Programming Languages
B. Wegbreit Editor

The Denotational Semantics of Programming Languages

R.D. Tennent
Queen's University, Kingston, Ontario

This paper is a tutorial introduction to the theory of programming language semantics developed by D. Scott and C. Strachey. The application of the theory to formal language specification is demonstrated and other applications are surveyed. The first language considered, LOOP, is very elementary and its definition merely introduces the notation and methodology of the approach. Then the semantic concepts of environments, stores, and continuations are introduced to model classes of programming language features and the underlying mathematical theory of computation due to Scott is motivated and outlined. Finally, the paper presents a formal definition of the language GEDANKEN.

Key Words and Phrases: semantics, programming language, applicative, imperative, environment, store, continuation, theory of computation, higher-order function, recursive definition, LOOP, GEDANKEN

CR Categories: 4.22, 5.24

This work was supported in part by the National Research Council of Canada, Grant A8990. Author's address: Department of Computing and Information Science, Queen's University, Kingston, Ontario, Canada.

[1] Copyright © 1956, 1957, 1958 by Frederick Winsor. Reprinted by permission of Simon and Schuster, Inc.

1. Introduction

> See the little phrases go,
> Watch their funny antics.
> The men who make them wiggle so
> Are teachers of semantics.
>
> F. Winsor
> (Space Child's Mother Goose[1])

A great deal of progress has been made in the last few years towards the development of a theoretical framework appropriate to formal analysis and specification of the semantical aspects of computer languages. Despite the complexity and variety exhibited by modern programming languages, it has been shown by D. Scott and C. Strachey and their colleagues at Oxford University that a remarkably small number of fundamental semantic constructs provide an adequate conceptual basis for defining concise formal models of their meanings. This paper is a tutorial exposition of these concepts and demonstrates their usefulness to the formal definition of programming languages. Most of the references to the literature are provided in a bibliography at the end of this paper.

There are several applications which motivate analysis of the semantic structure of programming languages. A formal definition of a language provides a precise and complete reference standard for users and implementers, so that the omissions, contradictions, and ambiguities typical of informal semantic specifications such as those in the Algol 60 Report [Naur, 1963] may be avoided. Even if a formal definition were not comprehensible to average programmers, it could provide the basis for an accurate informal description.

A general language-independent framework of semantical concepts would help to standardize terminology, clarify similarities and differences between languages, and allow rigorous formulation and proof of semantic properties of languages. A language designer could analyze proposed constructs to help find undesirable restrictions, incompatibilities, ambiguities, and so on.

A theory of semantics should contribute to systematic composition and verification of programs, especially compilers. Indeed, a general notation for semantic specification would permit the development of a true compiler-generator, just as BNF led to the development of parser generators. Many of these goals have not yet been achieved, but sufficient progress is being made (see the references) to suggest that semantic analysis of programming languages will play a significant role in the development of computer science.

2. Basic Concepts

> 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.
>
> [Scott and Strachey, 1971]

2.1 Interpretation Functions

In mathematical logic, a *semantic interpretation* for a formal language is specified by defining mappings of the syntactic constructs of the object language into their abstract "meaning" in an appropriate mathematical model. For example, a class of numerals would be interpreted by mapping every possible numeral into the number it denoted. Similarly, if the object language is that of an applied predicate calculus, then every closed well-formed formula would be mapped into a truth value (*true* or *false*) relative to a domain of interpretation and specified meanings for the constant, function, and predicate symbols.

It is possible to define the semantics of programming languages using essentially the same approach. The abstract meanings appropriate to nontrivial computer languages are more complex and less familiar than truth values and numbers, but are no less mathematical. In order to demonstrate the practicability of this approach and to establish some notational and methodological conventions, we begin by considering a variant of an extremely simple language called LOOP [Meyer and Ritchie, 1967].

2.2 Syntax

In specifying a syntax as the domain of a semantic interpretation, it is convenient to be able to avoid some semantically irrelevant complications such as operator precedence and associativity by providing only an "abstract" form of syntax. In effect, an abstracted syntax specifies the compositional structure of programs while leaving open some aspects of their concrete representations as strings of symbols.

An abstract syntax of the LOOP language may be specified as shown in Figure 1.

The domains section specifies that the symbols Ξ, E, Γ, and Ψ are to be used as metavariables (possibly with primes or subscripts) over the sets of variables, expressions, commands, and programs, respectively. These are denoted **Var**, **Exp**, **Cmd**, and **Prog**, and are defined by BNF-like productions in which the metavariables stand for arbitrary elements of the corresponding syntactic category. *Capital* Greek letters are used as metavariables over syntactic domains, which are denoted by boldface symbols of *three or more* letters.

2.3 Semantics

To specify the meaning of programs in the language given above we need to define three interpretation functions:

$\mathcal{E}$: **Exp** $\rightarrow$ . .
$\mathcal{C}$: **Cmd** $\rightarrow$ . . .
$\mathfrak{M}$: **Prog** $\rightarrow$ . . .

where the co-domains will have to be constructed to model the type of "meaning" appropriate to the corresponding syntactic category. Script capital letters such as $\mathcal{E}$ will be used to denote semantic interpretation functions; i.e. mappings from syntactic constructs into their mathematical meanings.

The meaning of a LOOP program is the input-to-output function it computes; hence, if **N** is the set of (non-negative) integers, then the functionality (domain and co-domain) of $\mathfrak{M}$ may be specified as being:

$\mathfrak{M}$: **Prog** $\rightarrow$ (**N** $\rightarrow$ **N**)

i.e. every program will have a unique meaning given by $\mathfrak{M}$, and this meaning will be the function which maps every possible "input" number into the "output" that should be produced by all correct implementations of LOOP. The convention that the operator "$\rightarrow$" associates to the right will allow us to write $\mathfrak{M}$: **Prog** $\rightarrow$ **N** $\rightarrow$ **N** without parentheses.

The "value" of any expression is an integer; however, its *meaning* is more complex because in general the value may depend upon the state of the variables when the expression is evaluated. If **S** is the set of all possible "states" of the variables, then the functionality of $\mathcal{E}$ is:

$\mathcal{E}$: **Exp** $\rightarrow$ **S** $\rightarrow$ **N**

i.e. the meaning of an expression is a function that when applied to the current state, gives the value of the expression relative to that state. Use of a functional meaning here may seem surprising at first, but it is natural since an expression should have a *unique* value for *every* possible state.

Any state σ defines a correspondence between every variable and the integer that is its current "contents";

Fig. 1. Abstract syntax of the LOOP language.

Syntactic domains:

Ξ: **Var**	variables
E: **Exp**	expressions
Γ: **Cmd**	commands
Ψ: **Prog**	programs

Productions:

E ::= 0	constant
| Ξ	variable
| **succ** E	operation
Γ ::= Γ_1 ; Γ_2	sequencing
| Ξ := E	assignment
| **to** E **do** Γ	repetition
| (Γ)	parenthesization
Ψ ::= **read** Ξ; Γ; **write** E	

hence, it is convenient to model states by functions with domain **Var** and co-domain **N**:

$\sigma : \mathbf{S} = \mathbf{Var} \to \mathbf{N}$ states

Then for any variable Ξ, $\sigma[\![\Xi]\!]$ is the contents of Ξ in state σ; the symbols $[\![$ and $]\!]$ are used to enclose syntactic elements in order to separate the object and metalanguages.

To "update" a state the notation $\sigma[\nu/\Xi]$ is used to mean the state σ' such that $\sigma'[\![\Xi]\!] = \nu$ and for all $\Xi' \neq \Xi$, $\sigma'[\![\Xi']\!] = \sigma[\![\Xi']\!]$; that is, σ' is the same function as σ except at the argument Ξ which is mapped into ν. Semantic domains such as **N** and **S** will always be denoted by boldface symbols of *one or two* letters and *lower-case* Greek letters will be used as metavariables over such domains.

The meaning of a LOOP command is then a state transition function:

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

Then for any command Γ, $\mathcal{C}[\![\Gamma]\!]$ is the state transition function it specifies and $(\mathcal{C}[\![\Gamma]\!])(\sigma)$ is the state after execution of Γ if σ is the state before its execution. To minimize the number of parentheses the conventions are to assume that function application associates to the left and to omit parentheses (but not $[\![$'s and $]\!]$'s) around single-symbol arguments; hence, we may write: $\mathcal{C}[\![\Gamma]\!]\sigma$.

We may now complete the definition of the three interpretation functions by specifying their results when applied to typical elements of their domains. When a program of the form "**read** Ξ; Γ; **write** E" is executed, the implementation must carry out the following sequence of actions:

1. An initial state σ_i is established in which all variables are initialized to zero.
2. An integer ν is read in and stored in the variable Ξ.
3. The body of the program, Γ, is executed, resulting in some final state σ_f.
4. The expression E is evaluated relative to σ_f and this value is output.

Mathematically:

$\mathfrak{M}[\![\mathbf{read}\ \Xi;\ \Gamma;\ \mathbf{write}\ \mathrm{E}]\!]\nu$
$= \mathcal{E}[\![\mathrm{E}]\!]\sigma_f$
where $\sigma_f = \mathcal{C}[\![\Gamma]\!](\sigma_i[\nu/\Xi])$
where $\sigma_i[\![\Xi']\!] = 0$ for all $\Xi' : \mathbf{Var}$

The definition of the syntactic domain **Exp** is recursive, and so is the definition of its interpretation function; for each production defining **Exp** there is a corresponding clause in the definition of $\mathcal{E}$:

$\mathcal{E}[\![0]\!]\sigma = 0$
$\mathcal{E}[\![\Xi]\!]\sigma = \sigma[\![\Xi]\!]$
$\mathcal{E}[\![\mathbf{succ}\ \mathrm{E}]\!]\sigma = \mathcal{E}[\![\mathrm{E}]\!]\sigma + 1$

Note the difference between the numeral 0 which is part of the given object language, and the abstract concept of zero which we have chosen to denote by *0* in our metalanguage.

For the specification of $\mathcal{C}$ we define first the "iterations" of any function $\phi : X \to X$ to be the functions $\phi^\nu : X \to X$ for any $\nu = 0, 1, 2, \ldots$ such that $\phi^0(\chi) = \chi$ and $\phi^{\nu+1} = \phi \circ \phi^\nu$ where "$\circ$" is the usual function composition operation; that is, $\phi^{\nu+1}(\chi) = \phi(\phi^\nu(\chi))$ for all χ. Then:

$\mathcal{C}[\![(\Gamma)]\!] = \mathcal{C}[\![\Gamma]\!]$
$\mathcal{C}[\![\Gamma_1\ ;\ \Gamma_2]\!] = \mathcal{C}[\![\Gamma_2]\!] \circ \mathcal{C}[\![\Gamma_1]\!]$
$\mathcal{C}[\![\Xi := \mathrm{E}]\!]\sigma = \sigma[\mathcal{E}[\![\mathrm{E}]\!]\sigma/\Xi]$
$\mathcal{C}[\![\mathbf{to}\ \mathrm{E}\ \mathbf{do}\ \Gamma]\!]\sigma = ((\mathcal{C}[\![\Gamma]\!])^\nu)\sigma$
where $\nu = \mathcal{E}[\![\mathrm{E}]\!]\sigma$

In the case of the iteration construction, the formal definition gives definite answers to questions such as when and how often the control expression is to be evaluated, and what the behavior should be if this value is zero. In the first two clauses of the definition the use of functional meanings permitted omission of the "state" argument; that is:

$\mathcal{C}[\![\Gamma_1\ ;\ \Gamma_2]\!]\sigma = (\mathcal{C}[\![\Gamma_2]\!] \circ \mathcal{C}[\![\Gamma_1]\!])\sigma$ for all σ: **S**

but σ can be "canceled" on the right of both sides of this equation, because equality of the functions implies equality of the function results for all elements of their domain.

2.4 Discussion

It is important to realize that the definition of LOOP does not impose or imply any arbitrary constraints on implementations of the language, as would a more concrete implementation-oriented model, because nothing is specified about *how* the various functions must be computed or represented; the definition requires of an implementation merely that it computes the *right* mathematical function and this is what is more appropriate in a "standard" specification. The role of *operational* models of language semantics is to formalize language implementation methods so that their correctness may be verified by reference to the standard definition [Milne, 1974].

As an aid to verification of programs written in the defined language, it is convenient to tabulate useful rules of inference for constructs in the language, as in Hoare and Wirth [1973]. Such rules can be validated relative to a denotational definition when the conditions on their applicability are made explicit [Ligler, 1975a, 1975b].

