TOWARD A MATHEMATICAL SEMANTICS FOR COMPUTER LANGUAGES

by

Dana Scott

and

Christopher Strachey

Oxford University
Computing Laboratory
Programming Research Group-Library
8-11 Keble Road
Oxford OX1 3QD
Oxford (0865) 54141

Oxford University Computing Laboratory

Programming Research Group

OXFORD UNIVERSITY COMPUTING LABORATORY
PROGRAMMING RESEARCH GROUP
45 BANBURY ROAD
OXFORD

TOWARD A MATHEMATICAL SEMANTICS FOR COMPUTER LANGUAGES

by

Dana Scott
Princeton University

and

Christopher Strachey
Oxford University

Technical Monograph PRG-6
August 1971

Oxford University Computing Laboratory,
Programming Research Group,
45 Banbury Road,
Oxford.

Department of Philosophy,
1879 Hall,
Princeton University,
Princeton, New Jersey 08540.

Oxford University Computing Laboratory,
Programming Research Group,
45 Banbury Road,
Oxford OX2 6PE.

This paper is also to appear in *Proceedings of the Symposium on Computers and Automata*, Microwave Research Institute Symposia Series Volume 21, Polytechnic Institute of Brooklyn, and appears as a Technical Monograph by special arrangement with the publishers.

References in the literature should be made to the *Proceedings*, as the texts are identical and the Symposia Series is generally available in libraries.

ABSTRACT

Compilers for high-level languages are generally constructed to give the complete translation of the programs into machine language. As machines merely juggle bit patterns, the concepts of the original language may be lost or at least obscured during this passage. The purpose of a mathematical semantics is to give a correct and meaningful correspondence between programs and mathematical entities in a way that is entirely independent of an implementation. This plan is illustrated in a very elementary way in the introduction. The first section connects the general method with the usual idea of state transformations. The next section shows why the mathematics of functions has to be modified to accommodate recursive commands. Section 3 explains the modification. Section 4 introduces the environments for handling variables and identifiers and shows how the semantical equations define equivalence of programs. Section 5 gives an exposition of the new type of mathematical function spaces that are required for the semantics of procedures when these are allowed in assignment statements. The conclusion traces some of the background of the project and points the way to future work.

CONTENTS

Toward a Mathematical Semantics for Computer Languages

0. INTRODUCTION. The idea of a *mathematical* semantics for a language is perfectly well illustrated by the contrast between *numerals* on the one hand and *numbers* on the other. The numerals are *expressions* in a certain familiar language; while the numbers are mathematical *objects* (abstract objects) which provide the intended *interpretations* of the expressions. We need the expressions to be able to communicate the results of our theorizings about the numbers, but the symbols themselves should not be confused with the concepts they denote. For one thing, there are many *different* languages adequate for conveying the *same* concepts (e.g. binary, octal, or decimal numerals). For another, even in the *same* language many different expressions can denote the same concepts (e.g. 2+2, 4, 1+(1+(1+1)), etc.). The problem of explaining these *equivalences* of expressions (whether in the same or different languages) is one of the tasks of semantics and is much too important to be left to syntax alone. Besides, the mathematical concepts are required for the *proof* that the various equivalences have been *correctly* described.

In more detail we may consider the following explicit syntax for binary numerals:

NUMERALS

$$\nu ::= 0 \mid 1 \mid \nu 0 \mid \nu 1$$

Here we have used the Greek letter ν as a *metavariable* over the syntactical category of numerals, and the category itself is being given a *recursive definition* in the usual way. Thus, a numeral *is* either one of the digits 0 or 1 or is the result of suffixing one of these digits to a previously obtained numeral. Let the set of all numerals be called Nml for short.

Semantically speaking each of the numerals is meant to denote a unique number. Let N be the set of numbers. (The elements of Nml are expressions; while the elements of N are mathematical objects conceived in abstraction independently of notation.) The obvious principle of interpretation provides a function, the *evaluation mapping*, which we might call $\mathcal{V}$, and which has the functional character:

$$\mathcal{V} : \mathrm{Nml} \to \mathrm{N}.$$

Thus for each $\nu \in \mathrm{Nml}$, the function value

$$\mathcal{V}[\![\nu]\!]$$

is the number denoted by ν.

How is the evaluation function $\mathcal{V}$ determined? Inasmuch as it is to be defined on a recursively defined set Nml, it is reasonable that $\mathcal{V}$ should itself be given a recursive definition. Indeed by following exactly the four clauses of the recursive definition Nml, we are motivated by our understanding of numerals to write:

$$\begin{aligned}
\mathcal{V}[\![0]\!] &= 0 \\
\mathcal{V}[\![1]\!] &= 1 \\
\mathcal{V}[\![\nu 0]\!] &= 2 \cdot \mathcal{V}[\![\nu]\!] \\
\mathcal{V}[\![\nu 1]\!] &= 2 \cdot \mathcal{V}[\![\nu]\!] + 1
\end{aligned}$$

Here on the left-hand side of the equations, $\mathcal{V}$ is being applied to expressions; while on the right-hand side the values are given. To borrow the relevant terminology from logic, the numerical ex-

pressions belong to the *object language*; whereas the definition of $\mathcal{V}$ is given in the *metalanguage*. To be able to write down explicitly the definition of $\mathcal{V}$, some metalinguistic symbolization is of course required. The metalinguistic expressions must at all cost be distinguished from those in the object language. (We have put the object language 0 and 1 in Roman type-face and the metalanguage *0*,*1*,*2* in italics. Logicians often take further precautions by enclosing the object language expressions in quotes; the quotation-expressions can then be regarded as part of the metalanguage, and thus the languages are "insulated" from one another. In this paper, however, our object languages are simple enough making the use of such devices less critical. The separation needs to be observed nevertheless, and in the semantic equations we have enclosed the object language expressions in the special brackets [ ] merely as an aid to the eye.)

Granted that there is a distinction between symbol and object, it may still seem that the above equations for $\mathcal{V}$ are circular or nearly vacuous in content. Such a conclusion is wrong, however, because there is an easily appreciated point to the definition: namely, the explication of the *positional notation*. In our metalanguage we need never have heard of decimals or binaries. We do require, though, the concept of *number*, the concepts of *zero* and *one*, the concepts of *addition* and *multiplication*. (By definition *2=1+1*, say, and if we want, the whole *theory* of numbers could be conveyed in the metalanguage with the help of *Roman numerals* augmented with a few tricks from algebra such as the use of *operation symbols* and *variables*.) These concepts, fundamental as they are, cannot be strictly said to *imply* the positional notation. In fact, the clever use of 0 to help form strings of digits was a *discovery of language*. This discovery in no way changed the abstract nature of number, but it was a tremendous help in popularizing the use of arithmetic ideas - and there seems to be a perfectly good parallel here with computer languages many of which contain in their syntactical structures quite as clever discoveries of language.

One point that encourages confusion in thinking about numbers is the possibility of having a *complete* and *canonical* naming system for them. In the illustrative syntax for binary numerals we have been considering, such strings as 001101 were allowed. As everyone knows the initial run of 0's is unnecessary for we can show that

$$\mathcal{V}[\![001101]\!] = \mathcal{V}[\![1101]\!] = \text{XIII}.$$

This type of straight-forward deletion gives us the only possible equivalences in this very simple language. The *reduced* numerals (i.e., numerals of the forms 0 and 1ν, where $\nu \in \text{Nml}$ is arbitrary) are then in a one-one correspondence with the numbers. We can then work exclusively with these normal forms, and it is so easy to think of these expressions as *being* the numbers - especially if one is familiar with only one notational system. The attitude is wrong-headed, however. But for many activities there may be no real harm, since the confused mind will give the *same answers* as the clear-headed person. The notationally bound thinker may often be distinguished by the way he feels that he has to specify all his operations by algorithmic symbol manipulations (as in digitwise addition of numerals). Again there may be no real harm in this - if the algorithms are correctly given. And in the case of numbers the two approaches can be brought together (the symbolic and the conceptual), for our system of reduced numerals can, by a slight amount of good will, be regarded as a *model* for number theory. Since we know that *all* such models are isomorphic, there is not much mathematical advantage in using one model over another. This is *a* sense in which numbers can be consistently confused with numerals. But the confusion does not really do us any good either.

The reasons why the number/numeral confusion should be avoided are many. For one reason, we may turn the isomorphism argument the other way round: If all models of number theory are isomorphic, you may *not* want to single out a particular one at all. Because the semantical ideas will equally well apply to all of them, you may want to leave open the possibility of *shifting* from one to another. This is somewhat analogous in computer language semantics

to allowing different representations to be used in different machines to implement the same language (in hopefully isomorphic ways). In the case of numbers not much advantage is bought by this freedom, but any attitude of restricting generality is a bad habit which can be misleading in analogous but more complicated situations.

A more important reason for not getting into this habit comes to the fore when one realizes that for some systems of mathematical concepts no fully adequate notational system is possible: the real numbers are the prime example. Of course it will be objected that this realm of mathematics is much too abstract, much too infinitary, much too distant from real-life computation. This objection cannot stand careful conceptual investigation, but a full answer would take us too far from the topic of this paper. A quite adequate answer concerns not the mathematically very pure structures such as the real numbers, but rather our theories of *classes* of similar but different structures. That is to say, for the semantic investigation of certain language features it may not be appropriate to single out one (isomorphism type of) structure, but for many reasons - generality, lack of knowledge, for the sake of experiment - we may want the *same* semantical equations to be employed over the *whole class* of structures. Since the structures need no longer be isomorphic, different structures may lead to quite different normal forms for expressions. (The mathematical theory of groups, for instance, could provide us innumerable such examples.) Hence, no one system of "numerals" would any longer suffice. Even if the separate structures could somehow each be symbolically constructed, the effort would be beside the point: what we are trying to get at are the *common features* of the structures. The various *ad hoc* details would only detract from this higher purpose.

