Technical Report CS/E 84-006 October, 1984

FUNCTIONAL ANALYSIS OF PROGRAMS

Richard Hamlet
Oregon Graduate Center
Beaverton, OR 97006

Harlan Mills
Department of Computer Science
University of Maryland
College Park, MD 20742

Abstract

Analysis of computer programs using a semantics that combines features of the operational and denotational methods is described. The method is an explanatory, analytic tool, a "program calculus" that allows program meaning to be obtained from program syntax, then compared to a desired meaning by simple set-theoretic methods. Meanings are functional, sets of ordered (input, output) pairs. A subset of Pascal is used to illustrate the theory.

Dr. Hamlet's research was partially supported by the Air Force Office of Scientific Research (Contract F49620-83-K-0018).

1 *Introduction*

Three important theoretical underpinnings of computer programming are:

(1) Syntax. Phrase-structure grammars (usually context-free) are used to constrain the form programs may take [1]. It is possible to give very precise descriptions using more complex grammar mechanisms [2,3], but a collection of "static semantics" informally given is the rule [4]. In any case, there is almost no disagreement about the appropriateness of defining language syntax using grammars.

(2) Semantics. Meanings can be assigned to programs in ways ranging from clever use of natural language [1] through semi-formal aids [5] to careful mathematical definitions [3,6,7]. Syntax is exploited to break a program into units whose separate meanings are defined, then combined to form the meaning of the whole. However, there is no consensus about the "best" semantic definitional technique. Furthermore, it is easy to confuse definition with specification (see (3) following) because each technique carries with it a natural method of reasoning about the meanings it defines.

(3) Specification. The definition of programming-language semantics captures what a program does mean; specification captures what one was intended to mean. It is natural to use a technique for specification that is closely related to the definitional technique--then it is possible to reason about programs, that is, prove things about them and their specifications.

This paper describes a functional approach to the study of programming following Mills [8, 9]. The technique is both rigorous and practical, using methods and ideas from operational and denotational semantics.

Our technique is an alternative to the Floyd/Hoare semantic theory, and it may help the reader to explain how that theory would be presented if it were the subject of this paper. (The discussion treats "truth" and "proof" informally, to make clear the computer-science issues at the expense of the logical ones. For a complementary treatment, see Apt [10].)

We believe that the proper fundamental view of programming is the imperative, operational one. People understand a program as a sequence of steps by which input is transformed to output. Each step corresponds to some syntactic part of the program, which authorizes or effects that step. In the case of looping and invoking procedures one part of the program may be used over and over; without loops or procedures each part has at most one use. By defining an appropriate intermediate "state" of a program's computation, the action of each program statement can be precisely described, as effecting the transformation of one intermediate state into the next. The entire process begins with a special initial state containing the input, and ends with a special final state containing the output. Definitions of this kind are natural, and have been given from the beginning, notably by Turing [16]. The definition is so natural, and procedural languages on the ALGOL 60 model are so well known, that operational definition may be tacitly assumed for such a language.

The difficulty with an operational definition is that it is only easy to use in tracing a particular computation with known input. Unless the input is given, the initial state is unknown, and one cannot easily proceed. Perhaps this is the source of the well-known simplification of reality that "computers can only process numbers." In any case, to reason successfully about programs requires a different technique. Floyd [19] was the first to devise one. He imagined first-order logic assertions attached to a program, a *preassertion* before each statement, and a *postassertion* after. For a particular statement with preassertion P and postassertion Q, the *verification condition* V(P,Q) is a formula constructed from P and Q that attempts to capture the statement's meaning. V(P,Q) is successful in this attempt if it holds if and only if the truth of P, followed by the execution of the statement, guarantees the truth of Q. A program may then be

proved as follows. Pre- and postassertions are devised for each statement such that each verification condition holds, and each postassertion implies the preassertion of the following statement. It can then be shown by induction that if the preassertion before the program's first statement (the *input assertion)* is true, then following execution of the program, the postassertion after the program's last statement (the *output assertion)* must be true.

The definition obviously involves a tacit definition of the program meaning, since otherwise the much-mentioned "execution of the program" makes no sense. Given the operational definition, it is possible to prove that certain verification conditions *are* successful in capturing the meaning so defined. This is what Floyd did, arguing informally that his verification conditions were the proper ones according to the usual understanding of statement meanings. When a program has been proved we can then be sure that should its input assertion be true, and should it execute and reach the last statement, that the output assertion will be true, where execution agrees with the tacit definition used to frame proper verification conditions.

But it is also possible to turn matters around and use Floyd's idea as a semantic definition. For each statement, we can *define* its meaning as that action which will make Floyd's verification condition hold. One attractive feature of this approach is that it eliminates both the messy operational definition and the messy proofs that the verification conditions capture it. A program proof then takes on an entirely different character. It is more a *calculation* of the program's meaning according to Floyd's definition, in that it establishes the pair of input and output assertions as a proper overall description. In some sense this pair is computed from the assumed verification-condition meanings for individual statements, through the mechanism of other, intermediate assertions necessary to complete the "proof." As anyone who has ever done a Floyd proof knows, this "calculation" is a tedious one, and unless it is guided by intuition about what the program actually does (using the tacit operational definition) it is a hopeless task.

Hoare's contribution [20] to the method was to recognize that Floyd's verification conditions provided strong constraints on the pre- and postassertions P and Q. Only for very peculiar pairs P,Q will verification condition V(P,Q) hold, and by describing those peculiar pairs he eliminated the necessity for establishing the truth of V(P,Q). When a pair P,Q satisfying the Hoare constraints for some statement are used around that statement, V(P,Q) is forced to hold. Hoare's rules are usually given in the form of a "logic" whose statements have the form

P { S } Q

where P and Q are pre- and postassertions for statement S, and the axioms and rules of inference of the logic determine which P and Q will make V(P,Q) hold, where V is the verification condition for S. The interpretation taken for this statement is, "if P holds and S executes then Q holds." (Hoare did not give even an informal argument that his rules for pre- and postassertion choice will force the truth of Floyd's verification conditions, but he did state that the work is equivalent.) A Hoare proof of a program then consists of choosing assertions that satisfy his rules. Some of the rules mechanically generate the needed assertions, others only give properties that they must have.

The same definitional situation obtains for a Hoare proof as for a Floyd proof: if a tacit operational definition is taken for the program it is necessary to prove that the Hoare rules agree with that definition; then the proof is a proof that the program's execution (so defined) must behave as the assertions state. On the other hand, if the Hoare rules are taken as defining what the program does, nothing need be proved about them, and the program proof becomes a calculation of assertions. In the latter view, Hoare's method is superior to Floyd's, since some of its

rules (notably for assignment) are very easy to use.

The usual textbook treatment of the Floyd/Hoare method tacitly assumes the operational definition of the programming language, does not prove that the Floyd/Hoare rules are in agreement with that definition, yet then claims that the proofs have significance beyond calculation from definitions. If conventional logic were presented in this way, it would define logical validity using proof theory, then claim that the theorems proved were true without an independent definition of truth. Indeed, logic presented for computer science students often does exactly that.

In this paper we hope to do better in three ways. First, since the intermediate states of the operational definition are an important part of our formal treatment, it is easier to give the operational definition, and to prove that our formalism agrees with it. Second, when the formalism is viewed as a device for calculating meaning from definitions, the calculation is often easier, more intuitive than with the Hoare method. Finally, many of the peculiar features of programming languages on which Hoare logic founders, notably those involving procedures and their parameters, our formalism handles without difficulty.

The rest of this paper is organized as follows:

<table>
<tr><th>Section</th><th colspan="2">Contents</th></tr>
<tr><td>2</td><td colspan="2">Describes the programming language and outlines its operational definition.</td></tr>
<tr><td>3</td><td colspan="2">Gives the denotational character of the semantic theory to be developed.</td></tr>
<tr><td>4</td><td colspan="2">Develops the complete theory for linear programs, the special case of no conditionals or loops.</td></tr>
<tr><td>5</td><td colspan="2">Defines the desired relation between specification and program meaning, that of correctness.</td></tr>
<tr><td>6</td><td>Conditional statements</td><td rowspan="3">Completes the semantic theory.</td></tr>
<tr><td>7</td><td>Iterative statements</td></tr>
<tr><td>8</td><td>Procedure statements</td></tr>
</table>

2 The Language and its Operational Semantics

In choosing a programming language to explain, when the subject is not that language but the explanation, there is a fine line between the real and the abstract. On the side of abstraction, it can be argued that the language should show off the features of the explanation to good advantage, and the peculiarities of real languages should not be allowed to confuse a clear picture. But the

real-side argument is equally cogent: if the explanation does not show that it can handle practical complications, it is a failure. We compromise on a subset of the Pascal language.

2.1 Features included in CF Pascal

As primitive data types we include only characters and files of characters. The virtue in this choice over numeric types is that it eliminates number theory--not the central language-theory idea--from the example. The only operations on characters are comparisons and "next in alphabetic sequence;" the latter is sometimes undefined.

Statements of our language present less difficult choices. There is an assignment, read/write, a conditional, iteration, and procedure invocation. The most important decision here is to allow the creation of intermediate results through assignment (and thus deal with "sequential" instead of "functional" programming), making statement sequence the fundamental construction of the language.

Declarations must be treated by any reasonable semantic theory, but block structure is not central to our language. The idea of two identical identifiers with different meanings is retained by allowing declared procedures to have local variables.

Of the many kinds of procedure call and parameter mechanisms, we have chosen strict call-by-reference and recursive procedures as the ones to include.

