Syracuse University

SURFACE

Electrical Engineering and Computer Science - Technical Reports

College of Engineering and Computer Science

1976

The Definition of Programming Languages

J. A. Robinson
Syracuse University

Follow this and additional works at: https://surface.syr.edu/eecs_techreports

Part of the Computer Sciences Commons

Recommended Citation

Robinson, J. A., "The Definition of Programming Languages" (1976). *Electrical Engineering and Computer Science - Technical Reports*. 44.
https://surface.syr.edu/eecs_techreports/44

This Report is brought to you for free and open access by the College of Engineering and Computer Science at SURFACE. It has been accepted for inclusion in Electrical Engineering and Computer Science - Technical Reports by an authorized administrator of SURFACE. For more information, please contact surface@syr.edu.

THE DEFINITION OF PROGRAMMING LANGUAGES

J. A. Robinson

October 1976

SCHOOL OF COMPUTER
AND INFORMATION SCIENCE
SYRACUSE UNIVERSITY

THE DEFINITION OF PROGRAMMING LANGUAGES*

J. A. Robinson

October, 1976

School of Computer and Information Science
Syracuse University
Syracuse, New York 13210

*This work supported by the Rome Air Development Center under contract F30602-75-C-0121.

THE DEFINITION OF PROGRAMMING LANGUAGES

0. Introduction

There is no need to argue in favor of concise, clear, complete, consistent, descriptions of programming languages, nor to recite the cost in time, energy, money and effectiveness which is incurred when a description falls short of these standards. Reliable, high-quality computer programming is impossible without a clear and precise understanding of the language in which the programs are written -- this being true quite independently of the merits of the language as a language.

In this study we tried to discover the current state of the methodology of definition of programming languages. We sought to separate the question (as far as it can be done) of *definition* from that of *design*. Our goal was to get at ways of *specifying* programming languages rather than ways of *inventing* them or of *implementing* them on machines. We realise however that these questions are not completely separable.

Many programming languages are poorly defined. Nowadays -- indeed ever since the influential and pioneering example of the formal definition of ALGOL-60 -- one finds (usually) that the *syntax* of a programming language is defined very clearly and compactly, but that the *semantics* is (usually) explained badly.

This discrepancy seems to be due to the fact that the BNF notation (with its accompanying framework of concepts for dealing with sets of strings over a given alphabet of characters) has become a powerful, standard tool for specifying a syntax; whereas nothing comparable has yet received universal acceptance for specifying a semantics. Until relatively recently, indeed, nothing comparable *existed*. However, since about 1970 there has been a fruitful research effort under way in *formal semantics* which has produced a sound mathematical theory of considerable power and elegance. The product of this research has been a system of notation (with an accompanying framework of concepts and results) which, in the opinion of the present writer, more than redresses the imbalance between the methodology of syntax and that of semantics.

There has also been, over the past decade or so, a trend towards improving the technique of syntax-specification, summed up in the distinction between *concrete* and *abstract* syntax. The earlier BNF-influenced notion was that a syntax was a system of sets of *strings of characters*. The *structure* of a given string had to be computed (as a labelled tree figure of some kind) by means of the operation of *parsing* the string. The details tended to be fussy, minute, many, and yet *inessential to the overall mission of the language.*

The best current practice is to present the syntax of a language in as *abstract* a form as is consistent with the objective of creating suitable vehicles for the intended semantic roles of the various syntactic constructs. Thus, one might specify, for example, that a *conditional expression* is simply a "thing" that has three *immediate constituents*: a *premiss*, a *conclusion*, and an *alternative*; and that the premiss is a *sentence* while the conclusion and the alternative are both *expressions*. Only two basic requirements on one's abstract specification need be imposed explicitly:

the requirement of *unique decomposition* and the requirement of *finite composition.*

The first requirement, for example, means that two conditional expressions are the *same* conditional expression if and only if they have the *same premiss,* the *same conclusion,* and the *same alternative.*

The second requirement says that there must be no *infinite* sequences

$$x_1, x_2, \dots$$

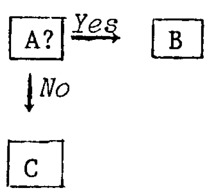
of expressions in which each next expression x_{j+1} is an immediate constituent of the preceding expression, x_j.