To bring this introductory sermon to a close: the point of our approach is to allow a proper balance between rigorous formulation, generality of application, and conceptual simplicity. One essential achievement of the method we shall wish to claim is that

by insisting on a suitable level of abstraction and by emphasizing the right details we are going to hit squarely what can be called *the* mathematical meaning of a language. In the trivial example of the binary numerals discussed above everyone will agree that the evaluation function $\mathfrak{V}$ is indeed correctly defined. That much is obvious. Note, however, that having accepted this fact, it is then possible to *prove* that certain numerical algorithms are *correct* (digitwise addition, say), and before we had the definition this question did not even make sense. (Specifically, digitwise addition is an operation $\nu_0 \oplus \nu_1$ defined on numerals ν_0, ν_1. What one needs to prove is that

$$\mathfrak{V}[\nu_0 \oplus \nu_1] = \mathfrak{V}[\nu_0] + \mathfrak{V}[\nu_1]$$

with symbolic addition on the left and conceptual addition on the right. It is not difficult to do this, but one needs an indnctive argument.) These are simple points, but it is easy to lose sight of them when the languages get involved.

1. STATES AND COMMANDS We begin by postulating that the interpretation of the language depends on the *states* of "the system". That is to say, computer oriented languages differ from their mathematical counterparts by virtue of their *dynamic* character. An expression does not generally possess *one* uniquely determined value of the expected sort, but rather the value depends upon the state of the system at the time of initialization of evaluation. What increases the dynamic character of the evaluation process is the fact that the act of evaluation may very well alter the initial state. Thus the working out of a compound expression can require several changes of state, and the treatment of a subexpression generally has to wait for the moment at which the state can be provided appropriate to its evaluation. Therefore the "algebra" of equivalences of such expressions need not be as "beautiful" as the well-known mathematical examples. This does not mean that the semantics of such languages will be *less* mathematical, only an order more complex.

Part of our assumption is that the states of the system form a *set* S, and the dynamic character of the language will require us to consider *transformations* of this S into itself: the state transformations. For the moment let us write (in our metalanguage)

$$[S \to S]$$

for the set of *all* state transformations (this set may require restriction later). By a *transformation* $f \in [S \to S]$ we understand the ordinary concept of a mathematical *function* defined on S with values in S. Functions in the mathematical sense are abstract objects - they can be *defined* in various linguistic forms, but after the definition is interpreted all that is left is the bare *correspondence* between *arguments* $\sigma \in S$ and *values* $f(\sigma) \in S$. In particular two functions which assume the same values for the same arguments are mathematically identical - even though they might have been defined in some object language in quite different ways.

The simplest way to regard the state transformations from σ to $f(\sigma)$ is that they give the results of executing a *command*. No explicit values are required; one is merely being asked to "move along". Of course certain "values" may be implicit in σ,

and they may be changed in passing to $f(\sigma)$ (e.g. ones *position*). The command, however, is concerned more with the overall change; other kinds of expressions can be used to extract from a state $\sigma \in S$ any relevant values. But one syntactic category in our language will be that of commands; let us call the set of these expressions Cmd. Given $\gamma \in$ Cmd, no matter how complex, the mathematical meaning of this expression is an associated state transformation $\mathcal{C}[\![\gamma]\!] \in [S \to S]$. That is, the semantics of commands is to provide us with a mapping:

$$\mathcal{C} : \text{Cmd} \to [S \to S],$$

just as the semantics of numerals gave us:

$$\mathcal{N} : \text{Nml} \to N.$$

What is vague here is that we have no idea what commands are, whereas numerals were standard. That we have at this stage no idea what states $\sigma \in S$ are is far less serious, because that is the part of the interpretation we are on purpose leaving open.

It is not difficult to be more explicit about the syntax of commands, however, because there are several quite natural ways of combining them. An initial syntax might look as follows:

COMMANDS

$$\gamma ::= (\gamma) \mid \phi \mid \text{dummy} \mid$$
$$\varepsilon \to \gamma_0 , \gamma_1 \mid \gamma_0 ; \gamma_1$$

Here the Greek letter γ is a metavariable over the category (Cmd) which is being given a recursive definition. On the right-hand side of the definition the $\gamma, \gamma_0, \gamma_1$ can be regarded as *previously* obtained commands, where the subscripts are required in a binary composition to allow for *different* commands to be chosen. If in one clause of the definition the *same* γ appeared twice, we would intend that the same previously obtained expression be used in both positions. There is no implied connection between the γ's in the separate clauses. The letters ϕ and ε are meant to refer to other syntactic categories yet to be explained. The expression dummy is a *constant* command (an "atomic" command expression).

As anyone can see (assuming that the categories of ϕ and ε

are simple) the set Cmd is going to be a context-free language. Also obvious is the fact that the language is *ambiguous* - thus $\gamma_0;\gamma_1;\gamma_2$ can be parsed in two different ways at least. Machines generally prefer their languages *unambiguous*; while humans enjoy a little uncertainty, or at least they find ways to overlook ambiguities by giving each other the benefit of the doubt. Sometimes ambiguities make no difference (as in $\gamma_0;\gamma_1;\gamma_2$), but at other times they are quite tiresome (as in $\varepsilon \to \gamma_0,\gamma_1;\gamma_2$). We shall deal with this problem presently, but in the meantime note the clause (γ) in the definition. This clause allows us to form $(\varepsilon \to \gamma_0,\gamma_1);\gamma_2$ or $\varepsilon \to \gamma_0,(\gamma_1;\gamma_2)$ which are similar to the ambiguous expression but which have one chance less of being ambiguous. With a sufficient nesting of parentheses all ambiguity can be eliminated; or, speaking more precisely, there is a completely unambiguous sublanguage of Cmd. The only trouble with us humans is that the majority find writing in this sublanguage a terrific bore; hence the tendency to the more dangerous syntax.

Before we can be more precise about Cmd and the interpretation $\mathcal{C}$, we have to discuss the ε. For the time being $\varepsilon \in$ Exp, the class of *Boolean expressions* (we shall allow other types of expressions later). As a starter at a syntax we can write:

EXPRESSIONS

$$\varepsilon ::= (\varepsilon) \mid \pi \mid \mathbf{true} \mid \mathbf{false} \mid \varepsilon_0 \to \varepsilon_1,\varepsilon_2$$

The same remarks about ambiguity apply. The π are certain atomic expressions which we shall not stop to detail now. The Boolean expressions true and false are constants, and $\varepsilon_0 \to \varepsilon_1,\varepsilon_2$ is the well-known conditional expression (short for: if ε_0 then ε_1 else ε_2, which some may prefer).

What of interpretation? In the first place we postulate the set T of *truth values* which contains the elements *true* and *false* corresponding to true and false. But the meaning $\mathcal{E}[\![\varepsilon]\!]$ is not going to be simply an element of T, for in general values must depend on states. Besides this evaluation may *cause* a change of state. Thus the correct functional character of $\mathcal{E}$ is:

$$\& : Exp \to [S \to [T \times S]].$$

This means that given $\epsilon \in Exp$ and $\sigma \in S$, then

$$\&[\![\epsilon]\!](\sigma) = \langle t, \sigma' \rangle$$

where $t \in T$ is the value of ϵ given σ, and σ' is the resultant state after the evaluation and may differ from σ as it may have been changed by the evaluation. (Here $T \times S$ is the usual *cartesian product* of sets and $\langle \cdot, \cdot \rangle$ is the *pairing* function.)

Before giving the clauses that define $\mathcal{C}$ and $\&$, it is useful to introduce several mathematical operations on functions. If $f : B \to C$ and $g : A \to B$, then as usual we write $f \circ g : A \to C$ for the *composition*, where

$$(f \circ g)(\alpha) = f(g(\alpha))$$

for all $\alpha \in A$. We have further

$$P : A \to [B \to [A \times B]]$$
$$M_0 : A \times B \to A, \text{ and}$$
$$M_1 : A \times B \to B,$$

where

$$P(\alpha)(\beta) = \langle \alpha, \beta \rangle,$$
$$M_0(\langle \alpha, \beta \rangle) = \alpha, \quad \text{and}$$
$$M_1(\langle \alpha, \beta \rangle) = \beta$$

for all $\alpha \in A$, $\beta \in B$. Finally, if $p : A' \to [B \to C]$ and $q : A \to A' \times B$, then $p * q : A \to C$, where

$$(p * q)(\alpha) = p(M_0(q(\alpha)))(M_1(q(\alpha)))$$

for all $\alpha \in A$. On all of these operations and special functions we ought to write some kind of ABC subscripts, because strictly speaking they depend on the choice of the various sets; we have omitted these subscripts as they can easily be deduced from the context. The same goes for the *identity* function

$$I : A \to A,$$

where

$$I(\alpha) = \alpha$$

for all $\alpha \in A$.