In summary, semantic analysis of a programming language is founded on its denotational definition, but includes, on the one hand, formal models of implementations and, more concretely, implementations themselves, and, on the other hand, statements of "surface properties" of language constructs and, more abstractly, deeper theorems about the language as a whole.

The remainder of this paper considers denotational

definitions of languages that are more complex and practical than LOOP. Semantic constructs needed to model three broad classes of programming language features are presented and the mathematical theory underlying these constructions outlined. Then, as a demonstration of this conceptual framework, we give a definition of the language GEDANKEN [Reynolds, 1970], and conclude with a guide to the literature in the area.

3. Expressions and Environments

> The commonplace expressions of arithmetic and algebra have a certain simplicity that most communications to computers lack.
>
> [Landin, 1966]

3.1 Applicative Languages

The characteristic semantic feature of expressions is that they are "evaluated"; that is, the semantic interpretation of an expression ultimately defines its "value." Furthermore, for "pure" expressions, it is *exclusively* the value that has semantic importance. This linguistic property is termed *referential transparency* [Quine, 1960], for it allows a subexpression to be replaced by any other expression having the same value without any effect on the value of the whole. Languages or language subsets having the property of referential transparency are termed *applicative*; other adjectives that have been used include declarative, denotative, descriptive, and functional.

In most programming languages other than "pure" LISP [McCarthy, 1960] expressions are not purely applicative; however, in this section we shall be excluding from consideration any of the *imperative* (i.e. non-applicative) aspects of programming languages, such as updating assignments, jumps, and intermediate input/output which spoil referential transparency by introducing the possibility of "side effects" or transfers of control during expression evaluations.

On the other hand, the expression concept is not as shallow as is implied by the syntactical descriptions of most programming languages, in which the only forms of expression recognized as such are atomic constituents (constants, identifiers, etc.) and operator-operand combinations in a variety of syntactical arrangements such as prefix, infix, distributed (e.g. **if** ... **then** ... **else** ...) and so on. The applicative subsets include other forms as well, as may be seen by considering the following which are forms of expression typical of mathematical discourse:

(i) let $x = 5$ in $x + 3$,

(ii) $x + 3$ where $x = 5$,

(iii) let $f(x) = x + 3$ in $f(5)$,

(iv) $\sum_{i=1}^{5} i + 3$,

(v) $f(5)$ where $f(n) = \begin{cases} 1, & \text{if } n = 0 \\ n \times f(n-1), & \text{otherwise.} \end{cases}$

These forms of expression all involve the idea of *binding* an identifier to a denotation; the corresponding constructions in programming languages are the various forms of local declarations, including local variables, function definitions, formal parameter lists, iteration control variables, and so on. Because of binding constructions, it is necessary in general to evaluate an expression relative to an *environment* which provides a value for each free variable in the expression.

3.2 Abstractions

A more primitive form of binding construction in some programming languages is the *abstraction*, which appears as the "lambda expression" in LISP [McCarthy, 1960] and the "routine denotation" in Algol 68 [Branquart et al., 1971]. In the notation of Church [1941], an abstraction expression has the form "λI.E", where I is an identifier (the *bound variable*) and E is an expression (the *body*, usually containing I). Informally, the value of λI.E (in some environment) is the *function* that maps an argument value to which it is applied into the value of E relative to the environment extended by binding I to the argument value; for simplicity, we assume that the functionality of the function is evident from context. For example, in any environment $\lambda x.0$ denotes the constant zero function, $\lambda x.x$ denotes the identity function, $\lambda x.x^2$ denotes the squaring function, and $\lambda x.x + y$ denotes the function whose result is the sum of its argument and the value of y in that environment.

Because the value of an abstraction expression is a function, it may appear as the operator part of an operator-operand combination; for example, we may rewrite:

$square(5)$ where $square(x) = x^2$

as

$(\lambda x.x^2)(5)$.

This is an example of referential transparency with respect to the operator part of a combination.

Abstraction expressions are not often used in practical programming, but they play a fundamental role in the semantical analysis of applicative languages which will be described in the next section; the notation will also prove useful as part of the metalanguage for semantic definitions.

3.3 Applicative Structure

We have discussed three classes of expressions:

(i) atomic constituents, denoting fixed or locally defined values,

(ii) operator-operand combinations, denoting the application of a function to an argument (possibly a tuple),

(iii) various identifier-binding constructions.

It turns out that identifier-binding constructions may

be regarded as being merely more readable representations ("syntactic sugarings") of certain arrangements of abstractions, atomic constituents, and operator-operand combinations. The analysis of an expression in terms of these primitive constructs is called its *applicative structure*.

For instance, the use of auxiliary definitions in examples (i) and (ii) of Section 3.1 is semantically equivalent to a combination of the form $(\lambda x.x+3)\ (5)$ in which the operator part is an abstraction; that is, if $\mathcal{E} : \mathbf{Exp} \rightarrow \ldots$ is the semantic interpretation function for expressions, then:

$$\begin{aligned} & \mathcal{E}[\![\text{let } x=5 \text{ in } x+3]\!] \\ = \ & \mathcal{E}[\![x+3 \text{ where } x=5]\!] \\ = \ & \mathcal{E}[\![(\lambda x.x+3)\ (5)]\!] \end{aligned}$$

Similarly:

$$\begin{aligned} & \mathcal{E}[\![\text{let } f(x) = x+3 \text{ in } f(5)]\!] \\ = \ & \mathcal{E}[\![\text{let } f = \lambda x.x+3 \text{ in } f(5)]\!] \\ = \ & \mathcal{E}[\![[\lambda f.f(5)](\lambda x.x+3)]\!], \end{aligned}$$

illustrating the use of an abstraction as an operand.

The summation notation may be analyzed as follows:

Sigma(*1*, *5*, $\lambda i.i+3$)

where *Sigma*(*a*, *b*, *g*) = **if** *a* > *b* **then** *0* **else** *g*(*a*) + *Sigma*(*a* + *1*, *b*, *g*), but this introduces the problem of *recursive* (self-referential) definitions, also seen in example (v). Recursive definitions raise some very important questions as to the existence and uniqueness of the defined entity which will be considered in Section 4; here, we are interested only in the problem of finding an appropriate applicative structure for such definitions.

The recursion of example (v) may be considered to be the following *equation* in a function variable *f*:

$f = \lambda n.$ **if** $n = 0$ **then** *1* **else** $n \times f(n-1)$

i.e. we are interested in functions that satisfy the above equation for *f* at all argument values *n*. Such equations may be assigned a unique solution by assuming the existence of a functional *Y* having the following *fixed-point* property: when *Y* is applied to a function *F* the result is an entity *f* such that the equation $f = F(f)$ is satisfied; that is, $Y(F) = F(Y(F))$. In example (v), *F* is the function-valued functional defined by:

$F(g) = \lambda n.$ **if** $n = 0$ **then** *1* **else** $n \times g(n-1)$

and *f* is the factorial function; hence it may be analyzed as being semantically equivalent to:

$f(5)$ where $f = Y(F)$
where $F(g)(n) =$ **if** $n = 0$ **then** *1* **else** $n \times g(n-1)$

and then, using transformations already discussed, to:

$(\lambda f.f(5))$
$[(\lambda F.Y(F))$
$(\lambda g.\lambda n.$ **if** $n=0$ **then** *1* **else** $n \times g(n-1))]$

Similarly, *Sigma* may be explicitly defined as follows:

Sigma = *Y*(*F*)
where $F(S)(a, b, g) =$ **if** $a > b$ **then** *0*
else $g(a) + S(a+1, b, g)$

The mathematical definition of *Y* will be discussed in Section 4.3.

These examples have provided evidence for the thesis due to Landin [1964] that applicative languages are "spanned" by the "basis" of atomic constituents, combinations, and abstractions. In the next section, we discuss the semantics of applicative languages and, without loss of generality, may restrict consideration to only this small number of constructs.

3.4 Semantics

We consider the following archetypal expression language, called AEXP (applicative expressions):

B : **Bas**	bases
I : **Ide**	identifiers
E : **Exp**	expressions
E ::= B	base
| I	identifier
| λI.E	abstraction
| E_1E_2	combination
| (E)	parenthesization

We shall not specify any particular bases (constants) or domain of interpretation, but assume that there is a space of expressible values **E** and an interpretation function $\mathcal{B} : \mathbf{Bas} \rightarrow \mathbf{E}$. Now, in order to determine the value of an expression which may in general contain free identifiers, it is necessary to know the values to which such identifiers are bound in that local context. In AEXP the binding must be established by both an abstraction (which defines its textual scope) and a combination (whose operand supplies the value to be denoted); the set of associations of identifiers and their denotations in any context is termed the *environment*.

A convenient model for environments is as follows:

$\rho : \mathbf{U} = \mathbf{Ide} \rightarrow \mathbf{D}$ environments

where **D** is a space of denotable values, so that the value to which an identifier I is bound in the environment ρ is $\rho[\![\mathrm{I}]\!]$. In order to bind I to a value $\delta : \mathbf{D}$ we use the "updating" notation of Section 2.3; that is, $\rho[\delta/\mathrm{I}]$ is the environment in which I is bound to δ and is otherwise the same as ρ. In general, the values that are denotable in a language (**D**) need not be as extensive as those that are expressible (**E**); for example, in Algol 60 an **integer**, **real**, or **Boolean** value is not denotable, but can only be *dynamically* assigned to a named storage location or array component [Strachey, 1972].

Among the expressible values in AEXP are functions:

$\phi : \mathbf{F} = \mathbf{D} \rightarrow \mathbf{E}$

and we use the notation "ϕ in **E**" to represent the *injection* of the function ϕ into the space of all expressible values, and similarly "δ in **E**." The notation "$\epsilon \mid \mathbf{F}$" represents the *projection* of the expressible value ϵ onto **F**, and similarly "$\epsilon \mid \mathbf{D}$" for the projection onto **D**; these operations will be defined more exactly in Section 4.2.

The semantic interpretation function for AEXP is then:

$$\boxed{\mathcal{E} : \mathbf{Exp} \to \mathbf{U} \to \mathbf{E}}$$

$\mathcal{E}[\![(\mathrm{E})]\!]\rho = \mathcal{E}[\![\mathrm{E}]\!]\rho$
$\mathcal{E}[\![\mathrm{B}]\!]\rho = \mathcal{B}[\![\mathrm{B}]\!]$
$\mathcal{E}[\![\mathrm{I}]\!]\rho = \rho[\![\mathrm{I}]\!]$ in **E**
$\mathcal{E}[\![\lambda\mathrm{I}.\mathrm{E}]\!]\rho = \phi$ in **E**
  where $\phi(\delta) = \mathcal{E}[\![\mathrm{E}]\!](\rho[\delta/\mathrm{I}])$
$\mathcal{E}[\![\mathrm{E}_1\mathrm{E}_2]\!]\rho = \phi(\delta)$
  where $\phi = \mathcal{E}[\![\mathrm{E}_1]\!]\rho \mid \mathbf{F}$
  and $\delta = \mathcal{E}[\![\mathrm{E}_2]\!]\rho \mid \mathbf{D}$

In words, the meaning of a parenthesization is that of the parenthesized expression; the value of a base is given by applying $\mathcal{B}$ to it, independently of the environment, while that of an identifier is obtained by applying the given environment to it; the value of an abstraction is, as before, the function whose result for any argument value is the value of the body relative to an environment that binds the bound variable of the abstraction to the argument value; the value of a combination is obtained by applying the value of the operator part to the value of the operand part.

Note that for the evaluation of an abstraction body, the environment used is the one relative to which the *abstraction* expression is evaluated (rather than the operator-operand combination that caused the activation); i.e. the scope of identifiers is always *statically* determined, as in Algol 60. To achieve this form of binding for nonlocal variables in LISP the FUNCTION operator must be used; see Gordon [1973].

Although this completes the definition of the interpretation, its simplicity is deceiving; in the following we consider whether the sets and functions assumed in this model are mathematically well-defined.

4. Mathematical Foundations

> The generalization from computable to continuous functions is much like the generalization from algebraic to real numbers. In both cases one moves from a small but subtle set, determined by a certain kind of finite implicit representation, to a larger but structurally simpler set which can be constructed by limiting processes.
>
> [Reynolds, 1973]

4.1 Motivation

The definition of $\mathcal{E}$ in Section 3.4 appears quite reasonable, but it conceals some rather serious mathematical problems. These arise because semantic models typically make use of functions of *higher order*; i.e. functions whose arguments or results are functions or other infinite objects. For example, the meaning of an expression specified by the interpretation function $\mathcal{E}$ is a function whose argument is an environment function. Some examples of computational phenomena which are modeled most naturally by using higher-order functions are procedures whose parameters or results are procedural, the input and output streams of a nonterminating program such as an operating system or an ordinary program stuck in a loop, and "reentrant" data structures.