With such an abstract specification, the particular details of *how* a conditional expression shall be written, e.g. whether as one of the following

A → B, C

If A *then* B *else* C

si A *alors* B *ou* C



B *provided that* A *and otherwise* C

or some other "syntactically sugared" bit of text, does not matter: so long as the *premiss* A, the *conclusion* B and the *alternative* C are computable from the text by the *selection* functions provided as part of the concrete, sugared representation.

The benefits of using abstract syntax specification are obvious and will not be argued here. Suffice it to remark that, in cases where the language being specified is intended (as, e.g., with JOVIAL) to be implemented on a wide variety of machines and for a large community of users, one gains *flexibility* and *portability* at the expense merely of the writing of "concrete-to-abstract" *translators*, and "abstract-to-concrete" *representors*. The abstract syntax is a suitable *interlingua* between the different implementations of the same language.

In the opinion of the present writer there is a widespread understanding of and adherence to the principle of specifying syntax abstractly. At any rate, the techniques appropriate for doing so are not new or difficult. In the sequel a relatively unorthodox viewpoint (that syntactic objects are *functions*, namely from a finite set of *selectors* to a set of other syntactic objects as their *immediate constituents*) is presented as part of a general approach to programming language definition in which every entity is construed as a function of some kind, or else as unanalyzed, primitive. However, there is nothing particularly novel in this point of view. It is the method of handling *semantic* definitions which is new and important.

The main content of this study will therefore be a summary of the best current principles and practice of *formal semantic specification.*

0.1 Abstract vs. Concrete Semantics

In preparing the critical survey we studied several contrasting methods of programming language definition: (a) the method used by the VIENNA group to define PL/I and which has subsequently been applied to other languages, such as ALGOL-60; (b) the SEMANOL method, developed by a group at TRW for RADC, as a method for specifying the language JOVIAL; (c) the SECD method devised by Landin [7] for defining the semantics of programming languages based on the lambda-calculus; and subsequently [8] elaborated to a method based on the SHARING machine; (d) the AXIOMATIC method introduced by Hoare [9] for specifying the semantics of isolated programming language constructs such as *assignment*, *while-do*, and *if-then-else*, principally for the purpose of providing a formal basis for proving properties of programs built out of these constructs; (e) the OXFORD method devised by Strachey and Scott [6] as an extension of the well-established TARSKI set-theoretic semantics [10] used in mathematical logic for several decades.

The literature on these various methods is extensive, and is only just beginning to include expositions designed for the general computing public (as opposed to the specialist research workers in the field of programming language design and implementation).

There is, in the opinion of the present writer, one outstanding issue which forces itself on the attention of the reader of the programming language definition research literature: the issue of whether to specify semantics in a *concrete*, or an *abstract*, way.

Concrete specifications are those which, in one way or another, involve the description of a MACHINE or INTERPRETER, and take the form, in one way or another, of stating how the interpreter behaves when given a program text, together with some data-describing text, as input.

The interpreter may be described at various levels of conceptual abstraction -- at one end, details of behavior being insisted upon; at the other, merely broad essentials being laid down -- but the general principle at work is the same one: to specify an OPERATIONAL semantics by saying *what will happen* when the defining machine is driven by the defined program in the presence of given data objects.

Abstract specifications of semantics avoid the introduction of the definitional interpreter [11]. They proceed instead by *describing the entity denoted by the program text* -- the "meaning" of the text -- in a way that is independent of any particular concrete realization or representation of that entity.

The advantages of abstract over concrete semantics are similar to those of abstract over concrete syntax: essential features are isolated from the irrelevant detail which accompanies particular realizations of them; a wider range of implementations -- possibly involving completely different conventions, codings, and software technologies -- is admitted; correctness proofs for compilers and interpreters are immeasurably simplified and shortened; documentation of the language is made concise and transparent.

It is in the OXFORD methodology of definition that the principle of abstract semantics reaches its highest development to date. The Oxford definition method will in the opinion of the present writer evolve, with use, into the standard tool of the professional programming language designer for specifying, developing, analyzing, and explaining not only constructs but entire languages.

We therefore turn to a closer look at the ideas involved.