Turning now to the mapping & we have clause by clause:

$$\&[\![(\varepsilon)]\!] = \&[\![\varepsilon]\!]$$

because parentheses add nothing to meaning.

$$\&[\![\pi]\!] = (\text{some given } S \to [T \times S]),$$

because each atomic expression π must have a given meaning. (Which meaning this is, need not concern us here, since at the moment all we consider is the *form* of the semantic definition.)

$$\&[\![\mathsf{true}]\!] = P(\mathit{true}), \text{ and}$$
$$\&[\![\mathsf{false}]\!] = P(\mathit{false}),$$

because true and false are constants which can be evaluated "instantaneously" without change of state. This means that for all σ,

$$\&[\![\mathsf{true}]\!](\sigma) = \langle \mathit{true}, \sigma \rangle \text{ and}$$
$$\&[\![\mathsf{false}]\!](\sigma) = \langle \mathit{false}, \sigma \rangle.$$

$$\&[\![\varepsilon_0 \to \varepsilon_1, \varepsilon_2]\!] = \mathit{Cond}(\&[\![\varepsilon_1]\!], \&[\![\varepsilon_2]\!]) * \&[\![\varepsilon_0]\!],$$

where the function

$$\mathit{Cond}: [S \to T \times S] \times [S \to T \times S] \to [T \to [S \to T \times S]]$$

is such that

$$\mathit{Cond}(e_1, e_2)(t) = t \to e_1, e_2$$

so that

$$\mathit{Cond}(e_1, e_2)(t)(\sigma) = \begin{cases} e_1(\sigma) & \text{if } t = \mathit{true}, \\ e_2(\sigma) & \text{if } t = \mathit{false}, \end{cases}$$

for all e_1, e_2, t, σ in appropriate sets.

It is now possible to give the clauses of the definition of $\mathfrak{C}$:

$$\mathfrak{C}[\![(\gamma)]\!] = \mathfrak{C}[\![\gamma]\!],$$
$$\mathfrak{C}[\![\phi]\!] = (\text{some given } S \to S),$$
$$\mathfrak{C}[\![\mathsf{dummy}]\!] = I,$$
$$\mathfrak{C}[\![\varepsilon \to \gamma_0, \gamma_1]\!] = \mathit{Cond}(\mathfrak{C}[\![\gamma_0]\!], \mathfrak{C}[\![\gamma_1]\!]) * \&[\![\varepsilon]\!],$$
$$\mathfrak{C}[\![\gamma_0 ; \gamma_1]\!] = \mathfrak{C}[\![\gamma_1]\!] \circ \mathfrak{C}[\![\gamma_0]\!].$$

These functional equations can all be read in words: $\mathcal{C}[\![(\gamma)]\!]$ is of course the same as $\mathcal{C}[\![\gamma]\!]$. Next $\mathcal{C}[\![\phi]\!]$ is a given state transformation, since ϕ is taken for the moment as a primitive (or atomic) command. (Later we could introduce some syntax for the category of ϕ if we wished, and this would require further semantic equations of a similar sort.) The dummy-*command* is next being interpreted as the "do nothing" transformation. The *conditional-command* on the other hand requires a conditional operator similar to the one used for expressions, except that the domain of definition is altered to make it appropriate to commands. Specifically we have:

$$Cond : [S \to S] \times [S \to S] \to [T \to [S \to S]].$$

Some subscripts ought to be introduced to indicate the difference between the two kinds of *Cond*-functions, but we are relying on the context to make this clear. Finally $\gamma_0;\gamma_1$ is being interpreted as the *sequencing* operation on commands, which in terms of state transformations is simply the *composition of functions*. Note, however, the *order* of application. The "positional notation" of the command language conventionally places the first command to be executed on the left and the following one on the right. The convention with mathematical composition of functions is just the opposite. (The reason being the common use of $f(x)$ rather than xf for the function-value notation.)

In format these equations for $\mathcal{C}$ and $\mathcal{E}$ are not much different from those for $\mathcal{V}$ as applied to the numerals. In all cases there is a syntactical definition in several clauses. The semantical definition is "syntax-directed" in that it follows the same order of clauses and transforms each language construct into the intended operation on the meanings of the parts. In the case of numerals the meanings were familiar mathematical objects on which many familiar operations (addition, multiplication, etc.) were already defined. In the case of commands and Boolean expressions, the values were not so well known, nor were the operations (such as *Cond*) very common. Nevertheless, the domains $[S \to S]$ and $[S \to T \times S]$ are quite appropriate to the ideas that are being

explained, and the various operations ($\circ, *, P, M_0, M_1$) are natural ones for these domains.

That it is necessary to construct meanings out of *functions* ([S → S]) may seem bothersome, but it should not be deplored. After all the idea of function is one of the greatest of our mathematical discoveries. The Calculus would be impossible without functions, and the development of the subject would be unthinkable without the use of various operators (derivative, integral, etc.) and consideration of the equations involving them. Though formal languages are not generally made explicit, still there is constant translation from intuitive ideas to mathematical concepts (*velocity* means derivative, *area* means integral, *superposition* means addition of functions, and so on). In the case of computer languages we cannot say yet that we have introduced concepts anywhere near as important as those of the Calculus, but the *spirit* of mathematizing ideas can and ought to be carried over. That the project is a useful activity remains to be demonstrated, but the treatment of *recursion* in the next section ought to indicate some of the advantages of the mathematical approach.

Before turning to more important concepts, the question of *ambiguity* must be faced again. We have allowed our grammars to be ambiguous, and so strictly speaking the semantical mappings $\mathcal{C}$ and $\mathcal{E}$ are not well defined. (Von Nml was well defined because that grammar was not ambiguous.) Is there a mistake here? We think not. Our attitude is that the meaning of an expression depends on the way it is parsed. From this point of view the mappings $\mathcal{C}$ and $\mathcal{E}$ are defined *not* on expressions *simpliciter* but rather on the annotated *deduction trees* for expressions based on rules of the grammar. (If one wishes a linear notation, a fully bracketed language could be introduced in the usual way, and then the expressions we have written above would result by deletion of the brackets and subscripts.) As emphasized above these brackets (or trees) are intolerable to write; hence, as long as we keep our semantic equations "in step" with the syntactical defini-

tions and warn the reader of the oversimplification, it will be clear what is intended without the burden of any more notation. (In the terminology of Knuth [2, 3] we are still using only *synthesized* attributes in assigning meanings to expressions. Whether the *inherited-attribute* approach is convenient is something we must consider further, and there is no doubt that Knuth's way of introducing both bottom-up and top-down dependencies between semantic equations is an interesting notion. Be that as it may, the point of our paper is the study of *what* it is that semantic equations assign to expressions. The *path* by which the assignment is made is at the moment a secondary issue.)

2. RECURSION. The language of commands and expressions as introduced in the last section is at best of illustrative value. It was useful to see in not too horrible detail the connections between the syntactical and semantical equations. But the actual language used was of very limited expressive power. The command structure was of a very *direct* sort: the execution of such a command would procede along a "branch" of a *tree*, the commands being composed in sequence and the choice of path at a branch point being decided by evaluation of a conditional expression. In Scott [6] the expressive power of such a language was expanded by the introduction of a certain kind of *infinite* tree, but the mathematics of this approach would take us too far afield here. And in any case, it is rather fully explained in that paper. Our interest here lies in more conventional language features; in particular those that can be written down in finite space. This does not mean that thinking of finite expressions as being "unrolled" into infinite trees might not be mathematically illuminating, rather we do not have the time to discuss it in this paper.

The question that leads to infinite trees is of course that of *recursively defined* commands. By way of example suppose that exit is a primitive command, as is fudge. Moreover, suppose that test is a primitive Boolean expression. One would wish then to introduce a command *loop* by the equation:

$$loop = (\text{test} \rightarrow \text{fudge}; loop, \text{exit})$$

In other words, to loop means to alternate testing and fudging until a negative test is produced. At that moment a hasty exit is then required without further fudging. This is one of the simplest and most familiar examples. More generally it will be desired to introduce a whole *sequence* of interrelated commands $\xi_0, \xi_1, \ldots, \xi_{n-1}$ by a system of eqnations:

$$\xi_0 = \gamma_0, \ \xi_1 = \gamma_1, \ldots, \ \xi_{n-1} = \gamma_{n-1},$$

where we may think of ξ_0 as the *principal* command to be executed and the remaining ξ_i as *auxiliary* commands introduced to aid the definition. Now the γ_j command expressions will involve reference

to the ξ_i commands, just as in the *loop* example. Sometimes it is possible to eliminate the auxiliary commands at the expense of perspicuity by substitution of one equation in another. But at other times the elimination is not possible or not at all obvious. In any case it is a language feature of well-known convenience to allow simultaneous equations. All of this is very familiar ground.

Syntactically, to accommodate this recursive style of command definition, we must expand the language to allow for (temporary) *identifiers* which will refer to these auxiliary commands, the precise reference being controlled by a scheme of *declaration*. The exact style of identifiers need not concern us here; all we require is a syntactical category Id of expressions distinct from the other kinds of expressions mentioned so far. Besides this Id should be infinite to allow for as many auxiliaries as we please. We use the Greek ξ to range over Id.

Having provided identifiers, we then need to combine them with other command expressions. Of course an identifier standing alone will be allowed in the expanded Cmd. In this way identifiers can be included in the interiors of commands. Further the system of equations indicated above can be recorded in the following declaration scheme:

$$\S\ \xi_0,\xi_1,\ldots,\xi_{n-1} : \gamma_0,\gamma_1,\ldots,\gamma_{n-1}\ \S$$

where we have slightly torn apart the equations. The initial list of identifiers tells us to "watch out" because some auxiliaries are being introduced - in the order given. The following sequence of commands tells us how to use these auxiliaries with $\xi_i = \gamma_i$ intended - in the order as written. Recursion is permitted because the various ξ_i may occur within the γ_j. Having gone to the trouble of writing ξ_0 first, we may not only regard this expression as a scheme of declaration but also as an instruction to carry out ξ_0 *first*, thereby activating the other ξ_i as necessary under the control of the γ_j. (The pretty brackets § and § are not strictly necessary, since we could regard : as the declaration operation and employ the colourless (and) to block ambiguity - but one can carry

linguistic stinginess too far.)

