which gives, for each current state vector, the next statement or its label. This subject is taken up further in section 2.4. If a command does not specify the next command, the only allowable commands are assignments, function calls, and the like; these may be combined to form *straight-line programs*, which have been studied by a number of authors.

The state vectors in the *domain* of the effect of a command do not, normally, need program-counter components, since a command does the same thing no matter where it appears in a program. This suggests that instead of writing $f_C: \mathcal{S} \to \mathcal{S}$ we might write $f_C: \mathcal{S} \to \mathcal{T}$, where the state vectors in $\mathcal{T}$ have program-counter components and those in $\mathcal{S}$ do not. This is a special case of the use of *different* sets of state vectors for the domain and range of the state vector function of a command. General treatments of this kind include that of [Ershov 64] involving the "operating region" of an algorithm, which may change as the algorithm progresses, allowing new state vector components to appear, and the "directions" of [Elgot 68] (see also section 3.4). [Scott 67] and [Manna 69] are concerned with different domains (effectively sets of state vectors) for input, program, and output variables; while [Park 68] allows the set of legal expressions to change when a command affecting data structure is executed.

2.2 Expressions

A unified theory of state vector functions starts with the observation that they may be associated with other program components than instructions or commands. Of these, the simplest is the arithmetic (or Boolean) expression. Given a state vector and an expression, with or without side effects, we can find values for all variables appearing in the expression and then evaluate the expression using those values, producing an arithmetic or Boolean value as output. This was apparently first noticed in [McCarthy 63], where the notation value(t, ξ) is used; here t is a term (= expression) and ξ is a state vector. If we write $f_t(\xi)$ for value(t, ξ), we obtain the state vector function f_t corresponding to the term t.

Expressions are built up from terms and factors, and similarly their state vector functions are built up from the state vector functions of the terms and factors. McCarthy gives a four-way conditional expression defining value(t, ξ) when t is a variable, a constant, a sum, or a product. Similarly, Burstall defines plus(e, e') as the sum of the two expressions e and e'; he then gives the equation val(plus(e, e'), s) = val(e, s) + val(e', s), where val(e, s) is the value of e given the state vector s [Burstall 70]. Strachey writes R(ϵ, σ) for the value of the expression ϵ given σ as the content of the store, and makes the further point that if ϵ appears on the left-hand side of the assignment symbol (which might happen with a subscripted variable, for example) then a different function, which he calls L(ϵ, σ), should be used; the values of this function are addresses.

It is interesting to note that these points have been missed by a number of other authors, who treat all expressions in functional form. Thus Cooper treats a fixed collection of functions whose arguments are register values [Cooper 69]; also in [King 71] the arguments of the various functions are variable values. An extension of this system, found in [Igarashi 71], can be made to give the full generality of arithmetic expressions; here, however, the arguments of the various functions are not restricted to values of variables, but may, recursively, be values of other expressions as well.

Expressions with side effects have given several authors more trouble than was really necessary, and some of them seem to have been dimly aware of this. Thus we find [Floyd 67]: "Consider the statement a:=c + (c:=c + 1) + c, which has the effect of assigning $3c_0 + 2$ to a, where c_0 is the initial value of c, and assigning $c_0 + 1$ to c....Let us reluctantly postulate a processor, with a pushdown accumulator stack S...." Later, the handling of side effects is attempted in [Burstall 70] by attaching "points" (effectively labels) to the <u>middle</u> of an expression with side effects, thus separating out its evaluation over several "statements." A key to the true solution of this problem is found in [Strachey 66], where $R'(\epsilon, \sigma)$, for the expression (with side effects) ϵ and the content of the store σ, gives as value a <u>pair</u> (β, σ'), where β is the value as before and σ' is the new state vector. Strachey, however, does not give combination rules for the state vector functions of expressions. By combining Strachey's work with Burstall's, one obtains an analogue of Burstall's value formula given above:

sideeffect(plus(e, e'), s) = sideeffect(e', sideeffect(e, s))
val(plus(e, e'), s) = val(e, s) + val(e', sideeffect(e, s))

which reduces to the original formula if sideeffect(e, s) = s.

2.3 Assignments

An assignment without side effects is a special case of a command. The assignment of the constant $\underline{k}$ to the variable $\underline{v}$ corresponds to the state vector function $f(S) = S'$ where $S'(z) = S(z)$ for all $z \notin v$ (this is abbreviated $S' =_{\{v\}} S$ in [McCarthy and Painter 67] and $S' = S \| M'$ in [Wagner 68], where M' is the set of all variables except v) and $S'(v) = k$. This is called $a(v, k, S)$ in [McCarthy 63]; by writing this as $a_C(S)$ where C is $\{v \leftarrow k\}$, we may obtain the effect a_C of the assignment C. This notation, along with the earlier mentioned $c(v, S)$, continues to be used ([McCarthy 66], [Painter 67], [Kaplan 67], [Kaplan 68]).

An assignment with side effects is identical to a general command, as noted in [Good 70], since any variable may have its value changed by side effects. To put it another way, any assignment is equivalent to a sequence of assignments without side effects, as noted in [Floyd 67], although extra variables may need to be introduced (for example, when two variables interchange their values). The terminology of various authors with respect to assignments is displayed in Table 7.

Several authors, following McCarthy, restrict themselves to assignments without side effects. Engeler's definition of operations (= commands) is perfectly general [Engeler 67], but the only operations he actually considers are assignments without side effects. Assembly language instructions and their state vector functions are treated in [Painter 67]; here, of course, no side effects are present. In [Elgot and Robinson 64], it is explcitly considered as possible that the left side of an assignment, as well as the right side, might depend on the state vector; assignments are written "Insert F in G" (for G:=F) where F and G are both functions of the addresses a_0, ...,

Elgot and Robinson 64	"The following schema is typical of a large class of instructions which may appear in a program...'Insert F....in G....' ....In this instruction k and a are supposed to constitute the given state.... The next content function k' satisfies: k'(x) = k(x) for all x except x = G.... for which k(x) = F....."
Engeler 67	"Each operation is intended to produce....a map from A^ω into A^ω (where A^ω is the set of all sequences $\langle a_0, a_1, \ldots \rangle$ of elements of A). The following are the types of operations from which the admissible ones are selected.....$x_i := g_k(x_{j_1}, \ldots, x_{j_{m_k}})$, which transforms $\langle a_0, \ldots, a_i, \ldots \rangle$ into $a_0, \ldots, f_k(a_{j_1}, \ldots, a_{j_{m_k}}), \ldots \rangle$."
King 69	"A 'state vector' of a program is an (n+1)-tuple of values $\langle$N,a1,a2,...an$\rangle$ where N is an integer $1 \leq N \leq m$ (the program counter) ....Let vj=$\langle$i,b1,b2,...,bn$\rangle$ be an arbitrary vector in the sequence. Then if <u>si</u> has the form <u>assign</u>:{xk,f(x1,x2,...,xn),N}, the next vector in the sequence is vj+1=$\langle$N,b1,...,f(b1,b2,...,bn),...,bn$\rangle$ where the expression f(b1,b2,...,bn)....occurs at the (k+1)-th position in the state vector."
Luckham, Park, and Paterson 70	"$\sigma(i)$ is the i-th member of σ, $P(\sigma)(i)$ is the vector after $\sigma(i)$ is executed and $P(\sigma)(i,j)$ is its j-th component....The execution and computation sequences, σ_I and $P_I(\sigma_I)$.... are defined inductively as follows....If $\sigma_I(i+1)$ is the assignment $k.L_w := F^v_t(L_{u_1}, \ldots, L_{u_v})$, then....$P_I(\sigma_I)(i+1,j) = P_I(\sigma_I)(i,j)$ if $j \neq w$, and $P_I(\sigma_I)(i+1,w) = I(F^v_t)(P_I(\sigma_I)(i,u_1), \ldots, P_I(\sigma_I)(i,u_v))$."
Burstall 70	"<u>p</u> <u>has</u> <u>assignment</u>(<u>i</u>,<u>e</u>) <u>and</u> <u>p'</u> <u>follows</u> <u>p</u> <u>and</u> <u>s</u> <u>at</u> <u>p</u> $\Rightarrow$ <u>next</u>(<u>s</u>) <u>at</u> <u>p'</u> <u>and</u> <u>val</u>(<u>i</u>,<u>next</u>(<u>s</u>)) = <u>val</u>(<u>e</u>,<u>s</u>) <u>and</u> [<u>identifier</u>(<u>i'</u>) <u>and</u> <u>i'</u> $\neq$ <u>i</u> $\Rightarrow$ <u>val</u>(<u>i'</u>,<u>next</u>(<u>s</u>)) = <u>val</u>(<u>i'</u>,<u>s</u>)]"
Good 70	"Let M be a memory, and let u = $(u_1, \ldots, u_m)$ and w = $(w_1, \ldots, w_n)$ be vectors of cells in M. Also let $\overline{M}_w$ denote the vector of all cells not in the vector w. Assignment operations on M are written w:=f(u) where f is a total n-vector valued function.... An assignment operation can be initiated at any time t....The precise meaning of w:=f(u) is w(t+1) = f(u(t)) and $\overline{M}_w(t+1) = \overline{M}_w(t)$ where t is the time when the operation is initiated."

TABLE 7

a_j and of their contents as given by the current content function k. These considerations are also treated in [Strachey 66], where side effects are considered. In [King 71], assignment and go-to (transfer) functions are combined; assign:{xk, f(x1, x2, ..., xn), j} means "perform the assignment xk:=f(x1, x2, ..., xn) and then go to j." The state vector function associated with this assignment is formally defined, as is the case in [Burstall 70] with assignment(var, expr), and in [Luckham, Park and Paterson 70] with $k.L_i = F^n_j(L_{k_1}, L_{k_2}, \ldots, L_{k_n})$.

Left sides of assignments in abstract treatments are usually as simple as possible; it is normal to forbid them even to have subscripts, let alone side effects. (A side effect on the left side may be caused by a function call in a subscript expression, although this is forbidden, for example, in FORTRAN II.) In [Strachey 66], however, the general case is treated; his $L(\mathcal{E}, \sigma)$ is a "generalized address," where $\mathcal{E}$ is an expression and σ is a state vector. In our terminology we would say that the state vector function f_v corresponding to the left side v has values which are variables. That is, the range of this function is the domain of the state vectors (as content functions) under consideration. If v can be both a left side and a right side, the corresponding state vector functions f_v and g_v are related by the equation $g_v(S) = S(f_v(S))$ (a similar equation is given in [Park 68]). A side effect of a left side is exactly like a side effect of a right side -- it is a function from state vectors to state vectors. Strachey's $L'(\mathcal{E}, \sigma)$ is a function whose value is a pair (α, σ'), where α and σ' correspond respectively to the values of f_v and g_v above.

To construct the state vector function of a generalized assignment, in which either the left or the right side, or both, may have side effects, we must decide whether the left side or the right side is to be evaluated first. Two formulas, one for left-to-right and the other for right-to-left evaluation, are given in [Strachey 66]. Again combining Strachey's work with Burstall's, it is possible to replace

$$\begin{array}{l}\underline{\mathrm{val}}(\underline{\mathrm{i}}, \underline{\mathrm{next}}(\underline{\mathrm{s}})) = \underline{\mathrm{val}}(\underline{\mathrm{e}}, \underline{\mathrm{s}})\ \underline{\mathrm{and}} \\ [\underline{\mathrm{identifier}}(\underline{\mathrm{i}}')\ \underline{\mathrm{and}}\ \underline{\mathrm{i}}' \neq \underline{\mathrm{i}} \Rightarrow \underline{\mathrm{val}}(\underline{\mathrm{i}}', \underline{\mathrm{next}}(\underline{\mathrm{s}})) = \underline{\mathrm{val}}(\underline{\mathrm{i}}', \underline{\mathrm{s}})]\end{array}$$

in Burstall's formula (our Table 7) by

$$\begin{array}{l}\underline{\mathrm{right}}(\underline{\mathrm{left}}(\underline{\mathrm{i}}, \underline{\mathrm{s}}), \underline{\mathrm{next}}(\underline{\mathrm{s}})) = \underline{\mathrm{right}}(\underline{\mathrm{e}}, \underline{\mathrm{sideeffect}}(\underline{\mathrm{i}}, \underline{\mathrm{s}}))\ \underline{\mathrm{and}} \\ [\underline{\mathrm{identifier}}(\underline{\mathrm{i}}')\ \underline{\mathrm{and}}\ \underline{\mathrm{i}}' \neq \underline{\mathrm{left}}(\underline{\mathrm{i}}, \underline{\mathrm{s}}) \Rightarrow \\ \underline{\mathrm{right}}(\underline{\mathrm{i}}', \underline{\mathrm{next}}(\underline{\mathrm{s}})) = \underline{\mathrm{right}}(\underline{\mathrm{i}}', \underline{\mathrm{sideeffect}}(\underline{\mathrm{e}}, \underline{\mathrm{sideeffect}}(\underline{\mathrm{i}}, \underline{\mathrm{s}})))]\end{array}$$

for left-to-right evaluation (where $\underline{\mathrm{left}}(\epsilon, \sigma)$ and $\underline{\mathrm{right}}(\epsilon, \sigma)$ are Strachey's $L(\epsilon, \sigma)$ and $R(\epsilon, \sigma)$ respectively, and $\underline{\mathrm{right}}(\epsilon, \sigma)$ corresponds to Burstall's $\mathrm{val}(\epsilon, \sigma)$) and

$$\begin{array}{l}\underline{\mathrm{right}}(\underline{\mathrm{left}}(\underline{\mathrm{i}}, \underline{\mathrm{sideeffect}}(\underline{\mathrm{e}}, \underline{\mathrm{s}})), \underline{\mathrm{next}}(\underline{\mathrm{s}})) = \underline{\mathrm{right}}(\underline{\mathrm{e}}, \underline{\mathrm{s}})\ \underline{\mathrm{and}} \\ [\underline{\mathrm{identifier}}(\underline{\mathrm{i}}')\ \underline{\mathrm{and}}\ \underline{\mathrm{i}}' \neq \underline{\mathrm{left}}(\underline{\mathrm{i}}, \underline{\mathrm{sideeffect}}(\underline{\mathrm{e}}, \underline{\mathrm{s}})) \Rightarrow \\ \underline{\mathrm{right}}(\underline{\mathrm{i}}', \underline{\mathrm{next}}(\underline{\mathrm{s}})) = \underline{\mathrm{right}}(\underline{\mathrm{i}}', \underline{\mathrm{sideeffect}}(\underline{\mathrm{i}}, \underline{\mathrm{sideeffect}}(\underline{\mathrm{e}}, \underline{\mathrm{s}})))]\end{array}$$

for right-to-left evaluation. Similar formulas may be obtained for multiple assignments in ALGOL, where more than one assignment symbol (:=) is present.