One of the reasons that higher-order functions in semantic models are mathematically troublesome is that it is necessary to allow general *recursive* definitions. The traditional approach to specifying the mathematical meaning of a recursive function definition [Kleene, 1952; Morris, 1971; Cadiou, 1972] is to demonstrate that there is a "partial function" over a denumerable domain which is the unique limit of a sequence of partial functions, each of which is at least as well-defined as the preceding elements in the sequence. Of the many partial functions that might satisfy the defining equation, this limit is the least defined and is also the "natural" solution from the computational point of view. Now the question is: how can this approach be generalized to allow for recursive definitions of functionals such as $\mathcal{E}$ and *Sigma* whose arguments and results may also be "partial" functions, or functionals, or other infinite and recursively defined objects?

Another problem arises from the possibility of *self-application* of higher-order functions. It may be recalled that in the semantic model of AEXP the space of "functions" was defined to be: $\mathbf{F} = \mathbf{D} \to \mathbf{E}$; now if the functions are included in the space of denotable values **D**, this implies that for any total function ϕ, the meaning of $\phi(\phi)$ should be well-defined. As a more concrete example, consider the following unusual but legal definition of the factorial function in Algol 60 [Ledgard, 1971]:

```
integer procedure factorial(n);
  integer n;
  integer procedure f(g, m);
    integer procedure g;
    integer m;
    f := if m = 0 then 1 else m × g(g, m−1);
  factorial := f(f, n)
```

The definition of f is not recursive (statically self-referential) but the procedure is self-activating (dynamically re-entrant). Even when the object language under consideration does not allow such self-application of procedures, it may be very desirable to use self-applicable "functions" to provide a natural and representation-independent model of other aspects of the language [Scott, 1970].

The problem is that unrestricted use of "functions" which are self-applicable leads to variants of the "para-

doxical" contradictions of naive set theory; for example, if p were a predicate that is *true* just when its argument is a predicate which is *false* when applied to itself (i.e. $p(q) = \neg q(q)$), then we would have $p(p) = \neg p(p)$.

A mathematical theory of computation which provides satisfactory solutions to these problems has recently been developed by D. Scott, using ideas from lattice theory and topology. A detailed technical exposition of this theory is beyond the scope of this paper and not necessary for the discussion of semantical concepts which follows, but we shall give an informal overview of its main features.

4.2 Basic Concepts

The key point is that Scott's theory characterizes a class of "data types," termed *domains*, and a class of functions (including those of higher order) which are sufficiently general to allow natural models of computational phenomena (including recursion and self-application), but which are also sufficiently restricted to exclude set theoretic paradoxes and allow finite approximations. The restrictions are imposed by a number of axioms, which are justified by demonstrating that mathematically consistent spaces and mappings needed in semantic models may be constructed satisfying these restrictions.

Informally, the main feature of a Scott domain is that a sequence of better and better *approximations* ("partial" objects, in a very general sense) in a domain must converge to a well-behaved *limit*, also in the domain; then all "operations" defined on the data type must be *continuous* functions in order to preserve these limits. The underlying topology is not the usual one, so that this notion of continuity is much more general than that of analysis.

Primitive domains may be formed by adjoining to finite or denumerable sets such as {*true*, *false*}, or {..., 2, −1, 0, 1, 2, ...} two special objects "$\perp$" (termed *bottom*, representing information which is completely undetermined) and "$\top$" (termed *top*, representing information which is consistent or overdetermined). The following may then be considered as primitive domains:

$\mathbf{N} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}^\circ$	integers
$\mathbf{T} = \{true, false\}^\circ$	truth values
$\mathbf{H} = \{\text{"a"}, \text{"b"}, \ldots\}^\circ$	characters

where $\{\ldots\}^\circ$ denotes the augmentation of the set by $\perp$ and $\top$. In such domains the notion of approximation is very elementary: $\perp$ approximates all elements, all elements approximate $\top$, and all other pairs are incomparable; hence there are no nontrivial limits or recursive definitions of elements in primitive domains, and the added structure is needed merely to satisfy the general requirements of the axioms and provide a basis for construction of more complex domains.

Nonprimitive domains may be constructed in a number of ways; if $\mathbf{D}$, $\mathbf{D}_1$, and $\mathbf{D}_2$ are *any* domains, then the following are also domains:

(i) $\mathbf{D}_1 \times \mathbf{D}_2$	product domain
(ii) $\mathbf{D}_1 + \mathbf{D}_2$	sum domain
(iii) $\mathbf{D}_1 \rightarrow \mathbf{D}_2$	function domain
(iv) $\mathbf{D}^n = \mathbf{D} \times \mathbf{D} \times \ldots \times \mathbf{D}$	domain of lists of length n
(v) $\mathbf{D}^* = \mathbf{D}^0 + \mathbf{D}^1 + \mathbf{D}^2 + \ldots$	domain of finite lists

Except for the special treatment of $\perp$ and $\top$, the elements of $\mathbf{D}_1 \times \mathbf{D}_2$ correspond to ordered pairs whose first components are elements of $\mathbf{D}_1$ and whose second components are in $\mathbf{D}_2$, and an element of $\mathbf{D}_1 + \mathbf{D}_2$ corresponds to an element of one of either $\mathbf{D}_1$ *or* $\mathbf{D}_2$. The domain $\mathbf{D}_1 \rightarrow \mathbf{D}_2$ consists of *continuous* functions from $\mathbf{D}_1$ to $\mathbf{D}_2$. $\mathbf{D}^n$ and $\mathbf{D}^*$ are the domains of n-tuples and all the finite lists, respectively, of elements from $\mathbf{D}$.

All of these constructed domains contain also the special elements $\perp$ and $\top$, and in some cases "partial" elements also, with approximation relations derived from those of the constituent domains. For example, in the case of a function domain $\mathbf{D}_1 \rightarrow \mathbf{D}_2$, f approximates g when $f(x)$ approximates $g(x)$ for all x: $\mathbf{D}_1$.

Several of the constructions may be combined in a domain definition; syntactically, it is assumed that of the binary domain operators "$\times$" has the highest precedence and "$\rightarrow$" the lowest (and associates to the right, as before). For a sum $\mathbf{X} = \ldots + \mathbf{Y} + \ldots$ we will use the following suffix notations for operations of *inspection*, *projection*, and *injection*, respectively:

(i) for χ: $\mathbf{X}$,
$$\chi\mathbf{EY} = \begin{cases} true, & \text{if } \chi \text{ corresponds to an element } \gamma\text{: } \mathbf{Y} \\ false, & \text{if } \chi \text{ does not correspond to an element of } \mathbf{Y} \end{cases}$$

(ii) for χ: $\mathbf{X}$,
$$\chi \mid \mathbf{Y} = \begin{cases} \gamma, & \text{if } \chi \text{ corresponds to } \gamma\text{: } \mathbf{Y} \\ \perp & \text{(erroneous), if } \chi \text{ does not correspond to any element of } \mathbf{Y} \end{cases}$$

(iii) for γ: $\mathbf{Y}$,
γ in $\mathbf{X} = \chi$, where χ: $\mathbf{X}$ corresponds to γ.

Another method for domain definition will be mentioned later, but we now turn to a consideration of continuous functions on domains. Recalling the structure of AEXP, it is reassuring that constant and identity functions over any domain are continuous, and that any function defined by abstractions and combinations is continuous when the constituent subexpressions define continuous functions on domains.

On primitive domains the requirement of continuity reduces to *monotonicity*: a function f is monotone when, if x approximates y then $f(x)$ approximates $f(y)$. Hence any "partial" function on a set may be extended to a continuous total function on the corresponding domain by defining $f(\perp) = \perp$ (since $\perp$ approximates everything), $f(\top) = \top$ (since $\top$ is approximated by everything) and $f(x) = \perp$ if the partial function is

undefined at x. Such extensions are termed *"doubly strict"*.

Less strict extensions of functions are also possible; if a function is constant with respect to one of its arguments, then it need not have the result $\perp$, even when that argument is undefined. For example, a continuous *conditional* function may be defined as follows: $\tau \rightarrow \epsilon_1, \epsilon_2$ is equal to $\perp$, ϵ_1, ϵ_2, or τ respectively, as the value of τ is $\perp$, *true*, *false*, or τ. Although the function is doubly strict with respect to τ, its result may be "defined" even when the alternative that is not selected happens to be $\perp$; of course, this provides a mathematical model of the *computational* treatment of a conditional expression.

4.3 Recursion

As discussed in Section 3.3, the problem of specifying a mathematical meaning for a general recursive definition is that of showing the existence of a fixed-point-finding function Y to produce an appropriate solution to equations of the form $f = F(f)$, given the higher-order transformation $F : \mathbf{D} \rightarrow \mathbf{D}$. Now if $\mathbf{D}$ is a domain, there is an approximation relation defined on it and a "worst" element $\perp$; then, by monotonicity, $F(\perp)$ approximates $F(F(\perp))$, and by induction:

$$\perp, F(\perp), F(F(\perp)), \ldots, F^i(\perp), \ldots$$

is a sequence of better and better approximations which, by continuity, converges to a limit f such that $F^i(\perp)$ approximates f for all $i \geq 0$ and $F(f) = f$. It can be shown that for any domain $\mathbf{D}$ there is a *continuous* fixed-point function $Y_D : (\mathbf{D} \rightarrow \mathbf{D}) \rightarrow \mathbf{D}$ such that for any continuous $F : \mathbf{D} \rightarrow \mathbf{D}$,

(i) $Y_D(F) = \mathbf{lim}_{i \rightarrow \infty} F^i(\perp)$ is a solution of the equation $f = F(f)$ and
(ii) any other solution of the equation is approximated by $Y_D(F)$.

This result is a generalization of the classical recursion theorem of Kleene [1952] in that the approximation relation allows the arguments and results of recursively defined functions to be "partially defined" objects of higher order, rather than either strictly defined or undefined. The argument can be generalized to give meanings to arbitrarily complex systems of mutually recursive definitions.

The self-application problem discussed in Section 4.1 is solved in Scott's theory by showing that domains, as well as domain elements, may be recursively defined. For example, it is possible to construct by a limiting process a mathematically consistent solution $\mathbf{D}$ to a *reflexive* domain "equation" of the form: $\mathbf{D} = \mathbf{B} + (\mathbf{D} \rightarrow \mathbf{E})$, where $\mathbf{B}$ is any domain of "basic" values and $\mathbf{E}$ is a domain which might include $\mathbf{D}$; that is, any $\delta : \mathbf{D}$ corresponds either to a basic value or to a function $\phi : \mathbf{D} \rightarrow \mathbf{E}$ applicable to every element of $\mathbf{D}$, including possibly its own representative in $\mathbf{D}$. This argument is very general and applies to all recursive or mutually recursive domain definitions involving any of the domain constructions we have discussed.

As another example of recursive domain definitions, consider adding the following simple subroutine facility to the LOOP language discussed in Section 2: the expression **proc** Γ defines a (parameter-less) procedure whose body is the command Γ. The procedure may be assigned to a variable and subsequently activated by the command **call** E.

To model this facility, the following mutually recursive equations must be solved:

$\mathbf{P} = \mathbf{S} \rightarrow \mathbf{S}$	procedures
$\mathbf{E} = \mathbf{N} + \mathbf{P}$	expressible values
$\sigma : \mathbf{S} = \mathbf{Var} \rightarrow \mathbf{E}$	states

and then the semantic interpretations of the new constructs are:

$$\mathcal{E}[\![\textbf{proc } \Gamma]\!]\sigma = \mathcal{C}[\![\Gamma]\!] \text{ in } \mathbf{E} \quad \text{and}$$
$$\mathcal{C}[\![\textbf{call E}]\!]\sigma = (\mathcal{E}[\![\mathbf{E}]\!]\sigma \mid \mathbf{P})\sigma$$

where, as before, $\mathcal{C} : \mathbf{Cmd} \rightarrow \mathbf{S} \rightarrow \mathbf{S}$, but now $\mathcal{E} : \mathbf{Exp} \rightarrow \mathbf{S} \rightarrow \mathbf{E}$. Except for the projections and injections between $\mathbf{E}$ and $\mathbf{N}$ that are now needed, the other constructs in LOOP may be interpreted as before.