The language that results from these choices is probably closer to IAL (of which MAD [11] and JOVIAL [12] are common examples) than to Pascal, but since the latter is much better known, we use its syntax, in what we call "CF Pascal" (for Character, File). Although for present purposes it is irrelevant that this language is of practical use, we have found it so for text-formatting problems, and for teaching introductory programming [13]. The exact scope of the CF subset is best given by the denotational theory in Sections 3-8: CF Pascal includes what we define, and where the Pascal definition [14] is clear, we define it that way.

2.2 Operational Definition

Although the full details of an operational definition for CF Pascal will not be given, enough will be presented to permit sample proofs that our analysis techniques are correct. In Sections 3-8, however, the denotational theory will be viewed as defining CF Pascal, and the reader who wants to show that these definitions are really theorems based on the operational definition will have to supply the proofs.

The essential element in a languages's operational definition is the ***instantaneous state description*** of a program. This object is successively transformed as input is converted to output by the program's actions. For CF Pascal this description consists of a *data state* and a *statement pointer*. The data state records the current association between program identifiers and their values; the statement pointer tells which part of the program will be used next. The program part indicated by the statement pointer authorizes a transition from the current instantaneous description to a subsequent one, according to rules peculiar to itself. It is these rules, for how each

statement changes one description into the next, that constitute the operational definition.

With only characters and files of characters, the values associated with quantities internal to CF Pascal programs come from an alphabet and strings over that alphabet, with the complication that file values are separated into the part already processed and that yet to be processed. That is, a file value is technically a pair consisting of a "past" string and a "future" string. Thus a data state of a CF Pascal program includes values from the two sets: characters and string pairs. Each value is connected with a program identifier, and represents its current "contents." For example, in a program where the identifier `OUTPUT` (of type `TEXT`) and `XXX` (of type `CHAR`) occur (note the special type font employed for parts of programs), a data state T might be the set of ordered pairs:

$$T = \{(\texttt{OUTPUT}, (\text{Page } 1, \Lambda)), (\texttt{XXX}, 2)\}$$

where Λ is the empty string. However, when no confusion can arise, we will display data-states in a simpler notation, using the type fonts to distinguish values from identifiers. For example, the data state above will be written:

$$T = \{\texttt{OUTPUT}=\text{Page } 1|, \texttt{XXX}=2\}$$

where the "=" separates identifier and value, and a vertical bar ("|") separates the future from past string in a file. In the example, `OUTPUT`'s future string is empty.

The statement pointer identifies the next part of the CF Pascal program to be used; in examples the lines may be numbered and the pointer given by number, or if there is no confusion, the statement may be named, e.g., "the IF statement."

Each *computation* by a CF Pascal program begins with an initial instantaneous description in which the statement pointer is to the program's first statement and the future string of the identifier `INPUT` is the contents of the standard input file, the `INPUT` past string is empty, and both strings for identifier `OUTPUT` are empty. The computation may terminate successfully, in which case the final instantaneous description has an undefined statement pointer, and a data state in which the past string for `OUTPUT` is the contents of the standard output file. In between initial and final instantaneous descriptions the computation comprises a sequence of instantaneous descriptions which reflects the changing data state and the flow of control. Each member in this sequence results from the previous member by the action of a part of the program, a statement.

Suppose that the current description is C, and its statement pointer indicates an assignment statement. Then the description D to follow in sequence is the same as C except that D's statement pointer is to the statement following the assignment, and in the data state of D the identifier on the left of the assignment may have a new value. This value is obtained by evaluating the right-side expression of the assignment. Where identifiers appear in this expression, their values are extracted from C. (This is the operational definition of the meaning of the assignment statement.) To take a slightly more complex example, if the statment pointer indicates an IF statement, the description D that must follow C in sequence may have a statement pointer to either the THEN part or the ELSE part, depending on evaluation of the Boolean expression, using identifier values from C. However, the data state of D is the same as the data state of C. (Details of expression evaluation are omitted; a careful definition begins with identifier and constant values, and follows the inductive definition that is implied by the language syntax. See

Section 3.2 for the careful denotational definition.)

When the statement pointer of current description C indicates a declaration of identifier X, the subsequent description D indicates the statement following the declaration; and, the data state of D is the same as that of C except that in it X is paired with *all possible values* of the appropriate type. This is a way of capturing the sense that a newly declared variable has no value before it acquires one from the action of a following statement.

When the statement pointer of current description C indicates a WHILE statement then the statement pointer of the following description D is to either the statement following the WHILE (if the Boolean condition evaluates to *false* in C), or the first statement of the WHILE body (it it evaluates to *true).* The data state of D is the same as that of C.

Compound statements like IF and WHILE make it necessary to define "next statement" in a somewhat peculiar way. In a WHILE statement, the next statement for the last part of the WHILE body is the original WHILE statement itself. In an IF statement, the next statement after the last part of the THEN is the one following the entire IF (that is, at the end of the ELSE).

The remainder of the operational definition of CF Pascal will not be given. Many of the details are similar to the tricks used to make the careful denotational definition in Sections 3-8 (notably for procedure invocation, where the ALGOL 60 copy rule is used), and supplying them is straightforward, if tedious. If full details were given, we would now have a complete definition of the computations of CF Pascal programs, the sequences of instantaneous descriptions beginning with an initial description and ending with a final one. This would allow us to make the following definitions of meaning for programs.

It may happen that a computation never reaches a successful conclusion. The most common reasons are (1) attempts to make use of variables that have not been given a single value, and (2) loops that fail to terminate. In the former case, there is no description that legally follows in the sequence, because in an expression that must be evaluated, an identifier is paired with all possible values in the current description. In the latter case, the sequence of descriptions is unending, because each description at the end of the loop body is followed by one at the beginning of the loop. (Unending recursion is another similar source of failure.) Then consider the complete collection of computations that *are* successful for a single program. These differ because the string in the standard input may take on arbitrary values, thus starting each computation off in a different way (and, typically, leading to a different sequence). This collection is a description of everything that the program can do, and by omission of failure sequences beginning with some strings in the standard input, also a description of what it does not do. But there is a more restricted meaning to the program than its complete collection of computation sequences. Its *input-output behavior* is described by examining only the initial and final descriptions in each member of the successful-computation collection. The initial descriptions contain input strings; the final descriptions contain output strings. As a string-to-string mapping, the meaning of the program is the function whose domain is all the strings appearing as input for successful computations, the function whose value on one of those strings is the output string in the corresponding final instantaneous description.

3 Denotational Definition of CF Pascal

CF Pascal programs, viewed as "black box" objects, have two special files, INPUT and OUTPUT; a program transforms the former into the latter. Because these files contain character sequences, the meaning of a program is a string-to-string mapping. It is the "denotational" view of semantics (usually credited to Scott [15]) to assign to a program text (itself of course a string) a meaning from the string-to-string maps, and to construct this meaning by first assigning appropriate mappings to program parts, then combining them into a meaning for the entire program.

3.1 Data-state Transformations

The denotational view of a data state is that it is a mapping from identifiers to values. For example the state

T = {OUTPUT=Page 1|, XXX=2}

given in the abbreviated notation of Section 2, has:

T(OUTPUT) = (Page 1, Λ), and T(XXX) = 2.

To express the transformation of one data state to another, our notational device is to employ the program fragment that effects the transformation, but to distinguish that fragment from a string (the program syntax) by surrounding it with a box. (The idea is due to Kleene [17].) Thus the denotational view that the meaning of a fragment is a mapping, and assigning meaning is associating a program string with its data-state map, is expressed by boxing the string. For example, in the state T above, the transformation effected by

```
WRITE('3')
```

would be shown as

WRITE('3') (T) = {OUTPUT=Page 13|, XXX=2}.

The notation is excellent for expressing examples; when we try to give the general case (for example if T were not a particular data state, but any state), a functional notation using ordered pairs is an improvement. For this example,

WRITE('3') = {(T, U): T = U except that the past string of U(OUTPUT) is the past string of T(OUTPUT) with 3 appended}.

3.2 Meaning of Expressions

The meanings of CF Pascal statements and programs can be constructed from the meanings of CF Pascal expressions. Because the language is severely restricted, there are very few of these. The meaning of an expression is a mapping from data states to the appropriate value range (characters for CHAR expressions, string pairs for files, and { *true, false* } for Boolean expressions). If X is a variable (necessarily of type CHAR or TEXT). then

$$[\![X]\!](T) = T(X).$$

For example, if

$$T = \{OUTPUT=Page\ 1|, XXX=2\}$$

then

$$[\![OUTPUT]\!](T) = Page\ 1$$

and

$$[\![XXX]\!](T) = 2.$$

The constants in CF Pascal are of type CHAR, and their meanings are the obvious ones:

$$[\![\text{'A'}]\!](T) = A$$
$$[\![\text{'B'}]\!](T) = B$$

etc.

for all data states T.

The only other CHAR expression uses the built-in function SUCC, and its meaning is defined inductively. As a first attempt, we might try:

$[\![SUCC(e)]\!](T)$ = the character following $[\![e]\!](T)$ in sequence,

where *e* is any CHAR expression, and "in sequence" is the lexicographic character ordering defined for Pascal. Because the character ordering may differ from machine to machine, and to illustrate the treatment of routine errors, we here take SUCC to be defined on A through Y in the obvious way, but make $[\![SUCC('Z')]\!]$ undefined for all data states. Similarly, the digits 0 through 8 have the obvious successors, but $[\![SUCC('9')]\!]$ is undefined. We say (for example) that "G follows F in sequence," but that nothing follows 9. Except for A-Y and 0-8, SUCC is undefined. Instead of adding an "undefined" clause to the definition above, the data state can be eliminated and the definition given as a collection of ordered pairs:

$[\![SUCC(e)]\!]$ = {(T, c): $[\![e]\!](T)$ is the character b, and c follows b in sequence}.