2.4 Conditions and Conditionals

A condition (or, as it is sometimes called, a predicate, or, especially in connection with correctness proofs, an assertion) is simply an expression whose value is true or false, rather than being a number, and differs from an arithmetic expression only in this regard. Indeed, the distinction between functions and predicates which is commonly made in the study of program schemes appears quite meaningless from the viewpoint of a programmer working with a declarationless language like LISP or APL; the conditional functions here are applicable to arbitrary arguments, and consider 0 (or NIL) as false and anything else as true.

The state vector function corresponding to a condition has values true and false; by taking the inverse image of true, one obtains a set of state vectors for which the condition holds. The association of a condition with a set of state vectors is especially useful when considering assertions (see section 5.1). If condition C implies condition C', then the set corresponding to C is contained in the set corresponding to C'. Conditions are built up from other conditions and from arithmetic expressions; the function equal(e, e'), which gives the condition that the two expressions (either arithmetic or Boolean) e and e' have equal values, is given in [Burstall 70], together with a definition of its state vector function: $\underline{val}(\underline{e}, \underline{s}) = \underline{val}(\underline{e}', \underline{s}) \Rightarrow \underline{val}(\underline{equal}(\underline{e}, \underline{e}'), \underline{s}) =$ true; $\underline{val}(\underline{e}, \underline{s}) \neq \underline{val}(\underline{e}', \underline{s}) \Rightarrow \underline{val}(\underline{equal}(\underline{e}, \underline{e}'), \underline{s}) =$ false.

A conditional expression is an expression of the form if b then x else y, where x and y are expressions (of any type) and b is a Boolean expression. Such expressions occur in ALGOL [Naur et al. 60] for arithmetic and for Boolean expressions x and y, and are called _arithmetic_ and _Boolean_ (as opposed to _simple arithmetic_ and _simple Boolean_) _expressions_. They are also studied in [McCarthy 63], although the term "conditional expression" is here more general, and encompasses recursive conditional definitions such as $n! = $ _if_ $n = 0$ _then_ 1 _else_ $n \cdot (n-1)!$. Leaving these out for the moment, McCarthy's discussion implies the state vector function relation (without side effects): if c is {_if_ b _then_ x _else_ y} then $f_c(\xi) =$ _if_ $f_b(\xi)$ _then_ $f_x(\xi)$ else $f_y(\xi)$, where ξ is a state vector and f_a is the state vector function corresponding to a. When side effects are present, this equation must be modified; for a left-to-right evaluation scheme, we have $f_c(\xi) =$ _if_ $f_b(\xi)$ _then_ $f_x(s_b(\xi))$ _else_ $f_y(s_b(\xi))$ and $s_c(\xi) =$ _if_ $f_b(\xi)$ _then_ $s_x(s_b(\xi))$ _else_ $s_y(s_b(\xi))$, where s_a is the side-effect function corresponding to a.

The quantities x and y in _if_ b _then_ x _else_ y may be arbitrary, as long as they are the same. They may, in particular, be left sides, as in (_if_ $a > b$ _then_ j _else_ k):=i [Strachey 66] (his terminology is $a > b \rightarrow j, k := i$) or subroutine names, as in (_if_ $a < b$ _then_ sin _else_ cos) [Landin 66b], as well as expressions. Most often, however, they are _commands_, in which case the "else y" may be eliminated if y does nothing. A conditional command may be treated like a conditional expression with side effects, by ignoring the value of the expression -- that is, by using only the second of the two equations above. We must, however, also allow for the state vector function, if any, which specifies the next command. If this is denoted by x_a for the command _a_ (the "exit function" of _a_) then, using the notation above, we have $x_c(\xi) =$ _if_ $f_b(\xi)$ _then_ $x_x(s_b(\xi))$ _else_ $x_y(s_b(\xi))$. If the "else y" is eliminated, we may set $x_c(\xi)$ to a special reserved word, such as _next_, when $f_b(\xi) =$ _false_, to indicate that the command has exited normally. More generally, this device allows arbitrary commands to have exit functions; a simple assignment has a constant exit function with constant value next. (This is different from the exit function being undefined, which it might be, for example, if a subscript were out of range.)

Many program models simplify the treatment of conditional commands by allowing only conditional transfers. The effect of a conditional transfer (without side effects) is the identity function, unless the state vectors have program-counter components, in which case these are the only components that are changed. These may involve one exit, as Burstall's ifthen(expr, pt) (meaning "if expr is true then go to pt"), or two exits, as Elgot and Robinson's "If R is true, go to H, and if R is false, go to $\overline{H}$," where R, H, and $\overline{H}$ are functions of the addresses $a_0, \ldots, a_j$ and their contents as given by the current state vector. Further examples appear in section 3.1, where we discuss restrictions on conditional instructions in models of programs.

3 MODELS FOR PROGRAMS

The mathematical definition of a <u>program</u> has been attempted by a large number of authors, as displayed in Table 8. We may notice immediately that many of these are not concerned with state vectors. Basically, "a <u>program</u> is a sequence of <u>instructions</u>....each of which specifies (i) an operation to be performed, and (ii) the next instruction to be executed" [Minsky 61]. It is thus possible to define programs in terms of operations without simultaneously defining operations in terms of state vectors.

In examining various program models, we observe that there are three important ways in which they differ:

(1) Some models allow only conditional transfer without side effects and assignment, while others treat more general forms of instructions.

(2) Some models are based on flowcharts or directed graphs, while others view a program simply as a set of statements.

(3) Some models define a <u>program</u>, while others define a more general object, called a <u>program scheme</u>, whose interpretations then correspond to programs.

We shall take these up one at a time, after which we shall discuss subroutines, blocks, and other program features included in only a minority of the program models studied.

	State vectors (or equivalent) used?	Two-branch or multi-branch model?	Directed graph model?	Programs or program schemes?
Yanov 58	No	Multi-branch	No	Schemes
Kaluzhnin 59	No	Two-branch	Yes	Schemes
Engeler 67	Yes	Two-branch	No	Programs
Floyd 67	No	Multi-branch	Yes	Programs
Narasimhan 67	No	Two-branch	Yes	Programs
Scott 67	No	Two-branch	No	Programs
Elgot 68	Yes	Multi-branch	No	Programs
Cooper 69	No	Two-branch	No	Schemes
Manna 70	No	Multi-branch	Yes	Schemes
Luckham, Park, and Paterson 70	No	Two-branch	No	Schemes
Good 70	Yes	Multi-branch	Yes	Programs
King 71	Yes	Two-branch	No	Programs

TABLE 8

3.1 Restrictions on Conditional Instructions

A number of authors have sought to simplify mathematical models for programs by placing two important restrictions on the conditional instructions which they contain:

(1) No conditional instruction may have more than two exits.

(2) No conditional instruction may change the value of any variable (except the program counter, if that is counted as a variable); that is, it must be a conditional transfer without side effects.

The first to make this particular simplification was apparently Kaluzhnin [de Bakker 69], who introduced, in 1959, the A-F graph schema. This is a directed graph model in which there are four types of node: the entrance node, the exit node, operator nodes (= generalized assignment), and discriminator nodes (= conditional transfer). In any interpretation, an operator node corresponds to a mapping from the domain of interpretation to itself, while a discriminator node corresponds to a mapping from the domain of interpretation to {true, false}. This corresponds to the state vector function associations made earlier in this paper, if the domain of interpretation is taken as the set of all applicable state vectors.

Kaluzhnin graphs have been re-introduced several times since. A glossary of terminology relating to this model is given in Table 9, and, for each of the authors cited in this table, a summary of the features of his model which distinguish it from the others under consideration is given in Table 10. It will be seen that a great number of arbitrary choices exist within the general restrictions above.

In the general, or multi-branch model, as opposed to Kaluzhnin's two-branch model, there is essentially only one type of state-

Kaluzhnin 67	Entrance	Operator node	Discriminator node	Exit
Engeler 67	Starting instruction (any type)	Operational instruction k: do ψ then go to p	Conditional instruction k: if φ then go to p else go to q	None (stop when transfer to undefined label)
Floyd 67	Starting point for the program	Assignment operation $x \leftarrow f(x,y)$	Conditional branch Φ	Halt for the program
Narasimhan 67	Initial vertex (any type)	Vertex with exactly one outgoing branch	Decision vertex	Terminal vertex
Scott 67	Start instruction	Operation instruction L: do F; go to L'	Test instruction L: if P then go to L' else go to L"	Halt instruction
Cooper 69	Always start at first statement	Assignment statement $L_k \leftarrow f(L_{i_1}, \dots, L_{i_n})$ Separate go to statement	Test statement $t(L_k)n_1,n_2$ (here n_1 is the false exit)	Halt statement
Bjorner 70	Entry nodes or begin-nodes	Operator nodes or action-nodes	Discriminator nodes or decision nodes	Exit nodes or end-nodes
Luckham, Park and Paterson 70	Always start at first in-	Assignment instruction $k.\ L_i := F_j^n(L_{k_1}, L_{k_2}, \dots, L_{k_n})$ Does not transfer	Transfer instruction $k.\ T_i(L_i)m,n$ For unconditional transfer, set m=n	Stop instruction
King 71	Always start at first	Assign statement assign: {xk, f(x1, x2, ..., xn), j}	Test statement test: {p(x1, x2, ..., xn), j1, j2}	Halt statement

TABLE 9

Kaluzhnin 59	Each graph corresponding to a program must have only one exit node (see also [Cooper 67] and [Manna 68]).
Narasimhan 67	Only three types of node in a basic flow chart (= program); the initial vertex (= node) may be any of these types.
Scott 67	Four types of statement to correspond to Kaluzhnin's four types of node; all statements are labelled.
Floyd 67	Fifth type of node in flowchart language, corresponding to a join of control, where exits from two nodes meet and continue as one.
Engeler 67	Only two statement types; the start may be either of the two, and termination is signalled by transfer to a nonexistent label.
Cooper 69	Assignment and unconditional transfer are separated; there are assignment, test, <u>go to</u>, and halt statements, and the initial statement may be any of these.
Bjorner 70	More than one <u>begin</u>-node allowed; graphs must be "clean" in a number of ways (no unreachable statements, no conditional may transfer to itself).
Luckham, Park and Paterson 70	No explicit unconditional transfer; all unconditional transfers are conditional transfers which go to the same label in either case.
King 71	Statement names, which are always integers from 1 to n, are used instead of labels; there are three statement types, and every program starts at statement 1.

TABLE 10

ment in a program. It specifies, for each state vector before the statement is executed, the next statement to be executed as well as the new state vector; in the terminology of section 2.4, it has an <u>effect</u> and an <u>exit function</u>. Yanov [Rutledge 64] refers to the "shift relation" which gives the next state vector (if Yanov's evaluations are interpreted as state vectors) and the "successor funcation" which gives the next statement, both as functions of the current evaluation. In [Elgot and Robinson 64], each element $b \in B$ (the set of words) is made to correspond to an instruction $h_b: \Sigma_0 \rightarrow \Sigma_0$, where each element of Σ_0 is a state (= state vector) giving each element $a \in A$ (the set of addresses) a value in B, and the program counter a value in A. This is more general than the above model, since it allows the possibility that the program-counter component of $\sigma \in \Sigma_0$ can affect address components of $h_b(\sigma)$ -- that is, that the effect of the instruction corresponding to b may depend on its location. Elgot and Robinson call a mapping h_b <u>location independent</u> if this cannot happen, and an entire machine (RASP) is called location independent if each of its instructions is.