This completes our brief overview of Scott's theory of computation; more detailed technical presentations may be found in the references. We have outlined how the theory solves the problems raised by the higher-order interpretation of AEXP, and we may now proceed to analyze more complex languages with the assurance that such semantic models are mathematically sound, provided that we limit ourselves to functions and domains defined using the methods we have discussed.

5. Commands and Stores

> Once a person has understood the way in which variables are used in programming he has understood the quintessence of programming.
>
> [Dijkstra, 1972]

5.1 Basic Model

The meanings of commands in the simple language LOOP were defined to be state transition functions, where a state could be modeled by a function from "variables" into their current contents. For more complex languages this simple approach is not adequate and it is necessary to structure their semantic models to incorporate both a textually determined environment (as discussed in Section 3) and a dynamically changing abstract *store*.

This complication arises because it is necessary to distinguish between *identifiers* (program variables, symbolic names, formal parameters) which are syntactic entities, and *locations* (storage variables, references, L-values, addresses, pointers) which are semantic. An identifier appears in a program and is statically bound to its denotation within the scope of its declara-

tion or binding construction; a location is in general computed (for example, by an array indexing operation or an indirect reference via a "pointer"-valued expression), and it may even be allocated or deallocated dynamically. The contents of a location may be irreversibly updated at any point, but identifier denotations "nest" so that upon leaving an "inner" scope, the previous environment reverts back. In many languages the class of values which are *denotable* by identifiers is not the same as those which are *storable* in locations; in Algol 60, for example, they are completely disjoint: only numbers and Booleans are storable, whereas the denotable values are locations, arrays, procedures, labels, switches, strings, and parameters called "by-name" [Strachey, 1972].

A semantic model which allows such distinctions to be made is obtained by interpreting expressions and commands relative to both an environment and a store, defined essentially as follows:

$\mathrm{I} : \mathbf{Ide}$	identifiers
$\alpha : \mathbf{L}$	locations
$\delta : \mathbf{D}$	denotable values
$\beta : \mathbf{V}$	storable values
$\rho : \mathbf{U} = \mathbf{Ide} \to \mathbf{D}$	environments
$\sigma : \mathbf{S} = \mathbf{L} \to \mathbf{V}$	stores

Typical semantic interpretation functions using these domains would be:

$$\mathcal{E} : \mathbf{Exp} \to \mathbf{U} \to \mathbf{S} \to \mathbf{E}$$
$$\mathcal{C} : \mathbf{Cmd} \to \mathbf{U} \to \mathbf{S} \to \mathbf{S}$$

where **E**, the generic domain of all expressible values, is usually constructed as the sum of **D** and **V**. Then $\mathcal{E}[\![\mathrm{E}]\!]\rho\sigma$ is the conventional value of E in the environment ρ when the state of the store is σ, and $\mathcal{C}[\![\Gamma]\!]\rho$ is the change of state of the store resulting from execution of Γ in ρ.

The primitive operations on stores are *content*: $\mathbf{L} \to \mathbf{S} \to \mathbf{V}$ and *update*: $(\mathbf{L} \times \mathbf{V}) \to \mathbf{S} \to \mathbf{S}$, defined as:

$$content\ \alpha\sigma = \sigma\alpha$$

and

$$update(\alpha, \beta)\sigma = \sigma[\beta/\alpha].$$

Then, in the store σ the value at a location denoted by an assignable identifier I in the environment ρ is $content(\rho[\![\mathrm{I}]\!])\sigma$, and an assignment command might be interpreted as follows:

$$\mathcal{C}[\![\mathrm{E}_1 := \mathrm{E}_2]\!]\rho\sigma = update(\alpha, \beta)\sigma$$
where $\alpha = \mathcal{E}[\![\mathrm{E}_1]\!]\rho\sigma \mid \mathbf{L}$
and $\beta = \mathcal{E}[\![\mathrm{E}_2]\!]\rho\sigma \mid \mathbf{V}$

This basic model must be elaborated if the language under consideration has *coercion* conventions; i.e. implicit accesses of location contents, type conversions, allocations, or other operations inserted in appropriate contexts for the convenience of programmers. The need for a specific coercion might be determined either from the expression and its textual context (i.e. at compile time), or from the value of the expression (i.e. at "run time"), depending on the conventions of the particular programming language. Another oversimplification, failure to allow for possible "side effects" of expression evaluation, is treated in Section 5.3.

5.2 Generalizations

Some straightforward extensions of this basic model can account for a variety of features found in many languages. Selectively updateable data structures such as arrays and records may be accommodated by allowing identifiers to be bound to lists of locations. "Pointer" values or references may be modeled by allowing locations or lists of locations to be storable values, so that a location may contain (a reference to) another or a data structure.

Dynamic allocation and deallocation of locations require a more complex model of the abstract store. The locations in any store must be partitioned into "active" and "inactive" areas, as follows:

$$\sigma : \mathbf{S} = \mathbf{L} \to (\mathbf{V} \times \mathbf{T}) \quad \text{stores}$$

Each location has associated with it (in addition to the usual stored value) a truth value "tag" to record whether it is active or inactive in the current store. Then the following additional primitive operations are appropriate:

$$area: \mathbf{L} \to \mathbf{S} \to \mathbf{T}$$
$$new: \mathbf{S} \to \mathbf{L}$$
$$lose: \mathbf{L} \to \mathbf{S} \to \mathbf{S}$$

where $area\ \alpha\sigma = true$ just if α is active in σ, $new\ \sigma$ is any location not in the active area of σ, and $lose\ \alpha\sigma$ is the store identical to σ but with location α deactivated (i.e. $area\ \alpha\ (lose\ \alpha\sigma) = false$). *content* and *update* must be redefined to take into account the added structure of **S**; we always want $content\ \alpha\ (update\ (\alpha, \beta)\sigma) = \beta$ when $area\ \alpha\sigma = true$, but language-dependent or even implementation-dependent properties might have to be considered when modeling the effects of attempting to access an inactive or uninitialized location.

Intermediate input and output operations have a dynamic and irreversible character, similar to assignment; hence, file components may be incorporated into a domain of stores as follows (we assume that only "basic" values may be read or written):

$\sigma : \mathbf{S} = (\mathbf{L} \to (\mathbf{V} \times \mathbf{T})) \times \mathbf{I} \times \mathbf{O}$	stores
$\mathbf{I} = \mathbf{B}^*$	input files
$\mathbf{O} = \mathbf{B}^*$	output files
$\beta : \mathbf{B}$	basic values

Then the following primitive operations may be used to model the reading of input and writing of output:

$$get: \mathbf{S} \to (\mathbf{B} \times \mathbf{S})$$
$$put: \mathbf{B} \to \mathbf{S} \to \mathbf{S}$$

get σ is a pair $\langle \beta, \sigma' \rangle$ such that β is the next value in the **I** component of σ and σ' is identical to σ but with that value removed; *put* $\beta\sigma$ appends the output value β to the **O** component of σ. Additional file components and primitive functions for opening, closing, rewinding, enquiring about, and so on may be necessary for modeling features of particular languages.

5.3 Procedures and Side Effects

The distinction between the statically determined environment and the dynamically changing store is most evident in the consideration of function and procedure definitions and activations. As in Section 3.4, the environment relative to which a procedure body is executed in a statically scoped language is that of its defining construction (extended by parameter bindings and local declarations), which may be different from that of the call; however, the store must be the one current when the procedure is activated, and the store resulting from that execution must be passed along when control returns to the point of call.

Hence, for procedures having an explicit parameter in the domain **D**, the store must also be supplied as an implicit parameter in the semantic model as follows:

$$\psi : \mathbf{P} = \mathbf{D} \to \mathbf{S} \to \mathbf{S} \quad \text{procedures}$$

Activation of a function (value-returning procedure) whose body involves the execution of commands may also result in side effects to the store, so that the possibly modified store must be returned as an implicit result as well as an implicit parameter of the function:

$$\phi : \mathbf{F} = \mathbf{D} \to \mathbf{S} \to (\mathbf{E} \times \mathbf{S}) \quad \text{functions}$$

Now, any expression whose evaluation can result in a function activation (or any other side effect-causing construct) might produce a change in the store, and so the functionality of the expression interpretation function must also be changed to accommodate this:

$$\mathcal{E} : \mathbf{Exp} \to \mathbf{U} \to \mathbf{S} \to (\mathbf{E} \times \mathbf{S})$$

For example, if we assume that the order of evaluation is left to right and that there are no coercions, an assignment command might have the following interpretation:

$$\mathcal{C}[\![\mathrm{E}_1 := \mathrm{E}_2]\!]\rho\sigma = \mathit{update}\ (\epsilon_1|\mathbf{L},\ \epsilon_2|\mathbf{V})\sigma''$$
$$\text{where } \langle \epsilon_2, \sigma'' \rangle = \mathcal{E}[\![\mathrm{E}_2]\!]\rho\sigma'$$
$$\text{where } \langle \epsilon_1, \sigma' \rangle = \mathcal{E}[\![\mathrm{E}_1]\!]\rho\sigma$$

Note that as the constituent expressions are evaluated, the state of the store changes from σ to σ' to σ'', and the updating operation is applied to σ''.

This form of model should not be construed as an endorsement of the use of side effects by programmers, but merely recognizes their existence in programming languages; the same is true of "shared" locations; i.e. locations accessible via more than one identifier [Ligler, 1975a].

5.4 Parameters and Declarations

Another manifestation of the differences between environments and stores is to be found in the parameter passing conventions of programming languages. In principle, parameter binding should merely extend the environment but changes to the store might be invoked for the convenience of the programmer; also, since a formal parameter must be bound to a denotable value whereas the argument might in general be any expressible value, coercions might be invoked when the actual parameter is inappropriate. For example, a call-by-value in Algol 60 binds the parameter not to the value of the actual parameter, but to a dynamically allocated location which is initialized by the actual parameter value, possibly coerced to obtain a storable value.

Analogous considerations apply to declarations because of their close association with abstractions, as discussed in Section 3.3. It is unfortunate that this association is obscured in many languages; for example, in PASCAL [Hoare and Wirth, 1973] a **var** *declaration* binds the identifier to a new storage structure, whereas a **var** *parameter* involves only an extension of the environment.

5.5 Elementary Control Structures

Many forms of "control structures" can be interpreted using just the concepts described so far; for example, assuming for simplicity that there are no side effects or coercions, basic conditional and iterative commands can be specified using the conditional function

$$\tau \to \sigma_1, \sigma_2$$

as follows:

$$\mathcal{E} : \mathbf{Exp} \to \mathbf{U} \to \mathbf{S} \to \mathbf{E}$$
$$\mathcal{C} : \mathbf{Cmd} \to \mathbf{U} \to \mathbf{S} \to \mathbf{S}$$
$$\mathcal{C}[\![\textbf{if E do } \Gamma]\!]\rho\sigma = (\mathcal{E}[\![\mathrm{E}]\!]\rho\sigma \mid \mathbf{T} \to \mathcal{C}[\![\Gamma]\!]\rho\sigma, \sigma)$$
$$\mathcal{C}[\![\textbf{while E do } \Gamma]\!]\rho\sigma = \mathit{repeat}\ \sigma$$
$$\text{where } \mathit{repeat}\ \sigma = (\mathcal{E}[\![\mathrm{E}]\!]\rho\sigma \mid \mathbf{T} \to \mathit{repeat}\ (\mathcal{C}[\![\Gamma]\!]\rho\sigma), \sigma)$$

The auxiliary definition of *repeat* is recursive, but could of course be written explicitly using the appropriate fixed-point operator.

For languages that use only control structures such as these, the meaning of any constituent is specifiable relative to only an environment (static context) and a store (dynamic history); however, the presence of control mechanisms such as **go to** statements, exits, error stops, backtracking, co-routines, and so on requires a complete restructuring of the interpretation framework, because the meaning of any constituent must be specified relative to a semantic component termed a *continuation* which models a possible computational *future*.