This expansion of expression now requires a revision of the syntax for commands whereby the principal syntactical equation (for γ) is accompanied by some auxiliary equations for sequences (namely ξ^n and γ^n):

COMMANDS

$$\gamma ::= (\gamma) \mid \phi \mid \text{dummy} \mid \xi \mid$$
$$\varepsilon \rightarrow \gamma_0, \gamma_1 \mid \gamma_0 ; \gamma_1 \mid$$
$$\S\ \xi^n : \gamma^n\ \S$$

$$\xi^n ::= \xi_0, \xi_1, \ldots, \xi_{n-1} \quad (n>0,\ \text{the } \xi_i \text{ distinct})$$
$$\gamma^n ::= \gamma_0, \gamma_1, \ldots, \gamma_{n-1} \quad (n>0,\ \text{the } \gamma_j \text{ arbitrary})$$

A word of explanation is in order here. Greek ξ is a metavariable over identifiers, while ξ^n by definition is a metavariable over n-tuples of *distinct* identifiers. (We keep $n>0$, so the n-tuples are nonempty.) The metavariable γ^n ranges over n-tuples of commands (again, $n>0$). In the last clause of the definition for γ note that the ξ^n and γ^n have the *same* n. Our language therefore is no longer context-free. But, if we may say so, who cares? Context-free languages have limited usefulness. Note, too, that we have not tried to torture ourselves with too rigorous a style of BNF syntactical definition. We deny that our syntax is unrigorous or even unaesthetic. On the other hand if someone has a really neat language definition system that is as easy to comprehend at this level of discussion, we shall be glad to consider it. The last thing we want to be is dogmatic about *language*: it is in the mathematization of *concepts* that we have a certain amount of dogma to sell.

For the time being we introduce no revision in the definition of Boolean expressions ε. Note that the command construction

$$\S\ \xi^n : \gamma^n\ \S$$

is, logically speaking, a *variable-binding operator*. The identifiers ξ^n are the *bound* variables (and, since a matching with the γ^n is intended, they must be kept distinct); whereas other identifiers which occur may occur as *free variables* because the construction can be iterated. A certain semantical device will have

to be introduced to handle this problem of scope of identifiers.

There are other problems, however, and the *loop*-example will suffice for illustration. In the official notation there are no equations for commands as such; rather our example above becomes

$$\S\ loop\ :\ (\mathsf{test} \to \mathsf{fudge};\ loop,\ \mathsf{exit})\ \S$$

which is a command with the understanding that $loop \in \mathrm{Id}$. What is the exact meaning of this command? Whatever it is our dogma says it must be a state transformation in $[S \to S]$. Let the command be called λ for short. We are asking what $\mathcal{C}[\![\lambda]\!]$ should be. Intuitively we want

$$\begin{aligned}\mathcal{C}[\![\lambda]\!] &= \mathcal{C}[\![\mathsf{test} \to \mathsf{fudge};\ \lambda,\ \mathsf{exit}]\!] \\ &= \mathit{Cond}(\mathcal{C}[\![\lambda]\!]\circ\mathcal{C}[\![\mathsf{fudge}]\!], \mathcal{C}[\![\mathsf{exit}]\!])\ast\mathcal{E}[\![\mathsf{test}]\!]\end{aligned}$$

To simplify our thinking here let:

$$\begin{aligned}\ell &= \mathcal{C}[\![\lambda]\!], \\ f &= \mathcal{C}[\![\mathsf{fudge}]\!], \\ e &= \mathcal{C}[\![\mathsf{exit}]\!], \quad \text{and} \\ t &= \mathcal{E}[\![\mathsf{test}]\!],\end{aligned}$$

where $\ell, f, e \in [S \to S]$ and

$t \in [S \to T \times S]$.

Of these f, e, and t are *known*, while ℓ is the *unknown* we seek. The functional equation for ℓ reads:

$$\ell = \mathit{Cond}(\ell\circ f, e)\ast t.$$

Some solution or other to this equation - if any exists - has the right to be called $\mathcal{C}[\![\lambda]\!]$, the meaning of the *loop*-command as a state transformation.

Now comes the rub. So far we have not analyzed the nature of the set S at all, because we opted for extreme generality. If we stick to this generality and allow the functions f, e, and t to be *arbitrary* functions, then it is easy to construct an example where *no* solution for ℓ exists in the above equation at all. The reason is simple: f, e, and t are *total* functions so interrelated that any attempt to define ℓ as required sets up an *infinite loop*: so that *no* choice of values can be made to satisfy the equation

as a functional equation between total functions.

The solution to this problem is easy enough and is well known: we modify our idea about the function space $[S \to S]$. We no longer demand that functions be total but understand the functions in $[S \to S]$ to be partial functions. Thus for certain $\sigma \in S$ and certain $g \in [S \to S]$ we allow $g(\sigma)$ to be undefined. With this convention it can then be shown that the equation for i does indeed have a solution in the partial function sense, and in fact it has a *least* solution. (By *least* we mean that the "graph" of the function is included in every other solution.) This approach is that of Park [4] and many others. (See references in Engeler [1].) Suitable as it is for many purposes and simple as it is, it is not quite the method we wish to adopt. Our method is related, but it is made a little more sophisticated in order to supply a closer analysis of the nature of the elements of S which is required for the explanation of other language features.

3. LATTICES AND FIXED POINTS. In the last section we found it necessary to expand $[S \to S]$ to allow for partial functions. The set of all partial functions is *partially ordered* by the relationship of one function's being *included* in the other. Under this partial ordering the set $[S \to S]$ takes on a structure which has quite pleasant properties. These properties can be formulated in an abstract way, so that the proof of the existence of solutions to fixed-point equations can easily be given. In order to regularize and generalize this argument, it turns out to be natural to derive the structure on $[S \to S]$ from structure on S. This is accomplished by *expanding* S until it becomes a partially ordered set itself - in fact, S will be made into a *complete lattice*. Just how this construction of an expanded S is to be done requires a closer examination of the kind of elements S should have. We will have to return to this question in more detail in §5. For the time being suppose that the expansion has been made.

Speaking a bit more generally for the moment, the structure of a complete lattice on a particular domain (set) D requires first a partial ordering which we write as

$$x \sqsubseteq y$$

for $x,y \in D$. This relationship is *reflexive*, *transitive*, and *anti-symmetric*. Next, if $X \subseteq D$ is a subset of D, we assume the existence of an element of D, called the *least upper bound* (*lub*) of the subset X, which we write as:

$$\bigsqcup X.$$

We have for all $y \in D$

$$\bigsqcup X \sqsubseteq y \text{ iff } x \sqsubseteq y, \text{ for all } x \in X;$$

and this condition uniquely characterizes $\bigsqcup X \in D$. A complete lattice *is* a partially ordered set in which *lub*'s always exist.

Among the *lub*'s in a complete lattice there are two extreme ones: the *lub* of *nothing* and the *lub* of *everything*. That is to say, the empty subset $\emptyset$ and the full subset D will both have *lub*'s to which we give special names:

$$\bot = \bigsqcup\emptyset \text{ and } \top = \bigsqcup D.$$

Note that for all $x \in D$ it is the case that

$$\bot \sqsubseteq x \sqsubseteq \top.$$

We can think of $\bot$ as the *weakest* element and $\top$ as the *strongest* element of D. The *ordinary* elements are somewhere in between, and $\bot$ and $\top$ should be considered rather extraordinary. (We can call them *bottom* and *top*.)

An intuitive way of reading the relationship $x \sqsubseteq y$ is to say that x *approximates* y. Thus x is worse and y is better. But take care, the sense of approximation being used here is a *qualitative one* of what we might style *direct approximation*. The statement $x \sqsubseteq y$ does *not* mean that x is *very near* y, but rather that x is a *poorer version* of y, that x is only partially specified and that it can be *improved* to y without changing any of the definite features of x. For example in the case of partial functions, $\sqsubseteq$ means *inclusion* of graphs (the graph of a function is just the set of ordered pairs of arguments and function values); hence, *improvement* means *adding* new ordered pairs. The smaller set of ordered pairs can indeed be said to be an approximation to the larger one. (In the case of partial functions treated by graphs in the ordinary way, the structure becomes a lattice only when $\top$ is added in a somewhat artificial way as a *top* element which is not represented as a set of ordered pairs. We shall discuss partial functions in a slightly different way below.) Additional examples of approximations treated in this way can be found in Scott [5] and [6].

If we take the notion of approximation seriously, then we have to rethink what we mean by *function*. Thus if

$$f : D \to D$$

and

$$x \sqsubseteq y,$$

then f should not juggle x and y around in too arbitrary a fashion. Indeed it ought to follow that

$$f(x) \sqsubseteq f(y);$$

because *if* we improve x to y, *then* in "calculating" $f(y)$ the calculation should be just an improvement over that for $f(x)$. Mathematically speaking the reasonable functions ought to be *monotonic* (i.e., $\sqsubseteq$ preserving).

Besides the intuitive motivation for monotonic functions, we have the well-known mathematical fact that *monotonic functions on complete lattices always have fixed points.* They even have *least* fixed points. This makes their use most convenient for our purposes.* Actually the functions we use - and which are appropriate to computation theory - have an even stronger property: they are *continuous*. (See the discussion in Scott [5] and [6].) We shall assume this stronger property but shall not go into the technical details in this paper. The reader should only be assured that normal functions are automatically continuous.