Every program in the multi-branch model is equivalent to one in the two-branch model; however, the equivalence must be constructed with care. In the directed graph model of [Good 70], each node has an associated operation which transforms the state vector, and all links leaving a node carry mutually exclusive traversal conditions. Thus an arbitrary node with <u>n</u> exits would seem to be equivalent to an assignment followed by <u>n</u>-1 two-way conditional branches. Such an equivalence, however, does not always hold without the introduction of extra variables, as shown by the ALGOL example **if** x=y **then** **go** **to** a **else** **go** **to** b where **real** **procedure** y; **begin** y:=z; x:=z **end**; . Here, if the node operation consists of y:=z followed by x:=z, we have no way of telling where to go next, since x, y, and z are now all equal. We may obtain equivalence by introducing a new Boolean variable, test; the node operation is now test:=(x=y); y:=z; x:=z; , and the traversal conditions are test=true and test=false. The directed graph model of [Manna 70] is more carefully constructed; he associates an operation on the domain with each arc of the graph, rather than with each node, and the operation is presumed to take place after the test is made. Under these conditions, the equivalence described above may be carried out without the necessaity of introducing extra variables. (Manna refers to elements of a domain, in the above sense, as state vectors, but makes no further study of their structure.)

The complexity of modern programming languages has reached a point at which even the assignment and conditional transfer statements which the two-branch model was intended to reflect are no longer well represented by that model, in particular if side effects are present. Further language features, such as FORTRAN assigned and computed GO TO's and arithmetic IF's and ALGOL go to's involving switches, are also not well represented in the two-branch model. We may always carry out the equivalence above, but the equivalent two-branch graph obscures timing considerations for computed GO TO's and the like, and is sometimes almost impossibly clumsy, as when describing the effect of a block with several exits.

3.2 Directed Graphs and Statement Sets

Our second point of difference among program models concerns the choice of a geometric versus an algebraic representation. Flow diagrams, or flow charts, have long been used by programmers (see, for example, [Goldstine and von Neumann 47]). A flowchart, in the sense considered here, is a finite directed graph whose nodes are statements of a program and whose links correspond to paths of control from one statement to another. We note that at least two other graph models of computations have been studied; in one of these, the links represent data dependency, in the sense that a link from A to B signifies that B cannot be started until certain data has been calculated by A [Martin and Estrin 68]; the nodes in the other are effectively finite paths in the normal flowchart which start at initial nodes, and a link from A to B signifies that the path A is obtained from B by deleting its last node [Scott 71].

Flow diagrams, of course, are also used in Kaluzhnin's model, discussed in the preceding section. Kaluzhnin and most other authors have used different terms for a computational process and its graph; the graph, by itself, presumably contains no information as to what a node does, or when a given link is taken. The terminology which has been used in this connection by various authors is given in Table 11.

Geometric considerations give rise to a number of useful elementary facts about programs. A node is <u>initial</u> if there are no links which lead into it; it is <u>terminal</u> if there are no links which lead out of it. Initial and terminal nodes correspond in an obvious way to start and stop statements of a program. A statement may be identified as unconditional (an assignment or an unconditional transfer, for example) if it has only one link which leads out of it. A computation sequence corresponds in an obvious way to a <u>path</u> in the graph which leads from an initial node to a terminal node. If a given node in a graph is not on any such path, the corresponding statement may be

Goldstine and von Neumann 47	-----------	Flow diagram	Boxes	Arrows
Kaluzhnin 59	Graph schema	Interpretation	Nodes	Arrows
Floyd 67	Flowchart	Interpretation*	Vertices	Edges
Narasimhan 67	Directed graph	Basic flow chart	Vertices	Branches
Bjorner 70	Well-formed flowchart	-----------	Nodes	Links
Manna 70	Directed graph	Interpreted graph	Vertices	Arcs
Good 70	Directed graph	A-process	Nodes	Arcs

* Floyd's interpretations involve assertions in a proof, rather than specification of a domain, which he does not do explicitly.

TABLE 11

removed from the program without altering its effect; this process of removal may be carried out in a finite number of steps, as noted in [Karp 60]. In addition, the concept of a set of nodes which is such that every closed path, or loop, in the graph must contain at least one member of the set is useful in the proof of correctness of programs by the method of [Floyd 67] (see section 5.1).

It is also possible, of course, to use graphical properties to impose unrealistic restrictions upon programs. Most authors ([Bjorner 70] is an exception) require a single initial node, although clearly the graphs of many real programs, such as those which compute sines and cosines, have more than one initial node. In the preceding section we have noted a number of authors who have required that there be only one terminal node in such a graph; this restriction may have been suggested by ALGOL main programs, which have no halt statements and stop only at the physical end, but it does not, for example, cover the case of ALGOL blocks, which can exit at any point by transferring to a statement in a containing block. The fact that finite automata, as well as programs, have directed graphs has suggested a number of constructions of finite automata and regular expressions corresponding to programs ([Rutledge 64], [Engeler 67], [Kaplan 69a], [Bjorner 70]); in much of this work, however, the authors have been content simply to give the correspondence without producing any deep results for the theory of programs.

A more serious difficulty in using directed graphs is concerned with subroutines. If links are to lead from the exit node of a subroutine to all possible return locations, the decision as to which of these to take must depend on the value of an explicitly introduced return address variable; this seems unnatural, especially for algebraic languages. A highly complex treatment of subroutines treated as flowcharts substituted dynamically into others is given in [Narasimhan 67] in his treatment of hierarchical computation processes. If a program and its subroutines are assumed to correspond to separate directed graphs, it is possible to require that each instruction which calls a subroutine, instead of linking to that subroutine in the directed graph sense, correspond in the calling program to a state vector function which embodies the effect of that subroutine, and a similar statement holds for blocks. These matters will be further examined in section 3.4.

In the models of programs which do not use directed graphs, there is normally a set of statements, some or all of which are labelled in some way. The use of such a set of statements rather than a directed graph is much like the use of a program rather than a flowchart to illustrate how a problem is solved -- it is primarily a matter of taste. There is still another approach, however, according to which an entire program is represented by a single state vector function $q: \mathfrak{J} \to \mathfrak{J}$, where the state vectors in $\mathfrak{J}$ have program-counter components. The idea is that any $T \in \mathfrak{J}$ gives us the current statement as well as the current state vector, so that we can execute that statement and arrive at a new current state vector and a new current statement, which may then be combined to form $q(T)$. If we take this point of view, the directed graph of a program becomes a derivative notion, rather than a fundamental one; its nodes are the values of the program counter, and it has a link from $\underline{i}$ to $\underline{j}$ if and only if there exists $T \in \mathfrak{J}$ with $T(\lambda) = i$ and $T'(\lambda) = j$, where λ is the program counter and $T' = q(T)$. Any directed graph may appear in this way, and, in particular, there are no restrictions on the form of a graph such as are made in the two-branch model.

3.3 Program Schemes and Interpretations

One advantage in using directed graphs is that we can prove facts about a graph which are independent of the contents of the boxes. Thus by removing boxes which cannot participate in terminating computations as in [Karp 60], for example, we can often replace a given program with a smaller equivalent one. There is another class of program models which give much more general results on program equivalence and other topics by using only a little bit more information about a program than is given by its graph. These are called program scheme (or schema) models, and a particular program which is an example of any program scheme is called an interpretation of that scheme.

The extra information available in a program scheme derives from the notion of form in mathematical logic. A test which takes place in a flow diagram corresponds to a condition, or predicate; an operation, which changes the state vector, corresponds to a function. In a program scheme these predicates and functions are given by their form only. That is, we are effectively given a directed graph containing predicate symbols and function symbols, but nothing else is said about what the predicates or the functions actually do. In any interpretation, these symbols are made to correspond to actual predicates and functions over some domain of interpretation.

There are as many definitions of program schemes as there are papers on the subject, but generally these definitions fall into two broad classes, depending on the nature of the domain of interpretation. In the first group of models, the domain of interpretation is composed of state vectors. (This fact is often left "understood," but it is explicitly stated in [Manna 70].) A "function" corresponds to a map from the domain to itself; a predicate corresponds to a map from the domain to {true, false}. Thus functions and predicates correspond to the state vector functions of general commands and conditions respectively. We have already seen how interpretations of Kaluzhnin graphs follow this scheme; Yanov's model [redacted] is another example; Still another is found in [Cooper 67].

In the second group, the domain of interpretation is more like a set of values, and there are argument symbols (which correspond to variables) as well as function and predicate symbols. A function of n arguments now corresponds to a map from D^n into D, where D is the domain, rather than from D into itself; a predicate with n arguments corresponds to a map from D^n to {true, false}. Thus predicates are much as before, but functions now correspond to the state vector functions of expressions, rather than commands. Therefore, an explicit notion of assignment is introduced in order to include simple commands in the model. All of the types of program scheme in this group use the two-branch model (as introduced in section 3.1); but they differ in several other seemingly unimportant ways, as displayed in Table 12. Most of these models are not concerned with state vectors explicitly, although it is easy to define the relevant state vector functions in each case, and this is actually done in [Kaplan 69a] and in [Luckham, Park and Paterson 70]. The functional arguments in [Kaplan 69a], in addition, are recursively defined terms, whereas in the other models they are simply argument symbols.

Any program scheme has an interpretation in which the values of variables are strings of operators (or their labels, if every operator is labeled) and in which the effect of each operator upon

Paterson 68	Functions with several arguments	Predicates with one argument	Locations
Cooper 69	Function names with several arguments (finitely many)	Test names with one argument (finitely many)	Registers (finitely many)
Kaplan 69a	Function letters with several arguments (recursively defined; infinitely many)	Relation letters with several arguments (infinitely many	Variables (infinitely many); terms (arguments of relations)
Milner 70a	One function letter, with one argument	One test letter, with one argument	Register letters (infinitely many)
Milner 70b	Function letters with several arguments (infinitely many)	Predicate or test letters with several arguments (infinitely many)	Location letters (infinitely many)
Luckham, Park and Paterson 70	Operator symbols with several arguments (infinitely many)	Transfer symbols with one argument (infinitely many)	Location symbols (infinitely many)

TABLE 12

the current string is to append itself to that string. In schemes as defined by Yanov, the resulting value at the end of any computation is the string of those operators which were executed, in their execution order, or the trace of that computation (see Table 3). Thus, in particular, if two such schemes are equivalent in all interpretations, their traces must agree. An interpretation somewhat similar to this one is called the free interpretation in [Luckham, Park and Paterson 70], although here only right sides of assignments, rather than entire commands, are being interpreted. Further discussion of program schemes is postponed to section 5.3, where we discuss equivalence of programs.

Still another form of separation of the properties of a flowchart from the properties of the computations in the boxes is the "outer and inner language" concept of [Wilkes 68]. The BNF rules which define a language are here separated into those concerned with specifying the flowchart (outer syntax) and those concerned with types of commands and parts of commands (inner syntax).

3.4 Programs, Subroutines, and Blocks

We now return to the method of analysis of Part 2. What sort of state vector function do we wish to associate with a program? One answer is suggested at the end of section 3.2: if $\mathfrak{J}$ is the set of all applicable state vectors with program-counter components, where the values of the program counter are the statements of the program P, then the state vector function to be associated with P is a function from $\mathfrak{J}$ into itself. This function, in turn, is specified completely by the effects and exit functions of all the statements in the program. If the statements are $C_1, \ldots, C_n$, the respective effects are $f_1, \ldots, f_n$, and the respective exit functions are $x_1, \ldots, x_n$, then the function $q: \mathfrak{J} \rightarrow \mathfrak{J}$ satisfies $q(S, C_i) = (f_i(S), x_i(S))$, where elements of $\mathfrak{J}$ are identified in the obvious way with pairs (S, C) with S a state vector without a program-counter component and C a statement, that is, a value of the program-counter variable.

There is another kind of state vector function, however, that may also be associated with a program. If S is the current state vector at the beginning of a program, and we execute the entire program, then, when it ends, we have a new current state vector, which we may denote by e(S). If the program does not end if started at S, then we may consider e(S) to be undefined. Thus e is a function from state vectors (without program counter components) to state vectors; we shall call it the effect of the given program, and we note that it also depends upon the choice of starting statement.

The effect e is derivable from the execution function q mentioned above, although, generally, only in semi-computable fashion. ([Mazurkiewicz 71] calls e the tail function corresponding to q.) The easiest way to perform this derivation is to construct a computation sequence $T_0, T_1, \ldots,$ from q by the equation $T_{i+1} = q(T_i)$, for $i \geq 0$, whenever this is defined; the effect e is then given by $e(S) = S'$ where $T_0 = (S, F)$, $T_n = (S', X)$ is the last member of the sequence to be defined, F is the starting statement, and X is the final statement. Definitions of this type have been made by a number of authors, as summarized in Table 13. Alternatively, we may define $e(S) = e'(S, F)$ where $e'(S, C) =$ _if_ $q(S; C)$ is undefined _then_ S _else_ $e'(q(S, C))$; this is a recursive definition which may not always recursively terminate. This style is used in the definition of micro(π, ξ) in [McCarthy 66] and that of $E(\mathfrak{U}, D, \xi, x)$ in [Kaplan 69a].

Another related function may be derived for a block (in ALGOL, for example) which may terminate either normally or by transferring to any of a number of labels defined in blocks containing the given one. Let S be a starting state vector; we run the program and determine how it exits. If it exits by transfer to the label L, then we write $L = n(S)$. If it does not exit at all, then $n(S)$ is undefined. If it exits normally -- that is, to the next statement following the block -- then we set $n(S)$ equal to some special word, such as _next_. The function _n_ may be called the _next-label function_ of the program; like the effect, it depends upon our choice of the start statement. Subroutines, as opposed to blocks, can exit in only one place in many languages, and thus would not have next-label functions, although we may note that in ALGOL the same considerations apply in this case as well (of course, it is not very common for a procedure to terminate in this way, but it might, in order, for example, to transfer to an error exit.)