In this definition, $\boxed{\texttt{SUCC}(e)}$ can fail to be defined at a particular input state T for two reasons. It may happen that $\boxed{e}$ is not defined at T (and hence $\boxed{e}$ (T) is no character b as required), or that b has no following character c. In either case, this T never appears paired with any c in the defining set.

Finally, Boolean expressions can be given meaning. Only single comparisons between character expressions are part of CF Pascal, and the definitions all have the form:

$\boxed{e < f}$ (T) = *true* if $\boxed{e}$ (T) precedes $\boxed{f}$ (T) in sequence; *false* otherwise

for all expressions e and f (and similarly for the operators other than $<$). In contrast to the treatment of SUCC, we here assume that each pair of characters can be meaningfully compared, but only obvious situations like B $<$ X will occur in examples. Care is required here because of the occurrence of the box functions on the right side of the definition. Should one of them be undefined, then the meaning function of the Boolean expression is also undefined. This corresponds to complete evaluation (as opposed to short-circuit, McCarthy evaluation) of Boolean expressions, and to the arbitrary choice that once a runtime error occurs, the failure of meaning propagates. Failure to be careful about such definitional matters has long been a source of trouble in program proving [18].

3.3 External Strings and Data States

The meaning of a program must be a string-to-string function; the meaning of a program fragment is a data-state-to-data-state function. To bring these into line requires associating a string that is presumed to constitute the input file with an appropriate internal data-state string pair attached to INPUT, and similarly associating the internal OUTPUT pair (initially empty in both past and future parts) with the program output. The necessary actions can be imagined to be the meaning functions of the program header and terminating period, program parts whose denotational meanings therefore transform strings to states, and states to strings, respectively.

$$\boxed{\texttt{PROGRAM P(INPUT, OUTPUT);}\ T\ \texttt{.}}$$
$$= \boxed{\texttt{PROGRAM P(INPUT, OUTPUT)}} \circ \boxed{T} \circ \boxed{\texttt{.}}$$

where T represents the bulk of the program, and $\circ$ is functional composition, written with the first-applied function at the left. Continuing the definition, for any strings D and E,

$\boxed{\texttt{PROGRAM P(INPUT, OUTPUT)}}$ $(D) = \{\texttt{INPUT=}|D, \texttt{OUTPUT=}|\}$

$\boxed{\texttt{.}}$ (..., OUTPUT=E|, ...) = E.

3.4 Calculating Program Meaning

The semantics to be presented below is a functional "calculus" that allows step-by-step computation of a program's meaning. By anticipating a definition to be subsequently presented, we can now illustrate this calculus:

$$\boxed{\texttt{BEGIN IF } e \texttt{ THEN END}} = \{(T,T): \boxed{e} \text{ is defined at T}\}.$$

This function is defined, and identity, just where $\boxed{e}$ is defined. A program's meaning is defined to be the (functional) composition of the meanings of its fragments, taken in order. Then we can calculate the meaning of

```
PROGRAM P1(INPUT, OUTPUT);
 BEGIN
  IF SUCC('Z') > 'A' THEN
 END.
```

applied to the string D:

$$\boxed{.}\,(\boxed{\texttt{BEGIN IF SUCC('Z') > 'A' END}}\,(\{\texttt{INPUT=}|D, \texttt{ OUTPUT=}|\})).$$

The function

$$\boxed{\texttt{BEGIN IF SUCC('Z') > 'A' THEN END}}$$

is by definition either the identity function or undefined, depending on the evaluation of the two CHAR expressions. Both have (or fail to have) values independent of the state, namely

$$\boxed{\texttt{SUCC('Z')}}\,(T) = \text{character following } \boxed{\texttt{'Z'}}\,(T) \text{ (if any)}$$
$$= \text{character following Z (if any)}$$

but in fact there is none. It is therefore irrelevant that

$$\boxed{\texttt{'A'}}\,(T) = A$$

because the innermost function is undefined, and the result is that the program P1 means a function that is everywhere undefined, that is, the empty function.

4 Statement Sequences and Linear Programs

The fundamental rule of the program calculus is functional composition. To calculate the meaning of a sequence of CF Pascal statements, first obtain the functional meaning for each one, then compose those functions. Statements are separated by semicolons within BEGIN - END brackets,

so:

$$\boxed{\texttt{BEGIN S1; S2; . . . ; Sn END}}\,(T) = \boxed{\texttt{Sn}}\,(\ldots(\boxed{\texttt{S2}}\,(\boxed{\texttt{S1}}\,(T)))\ldots),$$

or in the purely functional notation,

$$\boxed{\texttt{BEGIN S1; S2; . . . ; Sn END}} = \boxed{\texttt{S1}} \circ \boxed{\texttt{S2}} \circ \ldots \circ \boxed{\texttt{Sn}}\ .$$

There is here (and throughout this presentation) a lack of precision caused by omitting detailed syntactic analysis. Presented with a program text in a box, how is the text to be broken up into units to which the definitions are applied? We have used words like "statement" as if they had precise meaning in any such text. And of course, if care were used, they do have precise meaning, given by the derivation tree for the program. In that tree at any level there is precisely one "statement," indicated by the nonterminal of that name in an appropriate CF Pascal grammer. This use of grammar as a basis for the semantics goes back to ALGOL 60 [1], and may be the most important feature of grammar-based syntax. Here we do not make the correspondence precise, but in examples we order the compositions as they appear in the derivation tree.

4.1 Null Statement

Although the null statement often only enters programs by accident, it is a legitimate part of CF Pascal. Thus define:

$$\boxed{\ } = \mathrm{I}.$$

4.2 Variable Declarations

The declarations within a program have a role in the program calculus similar to that of the program header: they modify the data state so that it contains the proper identifier names for the remainder of the program to process. Each **VAR** declaration has a functional meaning that transforms a data state in which its identifier does not appear, to one in which it does appear. The definition must make clear what happens in the anomalous situation that a variable is used before it is assigned a value. To reflect the possibility that any value might be subsequently acquired, we define the meaning of the declaration to be a relation including all values of the appropriate kind. That is, for example

$$\boxed{\texttt{VAR V: CHAR}} = \{(T, W)\colon W = T \cup \{(\mathbf{V}, x)\colon x \text{ is a value of type } \mathbf{CHAR}\}.$$

This relation in which all possible values are paired with **V** is awkward to write, so we introduce a shorthand of "?" for the multiple values. Then for example the program part

VAR Fresh: CHAR

transforms execution state

{INPUT=|ABC, OUTPUT=|}

to execution state

{INPUT=|ABC, OUTPUT=|, Fresh=?}.

In the box notation:

$$\boxed{\texttt{VAR Fresh: CHAR}}\ (\{\texttt{INPUT=|ABC, OUTPUT=|}\}) = \{\texttt{INPUT=|ABC, OUTPUT=|, Fresh=?}\}.$$

The scope of an identifier introduced by a VAR declaration is the BEGIN ... END that follows it. Outside this scope the variable ceases to exist, hence the definition must arrange to remove the new identifier as well. The inverse of the VAR function does just the right thing, no matter whether the "? value" has been overlaid, or not. Thus the definition is

$$\boxed{\texttt{VAR } v\texttt{: CHAR; BEGIN . . . END}} = \boxed{\texttt{VAR } v\texttt{: CHAR}} \circ \boxed{\texttt{BEGIN . . . END}} \circ \boxed{\texttt{VAR } v\texttt{: CHAR}}^{-1}.$$

The extension of the definition to a TEXT variable, and to several variables, is straightforward.

When a data state T is not a function because in it some identifier V is associated with all potential values (in the shorthand, V=? appears in T), then $\boxed{\texttt{V}}$ is undefined on T.

4.3 Assignment Statements

The intuitive meaning of the assignment statement as a data-state transformation is that the identifier on the left side ceases to be associated with its old value, and instead becomes associated with a value obtained from the right side. In CF Pascal, assignment statements are of the restricted form:

$v := e$

where v is a variable declared as CHAR and e is either a variable declared as CHAR or a literal character enclosed in single quotation marks, or a nest of SUCC function calls founded on such a variable or literal. Section 3.2 has formally defined the box function for such expressions as a mapping from data states to character values (which may be undefined for some uses of SUCC). The investment in notation now pays off in a concise definition of the meaning of an assignment statement:

$\boxed{v := e}$ = {(T, U): U is the same data state as T except that U(v) = $\boxed{e}$ (T)}.

As usual, the definition includes the implicit case that should $\boxed{e}$ be undefined, then so is the assignment-statement function undefined.

Here are four examples:

$\boxed{\texttt{V1 := 'C'}}$ ({V1=A, V2=B}) = {V1=C, V2=B},
$\boxed{\texttt{V2 := V1}}$ ({V1=A, V2=B}) = {V1=A, V2=A},
$\boxed{\texttt{V1 := SUCC(V1)}}$ is undefined on the state {V1=Z},
$\boxed{\texttt{V2 := V2}}$ is undefined on the state {V2=?}.

4.4 WRITE Statements

Whatever a program may do internally to its execution state, the result cannot be observed by a person unless WRITE statements are included to communicate the internal state to the outside world. In CF Pascal, a WRITE statement may include only a sequence of expression arguments (of the kind defined in Section 3.2). Let there be one argument, e.

$\boxed{\texttt{WRITE(}e\texttt{)}}$ = {(T, U): U = T except that the past string of U(OUTPUT) is T(OUTPUT) with $\boxed{e}$ (T) appended}.

The generalization to a sequence of expressions is that their values are appended in the order written.