1. The Oxford Definition Method

A recent paper by R. D. Tennant [1] is a good tutorial introduction to what will in this study be referred to as the *Oxford Definition Method,* (ODM) originated by D. Scott and the late C. Strachey at Oxford University in England. According to Strachey [2] their collaboration began in the autumn of 1969. It ended with Strachey's recent untimely death (1975).

The ODM approach takes as its main mathematical point of departure a beautiful and important new mathematical theory, due to Scott, which deals with the foundations of our ideas about computation, especially as these bear upon *approximating* (describing) *infinite objects by means of finite ones.* The details of Scott's theory are quite difficult and will not be given here. In any case, Scott's theory is rather like Dedekind's "Schnitt" theory of the real numbers, in two respects: not only technically, but also in that one is assured, by its existence, that certain intuitive conceptions are "consistent" and "safe", and may be rigorously justified by precise constructions, *without having to perform the constructions* when using the conceptions routinely and intuitively.

The intuitions addressed by Scott's theory concern the collections of *functions* which appear to be the subject-matter of such rich programming languages as the full ("typeless") lambda-calculus of Church and Curry, which underlies so many of the modern high-level programming languages in use today. For the purposes of this study we need only say that Scott's theory rigorously justifies the use, as one's defining metalanguage, of the typeless, classical lambda calculus enriched with a suitable collection of primitives such as the conditional expression and the traditional set-theoretical notation for denoting the set of all functions with a given domain and range. We shall rely on this justification in the simplest and most direct way, namely by using such a metalanguage freely and uncritically in an intuitive, informal fashion. We proceed by going through some examples of the use of ODM.

1.1 Environments

The ODM approach is to define every significant structured entity of a programming language as a mathematical *function*. This is a deliberate attempt to avoid unnecessary specification and to employ *abstract* objects, rather than representations of them, as elements of the specified system.

An easy introductory example of this is the analysis of an *environment* of a computation -- namely the association between the *identifiers* in a program and their *values* or *denotations* -- as a function from the set ID of identifiers to the set D of denotable values. Thus, the environments form a set ENV, and the equation defining it is

$$\mathrm{ENV} = (\mathrm{ID} \to \mathrm{D})$$

where the notation on the right is the traditional one for the set of functions from ID to D. In the ODM framework of ideas, the set $(\mathrm{ID} \to \mathrm{D})$ does not in fact contain *all* functions whatsoever from ID to D, when ID is an *infinite* set, but only those which, in Scott's theory, qualify as "continuous". In the present study we can safely neglect this foundational aspect of ODM and proceed on an informal intuitive basis.

A particular environment ρ:

$$\rho \in \mathrm{ENV}$$

behaves as one would expect: it gives the denotation

$$\rho\xi$$

of every identifier ξ in the set ID. By $\rho\xi$ we mean *the value of the function*

ρ *at the argument* ξ, and we indicate the *application* of a function f to its argument a in general by writing down f followed by a:

$$fa$$

When fa is again a function g (as we shall see, in ODM applications this is very often the case), and when a is itself a function (same comment) we may write

$$fa\ b$$

to indicate

$$(fa)b$$

i.e.

$$g\ b$$

but if ab is a function h then we would write

$$f(ab)$$

to signify that the intended result is fh, not gb. This convention is called *association to the left*. It is convenient to have an explicit notation for *application* of functions to arguments; and we shall sometimes use the solid wedge ▶ for this. Thus

$$f \blacktriangleright x$$

means the same as

$$fx$$

and we can even reverse the direction of the wedge if we like, writing

$$x \blacktriangleleft f$$

The wedge points *from* the function being applied, *to* the argument.

1.2 Stores

It is easy to give, using the ODM approach, a clear and satisfactory analysis of a computer memory setup, and of the mechanism of *assignment* which is present in virtually all programming languages.

A *store* is taken to be a function σ in the set

$$S = (L \to V)$$

of functions from *locations* α in L to *storable values* β in V.

A particular σ is thus a possible *state* of the memory.

Two particular functions may then be defined.

(1)

define:	$contents \in L \to (S \to V)$
by:	$(contents \blacktriangleright \alpha) \blacktriangleright \sigma = \sigma \blacktriangleright \alpha$

and

(2)

define:	$update \in (L \times V) \to (S \to S)$
by:	$(update \blacktriangleright (\alpha, \beta)) \blacktriangleright \sigma = \sigma'$
where:	$\sigma' = \lambda \alpha'.\ \textit{if}\ \alpha = \alpha'\ \textit{then}\ \beta\ \textit{else}\ (\sigma \blacktriangleright \alpha')$

The use of the *lambda notation*

$$\lambda\, x.a$$

to denote the function whose value at t is the result of evaluating the expression a in an environment in which x is bound to t, is assumed to be familiar.