We may thus associate two functions with a program which are of exactly the same form as the two functions associated with a general command. The obvious conclusion is that these functions should become the effect and exit function, respectively, of any _call_ to the given program as a subroutine. Thus _a call does not cor-_

Yanov 58	$I(\mathfrak{U}): D \rightarrow D \times T$, where D is the (state vector or evaluation) domain, T is the set of terminal operators, I is an interpretation of the program scheme $\mathfrak{U}$.
Elgot and Robinson 64	$f: B^r \rightarrow B^s$, where B is the set of words, is computed (this notion is defined) by the program π at datum locations $d_0, \ldots, d_{r-1}$ and value locations $v_0, \ldots, v_{s-1}$, all of which are addresses.
Engeler 67	Given a construction and an input (state vector), we have an output (state vector); does not explicitly say that the output is a function of the input.
Scott 67	$\mathfrak{M}_\pi: X \rightarrow Y$, where X is the input (state vector) domain, Y is the output domain, π is a program, and $\mathfrak{M}$ is a machine.
Orgass and Fitch 69b	The program p determines the output state s_2 with respect to the input state s_1 if s_1, p, and s_2 are related by the programming language Ξ.
Wegner 70	The value of a computation (that is, of a sequence of information structures) is some function of the last element of the sequence.
Luckham, Park and Paterson 70	$Val(P_I)$ is the value of the program P under the interpretation I which, in particular, specifies an initial state vector; $Val(P_I)$ is then the final state vector.
King 71	Given an execution sequence and an initial state vector, we have a final vector; does not explicitly say that the final vector is a function of the initial vector.

TABLE 13

respond to a link (in the directed graph) to a subroutine entrance; a return does not correspond to links to all return addresses. Instead, each subroutine and each block is represented by a separate directed graph. Each usage of a block and each call to a subroutine corresponds to a general command whose effect and exit function are the effect and next-label function, respectively, of the corresponding block or subroutine.

The effect e and the exit function x may be combined into a single "overall effect" o, such that o(S) is the pair (e(S), x(S)) for any state vector S. Any command then has an overall effect, and if commands are combined to form a program (with a specified starting point), that program also has an overall effect. Overall effects are similar to directions as studied in [Elgot 68]; a direction of rank $\langle n; p, \omega q\rangle$ is a function of state vectors involving n variables whose values are pairs consisting of state vectors involving p variables and integers i, $0 \leq i < q$. The integer is supposed to denote the choice of exit. Given a set of directions D, we may form the set $\bar{D}$ of all directions corresponding to programs which are such that each command in them has a direction belonging to D. The bar operator taking D into $\bar{D}$ is a closure operator in the topological sense; that is, it satisfies $D_1 \subseteq D_2 \Rightarrow \bar{D}_1 \subseteq \bar{D}_2$, $D \subseteq \bar{D}$, and $\bar{\bar{D}} = \bar{D}$ (the last of these being the most important and the hardest to prove).

When a subroutine is called as a function inside an expression, it has a value v(S) for any state vector S given by v(S) = S'(n), where S' = e(S) for the effect e of the subroutine, and where n is the name of the output variable (normally an accumulator in machine language, or the function name in an algebraic language).

Thus a function call plays the same rôle as an expression. (The association of a value function of this kind with every string in a language J is called a realization of J in [Skowron 71].) The effect of the subroutine becomes part of the side effect of the expression in which the function call is contained, and its next-label function affects the exit function of any command containing that expression. Usually, of course, an assignment in ALGOL, for example, exits normally to the next instruction, and so the value of its exit function is next, for any state vector. This is not the case, however, if it contains an expression including a function call which sometimes or always transfers to a label in the block containing the given assignment, or in some block containing that one. In this case the value of the exit function is determined, in a computable way, from the next-label functions of function calls in the command.

A subroutine with formal parameters has an effect for each choice of actual parameters. That is, there is a mapping from the set of possible choices of actual parameters into the set of possible effects (and next-label functions). This mapping depends upon the methods of calling parameters (call by name, by value, by address, etc.), but there can be no doubt that it exists, although it may be difficult to determine and may even be uncomputable. Alternatively, the formal parameters may be counted as variables whose values are the corresponding actual parameters or their state vector functions (<u>not</u> their values, unless they are called by value); the subroutine then has a single effect and next-label function whose state vectors involve these components. If a subroutine contains a recursive call, this is replaced by its effect and next-label function, just as in the case of a non-recursive call; this leads, of course, to a recursive definition of the effect.

4 PROGRAM SYNTAX AND SEMANTICS

There are serious doubts as to whether a "program theory," universally applicable to programming situations, could be built up around any of our definitions of programs, flowcharts, program schemes, and so on, or, for that matter, around any single mathematical definition. Programming languages in practice are so diverse that it is unrealistic to expect programs in all of them, under all situations, to correspond to any one model. If we are going to prove assertions about programs written in some language, we must relate the way in which a program is formed in that language (the syntax of the program) to the results which the program produces (the semantics of the program). This is the subject of the present part of this paper.

Our approach to program semantics continues a line of investigation initiated by de Bakker, who summarizes nine different language-definition methods without, however, attempting a systematic evaluation of them [de Bakker 69]. We shall first discuss the method of Knuth, which is the last of the methods surveyed by de Bakker, and show how state vectors and state vector functions provide exactly the mechanism needed to apply this method to arbitrary programming languages. We then compare this with the method of McCarthy, both in its original form [McCarthy 63] and as it has been applied in the Vienna definition language, and show that Knuth's ideas can be used in this framework as well. Finally, we discuss a number of other forms of semantics which are based on rigorous definitions of translators or translation schemes. [redacted]

4.1 Knuth's Semantic Attributes

Most programming languages, such as ALGOL, are not context-free [Floyd 62], but may be presented as subsets of context-free languages (that is, in BNF with suitable restrictions, such as that every variable used must be declared), and we may therefore speak of their derivation trees. One way to associate a meaning, or semantics, with a program is to associate a meaning with every substring of that program which occurs on its derivation tree. The meaning of any nonterminal -- including <program> -- is then derived by means of equations which are associated with the production rules involving that nonterminal. This is the central idea behind semantic attributes ([Knuth 68a], [Knuth 71a]).

Any nonterminal may have one or more attributes. These are functions applicable to any string which is an instance of the given nonterminal. Thus if "value" is an attribute of <expression> (where <expression> is defined so as to involve no variables, only constants) then value(e) is defined for any string e which can be shown to be an expression by the given syntactic rules. If these are

```
<expression> ::= <integer> | <expression> '+' <integer>
<integer> ::= <digit> | <integer> <digit>
<digit> ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
```

then we expect value('20'), value('10+10'), and value('13+6+1') to be all equal to 20 (and value('+20') to be undefined, because '+20' is not an expression according to the given rules). In order to define this rather complicated function of strings, let us rewrite the above rules in the style of [Knuth 68a] and give their associated semantic rules; here E, I, and D stand for "expression," "integer," and "digit" respectively, while v_X means the value of X:

$E \rightarrow I$	$v_E = v_I$
$E_1 \rightarrow E_2 + I$	$v_{E_1} = v_{E_2} + v_I$
$I \rightarrow D$	$v_I = v_D$
$I_1 \rightarrow I_2\ D$	$v_{I_1} = 10 \cdot v_{I_2} + v_D$
$D \rightarrow 0$	$v_D = 0$
$D \rightarrow 1$	$v_D = 1$
$D \rightarrow 2$	$v_D = 2$
$D \rightarrow 3$	$v_D = 3$
$D \rightarrow 4$	$v_D = 4$
$D \rightarrow 5$	$v_D = 5$
$D \rightarrow 6$	$v_D = 6$
$D \rightarrow 7$	$v_D = 7$
$D \rightarrow 8$	$v_D = 8$
$D \rightarrow 9$	$v_D = 9$

Now we have values for integers and digits, as well as expressions. The value of '1' is 1 as a digit, and therefore as an integer also. The value of '13' is $10 \cdot 1+3 = 13$, since '13' is the integer '1' (with value 1) followed by the digit 3 (with value 3). Similarly, the value of '13+6' is 19, and that of '13+6+1' is thus 20.

In his papers, Knuth does not consider the semantics of actual programming languages, although this has been attempted in [Knuth 71b] and in [Wilner 71]. But the considerations of the previous two parts of this paper should make it clear that state vector functions may be associated with nonterminals in a programming language as semantic attributes. In particular, if <u>e</u> is an expression (this time possibly involving variables as well as constants), we can associate with e a function f_e such that $f_e(S)$ is the value of e, calculated from the values of variables occurring in e as given by the state vector S. Similarly, a state vector function with values <u>true</u> and false may be associated with a Boolean condition, a state vector function whose values are state vectors with an assignment or other general command, and so on. This method is further discussed in [Maurer 72], where an application is made to describe a dialect of FORTRAN II.

A fundamental characteristic of Knuth's method is the use of two kinds of attributes: synthesized and inherited. Synthesized attributes of a nonterminal are given by equations involving attributes of descendants of that nonterminal, that is, other nonterminals below it on the derivation tree (farther away from the root). All the attributes in the examples above are synthesized. Inherited attributes of a nonterminal are given by equations involving attributes of its ancestors, that is, those nonterminals of which it is a descendant. An excellent example of an inherited attribute in a programming language is the type function which assigns to each identifier its type (real, integer, or whatever). This must be an attribute of each occurrence of such an identifier in the program, in order to determine the state vector functions of expressions. For example, the state vector function corresponding to A+B has values obtained by adding the values of A and B as real numbers or integers (or arrays, structures, etc.), and performing conversions as required, depending on the type-function attributes of A and of B. These, in turn, are inherited from the type-function attributes of the program or block in which the given expression A+B is contained; and these type-function attributes are synthesized from the declaration section in that program or block, where types are assigned to variables, and also from the body of the program in any language in which variables may be left undeclared and their types assigned by default.

Imposing conditions which must be satisfied by the semantic attributes allows us to define languages which are not context-free in a way which closely reflects their informal description. For example, the set of variables declared in an ALGOL block and the set of variables used in that block may both be easily made into synthesized attributes of the block. By requiring that the second set be contained in the first, we obtain a rigorous formulation of the condition that every variable used must be declared. While synthesizing the set of variables declared, we may impose a similar restriction which amounts to the statement that no identifier may be duplicately declared. The type-function attributes of the previous paragraph may be restricted so as to forbid or otherwise restrict mixed-mode arithmetic; while sets of labels defined and used in a block may be constructed and then restricted so as to forbid duplicate label definition or transfer to an undefined label. Further examples are discussed in [Maurer 72].

A top-down parser for any language may be modified to produce the values of any computable synthesized attributes of its nonterminals. When inherited attributes are present, the process of producing these values may be endless; however, there is a test for deciding whether it will ever be endless, given in [Knuth 68a] and in a corrected version in [Knuth 71a]. Thus such a parser may be expanded to obtain an interpreter for languages described in this way.

4.2 McCarthy's Abstract Syntax and Semantics

The concept of abstract syntax and semantics as studied in [McCarthy 63] and applied in [McCarthy and Painter 66], [Kaplan 67], [Painter 67], and the work of the Vienna group (see the following section) is much more general than the treatment of Knuth. It is unrealistic to expect a single program to be able to interpret a language arbitrarily described in this style. By imposing restrictions suggested by the concept of semantic attribute, however, we may obtain a subclass of language descriptions processable in such a way.

According to McCarthy, a language is best described entirely in terms of functions, and there are functions for syntax and for semantics. The syntactic functions are of three types: the "is" functions, the "make" functions, and the component functions. To illustrate for the string 'A*B+C*D', considered as the sum of two terms 'A*B' and 'C*D':

```
issum('A*B+C*D') = true
mksum('A*B','C*D') = 'A*B+C*D'
addend('A*B+C*D') = 'A*B'
augend('A*B+C*D') = 'C*D'
```

The "is" functions are predicates. The predicate (in the natural language sense) of the sentence "A*B+C*D is a sum" is "is a sum"; this is abbreviated "issum," and since the above sentence is true, the predicate issum assumes the value true when applied to A*B+C*D as argument. The "make" functions have as many arguments as there are components in the quantity being "made"; here, a sum is made from two components, an addend and an augend, and hence there are two arguments to the function "make sum," abbreviated mksum. The component functions, addend and augend here, have one argument each. All arguments and values (except the values of the "is" functions) are strings in some programming language.

Any BNF rule is equivalent to a syntactic definition of this kind. We must, however, give names to all compound alternatives and to their components. For example, the BNF rule

```
<expr> ::= <term> | <expr> '+' <term>
```

has two alternatives; the first is simple, the second is compound. Let us denote this compound argument by "sum" and its components (<expr> and <term>, leaving the '+' aside because it is a terminal) by "addend" and "augend" respectively. We may now construct a syntactic definition as follows:

$$\mathrm{isexpr}(s) = \mathrm{isterm}(s) \lor (\mathrm{issum}(s) \land \mathrm{isexpr}(\mathrm{addend}(s)) \land \mathrm{isterm}(\mathrm{augend}(s)))$$

$$(x = \mathrm{mksum}(u, v) \land \mathrm{isexpr}(u) \land \mathrm{isterm}(v)) \Rightarrow (\mathrm{issum}(x) \land u = \mathrm{addend}(x) \land v = \mathrm{augend}(x))$$

$$\mathrm{mksum}(u, v) = \mathrm{concatenate}(u, \mathrm{concatenate}('+', v))$$

We have purposely deviated slightly from McCarthy's definitional style, particularly with respect to the third equation above. McCarthy considers a syntactic definition to be abstract if it is independent of the choice and placement of terminal symbols and of the order of appearance of nonterminals; this third equation would, therefore, normally be omitted. Such an equation is, however, necessary to complete the definition in any concrete case. By replacing this equation by

```
mksum(u, v) = concatenate('(PLUS ', concatenate(u,
     concatenate(' ', concatenate(v, ')'))))
```

we obtain a definition of LISP-style sums; whereas replacing it by

```
mksum(u, v) = concatenate('ADD ', concatenate(u,
     concatenate(' TO ', v)))
```

gives a definition of COBOL-style sums. Any results obtained using the abstract syntax only, such as the compiler correctness proofs of [Kaplan 67] and [Painter 67], are immediately applicable to a whole family of languages obtainable by different concrete choices. A more completely worked-out example of concrete realizations is given in [McCarthy 66].