6. Control Structures and Continuations

> The problem with labels is that the evaluation of any sub-expression may result in going to one, in which case the computation which might have been planned on to complete the evaluation of the main expression will have to be forgotten about. This means that the function compiled for the sub-expression should be passed as an argument something which says what more is waiting to be done, so that the sub-expression can decide whether to do it or not.
>
> [Morris, 1970]

6.1 Basic Model

We may describe the interpretations to be considered here as *prophetic* since they must specify not merely the local result or effect of a construct, but its contribution to the final result of a complete program or process execution. To demonstrate the use of prophetic interpretations and continuations in a fairly simple context, we shall begin by examining an imperative extension to the language AEXP which requires a semantic model having a continuation component, but not a store component. Consider introducing the following construction into AEXP:

enter

.
.
.
with E **escape**
.
.
.

exit

enter and **exit** are a form of scope brackets; if an escape expression is evaluated within that scope, control transfers immediately to the nearest textually surrounding exit point, the value of E becoming the value of that **enter** . . . **exit** construction; otherwise, evaluation proceeds in the normal way.

The extensions may easily be incorporated into the syntactic definition of **Exp** as follows:

E ::= B
| I
| λI.E
| E_1E_2
| **enter** E **exit**
| **with** E **escape**
| (E)

But it should be clear that there is no simple way to extend the definition of ε given for the purely applicative language AEXP because any expression might have as a constituent an escape which can transfer control to a more global context and thereby prevent that expression from having a conventional value.

Now, consider interpreting an expression relative to a function which specifies what is to be done with the value of that expression if there is no escape. This function is called a *continuation* and in this context must be applicable to the value of an expression while returning an "answer"; i.e. the prophesized result of the *whole* program in which the expression is embedded. In a purely applicative language the answer must be an expressible value but for the sake of clarity and to allow for the generalizations that will soon be described, we introduce a generic domain **A** of *answers* which include expressible values and also other possible results such as error messages, intermediate output, and so on. Then we may construct the domain:

$\kappa : \mathbf{K} = \mathbf{E} \to \mathbf{A}$ expression continuations

and the prophetic interpretation function $\mathcal{E}'$ has functionality:

$\mathcal{E}' : \mathbf{Exp} \to \mathbf{U} \to \mathbf{K} \to \mathbf{A}$

so that $\mathcal{E}'[\![E]\!]\rho\kappa$ is always the final answer yielded by executing the complete program of which E is a part.

As before, parentheses contribute nothing to the meaning and we have $\mathcal{E}'[\![(E)]\!]\rho\kappa = \mathcal{E}'[\![E]\!]\rho\kappa$. Also, by examining the first three productions for **Exp** it may be seen that for these forms of expression there is no possibility of an escape during their evaluation, and so the final answer is obtained by simply applying the given continuation to the conventional value of the expression:

$\mathcal{E}'[\![B]\!]\rho\kappa = \kappa(\mathcal{B}[\![B]\!])$
$\mathcal{E}'[\![I]\!]\rho\kappa = \kappa(\rho[\![I]\!]\ \text{in}\ \mathbf{E})$
$\mathcal{E}'[\![\lambda I.E]\!]\rho\kappa = \kappa(\phi\ \text{in}\ \mathbf{E})$

However, the function ϕ must be defined to allow for the possibility of an escape out of the body of the abstraction (we assume that this is allowed). This is accomplished by letting ϕ take a continuation as an additional parameter; if there is no escape during the evaluation of the body the final answer will be obtained by applying this continuation to the usual result of the function. Thus the domain of functions may be defined as:

$\phi : \mathbf{F} = \mathbf{D} \to \mathbf{K} \to \mathbf{A}$ functions

and the function ϕ in the abstraction clause of the definition of $\mathcal{E}'$ is such that:

$\phi(\delta)\{\kappa\} = \mathcal{E}'[\![E]\!](\rho[\delta/I])\{\kappa\}$

Braces (rather than parentheses) are used to bracket continuation arguments as a notational aid.

Operator-operand combinations are interpreted by defining two new continuations (κ_1 and κ_2) to be applied to the values of the operator and operand expressions (ϵ_1 and ϵ_2, respectively) if there are no escapes during their evaluations:

$\mathcal{E}'[\![E_1E_2]\!]\rho\kappa = \mathcal{E}'[\![E_1]\!]\rho\kappa_1$
where $\kappa_1(\epsilon_1) = \mathcal{E}'[\![E_2]\!]\rho\kappa_2$
where $\kappa_2(\epsilon_2) = (\epsilon_1|\mathbf{F})(\epsilon_2|\mathbf{D})\{\kappa\}$

When both values are available the function can be applied to the explicit argument value, and the original

continuation κ is supplied as the additional argument.

Although structured quite differently, it can be shown that the definition of $\mathcal{E}'$ so far is essentially equivalent[2] to that for $\mathcal{E}$; however, the use of continuations provides the extra leverage that makes it possible to also treat imperative transfers of control.

An escape is modeled by ignoring the "normal" continuation and applying instead an "abnormal" continuation that models the computational future from the proper exit point. Since this exit point is textually determined, it is appropriate to make the corresponding continuation a component of the environment, as follows:

$\rho : \mathbf{U} = (\mathbf{Ide} \to \mathbf{D}) \times \mathbf{K}$ environments

For convenience, the notations $\rho[\![\mathrm{I}]\!]$ and $\rho[\delta/\mathrm{I}]$ will continue to be used to access and modify the $\mathbf{Ide} \to \mathbf{D}$ component of ρ; to access and modify the continuation component ρ (*exit*) and ρ [κ/*exit*] will be used.

Then the remaining clauses of the definition of $\mathcal{E}'$ are:

$\mathcal{E}'[\![\textbf{enter E exit}]\!]\rho\kappa = \mathcal{E}'[\![\mathrm{E}]\!](\rho[\kappa/exit])\kappa$
$\mathcal{E}'[\![\textbf{with E escape}]\!]\rho\kappa = \mathcal{E}'[\![\mathrm{E}]\!]\rho\{\rho(exit)\}$

The **enter . . . exit** construction simply modifies the second component of the environment for the contained expression to be the given continuation; then evaluation of an escape expression in that environment will cause the application of that continuation to the value which is to be escaped with, and the normal continuation is ignored.

It is also easy to account for "error stops"; for example, the following modified form of the definition of $\mathcal{E}'[\![\mathrm{E_1E_2}]\!]$ incorporates a dynamic test which produces an appropriate error message as the answer with no further computation if the value of the operator part turns out *not* to be a function:

$\mathcal{E}'[\![\mathrm{E_1E_2}]\!]\rho\kappa = \mathcal{E}'[\![\mathrm{E_1}]\!]\rho\kappa_1$
where $\kappa_1(\epsilon_1) = (\epsilon_1 \mathrm{IF} \to \mathcal{E}'[\![\mathrm{E_2}]\!]\rho\kappa_2\,,$
"error: function expected")

where κ_2 is as before. In Section 7 it will be shown how continuations allow intermediate outputs to be added to a program's answer (even when the program is nonterminating and generates an infinite stream of outputs).

6.2 Generalizations

The use of continuations and prophetic interpretations is also possible when the object language requires a store component in its semantic model. The following domains and functionalities would be appropriate for expressions in a language allowing dynamic changes of state:

$\mathcal{E} : \mathbf{Exp} \to \mathbf{U} \to \mathbf{K} \to \mathbf{S} \to \mathbf{A}$
$\kappa : \mathbf{K} = \mathbf{E} \to \mathbf{S} \to \mathbf{A}$ expression continuations
$\phi : \mathbf{F} = \mathbf{D} \to \mathbf{K} \to \mathbf{S} \to \mathbf{A}$ functions

where the other domains are as before. Then $\mathcal{E}[\![\mathrm{E}]\!]\rho\kappa\sigma$ defines the contribution of E to the answer when σ is the state of the store and κ is the normal continuation, to be applied to the value of E and the possibly updated state of the store after the evaluation (if there is no transfer of control).

Commands may be interpreted by a prophetic semantic function

$\mathcal{C} : \mathbf{Cmd} \to \mathbf{U} \to \mathbf{C} \to \mathbf{S} \to \mathbf{A},$

where C is a domain of *command continuations*, applicable to the state of the store after execution of the command:

$\theta : \mathbf{C} = \mathbf{S} \to \mathbf{A}$ command continuations

Then procedures with one parameter are modeled by elements of

$\mathbf{P} = \mathbf{D} \to \mathbf{C} \to \mathbf{S} \to \mathbf{A}.$

A command continuation is also the appropriate model for a label value; for example, a simple **go to** command with a fixed label may be interpreted as follows:

$\mathcal{C}[\![\textbf{go to I}]\!]\rho\theta\sigma = (\rho[\![\mathrm{I}]\!] \mid \mathbf{C})\sigma$

The continuation θ is ignored and control is transferred to the program-point represented by the label by applying to the current store the continuation to which the label identifier is bound.

If more general expressions can yield label values:

$\mathcal{C}[\![\textbf{go to E}]\!]\rho\theta\sigma = \mathcal{E}[\![\mathrm{E}]\!]\rho\kappa\sigma$
where $\kappa\epsilon\sigma' = (\epsilon \mid \mathbf{C})\sigma'$

By right-canceling store arguments and using the lambda notation, the following more compact form of the equation is obtained:

$\mathcal{C}[\![\textbf{go to E}]\!]\rho\theta = \mathcal{E}[\![\mathrm{E}]\!]\rho\{\lambda\epsilon.\epsilon \mid \mathbf{C}\}$

Also the specifications of the functionalities of the interpretation functions may be simplified to:

$\mathcal{E} : \mathbf{Exp} \to \mathbf{U} \to \mathbf{K} \to \mathbf{C}$ and
$\mathcal{C} : \mathbf{Cmd} \to \mathbf{U} \to \mathbf{C} \to \mathbf{C}$
where $\mathbf{K} = \mathbf{E} \to \mathbf{C}$.

6.3 Discussion

The semantic ideas we have introduced—environments, stores, and continuations—provide a conceptual framework for formal semantic specification of almost all features of high-level programming languages. Topics which are beyond the scope of this tutorial

[2] This definition, however, differs from that of $\mathcal{E}$ in that it specifies a "call-by-value" form of parameter evaluation: the operand is evaluated just once before the activation of the function. A "call-by-name" approach (in which the actual parameter is in principle evaluated only for references to the formal parameter) could also be specified using more complex environments applicable to a continuation as well as an identifier: $\mathbf{U}' = \mathbf{Ide} \to \mathbf{K} \to \mathbf{A}$ and "functions" which are modeled by elements of $\mathbf{F}' = (\mathbf{K} \to \mathbf{A}) \to \mathbf{K} \to \mathbf{A}$; see Reynolds [1974b], who also discusses how "call-by-value" may be specified without using continuations.

and are treated elsewhere are nondeterminism, compile-time types, and more complex control structures such as backtracking, coroutines, and parallelism. Well-known languages for which denotational definitions exist include Algol 60, Algol 68, PASCAL, LISP and SNOBOL.

As an example of a complete definition we shall consider the language GEDANKEN [Reynolds, 1970] which elegantly exemplifies many features of practical programming languages without the restrictions and syntactic complexities necessary in languages such as Algol 68 and PASCAL to achieve efficient implementation and compile-time verifiability. Because the description of GEDANKEN is readily accessible, we shall not give a concrete syntax or an informal description of its semantics; the explanations given by Reynolds should be consulted, especially for the unusual features of functional data structures and implicit references.

7. Semantic Specification of GEDANKEN

> The recent development of programming languages suggests that the simultaneous achievement of simplicity and generality in language design is a serious unsolved problem.
>
> [Reynolds, 1970]

7.1 Notation

The following conventions will be convenient:

(i) $\lambda\alpha.\lambda\beta.\ldots$ will be abbreviated to $\lambda\alpha\beta.\ldots$

(ii) Nested continuations of the form: $\alpha\{\beta\{\gamma\}\}$ will be written:

$\alpha;$
$\beta;$
γ

The semicolon has lower precedence than application, but it does not terminate a lambda expression.

(iii) Injections and projections will often be omitted when there can be no confusion as to the target domain.

(iv) Symbols such as τ_{TRUE}, θ_{ERROR}, and ϕ_{VAL} denote predefined values in the initial environment.

(v) The ith component of a tuple α will be denoted $\alpha \downarrow i$.