1.3 Expressions

The expressions of a programming language comprise a set EXP of things which, as we shall see in the example of the next section, can also be construed, in the spirit of ODM, as functions. For the present discussion we assume nothing about the structure of expressions, but proceed to discuss the processes involved in assignment under the assumption that each expression ε in EXP, when *evaluated*, yields some *value* which depends on the particular structure of ε and on the environment in which the evaluation takes place.

Assignment involves an expression ε_o which denotes a location, and an expression ε_1 which denotes a value; and the *assignment command:*

$$\varepsilon_o := \varepsilon_1$$

causes the value of ε_1 to be stored in the location which is the value of ε_o. (ε_o might be arbitrarily complex; e.g. it might be

$$A[i+j,\ B[k+i]]$$

where A and B are arrays).

Two functions, L and R (for "left-value" and "right-value") are introduced to analyse this operation.

(1)	*define:* $L \in (\mathrm{EXP} \rightarrow (\mathrm{ENV} \rightarrow (\mathrm{S} \rightarrow (\mathrm{L} \times \mathrm{S}))))$ *by:* (detailed specification of L)
(2)	*define:* $R \in (\mathrm{EXP} \rightarrow (\mathrm{ENV} \rightarrow (\mathrm{S} \rightarrow (\mathrm{V} \times \mathrm{S}))))$ *by:* (detailed specification of R)

The detailed specifications of L and R will depend on the syntactic structures of the expressions ε in EXP. But the general idea is straight forward. $(L \blacktriangleright \varepsilon_o) \blacktriangleright \rho$ is, as the type specification in (1) shows, a function in

$$(\mathrm{S} \rightarrow (\mathrm{L} \times \mathrm{S}))$$

which, when applied to a store σ_o, yields a *location* α and a (possibly altered) store σ_1

(3) $$((L \blacktriangleright \varepsilon_o) \blacktriangleright \rho) \blacktriangleright \sigma_o = (\alpha, \sigma_1)$$

The possibly altered store σ_1 is to allow, in the analysis, for possible *side-effects* which may occur during the evaluation of ε_0 in ρ. Intuitively, α is the location in the memory which ε_0 denotes in ρ.

In carrying out the operation $\varepsilon_0 := \varepsilon_1$, then, the *first* step is to do equation (3). The *second* step is to get the value of ε_1 by using the function $\mathcal{R}$. By referring to the type of $\mathcal{R}$ given in (2), we see that

$$(\mathcal{R}\ \varepsilon_1)\rho$$

is a function in

$$(S \rightarrow (V \times S))$$

which, when applied to a store σ_1, yields a *value* β and a (possibly altered, again because of side effects of the evaluation process) store σ_2:

(4) $$((\mathcal{R}\ \varepsilon_1)\rho)\sigma_1 \;=\; (\beta,\ \sigma_2)$$

Intuitively, β is the value denoted by ε_1 in the environment ρ of identifier-bindings when the memory is in state σ_1. In this model of storage and assignment, the *environment* is not changed by assignments, but rather the *state of the memory*, (the *store*), is what changes.

With α and β as given by (3) and (4), the remaining work involved in the meaning of $\varepsilon_0 := \varepsilon_1$ is given by:

(5) $$(\mathit{update}\ (\alpha,\ \beta\ ))\ \sigma_2 = \sigma_3$$

That is, the overall effect of executing $\varepsilon_0 := \varepsilon_1$ when the environment is ρ and the store is σ_0 is to change the store to σ_3.