In general, a definition of this form will employ two types of "is" functions, one for each defined nonterminal (isexpr here) and one for each named alternative (here issum). The function isterm presumably corresponds to some other BNF rule starting with <term> ::= . "Make" functions will correspond to each named alternative, while component functions correspond to its components. It must be pointed out that in much work of this kind a very restricted definition of the term "function" is employed, according to which a predicate like isexpr is not a function and must always be called a predicate.

There is also a provision in the theory for semantic functions, and these can be made to correspond directly to semantic attributes. Each function has one more argument than the corresponding attribute, namely the string with which it is associated. Thus if the value of a term $\underline{t}$ is $f_t(\xi)$, given the current state vector ξ, then we may construct a function value(t, ξ) to give this value. Likewise if $f_\pi(\xi)$ is the new state vector after the ALGOL program π has been executed from the initial state vector ξ, we may construct a function algol(π, ξ) to give this new state vector. (Both these examples are taken from [McCarthy 63].) The equations which define semantic functions in this style may now be derived directly from the semantic attribute equations in both the synthesized and the inherited case. More general equations, corresponding to neither of these cases, may also be derived, such as those in the second concrete realization of Micro-ALGOL given in [McCarthy 66].

4.3 The Vienna Method

An extension of the concepts of abstract syntax and semantics, which has been used to describe PL-I ([Lucas and Walk 69], [Wegner 72]), ALGOL [Lauer 68], and, in a simplified form, BASIC [Lee 72], is also immediately adaptable to treatment in analogy with Knuth's work.

The Vienna definition language is built upon methods for building and analyzing trees. We shall be directly concerned with two types of tree; the first of these corresponds to the derivation tree of a program or other nonterminal in a context-free language, while the second corresponds to a state vector under the special conditions that variables may be arrays, structures, and the like.

A tree is a directed graph, and hence we may speak of its nodes and links. A composite object, in the Vienna definition language, is a tree whose links are all labeled, and whose terminal nodes are also labeled. The node labels are called elementary objects, and the labels on the links are called selectors. When a composite object represents a derivation tree, the elementary objects are the terminal symbols, and the selectors correspond roughly to the component names as discussed in the previous section. A syntactic equation corresponding to a BNF rule with several nonterminals on the right, but without alternatives, involves the names of the selectors explicitly. Thus

is-assign-st = (<s-left-part:is-var>,<s-right-part:is-expr>)

(taken from [Wegner 72]) corresponds to a class of partial subtrees of derivation trees of which the following is typical:

$$\begin{tikzcd}[column sep=small] & Z{=}A{+}B \arrow[dl, "\text{left-part}"'] \arrow[dr, "\text{right-part}"] & \\ Z & & A{+}B \end{tikzcd}$$

Here it is assumed that is-assign-st('Z=A+B'), is-var('Z'), and is-expr('A+B') are all true -- that is, Z is a variable, A+B is an expression, and therefore Z=A+B is an assignment statement -- and that left-part('Z=A+B') = 'Z' and right-part('Z=A+B') = 'A+B'. None of our three nodes is terminal, but we have labeled each node with the string corresponding to the subtree whose root is that node, although in practice this would ultimately become too cumbersome. A colon in such a rule associates a selector on the left with either an object (composite or elementary) on the right, or, metalinguistically, a predicate satisfied by such objects. Note that these rules are abstract in the sense of the preceding section; they do not depend on particular choices relating to terminal symbols.

Let us now restrict BNF in such a way that concatenation and alternation never occur in the same rule. This can always be done by introducing extra nonterminals; thus the example

```
<expr> ::= <term> | <expr> '+' <term>
```

of the previous section would be replaced by something like

```
<expr> ::= <term> | <sum>
<sum> ::= <expr> '+' <term>
```

A rule of the first kind above is now replaceable directly by an abstract syntactic equation involving predicates alone; thus, here, we would write

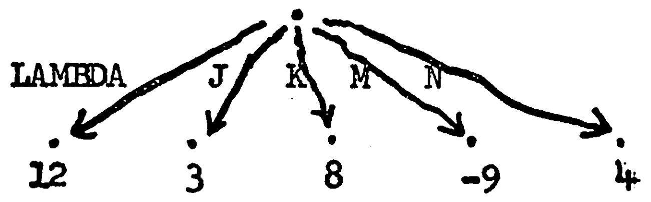
$$\text{isexpr} = \text{isterm} \vee \text{issum}$$

whereas a rule of the second kind, involving no alternatives, is replaceable by a syntactic equation involving selectors as illustrated above for assignment statements.

In formulating the semantic rules in the Vienna method as applied to a language such as PL/I, it is helpful to consider state vectors as composite objects. For example, the state vector

S = {LAMBDA = 12, J = 3, K = 8, M = -9, N = 4}

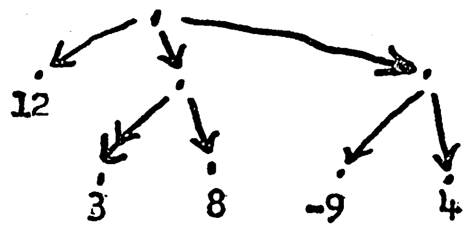
may be represented as the following composite object:



The form of this as a tree, of course, is quite special. In PL/I, however, arbitrary (disjoint) sets of variables may be grouped to form structures. Thus we may form a structure called I out of J and K and a structure called L out of M and N by writing

DECLARE 1 I, 2 J, 2 K, 1 L, 2 M, 2 N;

We may then use assignment statements such as I = I + L, which is equivalent to J = J + M followed by K = K + N. The state vector above would have the following tree form under these conditions:



Thus finite trees of arbitrary form can represent PL/I state vectors. (Assignments such as I = I + L involving arrays are also possible in PL/I, as in APL; the meaning of such assignments is much the same as with structures, and the corresponding state vectors have similar form as composite objects. Arrays without such assignments, of course, are also permitted in most other higher-level languages.)

Semantic functions may now be constructed in much the same way as described in the preceding section. Thus the following might be given as a description of sums without type conversion or side effects, where ξ represents a state vector represented as above as a composite object:

is-sum = (<s-addend:is-expr>,<s-augend:is-term>)

value(sum, ξ) = plus(value(addend.sum,ξ), value(augend.sum,ξ))

The first of these rules is syntactic, the second semantic. The period in "addend.sum" and "augend.sum" gives the result of selection by means of the selector on the left applied to the object on the right; thus "addend.sum" is the addend of _sum_, which is presumably a composite object, one of whose selectors is _addend_. If _name_ stands for a variable name, then value(name, ξ) = name.ξ. Assignments may be handled by an operator μ, defined in the Vienna language, which is somewhat analogous to McCarthy's _a_; the value of μ(_t_, <_s_:_x_>) is the composite object _t_ as modified by changing _s.t_ to be _x_. For the definition of assignment statements given above, we might write

effect(assign-st, ξ) = μ(ξ, <left-part.assign-st:value(right-part.assign-st, ξ)>)

again without type conversion or side effects.

The Vienna definition language includes a concept of the _state_ of a computation in PL/I. This includes both program and data components, as well as a wide variety of other possible components. The successive states of a computation in PL/I, according to the Vienna definition, correspond to the successive states of the computer which executes it, whereas in our own definitional style this is true only in the absence of subroutines; a subroutine corresponds to a separate computation, and the next state after a subroutine call is the state upon return from the subroutine.

4.4 Translation-Oriented Definitions

Any method of language definition must take account, in some way, of the fact that the effect of an arbitrary program cannot be determined recursively. We have done this by introducing a single semicomputable function by which the effect of a program is derived from its execution function. There is another style of definition, however, in which a program in a language is defined by means of a translation from that language into some extremely elementary language. The definition of the translation may then be entirely recursive, although the definition of the elementary language involved must be left as axiomatic. The best-known examples of elementary languages of this kind are ISWIM [Landin 66a], which has been used to describe ALGOL [Burstall 67], and the elementary notation for algorithms which is used to describe the language EULER [Wirth and Weber 66].

Of the two, Wirth and Weber's notation looks much more like a "normal" programming language. It has labels, assignment statements, simple and conditional branching statements, unary and binary operators, simple and subscripted variables, lists, and built-in functions. The value of a variable may be a reference, or pointer, to another variable, and indirect assignments to the variable v may appear, which assign new values to the variable to which v currently points. The definition of EULER in terms of this notation is formally and rigorously stated. No attempt is here made to reduce the elementary notation to simplest possible form, in the logical sense; it is quite clear, for example, that one type of statement, incorporating assignment and conditional transfer, would have sufficed.

The ISWIM notation, on the other hand, bears more of a resemblance to pure LISP; it is described in [Burstall 67] as "a more intelligible version of the lambda-notation." It has arithmetic and logical operators, conditional expressions, recursive and non-recursive definitions, and built-in functions including an output operator, but it has no labels and no branching. Any program constructed using assignment and conditional transfer may be effectively reduced to this form [McCarthy 60], so that ISWIM may be used to describe roughly the same class of languages as Wirth and Weber's notation can. The definition of ALGOL in terms of this notation is much more informal than that of EULER; on the other hand, ISWIM is the "cleaner" of the two notations, in that it would be harder, in ISWIM, to consolidate features to produce an even simpler language.

Semantic attributes may be used to describe the operation of a translator. With each nonterminal in the source language above the primary expression level we associate, as an attribute, some string in the destination language, or some function whose values are such strings. For example, we may associate with an arithmetic term a function which takes as argument a set of temporary cell names, and whose value is the code in the destination language needed to calculate the value of the term, using those names. (If no temporary cells are needed for that term, the function has a constant string value.) Two terms, or an expression and a term, are now combined to form an expression using an adding operator, and the strings or functions which are their attributes are combined into a single string or function which is an attribute of the expression. When an expression is used in an assignment, assuming that the temporary cell names can now be specified explicitly, the value of the associated function is taken, given these specific names as arguments. A translator for SIMULA, using destination strings as attributes, is described in [Wilner 71].

Translation-oriented language definitions are sometimes called "compiler-oriented," in contrast to "interpreter-oriented" definitions such as that of section 4.1. This causes confusion because the relative advantages of these methods have nothing to do with the relative advantages of interpreters and compilers. A compiler has two phases: translation and execution. In a translation-oriented method such as that of Wirth and Weber, the translation is formally defined, but the execution is either left axiomatic or defined informally. Our own methods are execution-oriented; they specify what is to be done next. They are linked to the operation of interpreters, since a (pure) interpreter has only one phase, an execution phase. They may also, however, be used with translation-oriented methods, to correspond to the execution phase of a compiler. For example, we could define Wirth and Weber's elementary notation for algorithms, using semantic attributes; if we did this, any language rigorously defined in this way would automatically become rigorously defined with respect to both translation and execution. Such a definition, however, appears more cumbersome from the viewpoint of proving correctness. We might, for example, optimize Wirth and Weber's translator, and prove the equivalence of the new translator with the old; but if someone presented us with an EULER translator based on entirely different principles from Wirth and Weber's, it is not at all clear how the correctness of such a translator could be proved.

Concerning ISWIM, it may be argued that this is not a language at all, but merely the lambda-notation of Church in a slightly altered form. Alternatively, one may argue that, if ISWIM is a language, then so is the notation we have been using, based on the fundamental mathematical properties of sets and functions. In this view, all language definition methods may be regarded as translation-oriented, differing only in that some provide translations into set-theoretic notation, as we have done, while others translate into basic logical or algorithmic notation. In one sense, of course, this is merely a matter of taste. No matter which notation we regard as fundamental and axiomatic, there are bound to be certain undefined terms, and the well-known logical paradoxes will crop up in one form or another. It is much less clear, however, what the ISWIM notation means, precisely, than what set-theoretical notation means; this is betrayed by the very name ISWIM (an acronym for If you See What I Mean). The original ISWIM [Landin 66a] is extended by Burstall, who even then is not quite satisfied with it [Burstall 67]. Wirth and Weber's notation, similarly, is only semi-formally defined, and the definition of the notation leaves a number of small details unclear.

Still another approach to the precise definition of compilers is furnished by the <u>production language</u> of [Floyd 61] and [Evans 64], used in the FSL compiler-compiler [Feldman 66] and excellently summarized in [Feldman and Gries 68] and elsewhere. Rather than defining a single compiler mathematically, production language allows us to give strict definitions of any of a whole class of compilers. It is essentially a special-purpose, compiler-writing-oriented language, which cuts compilers down to manageable size. It is itself defined informally, but it may easily be defined using state vector functions and semantic attributes. Using such a definition, any language with a compiler written in production language is automatically rigorously defined in roughly the same sense that EULER is.