What does all this theory have to do with the subject of semantics? Step by step the relevance is this: Commands (programs) are naturally thought of as defining state transformations. Recursive commands require partial functions. Solving for these partial functions is just finding (minimal) fixed points in certain functional equations. In general the existence of fixed points is justified by a lattice-theoretic argument. Therefore, if we can see the connection between lattices and partial functions, the relevance of the theory will be established.

Returning to S, we promised to expand it to a lattice. This can be done in many ways, but for simplicity suppose that the initial version of S was just an abstract set, S_0, say. In S_0 we assume no particular connections between the elements for the sake of argument. The expanded S results merely by the adjunction of the two, new "ficticious" elements $\perp$ and $\top$. The partial ordering $\sqsubseteq$

---

* The argument for fixed points is as follows. Let $f : D \to D$ be monotonic. Let $Y \subseteq D$ be the subset of all $y \in D$ such that $y \sqsubseteq z$ whenever $f(z) \sqsubseteq z \in D$. Let $x = \bigsqcup Y$. To show that $x = f(x)$, note first that $x \in Y$; because if $f(z) \sqsubseteq z$, then $y \sqsubseteq z$ for all $y \in Y$, so $x \sqsubseteq z$. Next note that $f(x) \in Y$; because if $f(z) \sqsubseteq z$, then $x \sqsubseteq z$, and so $f(x) \sqsubseteq f(z) \sqsubseteq z$ by monotonicity. Therefore $f(x) \sqsubseteq \bigsqcup Y = x$. But then $f(f(x)) \sqsubseteq f(x)$, again by monotonicity, so $x \sqsubseteq f(x)$ because $x \in Y$. Thus $x = f(x)$.

aside from satisfying the usual axioms, provides in addition only the relationships:

$$\perp \sqsubseteq x \sqsubseteq \top.$$

(For pictures of these and other partial orderings consult Scott [S] and [6].) This expanded S becomes a complete lattice in a rather trivial way, and the construction should not be taken as being typical.

The function space $[S \to S]$ is now regarded as being the set of *all monotonic functions* from S into S. (In more interesting lattices we shall restrict our function spaces to the continuous functions; in this example the restriction makes no difference.) For $\sigma \in S$ and $g \in [S \to S]$ when we formerly wrote that

$$g(\sigma) \text{ is } \textit{undefined}$$

we shall now write simply

$$g(\sigma) = \perp.$$

The new element $\perp$ can be regarded as an "embodiment" of the undefined. (The companion equation $g(\sigma) = \top$ could be read "$g(\sigma)$ is *overdefined*", but the utility of this concept is not as obvious.) Now if $f, g \in [S \to S]$ are any two functions, we can write

$$f \sqsubseteq g$$

to mean that

$$f(\sigma) \sqsubseteq g(\sigma)$$

for all $\sigma \in S$. This definition at once structures $[S \to S]$ as a partially ordered set and indeed as a complete lattice. This is a natural definition for g's being an improvement over f, if one reads it in words, and it corresponds to our previous ideas about functions. Thus if $f(\sigma) = \perp$ (is undefined), then $g(\sigma)$ is *unrestricted* and can be any element of S. If $f(\sigma)$ is better defined (say, $f(\sigma) = \sigma' \in S$), then $g(\sigma)$ can only be σ' or $\top$ if the relationship $f \sqsubseteq g$ is going to hold. Hence $f \sqsubseteq g$ means just about what we intended when we said that the ordinary graph of f is included in that of g.

Note that by the embodiment of $\perp$, what used to be *partial* functions are now *total* functions in the expanded sense, because $g(\sigma) = \perp$ is an allowed "value". This may seem like a silly thing to do, but the main mathematical point is that the lattice structure on $[S \to S]$ is now *derived*, by means of a simple definition, from the lattice structure on S. *And* by the very same regular process we can provide lattice structure on $[S \to S] \to [S \to S]$, and in general on *any* $[D \to D']$ - always remembering to use the set of continuous functions for this construction.

We can now make more precise that we mean by T as a lattice; namely $T = \{\perp, \mathit{false}, \mathit{true}, \top\}$, where $\perp \sqsubseteq t \sqsubseteq \top$ holds for $t \in T$, but $\mathit{false} \not\sqsubseteq \mathit{true}$ and $\mathit{true} \not\sqsubseteq \mathit{false}$. We have used T in the context $T \times S$, and in general any $D \times D'$ can be construed as a lattice if D and D' are. One has only to define:

$$\langle x,x' \rangle \sqsubseteq \langle y,y' \rangle \text{ iff } x \sqsubseteq y \text{ and } x' \sqsubseteq y',$$

for all $x,y \in D$ and $x',y' \in D'$. In this way all of the domains $[T \to [S \to T \times S]]$, etc. can be regarded as lattices, and by the general method fixed points can be obtained when necessary. In particular in the equation

$$\ell = \mathit{Cond}(\ell \circ f, e) * t$$

f,e,t were certain *constants* in their intended domains, and *Cond*, $\circ, *$ were certain functions (operators) on these domains. Under the present interpretation all these domains are lattices, and it can be checked that all these functions are indeed continuous. Therefore, the function

$$F : [S \to S] \to [S \to S],$$

where

$$F(\ell) = \mathit{Cond}(\ell \circ f, e) * t$$

is itself a continuous function; and we know that such functions have fixed points. The price of generality is high, but eventually there are some returns on your investment.

Another kind of pay-off was discussed in some detail in Scott [6]. In that paper the *syntactical* domains were taken to be lattices also, and it was found that the mapping

$$\mathcal{C}: \mathbf{Cmd} \to [\mathbf{S} \to \mathbf{S}]$$

was not only continuous but its *existence* could be proved by the very same lattice-theoretical argument *via* fixed points. That is a rather fundamental point and unifies the theory considerably.

The whole process of forming fixed points can be given a functional formulation. Let D be any complete lattice and let $[D \to D]$ be the lattice of continuous functions. Then there is a mapping

$$Y : [D \to D] \to D$$

such that for each $f \in [D \to D]$ the element $Y(f) \in D$ *is* the least fixed point of f. Hence

$$f(Y(f)) = Y(f)$$

will be satisfied. What is remarkable and particularly useful is that Y itself is *continuous*. Thus if we employ Y in various equations along with other continuous functions we can rest assured that the compound functions obtained are also continuous. This makes the theory very *smooth*, if the reader will forgive the pun.

In making up these lattices it is sometimes useful to join two lattices together into one. We write $D + D'$ to mean the result of taking a copy of D and a disjoint copy of D' and forming the *union*. To make this union a lattice we *identify* the $\bot \in D$ with the $\bot' \in D'$ and similarly for $\top \in D$ and $\top' \in D'$ ($\bot = \bot'$ and $\top = \top'$.) Thus, for "ordinary" elements of $D + D'$ we can say, roughly, that either they are elements of D or of D' but not both. The $\sqsubseteq$ relations are carried over directly with *no* connections imposed between the elements of the disjoint parts. We shall in §5 discuss considerably more complex constructions of lattices of a "recursive" nature, but first it is necessary to explain the semantical treatment of identifiers.

4. IDENTIFIERS AND ENVIRONMENTS. In §2 we introduced into our syntax for commands the *identifiers* ($\xi \in \mathrm{Id}$). An identifier standing alone is an "unknown" having no predetermined meaning of its own - in contrast to the constants. The way one wishes to use identifiers, however, is to give them *temporary* meanings which can be altered within the differing scopes of different operators. The way to indicate a temporary assignment of meanings is by a function

$$\rho : \mathrm{Id} \to [S \to S]$$

which we call (the current) *environment* of the identifiers. We use $[S \to S]$ here because in the elementary command language the values of the variables are to be command values. In other languages with other types of variables other types of values would have to be used.

Let us write for short:

$$\mathrm{Env} = [\mathrm{Id} \to [S \to S]].$$

Now it will no longer be true that a command has a "fixed" value, because our syntax allows $\gamma \in \mathrm{Cmd}$ to contain variables. What we have to do is to redefine $\mathcal{C}$ so that

$$\mathcal{C}: \mathrm{Cmd} \to [\mathrm{Env} \to [S \to S]].$$

That is to say, given $\gamma \in \mathrm{Cmd}$, we do not evaluate at once $\mathcal{C}[\![\gamma]\!]$ but rather have to provide the current $\rho \in \mathrm{Env}$ to find $\mathcal{C}[\![\gamma]\!](\rho)$ as a state transformation.

The details of this redefinition of $\mathcal{C}$ will require alterations of the environments. Our notation for this is the following. Suppose $\xi \in \mathrm{Id}$, $\theta \in [S \to S]$, and $\rho \in \mathrm{Env}$. Then

$$\rho[\theta/\xi] \in \mathrm{Env}$$

is that environment ρ' which is just like ρ except for the one identifier ξ where we define

$$\rho'[\![\xi]\!] = \theta$$

(Thus $\rho' = \rho[\theta/\xi]$ is the *modification* of the function ρ just at the argument ξ to have the prescribed value θ.) Generalizing this idea we can also write

$$\rho[\theta^n/\xi^n]$$

where $\theta^n \in [S \to S]^n$, the set of n-tuples of state transformations. Here ξ^n is a group of n distinct identifiers and the alteration changes all the n values of the original ρ. (These definitions require just a bit more rigor when Id is taken as a lattice in the more abstract version of syntax of Scott [6].)