If the meaning of commands α in the set CMD of the language is given by a function:

$$\mathcal{C} \in (\mathrm{CMD} \rightarrow (\mathrm{ENV} \rightarrow (S \rightarrow S)))$$

i.e. $(\mathcal{C}\ \gamma)\rho$ is a *state transition function* θ in $(S \rightarrow S)$ for each command γ and environment ρ, then part of the detailed specification of $\mathcal{C}$ would be:

$$(C(\varepsilon_o := \varepsilon_1)\rho)\sigma_o = \sigma_3 \quad \textit{where}$$
$$\sigma_3 = \textit{update}\ (\alpha,\ \beta)\ \sigma_2 \quad \textit{where}$$
$$(\beta,\ \sigma_2) = ((R\ \varepsilon_1)\rho)\sigma_1 \quad \textit{where}$$
$$(\alpha,\ \sigma_1) = (L\ \varepsilon_o)\rho)\ \sigma_o$$

or equivalently

$$C(\varepsilon_o := \varepsilon_1)\rho = \lambda\sigma_o.\ \textit{update}\ (\alpha,\ \beta)\sigma_2\ \textit{where}\ (\beta,\ \sigma_2) = ((\ R\ \varepsilon_1)\rho)\sigma_1\ \textit{where}\ (\alpha,\ \sigma_1) = ((\ L\ \varepsilon_o)\rho)\sigma_o.$$

using the lambda notation to define the state transformation θ explicitly.

1.4 Commands

In general, *commands* are program elements γ in a set CMD thereof whose syntax will be part of the specification of the programming language. There will be a *semantic function* C which gives the meaning of a command γ, just as the semantic functions L and R gave the "left" and "right" meanings of expressions ε in EXP.

The specification of the language will largely consist of the detailed definition of these semantic functions and of the syntactic structure of their (textual) domains.

A command γ is construed as denoting, in each environment ρ, a *state transition function* $\theta = C\gamma\rho$

$$\theta \in (S \rightarrow S).$$

The execution of a sequence $\gamma_o\ \gamma_1\ .\ .\ .\ \gamma_n$ of commands thus corresponds to the *transition* of the initial state σ_o successively to $\sigma_1 = C\gamma_o\ \sigma_o$, to $\sigma_2 = C\gamma_1\ \sigma_1$, etc. The *open wedge* $\triangleright$ notation represents *composition* of functions: $f \triangleright g$ being the function whose application to x yields the result given by:

$$(f \triangleright g) \blacktriangleright x = f \blacktriangleright (g \blacktriangleright x).$$

As with $\blacktriangleright$, the *direction* of $\triangleright$ may be reversed without changing the meaning, provided we also reverse its two operands:

$$f \triangleright g = g \triangleleft f$$

Then the composition of state transitions $\theta_o \ldots \theta_n$ (in the order of their application) is the state transition:

$$\theta = \theta_o \triangleleft \ldots \triangleleft \theta_n$$

and we may write the entire transition as:

$$\sigma_o \blacktriangleleft \theta = \sigma_o \blacktriangleleft (\theta_o \triangleleft \ldots \triangleleft \theta_n)$$
$$= \sigma_n$$

With this notation, the sequence of commands $\gamma_o \ldots \gamma_n$ corresponds to the state transition

$$(\mathcal{C}\gamma_o\rho) \triangleleft \ldots \triangleleft (\mathcal{C}\gamma_n\rho).$$

1.5 Sequencing

One of the more troublesome aspects of programming language semantics is the specification of "flow of control".

Suppose that $\gamma = \gamma_o \; ; \; \gamma_1$ is the command consisting of "γ_o followed by γ_1" -- to use the notation standard in many programming languages. Now in order to specify the state-transition for γ:

$$\mathcal{C}(\gamma_o \; ; \; \gamma_1)\rho = \theta$$

we shall have to know whether the execution of γ_o will involve a *jump* or not -- which in general we cannot know until γ_o is actually being executed, since, in general, whether γ_1 is the (dynamically) next command after γ_o, or whether some *other* command somewhere else in the program will be the dynamically next one, is not a fixed feature of the command γ_o ; γ_1, but of the *entire program* and its detailed potential interactions with possible environments and stores.

The ODM approach deals with this complication via an ingenious device, known as the device of *continuation functions*, which was hit upon independently around 1970 by several authors, including F. L. Morris [3] of Syracuse University.

The essential idea of the continuation function method is to define the semantics of a program element (such as a sequential command γ_o ; γ_1) *not* in isolation but rather *in context* as part of a complete program. (There is a flavor here of passing from *context-free semantics* to *context-sensitive semantics*.) To specify the transition function denoted by γ_o ; γ_1, in an environment ρ we must therefore, according to the continuation method, supply a context θ, the nature of which will be derived as the discussion develops, and define the transition as *depending upon* θ.