5 PROOFS OF ASSERTIONS ABOUT PROGRAMS

We shall now apply the logical framework which we have developed, and give techniques for proving various assertions about programs. Foremost among these are correctness (proving that a program produces given results under given conditions); termination (proving that a program never goes on indefinitely, except possibly under certain given conditions); and equivalence (proving that two programs produce the same results under the same conditions).

The problem of constructing an algorithm which will always solve any one of these problems is unsolvable for Turing machines [Davis 58]; and, since the functions computed by Turing machines are exactly the functions computed by Kaluzhnin graph schemes ([Peter 58], [Ershov 64]), no general algorithms may be constructed in this case either. Special classes of algorithms have been constructed for which these problems are solvable; for example, if we require all transfer instructions to be in the forward direction, and allow step-until-type loops and non-recursive subroutines, subject to certain obvious restrictions, we obtain a class of programs which always terminate and the effects of which are computable. (For further examples, see [Constable and Borodin 72].) Alternatively, a number of authors have reduced the general case of our problems to equivalent problems in logic, which, on the surface at least, appear as hard or harder to solve (see, for example, [Manna 69] on correctness, [Engeler 67] on termination, or [Milner 70b] on equivalence).

We shall be mainly concerned with the case in which some hint has been given as to why a program is correct, or why it terminates, or why two programs are equivalent. Our problem will then be to find types of hints which are applicable in a wide variety of programming situations, and which may always be cast in rigorous form.

5.1 Correctness

Debugging, as it is currently practiced, is a perfect example of what happens when a body of mathematical techniques is developed in the absence of a rigorous framework. Programs which are debugged quite thoroughly still develop strange errors, often after the programmers have departed. This state of affairs has been deplored in [McCarthy 65], [Naur 66], and elsewhere; although it is probably unrealistic to hope to eliminate debugging entirely, since even the mathematician "debugs" his new theorems before subjecting them to formal proof and publication.

In order to prove that a program actually does what it is supposed to, we can determine, for each of its commands, the relations which must hold among the variables just prior to the execution of that command at any time during the program. We may then attempt to prove for each command that if it is executed at a time when its associated relations hold, then the relations associated with the next command (in execution order) will always hold. If this can be done, it follows that if we start the program with the starting relations holding (and in some programs these will be null), then, no matter how the program twists and turns, every time a command is executed, the relations associated with the next command will hold. Thus, if we ever get to the end (which we may not), the relations associated with the last command will hold. This basic method of proof was first published in [Floyd 67] and independently, although much less rigorously, in [Naur 66]. Floyd credits the idea to Perlis and Gorn, but, as Knuth points out ([Knuth 68b], which also contains an excellent worked example) it can be traced back to von Neumann [Goldstine and von Neumann 47]. The relations among the variables, which are the key to the method, are called <u>assertions</u> by von Neumann, and we shall continue to refer to this method as the assertion method.

Floyd uses a directed graph model and attaches assertions to the edges (= links) of the graph; an edge with an associated assertion is said to be tagged. The assertion method takes a finite number of steps even if not every edge is tagged; as Floyd remarks, all that is necessary is to tag a set of edges which is such that every closed loop in the graph passes through at least one of them (in circuit theory, this is called a feedback arc set [Younger 63]) and which contains each entering (and exit) edge. Under these conditions, we determine the control paths [Cooper 69] between two tagged edges with no tagged edges in between, and prove for each such path that if it is started with its initial assertion true, and if it is actually followed to the end, its final assertion will be true when it ends. The total number of control paths is finite if and only if they are chosen relative to a feedback arc set.

The proofs of the verification conditions ("If P is true and we do command C then Q will be true") depend on the semantics of commands. Floyd requires that any reasonable semantic definition of a command be capable of producing such proofs, and he gives examples of verification conditions involving ALGOL commands and boxes in a flowchart language, but he does not show how these may be derived, and, in fact, is inclined to treat them as axiomatic (see also [Hoare 69], [de Bakker 69], [Kaplan 69a]). Using semantic attributes, however, we may derive verification conditions immediately. In fact, P and Q, being conditions, will correspond to state vector functions f_P and f_Q whose values are true and false, and thus also (see section 2.4) to sets s_P and s_Q of state vectors for which f_P and f_Q respectively yield the value true. Likewise, the command C will have a semantic attribute which is a state vector function e_C giving its effect. The condition that "if P is true and we do C then Q will be true" is then precisely the condition that if $S \in s_P$ then $e_C(S) \in s_Q$, for all state vectors S.

This analysis may be generalized. If C has several exits, and if Q tags an edge which is followed whenever the condition B is true (before C is executed), then B corresponds as before to a set s_B and the condition above becomes $S \in s_P \cap s_B \Rightarrow e_C(S) \in s_Q$. If B must be true after C is executed, rather than before, the condition becomes $(S \in s_P$ and $e_C(S) \in s_B) \Rightarrow e_C(S) \in s_Q$. If $C_1, \ldots, C_n$ are commands in a control path, with respective effects $e_1, \ldots, e_n$, we define $e_1' = e_1$, $e_i'(S) = e_i(e_{i-1}'(S))$ for $1 < i \leq n$, so that e_n' is the composition of all the e_i; if each C_i has only one exit, and P and Q are the initial and final conditions respectively, then $S \in s_P$ must imply $e_n'(S) \in s_Q$. If the control path is followed, after the command C_i, only if the condition B_i (corresponding to the set s_i) is true, then $e_n'(S) \in s_Q$ must be true only if $S \in s_P$ and $e_i'(S) \in s_i$ for each i, $1 \leq i \leq n$. Note that the conditions $e_C(S) \in s_Q$ and $e_n'(S) \in s_Q$ imply, in particular, that these functions are defined when applied to S; this must be assured by the form of P. For example, if C involves subscripted variables, P must imply that all subscripts are in range. If the set of all $e_C(S)$ for $S \in s_P$ is of the form s_R for some condition R, then R is the strongest verifiable consequent of P after the command C, and we write $R = T_e(P)$ [Floyd 67].

If the state vector S has domain M as a content function, then any function f(S) may be regarded as a function of two variables, $f(S(v), S|M-\{v\})$, for any variable $v \in M$. If f is the function f_Q corresponding to the predicate Q, and P is derived from Q by substituting the expression e for v wherever v occurs, then $f_P(S(v), S|M-\{v\})$ will be equal to $f_Q(f_e(S), S|M-\{v\})$, where f_e is the state vector function of e. In this case, if P is true and we do the command $\{v:=e\}$ then Q will be true. In fact, if S is any state vector and we do $\{v:=e\}$, producing S', where $S'(v) = f_e(S)$ and $S'|M-\{v\} = S|M-\{v\}$, then $f_Q(S') = f_Q(S'(v), S'|M-\{v\}) = f_Q(f_e(S), S|M-\{v\}) = f_P(S(v), S|M-\{v\}) = f_P(S)$. Thus P is true before execution of this command if and only if Q is true afterward; in particular, Q is the strongest verifiable consequent of P. This means that if we know Q we can easily derive an assertion P for which the verification condition through $\{v:=e\}$ will hold, merely by performing the substitution described above; and we may continue in this way in the backward direction along any control path. This is the so-called _back substitution method_, which appears as Axiom D0 in [Hoare 69]. We have defined the method only for constant **v**; generalization to subscripted left sides does not always work, as noted in [King 69].

When P = Q, the verification condition through $\{v:=e\}$ always holds provided that P does not involve **v** or any variable whose value might be changed by any side effect of e. The set of all such variables, for a given side effect or, more generally, for the effect of a given command, is called its _region of influence_ in [Elgot and Robinson 64] and its _output region_ in [Maurer 66]. The set of all variables _used_ by the effect of a command is called the _input region_ in [Maurer 66]; this concept can also be generalized to provide input regions for arbitrary state vector functions. [Wagner 68] combines the input and output regions into a _dependency-activity domain_, and [Igarashi 71] refers to them as R[A] and L[A] respectively, for the command A; this suggests the right and left sides of an assignment, although, for a general assignment with side effects, both sides of the assignment symbol may affect both regions.

It must be noted that Floyd's verification conditions are seemingly more general than those discussed here; his P and Q are not predicates but rather _vectors_ of predicates, corresponding to several entrances to a command and several exits from it. In practice, however, a command can always be taken to have a single entrance, unless it corresponds to a subroutine with several entry points; and in this case it will have different verification conditions corresponding to each entry point. We have chosen to consider the exits from a command one at a time, each one accompanied by a traversal condition, rather than a vector of exit predicates.

In proving the correctness of a program, the probability of error in the proof may be as high or higher than the probability of error in the program itself. Therefore, computer-aided methods are indicated. Computer aids to program verification are described in [King 69] and [Good 70]; further such programs, some of them unfinished, are referenced in [London 72b]. Some of these are to provide compilation as well as verification; in fact, an idea often mentioned (see, for example, [Dijkstra 68]) is to construct and verify programs simultaneously and interactively. A verifier may itself contain errors, of course; strictly speaking, no verifier-produced proof is to be trusted unless and until the verifier is itself verified in some manner, as well as the compiler (or assembler) for the language in which the verified program is written.

Most verifiers use the assertion method, although there are other methods of proving programs correct ([McCarthy 61], [Manna 69], [Burstall 69], [London 70b]) which may be computerized. A verifier may be written to act upon any language in which programs are written. Using the assertion method, we may even verify assembly language programs, so long as we do not consider instruction modification as a dynamic alteration of the flow diagram, as in [Goldstine and von Neumann 47], but rather regard an instruction word whose contents duringa program may be any of the commands c_1, ..., c_k as a k-way branch with the meaning "If c_i is stored here now, then do c_i."

5.2 Termination

When a program has finite computation sequences -- that is, when its execution does not run on indefinitely -- it is said to terminate. As we have just seen, the application of Floyd's assertion method to a program leaves open the possibility that the program may not terminate; thus in [King 69] a program which never terminates is always "correct." [Manna 69] calls a program correct only if it terminates; a program is partially correct if, for each input vector, it either terminates correctly or does not terminate at all.

Any program always terminates unless it contains loops; a program contains loops if and only if its commands cannot be arranged in sequence such that all transfers are in the forward direction. The best-known general treatment of termination is given in [Manna 68] and [Manna 70], improving an earlier treatment in [Floyd 67]. We shall present a simplified version of Manna's methods, and show that both the general and the simplified method are sufficient to prove the termination of any terminating program.

In general, what causes a loop to terminate is usually the behavior of the loop index, or controlled variable, or its equivalent. This is an expression which must increase (or decrease) every time around the loop, and which is bounded within the loop in some way. It is not enough to have a test within the loop which exits if an increasing loop index exceeds some bound, as illustrated by the ALGOL loop

```
a: j:=j+1; i:=0; if i>0 then go to b; i:=j; go to a; b:
```

where the integers i and j are initialized to zero. Here i assumes the successive values 0, 1, 2, ..., every time the label a is reached, but the loop is endless. One could require that the index never decrease during the loop, but this requirement is not necessary if we make use of information gained in proving correctness when proving termination. Specifically, some edge in each loop must be tagged (see the preceding section), and, if we measure increase of an expression around the loop from the tagged edge, it is enough (as far as that loop is concerned) that the assertion which tags that edge imply that the expression is bounded whenever that point is reached. We shall call an integer expression subject to these two conditions a controlled expression.

Controlled expressions are usually not difficult to find, provided we know intuitively why our given loop terminates. If the loop index $\underline{i}$ is decreasing, then $-\underline{i}$ is probably a controlled expression. If the loop index $\underline{i}$ is real (or, in PL/I, of scale FLOAT) then it probably must be incremented within its loop by at least some minimum amount ϵ each time (otherwise it might converge monotonically to a value less than its bound, and thus loop endlessly); the integer part of $\underline{i}/\epsilon$ is then probably a controlled expression. The bound must be constant, but this normally poses no problem; if the index $\underline{i}$ is bounded by the variable $\underline{n}$, the controlled expression is $\underline{i}-\underline{n}$. For an algorithm involving two integers which increase alternately (or, as in Euclid's algorithm, decrease alternately), the controlled expression is their sum, or its negative. For an interchange sort of a table A, the total number of pairs of integers (i, j) such that i < j and A[i] < A[j] is an expression whose value increases each time such a pair is placed in order. For the calculation of the limit of a convergent sequence $a_1, a_2, \ldots,$ in which a_i is compared with a_{i-1} each time, using a tolerance ϵ, the controlled expression is i; this increases by 1 each time, and is bounded by the constant N given by the convergence criterion (for every ϵ, including this particular ϵ, there exists N such that $|a_N - a_{N-1}| < \epsilon$).

In state vector terms, an integer expression corresponds to a state vector function f(S) whose values are integers. Defining the C_i, e_i, and e_i' for a control path as in the preceding section, the condition for f to be increasing along this path is $f(e_n'(S)) > f(S)$ for each state vector S; this holds whether or not the path is a loop. The boundedness condition, of course, is $f(S) \leq k$ for some constant k. Any integer-valued state vector function satisfying these conditions may be considered as a controlled expression, whether or not it is the state vector function of an expression syntactically producible in the given programming language. Thus, for the limit program of the preceding paragraph, if the variable a always contains the current a_i, then f(S) is that i such that $S(a) = a_i$. Likewise, for the sequential processing of a list or file, f(S) is that i such that the i-th element of the list or file is currently being processed; this increases by 1 each time, and is bounded by the known finite length of the file or the assumed finite length of the list.