We can now state the revised clauses of the semantical definition for $\mathcal{C}$. (The function $\mathcal{E}$ retains its former definition, because in this simple language Boolean expressions contain no identifiers.)

$$\mathcal{C}[\![(\gamma)]\!](\rho) = \mathcal{C}[\![\gamma]\!](\rho),$$
$$\mathcal{C}[\![\phi]\!](\rho) = (\text{some given } S \to S),$$
$$\mathcal{C}[\![\text{dummy}]\!](\rho) = I,$$
$$\mathcal{C}[\![\xi]\!](\rho) = \rho[\![\xi]\!],$$
$$\mathcal{C}[\![\varepsilon \to \gamma_0, \gamma_1]\!](\rho) = \mathit{Cond}(\mathcal{C}[\![\gamma_0]\!](\rho), \mathcal{C}[\![\gamma_1]\!](\rho)) * \mathcal{E}[\![\varepsilon]\!],$$
$$\mathcal{C}[\![\gamma_0 ; \gamma_1]\!](\rho) = \mathcal{C}[\![\gamma_1]\!](\rho) \circ \mathcal{C}[\![\gamma_0]\!](\rho),$$
$$\mathcal{C}[\![\S\xi^n : \gamma^n \S]\!](\rho) = M_0^n(Y(\lambda\theta^n.\mathcal{C}[\![\gamma^n]\!](\rho[\theta^n/\xi^n]))).$$

These clauses are quite similar to the previous ones, except that the environment is dragged along into the interpretation of each compound command. It is invoked whenever an identifier stands in the place of a command (giving $\rho[\![\xi]\!]$ in the fourth clause). It is altered whenever identifiers are bound as formal parameters. This last clause requires a gloss.

First off if $\rho' \in \mathrm{Env}$, then

$$\mathcal{C}[\![\gamma^n]\!](\rho') = \langle \mathcal{C}[\![\gamma_0]\!](\rho'), \ldots, \mathcal{C}[\![\gamma_{n-1}]\!](\rho') \rangle$$

where $\gamma^n = \gamma_0, \gamma_1, \ldots, \gamma_{n-1}$ is an n-tuple of commands. Therefore the metaexpression

$$\mathcal{C}[\![\gamma^n]\!](\rho[\theta^n/\xi^n])$$

can be regarded as a *function* of the n-tuple $\theta^n \in [S \to S]^n$ whose values are also in $[S \to S]^n$. The λ-expression

$$\lambda\theta^n.\mathcal{C}[\![\gamma^n]\!](\rho[\theta^n/\xi^n])$$

is just the name of that function in the domain

$$[S \to S]^n \to [S \to S]^n.$$

The Y-operator used in the equation above is then to have the

logical type

$$[[S \to S]^n \to [S \to S]^n] \to [S \to S]^n.$$

(Cf. the end of the last section taking $D = [S \to S]^n$.) The value of this Y-operator is an n-tuple, and M_0^n is the *selector* function such that

$$M_0^n(\langle\theta_0,\theta_1,\ldots,\theta_{n-1}\rangle) = \theta_0.$$

What we are doing here, then, is finding the least solution to the equation (really: a *system* of n equations)

$$\langle\theta_0,\theta_1,\ldots,\theta_{n-1}\rangle = \mathcal{C}[\![\gamma^n]\!](\rho[\theta^n/\xi^n])$$

and setting

$$\mathcal{C}[\![\S\xi^n:\gamma^n\S]\!](\rho) = \theta_0.$$

Our mathematical equations describe this process rather succinctly with the aid of the various functional operators. At first sight these operators seem horrendous, but actually they are not hard to read. Furthermore they *hide* just the right things leaving the structure and sequencing of the operations quite apparent. It would also seem to be an advantage to condense the various clauses to one or two lines: if one can actually write equations in detail, he may have a chance of proving a theorem. And his chances are improved if the equations are not too long. It remains to be seen, admittedly, whether the method is going to be really practical for more complex languages.

In the introduction we spoke of *models* for a theory, and it will be useful now to return to this discussion in the present context. The concepts of our language are separated into two kinds: the *primitive notions* and the *logical constructs* which are built on these. In the simple command language the primitive notions are the set of states S and the objects denoted by the π's and ϕ's. Let us introduce explicitly syntactical categories for these:

$$\pi \in \mathrm{Pred} \text{ and } \phi \in \mathrm{Op}$$

for the atomic predicates and operations. All the other concepts like the Boolean values, the conditional expressions, the sequencing and looping of commands are treated as logical notions with fixed

meanings. The only chance for variation then lies in the primitive concepts. The interpretation of them would be established by giving mappings

$$\mathcal{P} : \mathrm{Pred} \to [S \to T \times S]$$

and

$$\mathcal{O} : \mathrm{Op} \to [S \to S]$$

once S has been determined. We could then say that in broad outline models are given as:

$$\mathfrak{M} = \langle S, \mathcal{P}, \mathcal{O} \rangle$$

because once these features are specified the meanings of all expressions in the language are fixed. Of course all we have explained here is the *logical types* of $\mathcal{P}$ and $\mathcal{O}$, but that is all one needs to give the semantical definitions for

$$\mathcal{C}_{\mathfrak{M}} : \mathrm{Cmd} \to [S \to S]$$

and

$$\mathcal{E}_{\mathfrak{M}} : \mathrm{Exp} \to [S \to T \times S].$$

(That is, in the semantical definitions we should replace $\mathcal{C}$ by $\mathcal{C}_{\mathfrak{M}}$ and $\mathcal{E}$ by $\mathcal{E}_{\mathfrak{M}}$ and modify the atomic clauses to read:

$$\mathcal{C}_{\mathfrak{M}}[\![\phi]\!](\rho) = \mathcal{O}[\![\phi]\!] \text{ and } \mathcal{E}_{\mathfrak{M}}[\![\pi]\!] = \mathcal{P}[\![\pi]\!].$$

By the way, in Scott [6] the $\mathcal{O}$ and the $\mathcal{P}$ were treated as parameters and it was noted that with S fixed both $\mathcal{C}_{\mathfrak{M}}$ and $\mathcal{E}_{\mathfrak{M}}$ are *continuous* in these parameters.)

With this point of view we can specialize and vary $\mathfrak{M}$ in restricted classes that actually *are* models for some reasonable concept of computational structure - as we contemplated for various models of the theory of integers. Besides this we can *compare* two models. Thus if γ_0 and γ_1 are two commands, we say that they are *equivalent* in $\mathfrak{M}$ iff

$$\mathcal{C}_{\mathfrak{M}}[\![\gamma_0]\!] = \mathcal{C}_{\mathfrak{M}}[\![\gamma_1]\!].$$

It may very well be that γ_0 and γ_1 are equivalent in $\mathfrak{M}$ but not in $\mathfrak{M}'$. That may be an interesting fact. Whether it is or not, we can at least say what we mean with the aid of our semantical definitions.

5. PROCEDURES. The language features discussed up to this point have been of the most elementary sort and were kept simple just for the sake of illustration. Even so the mathematical entities associated with the commands and expressions were involved enough. Fortunately the level of complication that we have reached is a plateau on which a variety of other features can be accommodated without too much additional effort. Among the features pressing for recognition is of course the *assignment statement*. No programming language can be called practical if it does not include the assignment statement in some form. The issues surrounding the proper interpretation of the assignment statement, however, require a rather full treatment of their own, and this will have to be reserved for another publication (Strachey [11]). In this section we select only one concept - that of a procedure - to discuss in any detail, mainly because it fits in well with our previous discussion of identifiers and function spaces. Even with this addition the language remains fragmentary. (In the syntax we shall make provision for an assignment statement, but the semantics of it and some other related ideas will only be briefly sketched. These inclusions are made so that the reader can grasp something of the style of the languages we are considering.)

In order to be able to include other concepts in our language a substantial extension of the repertoire of expressions beyond the Boolean level will be necessary. In some languages this is done through the introduction of a host of syntactical categories. This may be a practical idea to aid automatic syntax checking and error detection, but for understanding the language as a whole it is sometimes a formidable hurdle. For the sake of exposition we pretend that all the type checking is going to be done at *run time*. Thus all the expressions that have *values* will be massed together into one category. Before we start to define the category we should stop to consider what the values are going to be.

We always need *Boolean values* (T), and we may as well throw in at this point (integer) *arithmetic* values (N). If we will

be getting into assignments, then some expressions will have *locations* (addresses) for values (L). In this paper we will not say too much about them, but we want to leave room for them. Next we bring up the suggestion that at some point one may want to store - or maybe pass as a parameter - a *command*. Hence we are going to allow elements of $C = [S \to S]$ as values of expressions also. Finally we come to procedures (P).

A procedure is very much like a mathematical function. Now some functions have restricted domains, while others are more widely defined. We do not wish to consider here typed functions, so we shall attempt to permit our functions a free range of arguments and values. The different sorts of values were just described in the last paragraph. Let us put them together into one space, the *value space*:

$$V = T + N + L + C + P.$$

Again for simplicity we restrict attention to *one-parameter* procedures; that is, the domain of a function will be V itself. The values will also turn up in V but the path cannot be so direct. Remember that every evaluation has to depend on the state of the system, and that any action generally has to effect a change of state. It will be just the same for procedures: evaluating a procedure at an argument may produce along with a value a change of state. This argument suggests that

$$P = [V \to [S \to V \times S]].$$

Thus if $p \in P$ and $x \in V$, then we cannot find a value from $p(x)$ until we look at $\sigma \in S$. Then we get

$$p(x)(\sigma) = \langle y, \sigma' \rangle$$

where y is the value and σ' is the (possibly) altered state. That seems just fine.