7.2 Syntactic Domains

B : **Bas**	bases
I : **Ide**	identifiers
E : **Exp**	expressions
Φ : **Abs**	abstractions
Π : **Par**	parameters
Ψ : **Prog**	programs
$\{\Lambda\}^\circ$	the null string

7.3 Productions

$\Psi ::= \mathrm{E}$
$\Pi ::= \mathrm{I}$
$\quad | \ \Pi_1, \Pi_2, \ldots, \Pi_n \quad (n \neq 1)$
$\quad | \ \Lambda$
$\quad | \ (\Pi)$
$\Phi ::= \lambda\Pi\mathrm{E}$
$\mathrm{E} ::= \mathrm{B}$
$\quad | \ \mathrm{I}$
$\quad | \ \Phi$
$\quad | \ \mathrm{E_1E_2}$
$\quad | \ \textbf{if } \mathrm{E_0} \textbf{ then } \mathrm{E_1} \textbf{ else } \mathrm{E_2}$
$\quad | \ \mathrm{E_1} \textbf{ and } \mathrm{E_2}$
$\quad | \ \mathrm{E_1} \textbf{ or } \mathrm{E_2}$
$\quad | \ \textbf{case } \mathrm{E_0} \textbf{ of } \mathrm{E_1}, \mathrm{E_2}, \ldots, \mathrm{E_n}$
$\quad | \ \mathrm{E_1}, \mathrm{E_2}, \ldots, \mathrm{E_n} \quad (n \neq 1)$
$\quad | \ \Lambda$
$\quad | \ \mathrm{E_1} = \mathrm{E_2}$
$\quad | \ \mathrm{E_1} := \mathrm{E_2}$
$\quad | \ \mathrm{E_1} ; \mathrm{E_2}$
$\quad | \ \ldots; \ \Pi \textbf{ is } \mathrm{E}; \ \ldots; \ \mathrm{I} \textbf{ isr } \Phi; \ \ldots; \ \mathrm{I}' : \mathrm{E}'; \ldots$
$\quad | \ (\mathrm{E})$

7.4 Semantic Domains

τ : **T**	$= \{true, false\}^\circ$	truth values
ν : **N**	$= \{\ldots, -2, -1, 0, 1, 2, \ldots\}^\circ$	integers
η : **H**	$= \{\text{"a"}, \text{"b"}, \text{"c"}, \ldots\}^\circ$	characters
ξ : **At**	$= \{ll, ul, \ldots\}^\circ$	atoms
B	$= \mathbf{T} + \mathbf{N} + \mathbf{H} + \mathbf{At}$	basic values
ϕ : **F**	$= \mathbf{E} \to \mathbf{K} \to \mathbf{C}$	functions
θ : **C**	$= \mathbf{S} \to \mathbf{A}$	label values
α : **L**		locations
Im	$= \mathbf{F} \times \mathbf{F}$	implicit references
Rf	$= \mathbf{L} + \mathbf{Im}$	references
ϵ : **E**	$= \mathbf{B} + \mathbf{F} + \mathbf{C} + \mathbf{Rf}$	expressible values
A	$= \{error\}^\circ + \mathbf{B} + (\mathbf{H} \times \mathbf{A})$	answers

i.e. an answer is either an error message, the value of the final expression of the program, or an intermediate output followed by another answer; this allows for any number of intermediate outputs (including zero or an infinite number), followed by an error message or the final value if the program terminates.

κ : **K**	$= \mathbf{E} \to \mathbf{C}$	expression continuations
χ : **X**	$= \mathbf{U} \to \mathbf{C}$	parameter continuations
D	$= \mathbf{E}$	denotable values
ρ : **U**	$= \mathbf{Ide} \to \mathbf{D}$	environments
V	$= \mathbf{E}$	storable values
σ : **S**	$= (\mathbf{L} \to (\mathbf{V} \times \mathbf{T})) \times (\mathbf{At} \to \mathbf{T}) \times \mathbf{H}^* \times \mathbf{H}^*$	stores

The second component of **S** associates a "tag" with each atom to record whether or not it has been generated; the third and fourth components are the input and output file, respectively. We assume finite inputs.

7.5 Primitive Store Functions

The following are similar to the primitives discussed in Section 5, except that for convenience they are used with continuations:

content: $\mathbf{L} \to \mathbf{K} \to \mathbf{C}$
update: $(\mathbf{L} \times \mathbf{V}) \to \mathbf{C} \to \mathbf{C}$
new: $\mathbf{K} \to \mathbf{C}$
get: $\mathbf{K} \to \mathbf{C}$
put: $\mathbf{H} \to \mathbf{C} \to \mathbf{C}$

The design of the language precludes access to inactive or uninitialized locations.

The function *gensym*: $\mathbf{K} \to \mathbf{C}$ activates a currently inactive atom; that is,

$$gensym\ \kappa\sigma = \kappa\xi\sigma'$$
where $\sigma' = \langle \sigma \downarrow 1,\ \sigma \downarrow 2[true/\xi],\ \sigma \downarrow 3,\ \sigma \downarrow 4 \rangle$
and $\sigma \downarrow 2(\xi) = false$.

7.6 Semantic Functions

$\mathcal{E} : \mathbf{Exp} \to \mathbf{U} \to \mathbf{K} \to \mathbf{C}$
$\mathcal{R} : \mathbf{Exp} \to \mathbf{U} \to \mathbf{K} \to \mathbf{C}$

The function $\mathcal{R}$ evaluates an expression using $\mathcal{E}$ and then automatically coerces the value.

$\mathcal{P} : \mathbf{Par} \to \mathbf{U} \to \mathbf{E} \to \mathbf{X} \to \mathbf{C}$

The function $\mathcal{P}$ is defined so that if possible $\mathcal{P}[\![\Pi]\!]\rho\epsilon\chi\sigma$ binds Π to ϵ and then applies the continuation χ to the resulting extended environment.

$\mathcal{F} : \mathbf{Abs} \to \mathbf{U} \to \mathbf{F}$
$\mathcal{M} : \mathbf{Prog} \to \mathbf{H}^* \to \mathbf{A}$
$\mathcal{B} : \mathbf{Bas} \to \mathbf{B}$

A subsidiary function $seq : \mathbf{E}^* \to \mathbf{F}$ will be used to convert a tuple of expressible values into a GEDANKEN function value:

$$\begin{aligned}
&seq\ (\epsilon_1, \epsilon_2, \ldots, \epsilon_n) \\
&\quad = \lambda\epsilon\kappa.\phi_{COERCE}\epsilon; \\
&\qquad \lambda\epsilon'.(\epsilon' \mathbf{E} \mathbf{N} \wedge 1 \leq \nu \leq n \to \kappa(\epsilon_\nu), \\
&\qquad\quad \epsilon' \mathbf{E} \mathbf{At} \to (\epsilon' = ll \to \kappa(1), \\
&\qquad\qquad\qquad \epsilon' = ul \to \kappa(n), \theta_{ERROR}), \theta_{ERROR}) \\
&\qquad\quad \text{where} \quad \nu = \epsilon' | \mathbf{N}
\end{aligned}$$

Another subsidiary function $coerce : \mathbf{K} \to \mathbf{K}$ simplifies the specification of coercions:

$$coerce\ \kappa = \lambda\epsilon.\phi_{COERCE}\ \epsilon\kappa$$

7.7 Semantic Equations

$$\boxed{\mathcal{E} : \mathbf{Exp} \to \mathbf{U} \to \mathbf{K} \to \mathbf{C}}$$

$$\mathcal{E}[\![\mathrm{B}]\!]\rho\kappa = \kappa(\mathcal{B}[\![\mathrm{B}]\!])$$
$$\mathcal{E}[\![\mathrm{I}]\!]\rho\kappa = \kappa(\rho[\![\mathrm{I}]\!])$$
$$\mathcal{E}[\![\Phi]\!]\rho\kappa = \kappa(\mathcal{F}[\![\Phi]\!]\rho)$$
$$\begin{aligned}
&\mathcal{E}[\![\mathrm{E}_1\mathrm{E}_2]\!]\rho\kappa \\
&\quad = \mathcal{R}[\![\mathrm{E}_1]\!]\rho; \\
&\qquad \lambda\epsilon_1.(\epsilon_1 \mathbf{E} \mathbf{F} \to \mathcal{E}[\![\mathrm{E}_2]\!]\rho\{\lambda\epsilon_2.\epsilon_1\epsilon_2\kappa\}, \theta_{ERROR})
\end{aligned}$$

$$\begin{aligned}
&\mathcal{E}[\![\textbf{if } \mathrm{E}_0 \textbf{ then } \mathrm{E}_1 \textbf{ else } \mathrm{E}_2]\!]\rho\kappa \\
&\quad = \mathcal{R}[\![\mathrm{E}_0]\!]\rho; \\
&\qquad \lambda\epsilon.(\epsilon \mathbf{E} \mathbf{T} \to (\epsilon \mid \mathbf{T} \to \mathcal{E}[\![\mathrm{E}_1]\!]\rho\kappa, \mathcal{E}[\![\mathrm{E}_2]\!]\rho\kappa), \theta_{ERROR})
\end{aligned}$$

$$\begin{aligned}
&\mathcal{E}[\![\mathrm{E}_1 \textbf{ and } \mathrm{E}_2]\!]\rho\kappa \\
&\quad = \mathcal{R}[\![\mathrm{E}_1]\!]\rho; \\
&\qquad \lambda\epsilon.(\epsilon \mathbf{E} \mathbf{T} \to (\epsilon \mid \mathbf{T} \to \mathcal{R}[\![\mathrm{E}_2]\!]\rho\kappa, \kappa(false)), \theta_{ERROR})
\end{aligned}$$

$$\begin{aligned}
&\mathcal{E}[\![\mathrm{E}_1 \textbf{ or } \mathrm{E}_2]\!]\rho\kappa \\
&\quad = \mathcal{R}[\![\mathrm{E}_1]\!]\rho; \\
&\qquad \lambda\epsilon.(\epsilon \mathbf{E} \mathbf{T} \to (\epsilon \mid \mathbf{T} \to \kappa(true), \mathcal{R}[\![\mathrm{E}_2]\!]\rho\kappa), \theta_{ERROR})
\end{aligned}$$

Note that the operands of **or** and **and** are evaluated "sequentially".

$$\begin{aligned}
&\mathcal{E}[\![\textbf{case } \mathrm{E}_0 \textbf{ of } \mathrm{E}_1, \mathrm{E}_2, \ldots, \mathrm{E}_n]\!]\rho\kappa \\
&\quad = \mathcal{R}[\![\mathrm{E}_0]\!]\rho; \\
&\qquad \lambda\epsilon.(\epsilon \mathbf{E} \mathbf{B} \to (\epsilon = 1 \to \mathcal{E}[\![\mathrm{E}_1]\!]\rho\kappa, \\
&\qquad\qquad\qquad \epsilon = 2 \to \mathcal{E}[\![\mathrm{E}_2]\!]\rho\kappa, \\
&\qquad\qquad\qquad \vdots \\
&\qquad\qquad\qquad \epsilon = n \to \mathcal{E}[\![\mathrm{E}_n]\!]\rho\kappa, \\
&\qquad\qquad\qquad \epsilon = ll \to \kappa(1), \\
&\qquad\qquad\qquad \epsilon = ul \to \kappa(n), \theta_{ERROR}), \theta_{ERROR})
\end{aligned}$$

$$\begin{aligned}
&\mathcal{E}[\![\mathrm{E}_1, \mathrm{E}_2, \ldots, \mathrm{E}_n]\!]\rho\kappa \\
&\quad = \mathcal{E}[\![\mathrm{E}_1]\!]\rho; \\
&\qquad \lambda\epsilon_1.\mathcal{E}[\![\mathrm{E}_2]\!]\rho; \\
&\qquad\quad \lambda\epsilon_2. \\
&\qquad\qquad \cdot \\
&\qquad\qquad\ \cdot \\
&\qquad\qquad\ \cdot \\
&\qquad\qquad \lambda\epsilon_n.\kappa\ (seq(\epsilon_1, \epsilon_2, \ldots, \epsilon_n))
\end{aligned}$$

$$\mathcal{E}[\![\Lambda]\!]\rho\kappa = \kappa\ (seq())$$

$$\begin{aligned}
&\mathcal{E}[\![\mathrm{E}_1 = \mathrm{E}_2]\!]\rho\kappa = \mathcal{R}[\![\mathrm{E}_1]\!]\rho; \\
&\qquad\qquad \lambda\epsilon_1.\mathcal{R}[\![\mathrm{E}_2]\!]\rho; \\
&\qquad\qquad\quad \lambda\epsilon_2.\phi_{NCEQUAL}\ (seq(\epsilon_1, \epsilon_2))\kappa
\end{aligned}$$

$$\begin{aligned}
&\mathcal{E}[\![\mathrm{E}_1 := \mathrm{E}_2]\!]\rho\kappa = \mathcal{E}[\![\mathrm{E}_1]\!]\rho; \\
&\qquad\qquad \lambda\epsilon_1.\mathcal{R}[\![\mathrm{E}_2]\!]\rho; \\
&\qquad\qquad\quad \lambda\epsilon_2.\phi_{NCSET}\ (seq(\epsilon_1, \epsilon_2))\kappa
\end{aligned}$$

$$\mathcal{E}[\![\mathrm{E}_1\ ; \mathrm{E}_2]\!]\rho\kappa = \mathcal{E}[\![\mathrm{E}_1]\!]\rho\{\lambda\epsilon.\mathcal{E}[\![\mathrm{E}_2]\!]\rho\kappa\}$$