The termination method of [Floyd 67] essentially involves finding a single controlled expression for a program, although it is to be decreasing and its values may lie in any well-ordered set. Using state vectors, it is easy to prove that in fact any terminating program has such an expression. Let T be a state vector with a program-counter component, and suppose that the computation sequence beginning with T terminates after n_T steps. Then we simply define $f(T) = n_T$, or, to produce an increasing controlled expression, $f(T) = -n_T$. Such an expression must, in general, involve a program-counter component, and, in addition, might not correspond to a recursive function.

The cycle method (Theorem 1) and the elementary path method (Corollary 1) of [Manna 70] involve a separate expression for each tagged edge. In the cycle method, each such expression must be increasing around every closed loop in the program that begins and ends at its corresponding edge; in the elementary path method, the value of the expression associated with the end of any elementary path (= control path) must be greater than the value of the expression associated with its beginning. The expressions are permitted to have values in well-ordered sets, as in Floyd's method; in the cycle method, the values of different expressions may lie in different well-ordered sets.

The advantage of Manna's method over Floyd's is that the well-ordered sets are simpler to construct, as illustrated in [Manna 70] by three examples. None of these involves the cycle method, which has the disadvantage, for large programs, that the total number of cycles in a program increases faster than the total number of commands. It might seem that the elementary path method is slightly unnatural, in that it does not deal with the values of a _single_ expression along a path unless that path is a loop. However, it is not enough to find such an expression for every path, as shown by the ALGOL loop

```
a: i:=i+1; j:=j-1; b: i:=i-1; j:=j+1; go to a;
```

which never terminates, even though i and j are bounded throughout, and i increases along the path from a to b while j increases along the path from b to a.

A rather complex set of conditions involving controlled expressions in our sense appears to give easy termination proofs for a wide class of large programs. Let us associate with each _control path_ a _sequence_ of expressions. Normally, each such path is determined to be inside k nested loops $L_1, \ldots, L_k$ whose controlled expressions are respectively $e_1, \ldots, e_k$, and this is the sequence which we associate with that path. No control path need be considered in this analysis unless it is inside at least one loop; that is, there is a way that the program can proceed from its end back to its beginning, if these are not the same. Define a path associated with $e_1, \ldots, e_k$ to be outer to any path associated with $e_1, \ldots, e_k, e_{k+1}, \ldots, e_{k+j}$. Define a path associated with $e_1, \ldots, e_k$ to be primary if e_k is a controlled expression and secondary if any closed loop containing it must contain some path outer to it (where possibly $j = 0$ in the definition of "outer"). Then a program terminates if every path is either primary or secondary and no e_j in the sequence associated with any path may decrease along that path.

5.3 Equivalence

In order to prove that two programs are equivalent, we must first define what we mean by equivalence. It would seem that two programs which do the same thing have, in our terminology, the same effect; but this definition is too restrictive for many practical purposes. The temporary variables used by two programs may have different values upon termination, and yet we may wish to regard them as equivalent since they produce the same effect upon the input and output variables. Igarashi [de Bakker 69] takes $A \underset{X}{\sim} B$, for the straight-line programs A and B, to mean that A and B produce the same effect on the variables belonging to the set X. A more general notion, involving separate sets of input and output variables, is introduced in [Manna 69] and independently (and much more informally) in [Orgass 70], where it is called relative equivalence. We may also define equivalence with respect to what is printed out, as in [Kaplan 69b], although, in our terminology (see section 1.2), this may be regarded as a special case of the above, since printout sheets and output tapes are regarded as variables with values.

Another kind of restricted equivalence resembles the notion of correctness with respect to an initial assertion. If two programs compute the square root of X in different ways, we would like to regard them as equivalent even though one or both of them may be "unsafe," that is, may give unpredictable results when the given value of X is negative. In [Manna 69], equivalence is defined with respect to an input predicate; two programs are equivalent if they have the same effect on any state vector for which the input predicate is true. To obtain ordinary equivalence, one uses the input predicate which is always true.

Equivalence of program schemes is normally taken to mean equivalence in all, or most, interpretations. For program schemes as defined by Yanov, this implies (see the discussion of traces of computations in section 3.3) that two equivalent schemes must have the same (finite) sequences of operations, as well as the same effects. (For programs, this last condition is called strict equivalence in [Orgass 70], as contrasted with the ordinary notion of operational equivalence.) One example of equivalence in Yanov's sense is furnished by the process of removing statements from a flowchart which cannot be reached by a path in the flowchart (as a graph) from any entrance node. Graphs containing such statements are called ineligible in [Karp 60], where it is noted that this condition may be detected in a finite number of steps; indeed, actual compilers often have error messages reading "NO WAY TO REACH THIS STATEMENT" or something similar, although they normally do not make full use of graph theory to mark all graphically unreachable statements in this way. Karp also treats the removal of statements from which a terminal node cannot be reached; such statements may appear in endless computations, but Yanov's notion of equivalence, as noted above, implies only that the finite operator sequences must agree.

Equivalence of the program schemes in [Luckham, Park and Paterson 70] in all interpretations is called strong equivalence. (Weak equivalence is a sort of "partial equivalence"; two program schemes are weakly equivalent if they give the same results under conditions when both are defined.) However, since a change of interpretation of such a scheme involves only right sides, and not left sides, of assignments in the scheme, strong equivalence is operational and not strict (in Orgass' sense, above). Transformations of flowcharts which produce transformed programs strongly equivalent (but not equivalent in Yanov's sense) to the original include the addition of new statements which do nothing, as well as the combining of two or more nodes into a single node. In these cases the effect of the transformed program is the same as that of the original, but the corresponding "finite sequences of operators usually do not even have the same length. Five such transformations are examined in [Cooper 67], by the use of which any node in a graph may be eliminated unless it links to itself, and thus any flowchart may be reduced to a form in which every node links to itself. If there are no other links, the flowchart is called proper cycle free (PCF); termination of PCF flowcharts is particularly easy to check, since it is the same as termination of each node-loop separately. Most flowcharts, however, are not reducible to PCF form, and Cooper gives a necessary and sufficient condition for this; a related result is found in [Bohm and Jacopini 66]. It is also possible to use Cooper's first transformation to show the equivalence of the two-branch and multi-branch graph models which we discussed in section 3.1.

For Yanov's notion of equivalence, there is a decision procedure involving reduction to canonical form. Strong equivalence, however, is undecidable, although three powerful tests for detecting it are given in [Kaplan 69a]. A formal theory of flowcharts is presented in [Kaplan 69b], in which certain basic equivalences between flowcharts are taken as axiomatic. For each of these, the transformed scheme is strongly equivalent to the original scheme. Kaplan's equivalences are specified explicitly in terms of functions and predicates, whereas Cooper gives only two examples in this style, leaving the others stated in graphical terms. Using his basic equivalences, Kaplan is able to prove further assertions about programs, using McCarthy's technique of recursion induction [McCarthy 61].

The treatment of flowchart equivalences as axiomatic leaves open the question as to which reasonable subsets of real programming languages satisfy the axioms and which do not. Strong equivalence, as we have seen, is defined for program schemes involving assignments to single fixed registers and functions and predicates with no side effects. The use of subscripted left sides, or of assignments to more than one register at once, does not affect any of Cooper's or Kaplan's basic equivalences; on the other hand, the form of both of Cooper's explicit examples and of three of Kaplan's six axioms must be altered if we allow predicates to have side effects. Even the transformations suggested by Karp do not preserve strong equivalence unless we are careful to replace endless loops by other endless loops, and not by error exits or anything else that would substitute a terminating computation for a non-terminating one.

A general method, which closely resembles that of Floyd, may be used to prove that a given programming language satisfies a given equivalence axiom. Let P and P' be two programs which act upon the (state vector) domain D, let K be a subset of the statements of P, and let f map K into the statements of P'. Suppose that we can prove for each statement $k \in K$ that if P and P' are started at k and f(k) respectively, each with current state vector $S \in D$, then the next time P reaches a statement $k' \in K$, with current state vector S', and the next time P' reaches a statement f(k") for some $k'' \in K$, with current state vector S", we will have k' = k" and S' = S". Under these conditions, for example, if k_i and $f(k_i)$ are unique entrances and k_f and $f(k_f)$ are unique exits, then P and P' are equivalent, and this statement may easily be generalized to several entrances and exits. If the set K satisfies Floyd's closed loop condition (see section 5.1) and the commands of P and P' are given by state vector functions, the conditions above may be checked in a finite number of steps. Of course, if P' is obtained by transforming a subset U of the statements of P, then f(k) = k for all $k \notin U$.

When P and P' act upon different domains D and D', the method above may be further generalized to involve a function g: D → D'. Such generalization is necessary, for example, when P' is obtained from P by transformations involving variable names. If P' uses temporary variables which P does not, then P and P' will, in general, be only relatively equivalent, involving a different kind of generalization. Axioms for relative equivalence are given in [de Bakker 69], some of which are originally due to Igarashi. Again, these are for assignments without side effects and with single variables on the left, and, for de Bakker's axioms, on the right as well; their form must be altered if these restrictions are relaxed. The same is true of Igarashi's axioms for conditional expressions, also presented by de Bakker.

Proofs of the correctness of compilers require at least the relative equivalence of a source program and the corresponding object program. Such proofs are given in [Kaplan 67], [Painter 67], and [London 72a]. In addition, if a compiler includes object code optimization, it is necessary to prove that the optimization methods used are valid in the sense that the optimized and unoptimized programs are always equivalent. Studies have been made (see, for example, [Allen 69]) of highly complex optimization methods for real programming languages, but proofs are usually not given, except for straight-line code, where the effect is always computable [Aho and Ullman 72]. Compilers for languages allowing subroutines must compile valid code for the handling of return addresses and parameters; the validity of this process is discussed, for return addresses, in [Maurer 73].

A C K N O W L E D G M E N T

This research was partially supported by National Science Foundation Grant GJ-31612.

R E F E R E N C E S

Aho and Ullman 72 — Aho, A. V., and Ullman, J. D., *Optimization of straight line programs*, SIAM J. Computing 1, 1 (1972), pp. 1-19.

Allen 69 — Allen, F. E., *Program optimization*, Annual Review in Automatic Programming 5 (1969), pp. 239-307.

Bjorner 70 — Bjorner, D., *Flowchart machines*, BIT 10, 4 (1970), pp. 415-442.

Bohm and Jacopini 66 — Bohm, C., and Jacopini, G., *Flow diagrams, Turing machines, and languages with only two formation rules*, Communications of the ACM 9, 5 (1966), pp. 366-371.

Burstall 67 — Burstall, R. M., *Semantics of assignment*, Machine Intelligence 2 (1967), pp. 3-20.

Burstall 69 — Burstall, R. M., *Proving properties of programs by structural induction*, Computer Journal 12, 1 (1969), pp. 41-48.

Burstall 70 — Burstall, R. M., *Formal description of program structure and semantics in first order logic*, Machine Intelligence 5 (1970), pp. 79-98.

Burstall 72 — Burstall, R. M., *An algebraic description of programs with assertions, verification, and simulation*, Proc. ACM Conf. on Proving Assertions about Programs (SIGPLAN Notices 7, 1; SIGACT News 14), January 1972, pp. 7-14.

Constable and Borodin 72 — Constable, R. L., and Borodin, A. B., *Subrecursive programming languages, part I: efficiency and program structure*, Journal of the ACM 19, 3 (1972), pp. 526-568.

Cooper 67 — Cooper, D. C., *Some transformations and standard forms of graphs, with applications to computer programs*, Machine Intelligence 2 (1967), pp. 21-33.

Cooper 69 — Cooper, D. C., *Program scheme equivalences and second-order logic*, Machine Intelligence 4 (1969), pp. 3-15.

Curry 63 — Curry, H. B., *Foundations of Mathematical Logic*, McGraw-Hill, New York, 1963.

Davis 58 — Davis, M., *Computability and Unsolvability*, McGraw-Hill, New York, 1958.

de Bakker 69 — de Bakker, J. W., *Semantics of programming languages*, in Advances in Information Systems Science 2, Plenum Press, New York-London, 1969, pp. 173-227.

Dijkstra 68 — Dijkstra, E. W., A constructive approach to the problem of program correctness, BIT 8, 3 (1968), pp. 174-186.

Ehlers 68 — Ehlers, H., Logic By Way Of Set Theory, Holt, Rinehart and Winston, Inc., New York, 1968.

Elgot and Robinson 64 — Elgot, C. C., and Robinson, A., Random-access stored-program machines, an approach to programming languages, Journal of the ACM 11, 4 (1964), pp. 365-399.

Elgot 68 — Elgot, C. C., Abstract algorithms and diagram closure, in Programming Languages, F. Genuys, ed., Academic Press, London-New York, 1968, pp. 1-42.

Engeler 67 — Engeler, E., Algorithmic properties of structures, Mathematical Systems Theory, 1, 3 (1967), pp. 183-195.

Ershov 60 — Ershov, A. P., Operator algorithms, I, Problems of Cybernetics 3 (1960), pp. 697-763.

Evans 64 — Evans, A., An ALGOL 60 compiler, Annual Review in Automatic Programming 4 (1964), pp. 87-124.