(1) $$(P(\gamma_o ; \gamma_1)\rho)\theta$$

as well as upon ρ. The analysis, by F. L. Morris and the others, of the nature of θ in this application, leads to the construing of its as a *transition*. Since (1) must also denote a transition, i.e. must be in the set

$$C = S \to S$$

we have that the type of P is

$$(CMD \to (ENV \to (C \to C))).$$

The intuitive interpretation of the "continuation function" θ in (1) is that it is the function corresponding to the remainder of the operation specified by the entire program, factored out (literally: in the sense of functional composition) and to be executed in the event that γ_o ; γ_1 *does not involve a jump to some other part of the program*. That is, θ represents the "normal continuation" of the flow of control.

As F. L. Morris remarks: [3]

> "the function compiled for the subexpression should be passed as an argument something which says what more is waiting to be done, so that the subexpression can decide whether to do it or not."

Thus, if the execution of γ does not involve a jump, we would specify that

$$(((P\ \gamma)\rho)\theta)\sigma_o = \theta(((C\ \gamma)\rho)\sigma_o)$$

i.e. the next state after σ_o would be reached by *first* going to the state

$$\sigma_1 = (C\ \gamma\rho)\sigma_o.$$

which is reached by executing the command γ, and *then* applying to *that* state, the *normal continuation* θ, to get:

$$\sigma_2 = \theta\sigma_1.$$

In the event of a jump, some other ("abnormal") transition, or continuation, will be specified as having to be applied to σ_1 to get σ_2. That is, some commands will "ignore their normal continuations".

In terms of this idea, the definition of P for the case of a sequencing command $\gamma_o\ ;\ \gamma_1$ is then:

$$P\ (\gamma_o\ ;\ \gamma_1)\rho\theta\sigma = (P\gamma_o\rho)(P\gamma_1\rho\theta)\sigma$$

which says, intuitively, that if θ is the normal continuation, then the transition:

(1) $$P(\gamma_o\ ;\ \gamma_1)\rho\theta$$

is explained as giving to the routine:

(2) $$P\ \gamma_o\rho$$

the normal continuation:

(3) $$P\ \gamma_1\rho\theta$$

so that, if in the execution of (2) there are no jumps, the remainder of the computation will be the application of the transition (3) to the state produced by executing (2).

The analysis of the meaning of a jump command

$$\textit{goto}\ \varepsilon$$

is now clear in terms of continuation functions. We shall have:

$$(\mathcal{P}(\textit{goto}\ \varepsilon)\rho)\theta = \theta'$$

where θ' is the transition function which is obtained by evaluating the "label expression" ε in the environment ρ:

$$\theta' = (\mathcal{E}\ \varepsilon)\rho$$

Thus the ODM approach yields a simple and straight forward analysis of the delicate concept of a *label value*. In Tennent's definition [1] of GEDANKEN [4] this is used very effectively to explain the semantics of the control aspects of that extremely powerful language, in which one can construct such unorthodox control regimes as co-routines and quasiparallel processes.

2. ASCL

The programming language ASCL ("A Small Continuation Language") is a language devised by C. Strachey and C. Wadsworth in [5] to illustrate the ODM concept of *continuation functions*. It also is well suited to illustrate the other features of ODM definitions, and we shall now go through its definition, as a more extended example of the ODM approach.

2.1 Syntactic Categories of ASCL

Four sets of syntactic entities:

ID CMD EXP FN

are declared and given the names exhibited in the line above. Simultaneously, four sets of (metalinguistic) variables are declared, namely the lower-case Greek letters

$$\xi \quad \gamma \quad \varepsilon \quad \phi$$

These variables, with or without numeric subscripts, will be used in the definitions with the ranges:

$$\xi \in \text{ID} \quad \gamma \in \text{CMD} \quad \varepsilon \in \text{EXP} \quad \phi \in \text{FN} .$$

Informally, we are told that

- ID is the set of *identifiers* of ASCL. No particular assumptions need be made (for the purposes of the definition) about the members of ID.
- CMD is the set of *commands* (this is the same as what is commonly called a *statement;* but the word "command" has the more appropriate "imperative" connotation).
- EXP is the set of *expressions* of ASCL.
- FN is the set of *primitive commands* of ASCL.