Enough of the program calculus has now been presented to handle a simple real program. For example, if P is:

```
PROGRAM WriteHello(INPUT, OUTPUT);
 VAR
  LetterL: CHAR;
 BEGIN
  LetterL := 'L';
  WRITE('H', 'E', LetterL, LetterL, 'O')
 END.
```

then if the input string is x the program header and VAR declaration establish the state

{INPUT=|x, OUTPUT=|, LetterL=?}

on which the program works as follows:

$$\boxed{\texttt{WRITE('H', 'E', LetterL, LetterL, 'O')}}\,(\boxed{\texttt{LetterL := 'L'}}\,(\{\texttt{INPUT}=|x,\ \texttt{OUTPUT}=|,\ \texttt{LetterL}=?\}))$$

$$=\boxed{\texttt{WRITE('H', 'E', LetterL, LetterL, 'O')}}\,(\{\texttt{INPUT}=|x,\ \texttt{OUTPUT}=|,\ \texttt{LetterL}=L\})$$

by the action of the assignment statement and statement composition. The values of the arguments in the WRITE statement are:

$$\boxed{\texttt{'H'}}\,(\{\texttt{INPUT}=|x,\ \texttt{OUTPUT}=|,\ \texttt{LetterL}=L\}) = H$$
$$\boxed{\texttt{'E'}}\,(\{\texttt{INPUT}=|x,\ \texttt{OUTPUT}=|,\ \texttt{LetterL}=L\}) = E$$
$$\boxed{\texttt{LetterL}}\,(\{\texttt{INPUT}=|x,\ \texttt{OUTPUT}=|,\ \texttt{LetterL}=L\}) = L$$
$$\boxed{\texttt{'O'}}\,(\{\texttt{INPUT}=|x,\ \texttt{OUTPUT}=|,\ \texttt{LetterL}=L\}) = O.$$

The WRITE statement thus changes the execution state to:

$$\{\texttt{INPUT}=|x,\ \texttt{OUTPUT}=HELLO|,\ \texttt{LetterL}=L\}.$$

The calculation of $\boxed{P}$ requires only the application of $\boxed{\texttt{VAR LetterL: CHAR}}^{-1}$, which removes `LetterL` from the state, and $\boxed{.}$, which extracts the past string of `OUTPUT`, with the result:

$\boxed{P}\,(y) = \text{HELLO}$ for any y.

4.5 READ Statements

The list of arguments in a CF Pascal READ statement can consist only of identifiers declared as CHAR variables. READ statement meaning is easy to give if enough characters are available in the data state (i.e., attached to `INPUT` as future string). The case of multiple variables in a READ statement is a straightforward extension of the single-variable case.

Suppose then that a data-state value for `INPUT` contains at least one character in its future string, say c. In the abbreviated form of a data state, write such a value as

`INPUT`=x|cy

where x and y represent the string parts of the value not of interest. Then the meaning of

`READ(Cv1)`

where the variable is suitable declared `CHAR` is

$\boxed{\texttt{READ(Cv1)}}$ ({INPUT=x|cy, ..., Cv1=v, ...}) = {INPUT=xc|y, ..., Cv1=c, ...}.

As an example, consider the program *P*:

```
PROGRAM Change2(INPUT, OUTPUT);
 VAR
  C2: CHAR;
 BEGIN
  READ(C2);
  WRITE(C2);
  READ(C2);
  WRITE('2')
 END.
```

We work out:

$\boxed{P}$ (ABC)

The program-header function and the VAR-declaration function establish the execution state:

{INPUT=|ABC, OUTPUT=|, C2=?}.

Then the successive statements yield:

$\boxed{\texttt{READ(C2); WRITE(C2); READ(C2); WRITE('2')}}$
({INPUT=|ABC, OUTPUT=|, C2=?})

= $\boxed{\texttt{WRITE(C2); READ(C2); WRITE('2')}}$ ({INPUT=A|BC, OUTPUT=|, C2=A})

= $\boxed{\texttt{READ(C2); WRITE('2')}}$ ({INPUT=A|BC, OUTPUT=A|, C2=A})

= $\boxed{\texttt{WRITE('2')}}$ ({INPUT=AB|C, OUTPUT=A|, C2=B})

= {INPUT=AB|C, OUTPUT=A2|, C2=B}.

Finally applying $\boxed{\texttt{VAR C2: CHAR}}^{-1}$ *o* $\boxed{\texttt{.}}$ yields

$\boxed{P}$ (ABC) = A2.

Reading past end of file is a runtime error in CF Pascal, so in the definition we must exclude those data states in which there are insufficient characters on the future string attached to INPUT. Define:

$\boxed{\texttt{READ(C)}}$ = {(T, U): the first character of the future string in T(INPUT) is c, and U = T except that (i) c is transferred from the past to the future string in U(INPUT) and (ii) U(C) = c}.

As usual, note that when there is no first character in T as required, there is no ordered pair in the function with first member T.

4.6 Analysis of Linear Programs

The analysis of a linear program is a straightforward, mechanical process. Each part function can be computed, and their composition found. In practice it is not easy to compose these functions, because the notation soon becomes unwieldy. The trace table [9] is a technique for organizing the computation for a series of assignments, which can be used for READ statements which "assign" as well. It amounts to a symbolic execution of the code in terms of initial values for the variables, leading to a set of equations which can be solved for their final values.

5 Proving Programs Correct

Before we complete the denotational definition of CF Pascal, two fundamental questions about program P should be raised:

(I) does $\boxed{P}$ agree with the intuitive meaning of P?

(II) Does P in fact do what it is intended to do?

5.1 Proving Denotational Meanings

A program's meaning defined operationally (Section 2) and denotationally (Sections 3-4, and continued in the sequel) are the same kind of objects: maps from strings to strings. The denotational definitions have of course been chosen to agree with intuition about what programs do, and perhaps it is obvious that those definitions are consistent with the operational ones. But a formal proof of this fact is possible.

Consider program P. The operational definition assigns a meaning to P: that string-to-string mapping M whose (input, output) pairs are the appropriate parts of the initial and final instantaneous descriptions of the successful computations of P. From the denotational definition $\boxed{P}$ can be calculated. We would like to prove that $M = \boxed{P}$.

Theorem (Denotational Consistency, linear case). For any linear program P, $\boxed{P}$ is the same as the meaning function M of the operational definition.

Proof. In fact, much more is true: in any computation with input x and output y:

The initial instantaneous description has state
B = [PROGRAM P(INPUT, OUTPUT)] (x);

Let the next statements of the computation be S1, S2, ... , Sk in order. Then the final instantaneous description has state D = [Sk] (...([S2] ([S1] (B)))...).

[.] (D) = y.

The first and last of these statements are proved by combining the definition of the box functions with the definition of initial and final states of a computation. If we take the middle statement as part of the theorem, the whole can be proved by induction on the number of statements in the BEGIN block of P.

Base case. If the program has no statements, i.e., P is

```
PROGRAM P(INPUT, OUTPUT);
 BEGIN
 END.
```

then the only computations are those whose initial and final instantaneous descriptions are the same, containing an empty OUTPUT (past) string, and an arbitrary INPUT (future) string. That is, for all inputs x, the output is Λ; or, M is the constant function with the empty string as value.

From Section 4, [BEGIN END] is the identity function, so [S1] (B) = B = D as required. Finally, from Section 3.3,

[P] = [PROGRAM P(INPUT, OUTPUT)] o [BEGIN END] o [.]

so

[P] (x) = [.] ({INPUT=|x, OUTPUT=|}) = Λ = y.

(A second base case, in which there are declarations and hence three instantaneous descriptions, is similar.)

Induction step. Now suppose that for all programs of less than m $\geq$ 1 statements, the augmented theorem holds. Consider a program P of m statements. By stripping off the last statement Sm, a program P' is created to which the inductive hypothesis applies; let its last statement be Sm´. The additional condition (for P') tells us that the penultimate data state of the original computation, say D´, is obtained from the initial state by

D´ = [Sm´] (... [S1] (B)...)

In the computation by P, the statement Sm operates on D´ to produce the final description D; in [P], the last state is [Sm] (D´). It therefore remains only to show that the action of Sm is the same operationally and denotationally. The proof is by cases, one for each kind of final statement.

Suppose the statement removed from P to form P' is an assignment

X := E

Then from the operational definition, in D, X has a value calculated from E's value in D′. This is exactly the transformation defined for $\boxed{\text{X := E}}$. The other cases for linear statements are similarly transparent. QED

In the sequel additional statements of CF Pascal will be denotationally defined, and for each the Theorem could be extended, by proving additional cases for the final statement. However, the technical complications are considerable, particularly for the iteration and procedure-call statements, and the additions to the proof will not be given.

5.2 Proving Programs Meet Specifications

Any program has a purpose, but that purpose may not require results in some exact form. For example, a program may be required to print the members of a set, without their order or the page layout being specified. These variations can be described by providing, for each instance of input data, all acceptable instances of output data, that is, a relation consisting of all such pairs. Therefore, a CF Pascal program specification is defined as any relation whose domain and range are the set of character strings.

Just as a program function may be difficult to describe in a well-known mathematical form, but nevertheless exists for every program; so a program specification may be difficult to set down, but is a mathematical relation nevertheless.

An important special case of a specification occurs when the acceptable pairs of input data and output data form a function; that is, for each instance of input data exactly one instance of output data is acceptable. This special case of a specification relation is therefore a specification function.