Rather than introduce additional notation, it is convenient to specify the block construction by first "stripping off" the nonrecursive declarations as follows:

$$\mathcal{E}[\![\Pi \textbf{ is } \mathrm{E}; \ldots]\!] = \mathcal{E}[\![(\lambda\Pi \ldots)(\mathrm{E})]\!]$$

where the "..." stands for the rest of the block; then:

$$\begin{aligned}
&\mathcal{E}[\![\mathrm{I}_1 \textbf{ isr } \Phi_1\ ; \ldots\ ;\ \mathrm{I}_m \textbf{ isr } \Phi_m\ ; \\
&\quad \mathrm{I}^1 : \mathrm{E}^1; \ldots\ ;\ \mathrm{I}^n : \mathrm{E}^n]\!]\rho\kappa = \theta_1 \\
&\quad \text{where} \quad \theta_j = \mathcal{E}[\![\mathrm{E}^j]\!]\rho'\{\lambda\epsilon.\theta_{j+1}\} \\
&\qquad\qquad\qquad \text{for} \quad j = 1, 2, \ldots, n-1 \\
&\quad \text{and} \quad \theta_n = \mathcal{E}[\![\mathrm{E}^n]\!]\rho'\kappa \\
&\quad \text{and} \quad \rho' = \rho \ldots [\phi_i/\mathrm{I}_i] \ldots [\theta_j/\mathrm{I}^j] \ldots \\
&\qquad\qquad\qquad \text{for} \quad i = 1, 2, \ldots, m \\
&\qquad\qquad\qquad \text{and} \quad j = 1, 2, \ldots, n \\
&\quad \text{and} \quad \phi_i = \mathcal{F}[\![\Phi_i]\!]\rho' \\
&\qquad\qquad\qquad \text{for} \quad i = 1, 2, \ldots, m
\end{aligned}$$

$$\mathcal{E}[\![(\mathrm{E})]\!]\rho\kappa = \mathcal{E}[\![\mathrm{E}]\!]\rho\kappa$$

$$\boxed{\mathfrak{R} : \mathbf{Exp} \to \mathbf{U} \to \mathbf{K} \to \mathbf{C}}$$

$$\mathfrak{R}[\![\mathrm{E}]\!]\rho\kappa = \mathcal{E}[\![\mathrm{E}]\!]\rho\ \{coerce\ \kappa\}$$

$$\boxed{\mathcal{P} : \mathbf{Par} \to \mathbf{U} \to \mathbf{E} \to \mathbf{X} \to \mathbf{C}}$$

$$\mathcal{P}[\![\mathrm{I}]\!]\rho\epsilon\chi = \chi(\rho[\epsilon/\mathrm{I}])$$

$$\mathcal{P}[\![\Pi_1, \Pi_2, \ldots, \Pi_n]\!]\rho\epsilon\chi$$
$$= \phi_{COERCE}\epsilon;$$
$$\lambda\epsilon'.(\epsilon'\mathbf{E}\mathbf{F} \to \chi_1(\rho),\ \theta_{ERROR})$$
$$\text{where}\quad \chi_i(\rho_i) = \epsilon'(i)\{\lambda\epsilon_i.\mathcal{P}[\![\Pi_i]\!]\rho_i\epsilon_i\chi_{i+1}\}$$
$$\text{for}\quad i = 1, 2, \ldots, n$$
$$\text{where}\quad \chi_{n+1} = \chi$$

$$\mathcal{P}[\![\Lambda]\!]\rho\epsilon\chi = \chi\rho$$

$$\mathcal{P}[\![(\Pi)]\!]\rho\epsilon\chi = \mathcal{P}[\![\Pi]\!]\rho\delta\chi$$

$$\boxed{\mathcal{F} : \mathbf{Abs} \to \mathbf{U} \to \mathbf{F}}$$

$$\mathcal{F}[\![\lambda\Pi\mathrm{E}]\!]\rho = \lambda\epsilon\kappa.\mathcal{P}[\![\Pi]\!]\rho\epsilon\{\lambda\rho'.\mathcal{E}[\![\mathrm{E}]\!]\rho'\kappa\}$$

$$\boxed{\mathfrak{M} : \mathbf{Prog} \to \mathbf{H}^* \to \mathbf{A}}$$

$$\mathfrak{M}[\![\Psi]\!]\langle \ldots, \eta_i, \ldots\rangle$$
$$= \mathfrak{R}[\![\Psi]\!]\rho_0\{\lambda\epsilon.(\epsilon\mathbf{E}\mathbf{B} \to \lambda\sigma.\epsilon \mid \mathbf{B},\ \theta_{ERROR})\}\sigma_0$$

where ρ_0 maps each predefined identifier into its initial denotation (these are defined in Section 7.8); in the initial store σ_0 all locations are inactive, only the atoms *ll* and *ul* have been generated, the input file is $\langle \ldots, \eta_i, \ldots\rangle$, and the output file is empty.

The definition of $\mathfrak{B} : \mathbf{Bas} \to \mathbf{B}$ (which merely interprets numerals and character strings) is omitted.

7.8 Predefined Values

$$\xi_{LL} = ll \qquad \xi_{UL} = ul$$

$$\tau_{TRUE} = true \qquad \tau_{FALSE} = false$$

$$\theta_{ERROR} = \lambda\sigma.error$$

$$\phi_{GOTO} = \lambda\epsilon\kappa.\phi_{COERCE}\epsilon;$$
$$\lambda\epsilon'.(\epsilon'\mathbf{E}\mathbf{C} \to \epsilon' \mid \mathbf{C},\ \theta_{ERROR})$$

$$\phi_{READCHAR} = \lambda\epsilon\kappa.get\ \kappa$$

$$\phi_{WRITECHAR} = \lambda\epsilon\kappa.\phi_{COERCE}\epsilon;$$
$$\lambda\epsilon'.(\epsilon'\mathbf{E}\mathbf{H} \to put\ \epsilon'\{\lambda\sigma.\langle\epsilon', \kappa\epsilon'\sigma\rangle\},\ \theta_{ERROR})$$

The output value is appended both to the output file (as a model of the dynamic change to the abstract store), and to the stream of "answers" (the ultimate result of the execution of the program).

$$\phi_{NCREF} = \lambda\epsilon\kappa.new;$$
$$\lambda\epsilon'.update\ (\epsilon', \epsilon)\{\kappa\epsilon'\}$$

$$\phi_{VAL} = \lambda\epsilon\kappa.(\epsilon\mathbf{E}\mathbf{L} \to content\ \epsilon\kappa,$$
$$\epsilon\mathbf{E}\mathbf{Im} \to (\epsilon \downarrow 2)\ (seq())\kappa,\ \theta_{ERROR})$$

$$\phi_{NCSET} = \lambda\epsilon\kappa.\phi_{COERCE}\epsilon;$$
$$\lambda\epsilon'.(\epsilon'\mathbf{E}\mathbf{F} \to \epsilon'(1)\kappa_1,\ \theta_{ERROR})$$
$$\text{where}\quad \kappa_1(\epsilon_1) = \epsilon'(2)\kappa_2$$
$$\text{where}\quad \kappa_2(\epsilon_2) = (\epsilon_1\mathbf{E}\mathbf{L} \to update\ (\epsilon_1, \epsilon_2)\{\kappa\epsilon_2\},$$
$$\epsilon_1\mathbf{E}\mathbf{Im} \to (\epsilon_1 \downarrow 1)\epsilon_2\{\lambda\epsilon.\kappa\epsilon_2\},$$
$$\theta_{ERROR})$$

$$\phi_{COERCE} = \lambda\epsilon\kappa.(\epsilon\mathbf{E}\mathbf{Rf} \to \phi_{VAL}\epsilon\ \{coerce\ \kappa\},\ \kappa\epsilon)$$

$$\phi_{IMPREF} = \lambda\epsilon\kappa.\phi_{COERCE}\epsilon;$$
$$\lambda\epsilon'.(\epsilon'\mathbf{E}\mathbf{F} \to \epsilon'(1)\ \{coerce\ \kappa_1\},\ \theta_{ERROR})$$
$$\text{where}\quad \kappa_1(\epsilon_1) = \epsilon'(2)\ \{coerce\ \kappa_2\}$$
$$\text{where}\quad \kappa_2(\epsilon_2) = (\epsilon_1\mathbf{E}\mathbf{F} \wedge \epsilon_2\mathbf{E}\mathbf{F}$$
$$\to \kappa\langle\epsilon_1, \epsilon_2\rangle,\ \theta_{ERROR})$$

$$\phi_{ATOM} = \lambda\epsilon\kappa.gensym\ \kappa$$

$$\phi_{ISREF} = \lambda\epsilon\kappa.\kappa(\epsilon\mathbf{E}\mathbf{Rf})$$

$$\phi_{ISINTEGER} = \lambda\epsilon\kappa.\phi_{COERCE}\epsilon\{\lambda\epsilon'.\kappa(\epsilon'\mathbf{E}\mathbf{N})\}$$

and similarly for $\phi_{ISBOOLEAN}$, ϕ_{ISCHAR}, etc.

The remaining predefined functions are either arithmetics (INC, DEC), comparisons (GREATER, CHARGREATER, NCEQUAL), or definable in terms of other predefined functions (REF, SET, UNITSEQ, NOT, INTTODIGIT, DIGITTOINT, VECTOR, NEQ, ADD, SUBTRACT, MULTIPLY, DIVIDE, REMAINDER, EQUAL); their specifications are omitted.

8. Bibliography

8.1 Introduction

We shall not attempt to reference every publication on semantics, but only those which are related to the material presented in this paper; additional material may be found in the following collections and in the papers referenced therein: Steel [1966], Engeler [1971], ACM [1971, 1972, 1973, 1975], and Rustin [1972].

Possible applications of a theory of programming language semantics are also discussed in McCarthy [1963a, 1963b], de Bakker [1969], and Hoare and Wirth [1973]. Language analyses and specifications using the denotational approach to semantics discussed in this paper may be found in Tennent [1973], Gordon [1973], Mosses [1974] and Milne [1974]; Tennent [1973] and Ligler [1975a] demonstrate applications to language design. Models and verification of language implementations are discussed in Milner and Weyrauch [1972], Morris [1972], Vuillemin [1973], Gordon [1973], and Milne [1974]. Program verification methods using an underlying logic based on Scott's theory of computation are described in Milner [1972] and Vuillemin [1973]. The following discuss the relations between denotational definitions and other aspects of semantic analysis: Reynolds [1972a], Hoare and Lauer [1973], Milne [1974], Donahue [1974], and Ligler [1975a, b].

8.2 Basic Concepts

Scott and Strachey [1971] is the first published description of their approach to programming language semantics. Abstract forms of syntactic description were introduced by McCarthy [1963a] and Landin [1964].

8.3 Expressions and Environments

The indispensable reference for this chapter is the classic by Landin [1964]; also useful are McCarthy [1960], Landin [1966], Strachey [1967], and Reynolds [1972a].

8.4 Mathematical Foundations

Scott [1970] is an outline of the axiomatic basis of his theory and its applications; the theory is developed in detail in Scott [1971a, 1971b, 1972b, 1972c, 1972d]. A fairly complete and systematic exposition is given in Reynolds [1972b]; related material may be found in Park [1969], Bekić [1969], Wadsworth [1971, 1975], de Bakker [1971, 1974], Milner [1973a], Egli [1973], Reynolds [1973, 1974b], and Milne [1974].

8.5 Commands and Stores

The model of storage has its origins in McCarthy [1963a] and Strachey [1966, 1967], and is discussed also in Burstall [1967], Reynolds [1970], Ledgard [1971], Strachey [1972], and Scott [1972a]. A rigorous treatment of some technical issues not raised in this paper is given in Milne [1974].

8.6 Control Structures and Continuations

The escape control structure is similar to the "program-point" of Landin [1966] and many other constructions that have been proposed more recently [Reynolds, 1972a; Knuth, 1974]. A typical application is described in Burstall [1968].