2.2 Syntax Equations for ASCL

The following two equations assert that CMD and EXP are the unions of several disjoint sets of syntactic *subcategories*, or *cases:*

CMD = FN + DUM + SEQ + IFC + WDO + GTO + BLK + ESC

EXP = ID + YES + NO + IFX + CMP

These "cases equations" give part of the information contained in a BNF equation, namely the breakdown into cases of the syntactic category denoted by the name on the left hand side. The remaining information, namely the structural descriptions of the syntactic subcategories, is given by a *structural equation* for each such subcategory. In the present example, the subcategories are:

DUM SEQ IFC WDO GTO BLK ESC

YES NO IFX CMP

Informally, we remark that:

- DUM is the set whose only element is the *dummy* command *"skip"*.
- YES is the set whose only element is the Boolean constant *"true"*.
- NO is the set whose only element is the Boolean constant *"false"*.
- SEQ is the set of compound commands often written as the (semi-colon-separated) *sequence* $\gamma_o ; \gamma_1$.
- IFC is the set of *conditional commands* often written *if* ε *then* γ_o *else* γ_1, or as: $\varepsilon \rightarrow \gamma_o, \gamma_1$.
- WDO is the set of *repetitive constructs* often written as *while* ε *do* γ.
- GTO is the set of *jumps* often written as *goto* ε.
- BLK is the set of *blocks* or *programs* often written as: *begin* $\gamma_o; \xi_1: \gamma_1; \ldots ; \xi_n: \gamma_n$ *end*, with each ξ_i being a *label* (or "tag") in the body of the block.
- ESC is the set of *escape commands* which return a value from a subroutine; these are often written as: *result is* ε, or *return* ε, or *escape with* ε.
- IFX is the set of *conditional expressions*, often written as *if* ε_o *then* ε_1 *else* ε_2, or as $\varepsilon_o \rightarrow \varepsilon_1, \varepsilon_2$.
- CMP is the set of *value-returning computations*, sometimes written as: *val of* γ.

These informal remarks are not part of the formal definition; nor are the particular concrete representations, such as: *if* ε *then* γ_0 *else* γ_1, for the various syntactic constructions. The syntactic structure is *abstract.* The equations which follow are descriptions of each syntactic construct as *a function of selectors.* To decompose each construct into its immediate constituents one merely *applies* the construct (as a function) to the selector(s) in its domain.

Note that FN and ID are not cited as *subcategories* since they have already been cited as *categories.*

Each subcategory is defined by a characteristic construction. The general form of the construction is given by an equation whose left-hand side is the name of the subcategory, and whose right-hand side is a *structural description* showing the immediate constituents and their categories, or an explicit listing of the members of the category (if the category is a finite set). In the ASCL subcategories there are three of the second kind, namely:

$$\mathrm{DUM} = \{\mathit{skip}\}$$

$$\mathrm{YES} = \{\mathit{true}\}$$

$$\mathrm{NO} = \{\mathit{false}\}$$

and eight of the first kind, namely:

$$\mathrm{SEQ} = \{\langle \mathit{first}\ \gamma_0\rangle \quad \langle \mathit{next}\ \gamma_1\rangle\}$$

$$\mathrm{IFC} = \{\langle \mathit{test}\ \varepsilon\rangle \quad \langle \mathit{positive}\ \gamma_0\rangle \quad \langle \mathit{negative}\ \gamma_1\rangle\}$$

$$\mathrm{WDO} = \{\langle \mathit{condition}\ \varepsilon\rangle \quad \langle \mathit{repeat}\ \gamma\rangle\}$$

$$\mathrm{GTO} = \{\langle \mathit{label}\ \varepsilon\rangle\}$$

$$\mathrm{BLK} = \{\langle \mathit{entry}\ \gamma_0\rangle \quad \langle \mathit{tag}_1\ \xi_1\rangle \quad \langle \mathit{command}_1\ \gamma_1\rangle \ \ldots\ \langle \mathit{tag}_n\ \xi_n\rangle \quad \langle \mathit{command}_n\ \gamma_n\rangle\}$$