The specification (relation or function) for a program is a mathematical form of what the program is supposed to do, a description of desired results. This form is entirely independent of any program to realize it, and in fact is the starting point for writing a program. It is important to recognize that a specification gives no information about how some program might perform to meet it. Since it is simply a collection of input-output pairs, it states what is to be done, without a hint of a method for doing it. On the other hand, once a particular program exists, its program function (which is the same kind of mathematical object as a specification function) defines what the program *does* do, without regard for any intentions the programmer may have had. Furthermore, using the program calculus, this meaning can be calculated step-by-step from the program text itself. The program function itself does not express how the program accomplishes what it does, but to calculate the program function requires full details of the program's inner workings. A central question of programming can be simply stated in these terms:

Given a specification and a program, does the program fulfill that specification?

The technical definitions necessary to state this question are already available. Given a program specification relation r and a program P, we say that P is correct with respect to r if and only if, for every member x of the domain of r (an instance of input data), P produces some member of the range of r which is paired with x in r. That is, for each input x, $(x, \boxed{P}(x)) \in r$.

Theorem (Program Correctness): Program P is correct with respect to specification relation r if and only if:

$$\text{domain}(\text{r } o \boxed{P}) = \text{domain}(\text{r})$$

Proof. The expression r o $\boxed{P}$ identifies all acceptable pairs of r computed by P. Therefore domain(r o $\boxed{P}$) identifies the set of input data for which P produces acceptable output data. Since domain(r) is the set of input data for which r specifies acceptable output data, the condition

$$\text{domain}(\text{r } o \boxed{P}) = \text{domain}(\text{r})$$

ensures that P produces acceptable output data for every instance of input data defined by r. QED

Note that P may execute successfully for input data not identified by r, but such pairs of $\boxed{P}$ are screened out of (r o $\boxed{P}$) by r. Note also that if P produces an unacceptable instance of output data, no member of r with that input data can be in r o $\boxed{P}$, and therefore domain(r o $\boxed{P}$) cannot coincide with domain(r).

In case the program specification is a function, the condition for program correctness can be simplied as follows:

Corollary (Program Correctness): Program P is correct with respect to specification function f if and only if

$$\text{f} \subseteq \boxed{P}\ .$$

Proof. The expression f o $\boxed{P}$ identifies all acceptable pairs of f computed by P, which must be f, itself. That is, P is correct with respect to f if and only if

$$\text{f } o \boxed{P} = \text{f}.$$

and thus if and only if f $\subseteq$ $\boxed{P}$. QED

6 Conditional Statements

Much of the power (and complication) in programs comes from their conditional statements, which provide the means of making decisions based not only on program input, but on intermediate values internal to the program. However, the meaning of a single conditional in isolation is easy to define.

6.1 Meaning of Conditionals

Let the program S be:

```
IF B
 THEN
   T
 ELSE
   E
```

where B is Boolean expression and T, E are statements. Then

$$\boxed{S}\,(\mathrm{U}) = \begin{cases} \boxed{T}\,(\mathrm{U}) \text{ if } \boxed{B}\,(\mathrm{U}) \\ \boxed{E}\,(\mathrm{U}) \text{ if } \neg\,\boxed{B}\,(\mathrm{U}) \\ \text{otherwise } \boxed{S} \text{ is not defined at U.} \end{cases}$$

For example if the program were:

```
IF V1 < V2
 THEN
   V1 := V2
 ELSE
   V2 := V1
```

Then

$$\boxed{S}\,(\{\mathtt{V1}{=}\mathrm{A},\ \mathtt{V2}{=}\mathrm{B}\}) = \{\mathtt{V1}{=}\mathrm{B},\ \mathtt{V2}{=}\mathrm{B}\}$$

because

$$\boxed{B}\,(\{\mathtt{V1}{=}\mathrm{A},\ \mathtt{V2}{=}\mathrm{B}\}) = \mathit{true},$$
$$\boxed{T}\,(\{\mathtt{V1}{=}\mathrm{A},\ \mathtt{V2}{=}\mathrm{B}\}) = \{\mathtt{V1}{=}\mathrm{B},\ \mathtt{V2}{=}\mathrm{B}\},$$

and the value of $\boxed{S}$ for this data state is given by the value of $\boxed{T}$.

In order to give a definition by cases without explicitly naming the data state, it is necessary to select one of two sets of ordered pairs according to the Boolean expression. The following standard trick accomplishes this:

$$\boxed{S} = \{(\mathrm{U}, \boxed{T}\,(\mathrm{U})) \colon \boxed{B}\,(\mathrm{U})\} \cup \{(\mathrm{U}, \boxed{E}\,(\mathrm{U})) \colon \neg\,\boxed{B}\,(\mathrm{U})\}.$$

The first set contains all state pairs in which the condition holds, and the second set those pairs in which it does not hold. It is important to note the way that failure of definition can occur here. There is no "evaluation" of these sets in any order. They simply contain or fail to contain certain pairs. For example, should $\boxed{E}$ fail to be defined for some U, that U will not occur in the second set, independent of $\boxed{B}$. Such a failure has no influence whatsoever on pairs in the first set.

However, failure of $[\![B]\!]$ in another matter. If this function fails to be defined for some U, neither condition holds, and U is not paired with anything in either set; that is, $[\![S]\!]$ is undefined for such a U.

The IF statement

```
IF B
 THEN
   T
```

could be given a similar definition, but it can also be agreed to mean the same as:

```
IF B
 THEN
   T
 ELSE
   {null statement}
```

so that its definition can be derived as follows:

$$[\![\texttt{IF } B \texttt{ THEN } T]\!](U) = \begin{cases} [\![T]\!](U) \text{ if } [\![B]\!](U) \\ U \text{ if } \neg\, [\![B]\!](U) \\ \text{otherwise undefined} \end{cases}$$

or, in the functional notation:

$$[\![\texttt{IF } B \texttt{ THEN } T]\!] = \{(U, [\![T]\!](U)) : [\![B]\!](U)\} \cup \{(U,U) : \neg\, [\![B]\!](U)\}.$$

In the second case, the identity function is applied to the state, since that is the meaning of the null statement.

6.2 Analysis of Conditional Statements

Composing the functions that result from IF statements is no more difficult in principle than composing imperative-statement functions, but the notational complication is even more severe. The trace table can be extended to help in practical cases. A conditional trace table [9] is a symbolic execution which includes calculation of the path predicates. For each path the values of variables can be computed symbolically, and thus the path function determined. The function of the whole is then a union of these path functions, each including its defining path condition.

7 *Iteration Statements*

With iteration-free programs, we have seen how to derive program meaning as a composition of the meaning of program parts, in which the number of parts is determined by the static program text. However, with iteration, program parts can be executed repeatedly. If the number of iterations were fixed, the part functions would be fixed compositions of simpler part functions. But the great power of iteration statements arises from a variable number of iterations, so we need not be surprised that the difficulty of dealing with iteration statements increases accordingly.

Although in a long iteration-free program there could be notational difficulty in calculating the program's meaning function, there is no difficulty in principle: each statement has its functional meaning, and the meaning of the whole is simply their composition. There is no mechanical way to deal similarly with iteration statements, but this section presents a definition of meaning, and a technique for proving that a function (nonmechanically obtained) is or is not the meaning function for a given loop.

7.1 *Meaning of Iteration Statements*

The power of iteration exacts a high price when it comes to calculating the meaning of WHILE statements. The pattern of our definitions has been to give the function of each statement-type in terms of its parts. The parts of a WHILE statement are evidently the condition (determining if the iteration should continue) and the loop body (what to do if it should continue). The difficulty is that the action of the loop body is repeated in forming the meaning of the loop as a whole, and this repetition occurs a number of times that is not explicit in the text. If the number of times k the iteration occurs were known, there would be a way out of the difficulty, for then the function of the entire WHILE statement would be a composition of its body's function exactly k times. For example, if it were magically given that

WHILE *B* **DO** *D*

"went around" exactly twice, then we could consider it equivalent to

BEGIN *D*; *D* **END**

and the result would be

$$\boxed{\textbf{WHILE } B \textbf{ DO } D} = \boxed{D;\ D}$$

There is yet a further complication in the WHILE statement. It may happen that a loop never terminates. In that case the "number of iterations" makes no sense, and the function of the loop is undefined for the state that began the repetition. Thus the number of iterations to completion is in all cases the key to the WHILE statement. If this number is k, then the function of the loop is k-fold composition of the function for the loop body; if the iteration continues without end, the function is undefined.

There is a direct way to capture the function of a loop, by asserting that the state resulting from the loop's execution is the result of the loop body executing k times, for some unspecified k. To be consistent with the intuitive meaning of the loop, this k (if it exists) has the property that

the loop condition is *true* before the 1st, 2nd, ... kth iteration, then is *false*. That is, after k iterations the condition fails for the first time. The WHILE statement meaning therefore consists of exactly those ordered pairs for which there is the appropriate k, with the output state being a k-fold action of the loop body on the input state.