Prophetic interpretations for modeling imperative control structures were proposed independently by Morris [1970] and Wadsworth [Strachey and Wadsworth, 1974]; see also Reynolds [1972a]. Mathematical semantic models of programming language features not discussed in this paper may be found in Tennent [1973], Reynolds [1974], Kahn [1973], Milner [1973b], Cadiou and Levy [1973], Gordon [1973], Milne [1974], and Cohen [1975].

8.7 Semantic Specification of GEDANKEN

The denotational definition given here may be compared with the "operational" description in Reynolds [1969]. Languages similar to GEDANKEN are defined in Tennent [1973] and Milne [1974].

Note added in proof. An important new book on the subject of this paper is to appear soon: *A Theory of Programming Language Semantics*, by R.E. Milne and C. Strachey (Chapman and Hall, London; Wiley, New York). It will include an account of the fundamental concepts of programming languages, a discussion of Scott's work, explanations of various sorts of semantics, and methods for establishing equivalence of programs and correctness of implementations.

Acknowledgments. The author is extremely grateful to J.C. Reynolds, C. Wadsworth, and R. Milne for their very helpful comments on earlier drafts of this paper.

Received November 1974; revised March 1975

References

ACM [1971]. Proc. ACM Symp. on Data Structures in Programming Languages. SIGPLAN Notices (ACM Newsletter) *6*, 2 (1971).

ACM [1972]. Proc. ACM Conf. on Proving Assertions about Programs. SIGPLAN Notices (ACM Newsletter) *7*, 1 (1972); also SIGACT News (ACM Newsletter) *14* (1972).

ACM [1973]. Conf. Rec. ACM Symp. on Principles of Programming Languages, Boston, 1973.

ACM [1975]. Conf. Rec. Second ACM Symp. on Principles of Programming Languages, Palo Alto, Calif.

de Bakker, J.W. [1969]. Semantics of Programming Languages. In *Advances in Information Systems Science, Vol. 2*, J.T. Tou, *Ed.*, Plenum Press, New York, 1969, pp. 173–227.

de Bakker, J.W. [1971]. *Recursive Procedures.* Mathematical Center, Amsterdam, 1971.

de Bakker, J.W. [1974]. *Least Fixed Points Re-visited.* Mathematical Center, Amsterdam, 1974.

Bekić, H. [1969]. Definable operations in general algebras, and the theory of automata and flowcharts (unpublished).

Branquart, P. [1971]. The composition of semantics in Algol 68. *Comm. ACM 14*, 11 (Nov. 1971), 697–708.

Burstall, R.M. [1967]. Semantics of assignment. In *Machine Intelligence 2.* American Elsevier, New York, pp. 3–20.

Burstall, R.M. [1968]. Writing search functions in functional form. In *Machine Intelligence 3.* American Elsevier, New York, pp. 373–385.

Cadiou, J.M. [1972]. Recursive definitions of partial functions and their computations. Tech. Rep. CS-266, Computer Sci. Dep., Stanford U., Stanford, Calif.

Cadiou, J., and Levy, J. [1973]. Mechanizable proofs about parallel processes. 14th Annual IEEE Symp. on Switching Theory and Automata, pp. 34–48.

Church, A. [1941]. *The Calculi of Lambda Conversion.* Princeton U. Press, Princeton, N.J.

Cohen, E.S. [1975]. A semantic model for parallel systems with scheduling. In ACM [1975], pp. 87–94.

Dijkstra, E.W. [1972]. Notes on structured programming. In *Structured Programming.* Academic Press, New York, pp. 1–82.

Donahue, J.E. [1974]. Mathematical semantics as a complementary definition for defined programming language constructs. Tech. Rep. CSRG-45, Computer Systems Research Group, U. of Toronto, Toronto, Canada.

Egli, H. [1973]. An analysis of Scott's λ-calculus models. TR-73-191, Dep. of Computer Sci., Cornell U., Ithaca, N.Y.

Engeler, E., *Ed.* [1971]. Symp. on Semantics of Algorithmic Languages. Springer-Verlag Lecture Notes Series no. 188, Springer-Verlag, Berlin, Heidelberg, New York.

Gordon, M.J.C. [1973]. Models of pure LISP. Experimental Programming Rep. No. 31. School of Artificial Intelligence, U. of Edinburgh, Edinburgh, Scotland.

Hoare, C.A.R., and Lauer, P.E. [1974]. Consistent and complementary formal theories of the semantics of programming languages. *Acta Inf. 3* (1974), pp. 135–153.

Hoare, C.A.R., and Wirth, N. [1973]. An axiomatic definition of the programming language PASCAL. *Acta Inf. 2* (1973), pp. 335–355.

Kahn, G. [1973]. A preliminary theory for parallel programs. Research Rep. 6, IRIA, France.

Kleene, S. [1952]. *Introduction to Metamathematics.* Van Nostrand, New York.

Knuth, D.E. [1974]. Structured programming with **go to** statements. *Computing Surveys 6*, 4 (Dec. 1974), 261–301.

Landin, P.J. [1964]. The mechanical evaluation of expressions. *Computer J. 6* (1964), 308–320.

Landin, P.J. [1966]. The next 700 programming languages. *Comm. ACM 9*, 3 (Mar. 1966), 157–164.

Ledgard, H. [1971]. Ten mini-languages, a study of topical issues in programming languages. *Computing Surveys 3*, 3 (Sept. 1971), 115–146.

Ligler, G.T. [1975a]. A mathematical approach to language design. In ACM [1975], pp. 41–53.

Ligler, G.T. [1975b]. Surface properties of programming language constructs. International Symp. on Proving and Improving Programs, Arc-et-Senans, France.

McCarthy, J. [1960]. Recursive functions of symbolic expressions and their computation by machine, I. *Comm. ACM 3*, 4 (April 1960), 184–195.
McCarthy, J. [1963a]. Towards a mathematical science of computation. In *Information Processing 1962*. Proc. IFIP Cong. 62. North-Holland Pub. Co., Amsterdam, pp. 21–28.
McCarthy, J. [1963b]. A basis for a mathematical theory of computation. *Computer Programming and Formal Systems*, P. Braffort and D. Hirschberg, *Eds.*, North-Holland Pub. Co., Amsterdam, pp. 33–69.
Meyer, A.R., and Ritchie, D.M. [1967]. The complexity of LOOP programs. Proc. 22nd ACM National Conference, pp. 465–469.
Milne, R.E. [1974]. The formal semantics of computer languages and their implementations. Ph.D. Th., Cambridge U. and Tech. Microfiche TCF-2, Oxford U. Computing Lab., Programming Research Group.
Milner, R. [1972]. Implementation and applications of Scott's logic for computable functions. In ACM [1972], pp. 1–6.
Milner, R. [1973a]. Models of LCF. Tech. Rep. CS-73-332, Computer Sci. Dep., Stanford U., Stanford, Calif.
Milner, R. [1973b]. Processes: a mathematical model of computing agents. Proc. Logic. Colloquium, Bristol, England.
Milner, R., and Weyrauch, R. [1972]. Proving compiler correctness in a mechanized logic. *Machine Intelligence 7*. Edinburgh U. Press, Edinburgh, Scotland, pp. 55–70.
Morris, F.L. [1970]. The next 700 formal language descriptions. (unpublished).
Morris, F.L. [1972]. Correctness of translations of programming languages, an algebraic approach. Tech. Rep. CS-72-303, Computer Sci. Dep., Stanford U., Stanford, Calif.
Morris, J.H. [1971]. Another recursion induction principle. *Comm. ACM 14*, 5 (May 1971), 351–354.
Mosses, P. [1974]. The mathematical semantics of Algol 60. Tech. Mon. PRG-12, Oxford U. Computing Lab., Programming Research Group.
Naur, P., *Ed.* [1963]. Revised report on the algorithmic language Algol 60. *Comm. ACM 6*, 1 (Jan. 1963), 1–17.
Park, D. [1969]. Fixpoint induction and proofs of program properties. *Machine Intelligence 5*. American Elsevier, New York, pp. 59–78.
Quine, W.V. [1960]. *Word and Object*. Technology Press, Cambridge, Mass., and Wiley, New York.
Reynolds, J.C. [1969]. GEDANKEN—a simple typeless language which permits functional data structures and coroutines. ANL-7621, Argonne National Labs., Argonne, Ill.
Reynolds, J.C. [1970]. GEDANKEN—a simple typeless language based on the principle of completeness and the reference concept. *Comm. ACM 13*, 5 (May 1970), 308–319.
Reynolds, J.C. [1972a]. Definitional interpreters for higher-order programming languages. Proc. 25th ACM National Conf., pp. 717–740.
Reynolds, J.C. [1972b]. Notes on a lattice-theoretic approach to the theory of computation. Dep. Systems and Information Science, Syracuse U., Syracuse, New York.
Reynolds, J.C. [1975]. On the interpretation of Scott's domains. *Symposia Mathematica, Vol. 15*. Academic Press, London pp. 123–135.
Reynolds, J.C. [1974a]. Towards a theory of type structure. Programming Symp. Paris. Springer-Verlag Lecture Notes in Computer Science, Vol. *19*. Springer-Verlag, Berlin, Heidelberg, New York, pp. 408–429.
Reynolds, J.C. [1974b]. On the relation between direct and continuation semantics. 2nd Colloquium on Automata, Languages, and Programming, Saarbrucken. Springer-Verlag Lecture Notes in Computer Science, Vol. 14, Springer-Verlag, Berlin, Heidelberg, New York.
Rustin, R., *Ed.* [1972]. *Formal Semantics of Programming Languages. Courant Computer Science Symposia 2*. Prentice-Hall, Englewood Cliffs, N.J.
Scott, D. [1970]. Outline of a mathematical theory of computation. Proc. 4th Princeton Conf. on Information Sciences and Systems; also Tech. Mon. PRG-2, Oxford U. Computing Lab., Programming Research Group, pp. 169–176.
Scott, D. [1971a]. The lattice of flow diagrams. In Engeler [1971]; also Tech. Mon. PRG-3, Oxford U. Computing Lab., Programming Research Group, pp. 311–366.
Scott, D. [1971b]. Continuous lattices. Proc. 1971 Dalhousie Conf. Springer-Verlag Lecture Note Series, No. 274, Springer-Verlag, Berlin, Heidelberg, New York; also Tech. Mon. PRG-7, Oxford U. Computing Lab., Programming Research Group.
Scott, D. [1972a]. Mathematical concepts in programming language semantics. AFIPS Conf. Proc., Vol. 40, 1972 SJCC AFIPS Press, Montvale, N.J., pp. 225–234.
Scott, D. [1972b]. Lattice theory, data types, and semantics. In Rustin [1972], pp. 65–106.
Scott, D. [1972c]. Lattice theoretic models for various type-free calculi. Proc. 4th International Cong. for Logic, Methodology, and the Philosophy of Science, Bucharest.
Scott, D. [1972d]. Data types as lattices. Unpublished lecture notes, Amsterdam.
Scott, D., and Strachey, C. [1971]. Towards a mathematical semantics for computer languages. Proc. Symp. on Computers and Automata, Polytechnic Institute of Brooklyn; also Tech. Mon. PRG-6, Oxford U. Computing Lab., pp. 19–46.
Steel, T., *Ed.* [1966]. *Formal Language Description Languages*. North-Holland Pub. Co., Amsterdam.
Strachey, C. [1966]. Towards a formal semantics. In Steel [1966], pp. 198–218.
Strachey, C. [1967]. Fundamental concepts in programming languages. In Notes for the International Summer School in Computer Programming, Copenhagen (unpublished).
Strachey, C. [1972]. Varieties of programming language. Proc. International Computing Symp., Cini Foundation, Venice; also Tech. Monograph PRG-10, Oxford U. Computing Lab. Programming Research Group.
Strachey, C., and Wadsworth, C. [1974]. Continuations, a mathematical semantics for handling full jumps. Tech. Mon. PRG-11. Oxford U. Computing Lab., Programming Research Group.
Tennent, R.D. [1973]. Mathematical semantics and design of programming languages. Ph.D. Th., Dep. of Computer Sci., U. of Toronto.
Vuillemin, J.E. [1973]. Proof techniques for recursive programs. Tech. Rep. CS-73-393, Computer Sci. Dep., Stanford U.
Wadsworth, C.P. [1971]. Semantics and pragmatics of the lambda calculus. Ph.D. Th., Oxford U.
Wadsworth, C.P. [1975]. The relation between lambda-expressions and their denotations (unpublished).