Or does it? Suppose the state space were a one-element space which could be dropped from consideration. Suppose that we are in a dropping mood and that we forget about N, L, and C as well. Then the equation for V comes down to:

$$V = T + P,$$

which after substitution reads:

$$V = T + [V \to V].$$

In words we can say that under the reduction every element of V is *either* a Boolean value *or* a function. It still sounds good, but there is trouble: in ordinary set theory there are *no such spaces*! Why? Because there is a cardinality question. V must have at least two elements; but if so, then by Cantor's Theorem the space of *all* functions $[V \to V]$ always has *more* elements than V. Hence, the equation is impossible.

Here is the place where the lattice-theoretic pay-off is especially generous. By restricting $[V \to V]$ to *continuous* functions the cardinality of the function space is considerably reduced. That is a help, but it is not enough just to have V and $T + [V \to V]$ in any one-one correspondence. The correspondence must be *continuous*; then everything is fine, because we can rest assured that all our functional equations involve only continuous functions. (Remember, to be able to use a function as an argument of other operators, we must keep it *inside* the proper spaces.) The way to achieve a continuous isomorphism is not quite obvious and demands an inductive construction. Some remarks are given in Scott [5] and further hints are found in Scott [6]. The full details will be presented in papers under preparation (see references in the bibliography). In any case the outcome is that the construction of such self-referential spaces is not only possible, but they can be made to suit a variety of purposes - as long as one can be happy with continuous functions. Since we can argue that computability theory *is* happy with continuous functions, all is well, and the existence of the big value space V can be taken for granted.

All right, what then is the (a) language that might go along with V? (The authors have the peculiar idea that the domains of our concepts can be quite rigorously laid out *before* we make final the choice of the language in which we are going to describe these concepts. And it may turn out that the *same* domain is suitable for *several* languages. This is not to deny that there may be some vague ideas

of language which influence our choice of domains. What we suggest is that, in order to sort out your ideas, you put your domains out on the table first. Then we can all start talking about them.) A possible format of the language would retain the distinction in category between Cmd and Exp. (This is a point one might wish to debate - but we do not have the space to do it here.) It is Exp that will undergo the major expansion over the earlier language, so we give Cmd first:

COMMANDS

$$\begin{aligned}\gamma ::= {} & (\gamma)\,|\,\phi\,|\,\mathsf{dummy}\,|\,\xi\,|\\ & \varepsilon \rightarrow \gamma_0,\gamma_1\,|\,\gamma_0;\gamma_1\,|\\ & \S\ \xi^{n}:\gamma^{n}\S\,|\\ & \varepsilon!\,|\,\varepsilon_0:=\varepsilon_1\end{aligned}$$

This looks almost the same as before except for the last two clauses. Since we will convert commands into expressions, the $\varepsilon!$ is needed for the reverse process. The $\varepsilon_0:=\varepsilon_1$ is the assignment statement taken as a command to make the required assignment.

Turning now to expressions we must take note of the five parts of V:

EXPRESSIONS

$$\begin{aligned}\varepsilon ::= {} & (\varepsilon)\,|\,\pi\,|\,\xi\,|\\ & \varepsilon{:}\mathsf{T}\,|\,\mathsf{true}\,|\,\mathsf{false}\,|\,\varepsilon_0\rightarrow\varepsilon_1,\varepsilon_2\,|\,\varepsilon_0=\varepsilon_1\,|\\ & \varepsilon{:}\mathsf{N}\,|\,\upsilon\,|\,\varepsilon_0\,\omega\,\varepsilon_1\,|\\ & \varepsilon{:}\mathsf{L}\,|\,\mathsf{hd}\,\varepsilon\,|\,\mathsf{tl}\,\varepsilon\,|\\ & \varepsilon{:}\mathsf{C}\,|\,{:}\gamma\,|\\ & \varepsilon{:}\mathsf{P}\,|\,\lambda\xi.\varepsilon\,|\,\varepsilon_0\,\varepsilon_1\,|\\ & \gamma\ \mathsf{resultis}\ \varepsilon\end{aligned}$$

Note that identifiers occur in both Cmd and Exp. Some may wish to avoid the overlap, but it actually does not cause any difficulty, because we will separate values in a moment.

Before trying to understand the features of this language, it will be well to state the exact logical types of the semantical functions. An identifier will be assigned an element *either* from C *or* from V depending on how it is to be used. This means that now we shall have to set:

$$Env = [Id \to C + V].$$

Strictly speaking C, V and C + V are all different domains even though the first two are matched with *parts* of C + V. We shall require a more precise notation to indicate this matching. A completely precise symbolism would be cumbersome, so we write

$$(\theta \text{ in } [C + V]) \text{ and } (\beta \text{ in } [C + V])$$

where $\theta \in C$ and $\beta \in V$ to indicate the *corresponding* elements of C + V. For $\delta \in C + V$ we write

$$\delta|C \text{ and } \delta|V$$

to indicate the "projection" from C + V onto its two parts. (In case δ corresponds to an element of C, then $\delta|V = \perp$; and if δ comes from V, then $\delta|C = \perp$. The lack of precision becomes clear for spaces such as C + C where one would have to distinguish between left- and right-hand parts.)

The logical types of the functions $\mathcal{C}$ and $\mathcal{E}$ now will be these:

$$\mathcal{C}: Cmd \to [Env \to [S \to S]]$$

and

$$\mathcal{E}: Exp \to [Env \to [S \to V \times S]]$$

We shall not state all the semantical equations, since either they have already been discussed enough for the simpler language, or they require too much additional development. But a few can be shown.

For the case of identifiers, we use:

$$\mathcal{C}[\![\xi]\!](\rho) = \rho[\![\xi]\!]|C$$

and

$$\mathcal{E}[\![\xi]\!](\rho) = \lambda\sigma.\langle\rho[\![\xi]\!]|V, \sigma\rangle,$$

which keep the types straight. Note that in the latter equation we had to make the right hand side a function of $\sigma \in S$ with values in $V \times S$. The point is that if we ask for the value of ξ as an expression relative to the environment and the state of the system, then the answer is to be just $\rho(\xi)|V$ *without* any change of state.

With the two new kinds of commands, we have in the first instance:

$$\mathcal{E}[\![\varepsilon!]\!](\rho) = Do*\mathcal{E}[\![\varepsilon]\!](\rho),$$

where *Do* is a special operator defined as follows.

$$Do : V \to [S \to S],$$

and

$$Do(\beta)(\sigma') = (\beta|C)(\sigma').$$

By this we mean to indicate that $C = [S \to S]$ is itself a part of V. Thus if

$$\mathcal{E}[\![\varepsilon]\!](\rho)(\sigma) = \langle\beta,\sigma'\rangle,$$

then we project β into $\beta|C$ and apply that to σ' obtaining

$$\sigma'' = (\beta|C)(\sigma').$$

That is the resultant change of state in executing $\varepsilon!$ so that

$$\mathcal{E}[\![\varepsilon!]\!](\rho)(\sigma) = \sigma''.$$

In the second instance, the assignment command, the sequence of events is more complicated.

In this paper we shall not try to write the equation for

$$\mathcal{E}[\![\varepsilon_0 := \varepsilon_1]\!](\rho)(\sigma),$$

but we can say in words more or less what happens. We first evaluate

$$\mathcal{E}[\![\varepsilon_0]\!](\rho)(\sigma) = \langle\beta,\sigma'\rangle,$$

and project β to $\alpha = \beta|L$, a location. Next we evaluate:

$$\mathcal{E}[\![\varepsilon_1]\!](\rho)(\sigma') = \langle\beta',\sigma''\rangle.$$

Now comes the scuffle. If β' is not in the part of V which comes from L, we set $\beta'' = \beta'$. However, if β' *does* correspond to a location, then we consider $\alpha' = \beta'|L$. At this point we reveal that these mysterious "states of the system" are the internal states of our hypothetical machine. That is to say, σ'' represents (among other things) the current state of the *memory* of the machine - a memory which provides *contents* for *locations*. Thus there is a function to be applied to extract the desired contents, and we can write:

$$\beta'' = Contents(\alpha')(\sigma'').$$

In any case we have $\beta'' \in V$. Finally there is one last transformation to be made: we have a location α and a value β'' and the current state σ''. All that remains is to *assign* β'' to α in σ'' obtaining:

$$\sigma''' = Assign(\alpha,\beta'')(\sigma'').$$

We then assert that the equation:

$$\mathcal{E}[\![\varepsilon_0 := \varepsilon_1]\!](\rho)(\sigma) = \sigma'''$$

makes the interpretation of the command what is usually intended. Well, that is reasonably precise, but it only becomes completely rigorous when we give an explicit *construction* of S as a domain of internal states along with the concomitant functions *Contents* and *Assign*. The exposition of these ideas is the task of Strachey [11].

Shifting now to the semantics of expressions, the compounds of the sort ε:D are meant as Boolean valued predicates which distinguish between the various parts of V; namely, D = T,N,L,C or P. Take note of the fact that T has been made a part of V so that after a Boolean value $t \in T$ has been found, it will have to be injected into V. For example, to find the value of

$$\mathcal{E}[\![\varepsilon_0 = \varepsilon_1]\!](\rho)(\sigma)$$

we have to evaluate ε_0, then ε_1, then see if they belong to the same part of V. If they do and the part is T,N, or L, then a test for *equality* is meaningful. We carry the test out, get a truth value, and then wrap it up into V. Remember too that the state of the system will have been changing.