The definition of meaning along these lines will now be given. Let WHILE statement W be

WHILE B DO D

where B is a Boolean expression and D is a statement. Then define

$$\boxed{W} = \{(T, U): \exists\, k \geq 0, \text{ such that } \forall 0 \leq i < k \\ (\boxed{B}(\boxed{D}^{i}(T)) \wedge \neg \boxed{B}(\boxed{D}^{k}(T)) \wedge \boxed{D}^{k}(T) = U)\}.$$

For example, consider the loop U:

```
WHILE V1 > '1' DO
 IF V1 = '8' THEN
  V1 := '0'
 ELSE
  IF V1 = '9' THEN
   V1 := '1'
  ELSE
   V1 := SUCC(SUCC(V1))
```

Suppose that the character value attached to V1 in the input state for U is x. If $x \leq 1$, then $k = 0$. If x is a digit, then $k = (10 - x)/2$ rounded up to the nearest integer. (For example, if $x = 9$, then $(10 - 9)/2 = 1/2$ or rounded up, $k = 1$.) It is clear without determining k that V1 will end up 1 or 0. If $x > 9$, the SUCC expression will necessarily go undefined, which in turn causes the assignment statement to mean the undefined function, and this means that in the definition of the WHILE statement there is no k for the case $x > 9$. Thus the meaning of the loop U is:

$$\boxed{U} = \{(T, T): T(\texttt{V1}) \leq 1\} \cup \{(T, A): 1 < T(\texttt{V1}) \leq 9 \text{ and } A = T \text{ except that } A(\texttt{V1}) \text{ is 1 or } \\ 0 \text{ as } T(\texttt{V1}) \text{ is an odd or even digit}\}.$$

7.2 Analysis of Iteration Statements

When the number of iterations cannot be easily determined, the loop controlled by the WHILE statement cannot be easily unwound as its loop body acting over and over. But it can always be unwound into the first time the body acts, and then the rest of the times, if that first execution is guarded by a test to cover the case that the loop doesn't execute at all. That is, the action of

WHILE B DO D

is equivalent to that of

BEGIN IF B THEN D; WHILE B DO D END.

The equivalence can be seen operationally by examining cases. First, suppose the original loop body D is never executed (because condition B fails immediately). Then in the expanded version the IF condition similarly fails, so the broken-out body is not executed, then in the repetition of the loop itself the condition B fails again, so the body is again not done. Thus the two programs agree in this case. Second, suppose the original loop in fact executed its body exactly once. Then the condition initially succeeds, but something in the body causes it to fail when tried a second time. In the expanded version exactly the same behavior is observed, with the IF condition succeeding, the broken-out body executing and the repeated WHILE condition then failing. The remaining cases in which the original loop executed more than once are similar; the broken-out body takes the first execution, and the repeated loop picks up the remainder.

The discussion in the preceding paragraph can be mirrored exactly in the formal meanings we have defined, where the functional meaning for each of the code fragments can be determined, and "equivalence" means that the functions are the same. The analysis by cases becomes a formal induction on the number of iterations required for the loop to terminate.

It may seem surprising that this simple device can help with the analysis of loops, but it does. The reason is that the repeated loop is exactly the same loop as the original, and this allows us to write an equation in which the loop function occurs twice. Equating the meaning functions for the two loops:

$$\boxed{\text{WHILE } B \text{ DO } D} = \boxed{\text{BEGIN IF } B \text{ THEN } D\text{; WHILE } B \text{ DO } D \text{ END}}\ .$$

In the compound statement of the second line the first part is a conditional, and the meaning of that can be worked out separately:

$$\boxed{\text{WHILE } B \text{ DO } D} = \boxed{\text{IF } B \text{ THEN } D} \circ \boxed{\text{WHILE } B \text{ DO } D}\ .$$

The function of the loop has explicitly reentered the equation. It will be clearer if we name this function so it can be easily recognized. Let

$$\mathrm{f} = \boxed{\text{WHILE } B \text{ DO } D}$$

Then the equation above is

$$\mathrm{f} = \boxed{\text{IF } B \text{ THEN } D} \circ \mathrm{f},$$

or

$$\mathrm{f}(\mathrm{T}) = \mathrm{f}(\boxed{\text{IF } B \text{ THEN } D}\ (\mathrm{T})).$$

This recurrence equation in function f is useful because very often we can guess (or believe a comment to discover) what a program is supposed to do. Then to prove that it does indeed do so, the guessed function can be substituted into the equation for f. Care is required in this operation,

however. The recurrence equation is one that the function f for the loop must satisfy; but it does not follow that any function satisfying the equation is in fact the function of the loop. A simple example will show the pitfall. Consider a loop that never terminates:

`WHILE 'A' = 'A' DO {nothing}`.

It is clear that this WHILE statement has a function that is empty--it contains no ordered pairs because the loop does not terminate on any input. But the recurrence reduces to

$f = f$

for this loop, since

$\boxed{\texttt{IF 'A' = 'A' THEN \{nothing\}}}$

is the identity function. Any function f satisfies $f = f$, yet all save the empty function are wrong for the loop.

The remedy for this problem is to add conditions which rule out extraneous solutions to the recurrence equation. One obvious condition, in view of the pitfall above, is to ensure that the function is not defined when the loop fails to terminate. That is, for f to be the loop function requires that

$\text{domain}(f) \subseteq \text{domain}(\boxed{\texttt{WHILE } B \texttt{ DO } D})$.

Another pitfall is shown by the loop:

`WHILE 'A' <> 'A' DO {nothing}`.

In this case the WHILE statement has a function that is the identity function; it acts as a null statement. But, again, since

$\boxed{\texttt{IF 'A' <> 'A' THEN \{nothing\}}}$

is the identity function, the recurrence equation reduces to

$f = f$

and any function satisfies the equation. All but the identity function are wrong for the loop, however. Many functions satisfy the domain constraint developed above as well; but, all except the identity function violate another easily checked condition when B does not hold:

$f(T) = T$ whenever $\neg \boxed{B}\ (T)$.

Happily, just these two additional conditions are sufficient to ensure that an f satisfying the

recurrence equation is the function of the WHILE statement.

Theorem (WHILE statement Verification): Let W be the program fragment

WHILE B DO D.

Then

$f = \boxed{W}$

if and only if:

1. domain(f) $\subseteq$ domain($\boxed{W}$)
2. f(T) = T whenever $\neg \boxed{B}$ (T)
3. f(T) = f($\boxed{\text{IF } B \text{ THEN } D}$ (T)).

Proof. First, suppose $f = \boxed{W}$. Then conditions 1.-3. must be established.

1. domain(f) $\subseteq$ domain($\boxed{W}$) because f and $\boxed{W}$ are the same function.

2. Suppose $\neg \boxed{B}$ (T) for some T. Then $\boxed{W}$ (T) = T by definition, and hence since $f = \boxed{W}$, we have f(T) = T as required.

3. It has been argued above that

$\boxed{\text{WHILE } B \text{ DO } D} = \boxed{\text{IF } B \text{ THEN } D; \text{ WHILE } B \text{ DO } D}$

and by definition of composition and W this is

$\boxed{W}$ (T) = $\boxed{W}$ ($\boxed{\text{IF } B \text{ THEN } D}$ (T),

so the same equation follows for f, since $f = \boxed{W}$.

Conversely, suppose

1. domain(f) $\subseteq$ domain($\boxed{W}$)
2. f(T) = T whenever $\neg \boxed{B}$ (T)
3. f(T) = f($\boxed{\text{IF } B \text{ THEN } D}$ (T)).

Then, we will show that $f = \boxed{W}$.

Let T be any member of domain($\boxed{W}$). That is, $\boxed{\text{WHILE } B \text{ DO } D}$ is defined for input state T. Therefore, by definition there exists a k (depending on T) such that:

$\boxed{B}$ ($\boxed{D}^{k}$(T)) is *false*, but for each $0 \leq i \leq k\text{-}1$, $\boxed{B}$ ($\boxed{D}^{i}$(T)) is *true*.

Then in hypothesis 3, substitute k-1 times for f:

$$\begin{aligned} f(T) &= f(\boxed{\texttt{IF } B \texttt{ THEN } D}\ (T)) \\ &= f(\boxed{\texttt{IF } B \texttt{ THEN } D}\ (\boxed{\texttt{IF } B \texttt{ THEN } D}\ (T))). \\ &= \ldots \\ &= f(\boxed{\texttt{IF } B \texttt{ THEN } D}^{k}(T)) \end{aligned}$$

using the associativity of composition. From the definition of the conditional, this is

$$f(T) = f(\boxed{D}^{k}(T))$$

since each of the evaluations is at a state where $\boxed{B}$ is *true*. On the right side of this equation, the state is one in which $\boxed{B}$ is *false*, so by hypothesis 2, f does not alter this state. Thus

$$f(T) = \boxed{D}^{k}(T).$$

On the other hand, by definition of the loop terminating after k iterations, $\boxed{W} = \boxed{D}^{k}$, so f = $\boxed{W}$.

We have shown that f and $\boxed{W}$ agree on domain($\boxed{W}$). They could only then disagree if f were defined somewhere that $\boxed{W}$ were not. But condition 1. forbids this. QED

For example, consider the WHILE statement *U* of the last section:

```
WHILE V1 > '1' DO
 IF V1 = '8' THEN
  V1 := '0'
 ELSE
  IF V1 = '9' THEN
   V1 := '1'
  ELSE
   V1 := SUCC(SUCC(V1))
```

whose function was claimed to be:

f = {(T, T): T(V1) ≤ 1} ∪ {(T, A): 1 < T(V1) ≤ 9 and A = T except that A(V1) is 1 or 0 as T(V1) is an odd or even digit}.

To prove this using the WHILE statement verification theorem we must show:

1. domain(f) ⊆ domain($\boxed{U}$)
2. f(T) = T whenever ¬ $\boxed{\texttt{V1 > '1'}}$ (T)
3. f(T) = f($\boxed{\texttt{IF V1 > '1' THEN } D}$ (T)),

where *D* is:

```
IF V1 = '8' THEN
 V1 := '0'
ELSE
 IF V1 = '9' THEN
  V1 := '1'
 ELSE
  V1 := SUCC(SUCC(V1))
```

1. The domain of f is evidently $\{c: c \leq 9\}$. The program evidently terminates on V1-values of 9 and 8, and for values of V1 less than 1. On the remaining digits the program advances along the sequence toward 9 or 8 by two steps, and so must halt.

2. Immediate from the definition of f.

3. Consider two cases for the data state T: If $T(V1) \leq 1$, then the box function of the conditional is the identity, and 3. holds. If $T(V1) > 1$, then the conditional box function is just $\boxed{D}$, so we require $f(T) = f(\boxed{D}(T))$, and this is evidently so by an analysis of the cases 8, 9, and 1-7, since the double SUCC preserves even- and oddness.