Feldman 66 — Feldman, J. A., A formal semantics for computer languages and its application in a compiler-compiler, Communications of the ACM 9, 1 (Jan. 1966), pp. 3-9.

Feldman and Gries 68 — Feldman, J. A., and Gries, D., Translator writing systems, Communications of the ACM 11, 2 (1968), pp. 77-113.

Floyd 61 — Floyd, R. W., A descriptive language for symbol manipulation, Journal of the ACM 8, 4 (1961), pp. 579-584.

Floyd 62 — Floyd, R. W., On the nonexistence of a phrase structure grammar for ALGOL 60, Communications of the ACM 5, 9 (1962), pp. 483-484.

Floyd 67 — Floyd, R. W., Assigning meanings to programs, Proc. Symp. Applied Math. 19 (Mathematical Aspects of Computer Science), American Mathematical Society, Providence, R. I., 1967, pp. 19-32.

Ginsburg 62 — Ginsburg, S., An Introduction to Mathematical Machine Theory, Addison-Wesley, Reading, Mass., 1962.

Goldstine and von Neumann 47 — Goldstine, H. H., and von Neumann, J., Planning and coding problems for an electronic computing instrument, reprinted in The Collected Works of John von Neumann, vol. 5, pp. 80-235, Pergamon Press, 1963.

Good 70 — Good, D. I., Toward a man-machine system for proving program correctness, Ph. D. Thesis, Computer Science Department, University of Wisconsin, 1970.

Hoare 69 — Hoare, C. A. R., An axiomatic basis for computer programming, Communications of the ACM 12, 10 (1969), pp. 576-580, 583.

Igarashi 71 — Igarashi, S., Semantics of ALGOL-like statements, in Lecture Notes in Mathematics 188, Semantics of Algorithmic Languages, Springer-Verlag, Berlin-Heidelberg-New York, 1971, pp. 117-177.

Kaluzhnin 59 — Kaluzhnin, L. A., Algorithmization of mathematical problems, Problems of Cybernetics 2 (1959), pp. 371-391.

Kaphengst 59 — Kaphengst, H., Eine abstrakte programmgesteuerte Rechenmaschine, Zeitschr. f. math. Logik und Grundlagen d. Math. 5 (1959), pp. 366-379.

Kaplan 67 — Kaplan, D. M., Correctness of a compiler for ALGOL-like programs, Artificial Intelligence Memorandum No. 48, Stanford University, 1967.

Kaplan 68 — Kaplan, D. M., Some completeness results in the mathematical theory of computation, Journal of the ACM, 15, 1 (1968), pp. 124-134.

Kaplan 69a — Kaplan, D. M., Regular expressions and the equivalence of programs, J. Computer and Systems Sci. 3, 4 (1969), pp. 361-386.

Kaplan 69b — Kaplan, D. M., Recursion induction applied to generalized flowcharts, Proc. 24th National ACM Conference (1969), pp. 491-504.

Karp 60 — Karp, R. M., A note on the application of graph theory to digital computer programming, Inf. and Control 3, 2 (1960), pp. 179-190.

King 69 — King, J. C., A program verifier, Ph. D. Thesis, Department of Computer Science, Carnegie-Mellon University, 1969.

King 71 — King, J. C., Proving programs to be correct, IEEE Transactions on Computers C20, 11 (1971), pp. 1331-1336.

Knuth 68a — Knuth, D. E., Semantics of context-free languages, Mathematical Systems Theory 2, 2 (1968), pp. 127-145.

Knuth 68b — Knuth, D. E., The Art of Computer Programming, Vol. 1: Fundamental Algorithms, Addison-Wesley, Reading, Mass., 1968.

Knuth 71a	Knuth, D. E., Semantics of context-free languages -- correction, Mathematical Systems Theory 5, 1 (1971), pp. 95-96.
Knuth 71b	Knuth, D. E., Examples of formal semantics, in Lecture Notes in Mathematics 188, Semantics of Algorithmic Languages, Springer-Verlag, Berlin-Heidelberg-New York, 1971, pp. 212-235.
Lambek 61	Lambek, J., How to program an infinite abacus, Canadian Math. Bulletin 4, 3 (1961), pp. 295-302.
Landin 64	Landin, P. J., The mechanical evaluation of expressions, Computer Journal 6, 4 (Jan. 1964), pp. 308-320.
Landin 65	Landin, P. J., A correspondence between ALGOL 60 and Church's lambda-notation, Communications of the ACM 8 (Feb.-Mar. 1965), pp. 89-101, 158-165.
Landin 66a	Landin, P. J., The next 700 programming languages, Communications of the ACM 9, 3 (1966), pp. 157-166.
Landin 66b	Landin, P. J., A formal description of ALGOL 60, in Formal Language Description Languages for Computer Programming, T. B. Steel, Jr., ed., North-Holland, Amsterdam, 1966, pp. 266-294.
Lauer 68	Lauer, P., Formal definition of ALGOL 60, Technical Report TR25.088, IBM Laboratory Vienna, 1968.
Lee 72	Lee, J. A. N., The formal definition of the BASIC language, Computer Journal 15, 1 (1972), pp. 37-41.
London 70a	London, R. L., Bibliography on proving the correctness of computer programs, Machine Intelligence 5 (1970), pp. 569-580.
London 70b	London, R. L., Proving programs correct: some techniques and examples, BIT 10, 2 (1970), pp. 168-182.
London 72a	London, R. L., Correctness of a compiler for a LISP subset, Proc. ACM Conf. on Proving Assertions about Programs (SIGPLAN Notices 7, 1; SIGACT News 14), January 1972, pp. 121-127.
London 72b	London, R. L., The current state of proving programs correct, Proc. 27th National ACM Conference, 1972.
Lucas and Walk 69	Lucas, P., and Walk, K., On the formal description of PL/I, Annual Review in Automatic Programming 6, 3 (1969).
Luckham, Park and Paterson 70	Luckham, D. C., Park, D. M. R., and Paterson, M. S., On formalized computer programs, J. Computer and Systems Sci. 4, 3 (1970), pp. 220-249.

McCarthy 60 McCarthy, J., Recursive functions of symbolic expressions and their computation by machine, Part I, Communications of the ACM 3, 4 (1960), pp. 184-195.

McCarthy 61 McCarthy, J., A basis for a mathematical theory of computation, Proc. Western Joint Computer Conf. (1961), pp. 225-238.

McCarthy 63 McCarthy, J., Towards a mathematical science of computation, Information Processing 1962, Proc. of IFIP Congress 62 (Munich), North-Holland, Amsterdam, 1963, pp. 21-28.

McCarthy 65 McCarthy, J., Problems in the theory of computation, Proc. of IFIP Congress 65 (Washington), Spartan Books, 1965, pp. 219-222.

McCarthy 66 McCarthy, J., A formal description of a subset of ALGOL, in Formal Language Description Languages for Computer Programming, T. B. Steel, Jr., ed., North-Holland, Amsterdam, 1966, pp. 1-12.

McCarthy and Painter 67 McCarthy, J., and Painter, J. A., Correctness of a compiler for arithmetic expressions, Proc. Symp. Applied Math. 19 (Mathematical Aspects of Computer Science), American Mathematical Society, Providence, R. I., 1967, pp. 33-41.

Manna 68 Manna, Z., Termination of algorithms, Ph. D. Thesis, Department of Computer Science, Carnegie-Mellon University, 1968.

Manna 69 Manna, Z., The correctness of programs, J. Computer and Systems Sci. 3, 2 (1969), pp. 119-127.

Manna 70 Manna, Z., Termination of programs represented as interpreted graphs, Proc. 1970 Spring Joint Computer Conf., pp. 83-89.

Martin and Estrin 67 Martin, D. F., and Estrin, G., Models of computations and systems -- evaluation of vertex probabilities in graph models of computations, Journal of the ACM 14, 2 (1967), pp. 281-299.

Maurer 66 Maurer, W. D., A theory of computer instructions, Journal of the ACM, 13, 2 (1966), pp. 226-235.

Maurer 72 — Maurer, W. D., A semantic extension of BNF, International Journal of Computer Mathematics, 1972.

Maurer 73 — Maurer, W. D., The validity of return address schemes, Information Sciences 6, 1 (1973).

Mazurkiewicz 71 — Mazurkiewicz, A. W., Proving algorithms by tail functions, Inf. and Control 18, 3 (1971), pp. 220-226.

Melzak 61 — Melzak, Z. A., An informal arithmetical approach to computability and computation, Canadian Math. Bulletin 4, 3 (1961), pp. 279-293.

Milner 70a — Milner, R., Program schemes and recursive function theory, Machine Intelligence 5 (1970), pp. 39-58.

Milner 70b — Milner, R., Equivalences on program schemes, J. Computer and Systems Sci. 4, 3 (1970), pp. 205-219.

Milner 72 — Milner, R., Implementation and applications of Scott's logic for computable functions, Proc. ACM Conf. on Proving Assertions about Programs (SIGPLAN Notices 7, 1; SIGACT News 14), January 1972, pp. 1-6.

Minsky 61 — Minsky, M. L., Recursive unsolvability of Post's problem of "tag" and other topics in the theory of Turing machines, Annals of Math. 74 (1961), pp. 437-455.

Narasimhan 67 — Narasimhan, R., Programming languages and computers: a unified metatheory, Advances in Computers 8 (1967), pp. 189-245.

Naur et al. 60 — Naur, P., et al., Report on the algorithmic language ALGOL 60, Communications of the ACM 3, 5 (1960), pp. 299-314.

Naur 66 — Naur, P., Proof of algorithms by general snapshots, BIT 6, 4 (1966), pp. 310-316.

Orgass and Fitch 69a — Orgass, R. J., and Fitch, F. B., A theory of computing machines, Studium Generale 22 (1969), pp. 83-104.

Orgass and Fitch 69b — Orgass, R. J., and Fitch, F. B., A theory of programming languages, Studium Generale 22 (1969), pp. 113-136.

Orgass 70 — Orgass, R. J., Some results concerning proofs of statements about programs, J. Computer and Systems Sci. 4, 1 (1970), pp. 74-88.

Painter 67 Painter, J. A., Semantic correctness of a compiler for an ALGOL-like language, Ph. D. Thesis, Computer Science Department, Stanford University, 1967.

Park 68 Park, D. M. R., Some semantics for data structures, Machine Intelligence 3 (1968), pp. 351-371.

Paterson 68 Paterson, M. S., Program schemata, Machine Intelligence 3 (1968), pp. 18-31.

Peter 58 Peter, R., Graphschemata und rekursive Funktionen, Dialectica 12 (1958), pp. 373-393.

Rutledge 64 Rutledge, J. D., On Ianov's program schemata, Journal of the ACM, 11, 1 (1964), pp. 1-9.

Scott 67 Scott, D., Some definitional suggestions for automata theory, J. Computer and Systems Sci. 1, 2 (1967), pp. 187-212.

Scott 70 Scott, D., Outline of a mathematical theory of computation, Proc. 4th Annual Conf. on Information Sciences and Systems, Princeton, 1970, pp. 169-176.

Scott 71 Scott, D., The lattice of flow diagrams, in Lecture Notes in Mathematics 188, Semantics of Algorithmic Languages, Springer-Verlag, Berlin-Heidelberg-New York, 1971, pp. 311-366.

Shepherdson and Sturgis 63 Shepherdson, J. C., and Sturgis, H. E., Computability of recursive functions, Journal of the ACM 10, 2 (1963), pp. 217-255.

Skowron 71 Skowron, A., Semantic translation of programming languages, Zeitschr. f. math. Logik und Grundlagen d. Math., 17 (1971), pp. 39-46.

Strachey 66 Strachey, C., Towards a formal semantics, in Formal Language Description Languages for Computer Programming, T. B. Steel, Jr., ed., North-Holland, Amsterdam, 1966, pp. 198-216.

Suppes 57 Suppes, P., Introduction to Logic, Van Nostrand, Princeton, N. J., 1957.

van Wijngaarden et al. 69 van Wijngaarden, A., et al., Report on the algorithmic language ALGOL 68, Numerische Math. 14 (1969), pp. 79-218.

Wagner 68 Wagner, E. G., Bounded action machines: toward an abstract theory of computer structure, J. Computer and Systems Sci. 2, 1 (1968), pp. 13-75.

Wegner 70 Wegner, P., Three computer cultures: computer technology, computer mathematics, and computer science, Advances in Computers 10 (1970), pp. 7-78.

Wegner 72 Wegner, P., The Vienna definition language, Computing Surveys 4, 1 (1972), pp. 5-63.

Wilkes 68 — Wilkes, M. V., The outer and inner syntax of a programming language, Computer Journal 11, 3 (1968), pp. 260-263.

Wilner 71 — Wilner, W., Declarative semantic definition, Ph. D. Thesis, Computer Science Department, Stanford University, 1971.

Wirth and Weber 66 — Wirth, N., and Weber, H., EULER: a generalization of ALGOL, and its formal definition, Communications of the ACM 9 (Jan.-Feb. 1966), pp. 13-25, 89-99.

Wirth 72 — Wirth, N., The programming language PASCAL, Acta Informatica 1 (1971), pp. 35-63.

Yanov 58 — Yanov, Yu. I., On the logical schemes of algorithms, Problems of Cybernetics 1 (1958), pp. 82-140.

Younger 63 — Younger, D. H., Minimum feedback arc sets for a directed graph, IEEE Transactions on Circuit Theory CT10, 2 (1963), pp. 238-245.