We need not discuss here the evaluation of numerals (ν) or arithmetic operations (ω). The mysterious operators $\uparrow$ and $\downarrow$ are for *referencing* and *dereferencing* - operations involving locations. Thus to evaluate:

$$\mathcal{E}[\![\uparrow\varepsilon]\!](\rho)(\sigma),$$

we have to find first:

$$\mathcal{E}[\![\varepsilon]\!](\rho)(\sigma) = \langle\beta,\sigma'\rangle.$$

Then we have to find a *new* (unused) location α in σ and take

$$\sigma'' = Assign(\alpha,\beta)(\sigma'),$$

making

$$\mathbb{E}[\![\uparrow\varepsilon]\!](\rho)(\sigma) = \langle\alpha,\sigma''\rangle.$$

In other words $\uparrow\varepsilon$ gives a *reference* to the value of ε. Obviously we want $\uparrow\varepsilon$ to be the opposite: ε is evaluated as having a value in L and then the *contents* of the current state of the system provide the value for $\uparrow\varepsilon$.

In the case of commands as expressions we take

$$\mathbb{E}[\![:\gamma]\!](\rho)(\sigma) = \langle\mathbb{C}[\![\gamma]\!](\rho) \text{ in } V,\sigma\rangle.$$

Note especially that *no* change of state has taken place and that $\mathbb{C}[\![\gamma]\!](\rho)$ has *not* been activated. The command has been "read", so to speak, but it has not been executed. Why do we do this? Because one may wish to store a command or to pass it as a parameter *without* executing it. In that way it can be set up and then set aside for later use.

Before we finish our survey of the semantics with a look at procedures, the resultis-construct can be given a quick explanation:

$$\mathbb{E}[\![\gamma \text{ resultis } \varepsilon]\!](\rho) = \mathbb{E}[\![\varepsilon]\!](\rho)\circ\mathbb{C}[\![\gamma]\!](\rho) \quad .$$

That is to say, do γ first, *then* evaluate ε. This combination is very similar to $\gamma_0;\gamma_1$ but is of a different syntactical category.

Finally, we come to procedures where the notation used in the language is just that of the functional abstraction/functional application sort. Abstraction is easy:

$$\mathbb{E}[\![\lambda\xi.\varepsilon]\!](\rho)(\sigma) = \langle f \text{ in } V,\sigma\rangle,$$

where $f \in P$ is defined by

$$f = \lambda\beta.\mathbb{E}[\![\varepsilon]\!](\rho[\beta \text{ in } C{+}V/\xi]).$$

Here $\beta \in V$ and

$$f : V \to [S \to V \times S]$$

comes out correctly by reference to the logical type of &. Note

that the evaluation of a functional abstraction requires no change of state. This is reasonable because none of the state transformations that may be lurking in ε can come out in the open until we know the value of σ. This cannot be known at the time the function is defined; we have to wait until it is applied to an argument.

Application can be interpreted in at least two ways. We take a more direct path - which might not be the most flexible. Thus to evaluate:

$$\mathcal{E}[\![\varepsilon_0\,\varepsilon_1]\!](\rho)(\sigma),$$

first find

$$\mathcal{E}[\![\varepsilon_0]\!](\rho)(\sigma) = \langle\beta,\sigma'\rangle,$$

and set $f = \beta|P$. Then find

$$\mathcal{E}[\![\varepsilon_1]\!](\rho)(\sigma') = \langle\beta',\sigma''\rangle,$$

and set

$$\mathcal{E}[\![\varepsilon_0\,\varepsilon_1]\!](\rho)(\sigma) = f(\beta')(\sigma'').$$

(The "indirect" route would test β to see if it were an L, if so all it would require is to look up the contents of that location and try to use that value. If β were not in the L part of V, then we would proceed as above.)

By the time we got around to the procedures there was not so very much to say. The point is of course that

$$\lambda\xi.\varepsilon$$

is but one clause of a highly recursive definition. The ε can be any compound expression. Thus the value of $\lambda\xi.\varepsilon$ is a complicated function. The construction of our value space allows us to treat this function (a mathematical object) just like any other value. It can be used as an argument of another function, it can be stored, it can be thought about in a mathematical way.

6. CONCLUSION There are many features missing from the language of the last section; to name a few: lists, structured values, initialization of parameters, more flexible parameter passing, recursive procedure definitions. This last is very important. Even though the space

$$[V \to [S \to V \times S]]$$

contains all the values of procedures, we gave no notation for recursively defined procedures the way we did for recursively defined commands. The reason for passing over that topic is the difficulty in keeping track of the state transformations involved in such definitions. This will have to be a topic for another essay.

Despite the shortcomings of the present exposition, we do feel, however, that we have demonstrated the possibility of a mathematical semantics for sophisticated languages. And we hope by now the reader understands what we mean by "mathematical semantics". In this approach the semantical functions give mathematical values to expressions - values related to some given model. The values of expressions are determined in such a way that the value of a whole expression depends functionally on the values of its parts - the exact connection being found through the clauses of the syntactical definition of the language. In this way the syntax is kept to a minimum so one can concentrate on the semantical interpretation.

The advantages of the method are many. In the first place we feel that it gets at the essence of meaning without having to formalize any bookkeeping, symbol tables, identifier lists, road maps or what have you - as is necessary in some language definitions. Furthermore, the method is conceptual and is not just a formal translation from one language into another. Sometimes the translation scheme is useful, but usually a full translation, say into the language of an "abstract" machine, makes it hard to discuss the features of the original language in isolation. As we have shown above we can move through the language one clause at a time, stopping to get a clear understanding of each construct by itself.

The present paper is one of a series that is the outcome of a collaboration which the authors started in the fall of 1969; further papers are mentioned in the references. As it now stands this theory falls into two rather distinct parts: the development of the appropriate mathematical apparatus, and its application to the application of programming languages. From a logical point of view the development of the mathematical foundations should obviously precede their application, but as often happens it is difficult to know exactly what mathematical apparatus is needed until some applications have been attempted.

The genesis of this approach is a paper given at a Working Conference on Formal Language Description Languages sponsored by IFIP in September 1964 (Strachey [10]). Although that paper contained the beginnings of the semantical ideas described here, it was quite unsatisfactory from a mathematical point of view. Not only was there no attempt at mathematical rigor, but the very existence of some of the objects used was not certain. For example, referring again to the domain mentioned in §5 which is quite naturally associated with interpretations of reasonable languages:

$$V = T + N + L + [S \to S] + [V \to [S \to V \times S]],$$

it is particularly important to note that such a domain *cannot* be constructed by ordinary set-theoretical means. Hence, the need for some such mathematical apparatus as we have presented here was forced on us. The present paper covers much the same ground as Strachey [10], but this time the mathematical foundations are secure. It is also intended to act as a bridge between the formal mathematical foundations and their applications to programming languages by explaining in some detail the notation and techniques we have found to be most useful.

Very much work remains to be done. An essential topic will be the discussion of the relation between the mathematical semantics for a language and the implementation of the language. What we claim the mathematics will have provided is the standard against which to judge an implementation. Thus if $\lambda\xi.\varepsilon$ is a function definition, then our semantics tells us which function - as a math-

ematical object - was intended. Any implementation must provide us with answers that are in complete harmony with this function in the same way we expect even the simplest desk calculator to be able to add. An interesting question here is whether the function defined by $\lambda\xi.\epsilon$ is calculable at all - in any sense. All of these questions are basic and do not even make sense without the proper mathematical foundation, which is just what we think our theory provides.

REFERENCES

[1] Erwin Engeler, ed., Semantics of Algorithmic Languages, Springer Lecture Notes in Mathematics, vol. 188 (1971).

[2] Donald E. Knuth, *Semantics of Context-Free Languages*, Mathematical Systems Theory, vol. 2 (1968), pp. 127-145.

[3] _____, *Examples of Formal Semantics*, in [1], pp. 212-235.

[4] David Park, *Fixpoint Induction and Proofs of Program Properties*, in Machine Intelligence 5 (1969), pp. 59-78.

[5] Dana Scott, *Outline of a Mathematical Theory of Computation*, in Proc. of the Fourth Annual Princeton Conference on Information Sciences and Systems (1970), pp. 169-176.

[6] _____, *The Lattice of Flow Diagrams*, in [1], pp. 311-366.

[7] _____, *Continuous Lattices*, in Proc. Dalhousie Conference, Springer Lecture Notes, in press.

[8] _____, *Lattice-theoretic Models for Various Type-free Calculi*, in preparation.

[9] _____, *Data Types as Lattices*, in preparation.

[10] Christopher Strachey, *Towards a Formal Semantics*, in Formal Language Description Languages, T. B. Steel, ed., North Holland (1966), pp. 198-220.

[11] _____, *An Abstract Model for Storage*, in preparation.

Programming Research Group Technical Monographs

This is a series of technical monographs on topics in the field of computation. Further copies may be obtained from the Programming Research Group, 45 Banbury Road, Oxford, England, or, generally, direct from the author.

PRG-1 Henry F. Ledgard
Production Systems: A Formalism for Specifying the Syntax and Translation of Computer Languages.
(£1.00, $2.50)

PRG-2 Dana Scott
Outline of a Mathematical Theory of Computation
(£0.50, $1.25)

PRG-3 Dana Scott
The Lattice of Flow Diagrams
(£1.00, $2.50)

PRG-4 Christopher Strachey
An Abstract Model for Storage
(in preparation)

PRG-5 Dana Scott and Christopher Strachey
Data Types as Lattices
(in preparation)

PRG-6 Dana Scott and Christopher Strachey
Toward a Mathematical Semantics for Computer Languages. (£1.00, $2.50)

PRG-7 Dana Scott
Continuous Lattices (£1.00, $2.50)