8 Procedures

The meaning of CF Pascal procedure-call statements should be easy to define, since a procedure body consists of statements whose meaning has already been given. (If the body contains procedure calls, the definition should close at this point.) Three ideas complicate the picture:

(1) Procedures have local variables, whose names may conflict with other variables in the program. Procedures use global variables from an environment different than the one at the point of call. In technical terms, the data state for a procedure's body may be quite different from the data state existing before and after its call.

(2) Procedures have parameters (called by strict reference in CF Pascal), which link the calling and called environments. There is the problem of aliasing: apparently distinct identifiers can refer to a single object.

(3) Procedures may be called recursively, introducing repeated instances of problems of (1) and (2) above, and the further difficulty that the meaning of a call may be defined in terms of another call on the same procedure.

These complications can be handled by a device based on the ALGOL 60 "copy rule" [1].

In substance, the meaning of a procedure-call statement is the meaning of the procedure declaration's body. Technical adjustments are needed to account for parameter names and local/global identifier conflicts, but once these are made, the body meaning can be largely obtained from the definitions in Section 4, 6, and 7. When a procedure call occurs within a procedure body there is no difficulty--the definition is simply applied again and eventually a lowest level is reached in which there are no more calls--unless the call is recursive.

A procedure statement must be associated with a declaration that supplies the procedure body, for example:

```
PROCEDURE HereAndThere;
  T;
X;
HereAndThere
```

has a declared body T, and the text X separates the declaration from the call. The meaning of the procedure statement is:

$$\boxed{\texttt{PROCEDURE HereAndThere; } T\texttt{; } X\texttt{; HereAndThere}} = \boxed{X} \; o \; \boxed{T} \; ,$$

but it is more convenient to write just

$$\boxed{\texttt{HereAndThere}} = \boxed{T}$$

and thus hide the process of identifying the body that goes with the call.

When there are **VAR** parameters, the situation is only a little more complicated. Given a declaration like:

```
PROCEDURE PwithP(VAR X: CHAR);
  T(X)
```

with body T(**X**) written to emphasize that it contains the parameter **X**, a call passing parameter **A** means:

$\boxed{\texttt{PwithP(A)}} = \boxed{T\texttt{(X←A)}}$, where T(**X**←**A**) means the body T(**X**) with each occurrence of **X** replaced by **A**.

The generalization of this definition to multiple parameters, and to **TEXT** parameters instead of **CHAR** parameters, is straightforward. (Students of ALGOL 60 will recognize the copy-rule semantics of call by name, which in the absence of arrays is the same as Pascal's strict call by reference.)

As an example of a procedure call, consider the program:

```
PROGRAM Call(INPUT, OUTPUT):
 VAR
  Next: CHAR;
 PROCEDURE CallMe(Flag: CHAR);
  VAR
   First: CHAR;
  BEGIN
   First := 'B';
   Flag := 'T'
  END;
 BEGIN
  Next := 'A';
  CallMe(Next)
 END.
```

When the procedure call that ends the program occurs, the data state (for input string x) is:

{INPUT=|x, OUTPUT=|, Next=A}.

Since this example is not concerned with INPUT and OUTPUT, in the sequel they will be omitted to simplify the state.

The call on CallMe has the meaning:

$$\boxed{\texttt{CallMe(Next)}}\ (\{\texttt{Next}=A\})$$

$$= \boxed{\texttt{VAR First: CHAR; BEGIN First := 'B'; Next := T END}}\ (\{\texttt{Next}=A\})$$

$$= \boxed{\texttt{VAR First: CHAR}}^{-1}(\boxed{\texttt{BEGIN First := ... END}}\ (\boxed{\texttt{VAR First: CHAR}}\ (\{\texttt{Next}=A\})))$$

$$= \boxed{\texttt{VAR First: CHAR}}^{-1}(\boxed{\texttt{BEGIN First := ... END}}\ (\{\texttt{Next}=A, \texttt{First}=?\})).$$

Working out the meaning of the body itself is straightforward:

$$\boxed{\texttt{BEGIN First := ... END}}\ (\{\texttt{Next}=A, \texttt{First}=?\})$$

$$= \{\texttt{Next}=T, \texttt{First}=B\},$$

so the result is

$$\boxed{\texttt{VAR First: CHAR}}^{-1}\ (\{\texttt{Next}=T, \texttt{First}=B\}), = \{\texttt{Next}=T\}.$$

In most cases, the calculation of a procedure's meaning can be done in terms of the formal parameter identifier, then actual parameter identifiers substituted into this meaning as needed. That is, to calculate `T(X←A)`, calculate `T(X)` (X←A). Since there may be a number of calls with parameters A, B, C, etc., this is superior to calculating `T(X←A)`, `T(X←B)`, `T(X←C)`, etc. each time from scratch. In the example above we would have

`CallMe(Flag)` = {(U, V): V = U except that the value of `Flag` (V) = T}

so

`CallMe(Next)` ({Next=A}) = `CallMe(Flag)` (Flag←Next)({Next=A})

= {(U, V): V = U except that the value of Next is T}({Next=A})

= {Next=T}

as before.

Unfortunately, this shortcut calculation of a procedure's meaning does not always agree with the definition, so care must be taken in using it. When parameters are "aliased," that is, different identifiers in the declaration become one and the same in the call, the shortcut can go wrong. For example,

```
PROGRAM Alias(INPUT, OUTPUT);
 PROCEDURE MessUp(FilV: TEXT);
  BEGIN
   WRITE('A');
   WRITE(FilV, 'B')
  END;
 BEGIN
  MessUp(OUTPUT);
 END.
```

has (from the declaration)

`MessUp(FilV)` = `WRITEL('A'); WRITEL(FilV, 'B')`

= {(U, V): V = U except that OUTPUT in V has one additional character A appended to its past string; also, FilV, if it is open for writing, has B similarly appended}.

That is, the "parametrized" meaning is that two files each get one character appended. If the formal parameter is replaced by the actual one in this meaning, the result is

`MessUp(OUTPUT)` = `MessUp(FillV)` (`FillV←OUTPUT`)

= {(U, V): V = U except that `OUTPUT` in V has one additional character A appended; also, `OUTPUT` has B similarly appended},

which specifies two contradictory characters as the one to be added to `OUTPUT`, instead of the meaning from the definition

`MessUp(OUTPUT)` = `WRITEL('A'); WRITEL(OUTPUT, 'B')`

that both characters are added to `OUTPUT`. This difficulty with the wrong names for parameters is called aliasing, because one identifier has two names; in the example, `OUTPUT` has its own name, but also the name `FillV`.

It is easy to identify situations in which aliasing might be a problem, and there to use the definition rather than the shortcut. This treatment of aliasing is a strength of the Mills theory compared to the Hoare theory. In the latter the proof rules do not work for aliased parameters, so programs with aliasing cannot be proved; in the former we must only be careful not to take shortcuts in the calculation.

8.2 Identifier Confusion

In pathological cases, the actual parameter identifiers that replace formal identifiers for the meaning of a call may conflict with local identifiers of the called procedure. The ALGOL 60 "copy rule" handles this case by requiring a systematic change of local identifiers that duplicate actual-parameter identifiers.

Another source of potential confusion is Pascal's use of static scope rules: a global identifier copied as part of a procedure declaration can be confused with the same identifier which is part of the calling state, having previously entered as a local. Again the copy rule states that such offending local identifiers must be systematically replaced.

Both kinds of confusion could be eliminated by adjusting the definition of section 8.2 to include a systematic replacement of the offending identifiers in the procedure body. For the nonrecursive case it is easier to just assume that CF Pascal programs do not reuse identifiers. But with recursion, the first pathology is unavoidable, as illustrated by

```
PROCEDURE Self(P: CHAR);
 VAR L: CHAR;
 BEGIN
  Self(L)
 END
```

in which `L` is necessarily confused. The case of recursion treated below indicates how such substitutions can be made in the definition.

When a procedure call occurs within a procedure there are just two possibilities: either this call and others that it may lead to eventually come down to a procedure body in which no calls occur; or, one of the procedures in the potential chain of calls has occurred previously in the chain, and the call is recursive. These alternatives are a consequence of the finite nature of programs: unless one of the procedures is recalled, all possible procedures will eventually be exhausted.

The consequence of blindly applying the definition of Section 8.1, without confusion of identifiers as described in 8.2, is happy: a definition of the meaning of the whole program results without difficulty. Similar application to a recursive sequence is less satisfactory: it results in a recurrence relation in which the function computed by the recursive procedure is defined in terms of itself. This section is devoted to the solution of such recurrence relations. Let us begin by eliminating the intrinsic identifier-confusion introduced by a recursive call.

Suppose a procedure has been declared, e.g.,

```
PROCEDURE Revise(P: CHAR);
  VAR L: CHAR;
  T(P,L)
```

where T(P,L) represents the remainder of the body that follows the local declaration, which may use both identifiers P and L. If **Revise** is called within T(P,L) with actual parameter L, then define

$$\boxed{\texttt{Revise(L)}} = \boxed{\texttt{VAR nL: CHAR; } T(\texttt{P}\leftarrow\texttt{L},\texttt{L}\leftarrow\texttt{nL})}$$

where nL is an identifier distinct from all others in the program, and the substitutions are made simultaneously so as not to interact. Thus the potential confusion of the parameter that must be inserted to obtain the meaning of the call, with the local that must be present, is avoided. At the next recursion, the call

```
Revise(L)
```

is copied into a procedure body with local declaration

```
VAR nL: CHAR
```

so there is no confusion at this level; at the next level the confusion is again eliminated by introducing nL, and so on.

For a recursive procedure call the meaning cannot usually be obtained by applying the definition to the body, since this would lead to an infinite regress trying to expand the recursive call. The situation is rather like the one for WHILE statements, and some of the same ideas can be used. We define the meaning of a procedure statement involving recursion, to be those pairs of states (T, U) such that U results from T by application of a finite number of calls, after which T is so transformed that the recursive call does not occur, so the definition closes. When there is no such finite sequence of calls for an input state T, then the box function of the procedure statement

is undefined at T.

Just as it is not always possible in practice to expand WHILE statements until they terminate, or to see the general case so that the number of iterations can be calculated, so it is not always practical to unwind recursions. It can happen that the values of the expressions controlling the recursion are too complex for analysis, and another technique is needed. For WHILE statements it was enough to unwind the loop once to obtain a recurrence equation; for recursive procedure statements such an equation can be obtained directly. There are already at least two procedure statements in the program: the original invocation, and the recursive call.

As an illustration, the program below "counts" all input files as of zero or nonzero length:

```
PROGRAM LenOIn(INPUT, OUTPUT);
 {Counts the number of characters on INPUT before the
  first #, using the numbers 0, 1, 1, 1, ...}
 VAR
  Nc, Number: CHAR;
 PROCEDURE Count;
  BEGIN
   READ(Nc);
   IF Nc = '#'
   THEN
    Number := '0'
   ELSE
    BEGIN
     Count;
     Number := '1'
    END
  END;
BEGIN
 Count;
 WRITELN(Number)
END.
```

The box function for the outer procedure statement involves the box function for a copy of `Count`'s body, including a procedure statement for `Count` itself. Making the expansion just once, and thus leaving `Count` when it occurs, gives the recurrence equation:

$$\boxed{\texttt{Count}} = \boxed{\texttt{READ(Nc); IF ... Count; Number := '1' END}}$$

$$= \boxed{\texttt{READ(Nc)}} \circ \boxed{\texttt{IF ... Count; Number := '1' END}}$$

The IF statement meaning is

$$\boxed{\texttt{IF Nc = '\#' THEN ... Number := 1 END}}$$

$$= \{(U, V): U(\texttt{Nc}) = \# \text{ and } V = U \text{ except that } V(\texttt{Number}) = 0\}$$

$\cup$ {(U, V): U(**Nc**) $\neq$ # and V = [**Number :='1'**] ([**Count**] (U))}.

Substituting in the original equation gives a formula for [**Count**] in terms of [**Count**] . The solution of such equations is in general difficult, but as for the WHILE statement it is often possible to guess a solution. Here the meaning of a **Count** procedure statement is evidently the function f:

f = {(U, V): V = U except that V(**Nc**) = #, V(**INPUT**) = s#|, and V(**Number**) = 0 if U(**INPUT**) = s|#; but V(**Number**) = 1 if U(**INPUT**) = s|r# , where s is any string and r any nonempty string}.

Substituting f into the equation for [**Count**] it can be seen that f is a solution.

Care is required, as it was with the WHILE statement, however, since the recurrence equation is only a necessary condition. The crucial example is the procedure

```
PROCEDURE Forever;
 BEGIN
  Forever
 END
```

whose recurrence equation is

[**Forever**] = [**Forever**] ,

and any function whatsoever is a solution. The meaning should be the empty function that is everywhere undefined, since this recursion never terminates. This gives us a guess for a verification theorem: the part function of a procedure statement involving a recursive call, is a solution to the recurrence equation whose domain agrees with the one for the code. A slightly stronger statement can be proved.

Theorem (Recursive Verification): Let **Pr** be a procedure statement for a procedure without parameters and including one recursive call, having the recurrence equation

[**Pr**] = M([**Pr**]).

M([**Pr**]) represents the result of calculating the part function of the procedure body, which will contain a single occurrence of **Pr**, which calculation must be carried out until [**Pr**] stands by itself. A function f = [**Pr**] iff:

1) domain(f) $\subseteq$ domain([**Pr**]), and
2) f is a solution to the recurrence equation in [**Pr**] .

Proof. The only-if direction is trivial, since [**Pr**] satisfies both conditions, and hence so does f = [**Pr**] . For the other direction, assume 1) and 2). Let T be any element of domain([**Pr**]).

Hence there is a copying sequence for the body of the procedure (say of length k) such that the recursive call occurs in each copy, then does not occur. That is, $M^k(\boxed{Pr})$ can be evaluated at T because it does not contain $\boxed{Pr}$ --all the procedure statements Pr have been eliminated in favor of the other statements in the procedure body. But since f also solves the equation by 2), we have $f(T) = M^k(\boxed{Pr})(T) = \boxed{Pr}(T)$.

Thus f and $\boxed{Pr}$ agree on domain($\boxed{Pr}$). They could only then disagree if f were defined somewhere that $\boxed{Pr}$ were not. But condition 1 forbids this. QED

Corollary (Recursive Verification): The meaning of a procedure statement Pr for a recursive procedure is the least-defined solution to the recurrence equation.

Proof. The least-defined solution must have a domain that is a subset of $\boxed{Pr}$, because $\boxed{Pr}$ is itself a solution, hence it satisfies the conditions of the theorem. QED

In a practical verification, the theorem is more useful than the corollary, because it is easier to identify the domain of the actual procedure-statement function than to show that a guessed solution to the recurrence equation is the least-defined one. In the example above, it is easy to show that the guess f has the same domain as the procedure-statement function $\boxed{Count}$. Thus the meaning of a recursive procedure statement can be obtained by guessing, then verifying the guess, in a way similar to that used for WHILE-statements in Section 7.

The Theorem has treated only a very special case of recursion in which only a single procedure with no parameters and a single recursion is involved. The technical details of the general case are considerable, since it involves several simultaneous recurrence equations for a multiple recursion, and procedure functions with different parameters in its several occurrences in the equations. For example, consider the case in which procedure P calls procedure Q and vice versa, and P does not call itself, but Q does call itself. Then in working out the meaning of a call on P the meaning of Q will appear, and working this out in turn will yield a recurrence relation defining Q's meaning in terms of itself and P's meaning. This illustrates the general situation: the result of analysis will be a system of m simultaneous equations in m functions, one for each procedure involved in a recursive chain of calls. Substitution may simplify this system (for example, substituting the P equation into the Q equation will eliminate P), but cannot solve it. It is evident that the case of recursion is far more complex than the case of WHILE statements, and a result entirely parallel to the WHILE verification theorem more difficult to obtain.

9 Summary and Conclusions

Using a subset of Pascal, a "program calculus" has been described that assigns functional meaning to programs. For programs containing no loops or recursive procedure calls, the application of this calculus is mechanical; in the more complicated cases, the meanings must be guessed and then checked. To prove a program correct for a specification in the form of acceptable input-output pairs then becomes an elementary set-theoretic problem. The specification is a set of pairs as is the program meaning; the program is correct just in case its meaning is a subset of the specification.

References

1. Naur, P. et al., Revised report on the algorithmic language ALGOL 60, *CACM* 6 (1963), pp. 1-17.

2. Knuth, D.E., Semantics of context-free languages, *Math. Systems Theory* 2 (1968), pp. 127-146.

3. van Wijngaarden, A. et al., Revised report on the algorithmic language ALGOL 68, *Acta Inf.* 5 (1975), pp. 1-236.

4. Wirth, N. and C.A.R. Hoare, A contribution to the development of ALGOL, *CACM* 9 (1966), pp. 413-432.

5. Wegner, P., The Vienna definition language, *Comp. Surveys* (1972), pp. 6-63.

6. Stoy, J.E., *Denotational Semantics: the Scott-Strachey Approach to Programming Language Theory,* MIT Press, 1977.

7. Tennent, R.D., *Principles of Programming Languages,* Prentice Hall, 1981.

8. Mills, H.D., The new math of computer programming, *CACM* 18 (1975), pp. 43-48.

9. Linger, R.C., Mills, H.D., and Witt, B.I., *Structured Programming: Theory and Practice,* Addison-Wesley, 1979.

10. Apt, K.R., Ten years of Hoare's logic: a survey--part I, *TOPLAS* 3 (1981), pp. 431-483,

11. Arden, B.W., The Michigan Algorithmic Decoder, University of Michigan Computing Center, Ann Arbor.

12. Shaw, C.J., JOVIAL--a programming language for real-time command systems, in *Annual Review in Automatic Programming,* Vol. 3, Pergamon Press, 1963, pp. 53-120.

13. Mills, H.D. et al., *Computer Programming,* Allyn and Bacon, to appear.

14. Wirth, N., The programming language Pascal, Acta Inf. 1 (1971), pp. 35-63.

15. Scott, D.S. and Strachey, C., Toward a mathematical semantics for computer languages, *Proc. Symposium on Computers and Automata* (J. Fox, ed.), Polytechnic Institute of Brooklyn, 1971.

16. Turing, A.M., On computable numbers, with an application to the entscheidungsproblem, *Proc. London Math. Society* Ser. 2 42 (1936), pp. 230-265.

17. Kleene, S.C., *Introduction to Metamathematics,* D. Van Nostrand, 1950.

18. Gerhart, S. and Yelowitz, L., Observations of fallibility in applications of modern programming methodologies, *IEEE Trans. Software Engineering* SE-2 (1976), pp. 195-207.

19. Floyd, R. W., Assigning Meanings to Programs, *Proc. Amer. Math. Soc.* 19 (1967), 19-31.

20. Hoare, C. A. R., An axiomatic basis for computer programming, *CACM* 12 (October, 1969), 576-580, 583.