$$\mathrm{ESC} = \{\langle \mathit{value}\ \varepsilon\rangle\}$$

$$\mathrm{IFX} = \{\langle \mathit{premiss}\ \varepsilon_0\rangle \quad \langle \mathit{conclusion}\ \varepsilon_1\rangle \quad \langle \mathit{alternative}\ \varepsilon_2\rangle\}$$

$$\mathrm{CMP} = \{\langle \mathit{code}\ \gamma\rangle\}$$

These structural equations declare that DUM is a finite set, whose only member is the entity *skip*. This is intended (as the semantic specifications which follow will formally show) to be a "dummy" command whose execution has no effect on the state of the computation. The sets YES and NO are, likewise, singletons, whose members are intended respectively to be the ASCL constants denoting *truth (tt)* and *falsehood (ff)*.

Each of the remaining equations has a right-hand side which is a *finite set of ordered pairs*, i.e., *a function with finite domain*. The elements of the domains of these functions are called *selectors*. Thus, the domains for each of these eight functions are:

SEQ {*first*, *next*}

IFC {*test*, *positive*, *negative*}

WDO {*condition*, *repeat*}

GTO {*label*}

BLK {*entry*, *tag*, *body*}

ESC {*value*}

IFX {*premiss*, *conclusion*, *alternative*}

CMP {*code*}

For example, a *conditional command* γ (i.e. a member of IFC) has three immediate constituents: they are found by applying γ to each of the three selectors *test*, *positive*, *negative*:

$$\varepsilon = \textit{test}\ \gamma$$

$$\gamma_0 = \textit{positive}\ \gamma$$

$$\gamma_1 = \textit{negative}\ \gamma$$

where we are writing the function γ to the right of its argument.

In general, application of a function to its argument may be indicated by writing the function on the left or on the right, whichever is convenient. To help, avoid confusion as to which direction is intended, we use the *solid wedge* notation ▶ for *application*. The thick end is where the *function* goes; the pointed end is the "point of application", where the argument goes. Thus

$$f \blacktriangleright x$$

is: *f applied to x*. This is exactly the same as

$$x \blacktriangleleft f$$

which is read: *f applied to x*, i.e. it is read from right to left. The application symbol ▶ can be used in any direction, including both vertical ones. Thus, *f applied to x* may also be written

$$\begin{array}{c} f \\ \blacktriangledown \\ x \end{array}$$

and

$$\begin{array}{c} x \\ \blacktriangle \\ f \end{array}$$

It is of course the usual convention to omit an explicit symbol for application; we shall do so only when it is clear "which direction application goes" in the given context.

In particular, the application of a syntactic construct to a selector goes from *right to left*, e.g.

$$first\ \gamma = first \blacktriangleleft \gamma$$

with the intended "pun" that the meaning

$$first\ \gamma = first \blacktriangleright \gamma$$

could also be thought of. It is quite usual to think of selectors as *functions* which "pick out" immediate constituents from a syntactic construct. For example, in LISP a nonempty list L has a *head (car)* and a *tail (cdr)*. We write *(car* L*)*, *(cdr* L*)* thinking of *car*, *cdr* as functions and L as argument. But in our system the same notation is used with the opposite wedge direction intended. Each structural equation has the reading: the set on the left is the set of all functions having the form exhibited on the right. For example the equation

$$\mathrm{SEQ} = \{\langle \mathit{first}\ \gamma_o\rangle\ \langle \mathit{next}\ \gamma_1\rangle\}$$

if written out rigorously would be:

$$\mathrm{SEQ} = \{\mathit{first},\ \mathit{next}\} \rightarrow \mathrm{CMD}$$

while the equation

$$\mathrm{IFC} = \{\langle \mathit{test}\ \varepsilon_o\rangle\ \langle \mathit{positive}\ \gamma_o\rangle\ \langle \mathit{negative}\ \gamma_1\rangle\}$$

would be

$$\mathrm{IFC} = \{\mathit{positive},\ \mathit{negative}\} \rightarrow \mathrm{CMD},\ \{\mathit{test}\} \rightarrow \mathrm{EXP}.$$

The second translation illustrates the "extended functionality notation". The usual notation for *the set of functions from the set A to the set B* is

$$\mathrm{A} \rightarrow \mathrm{B}$$

so that the assertion:

$$f \in (\mathrm{A} \rightarrow \mathrm{B})$$

means:

> f is defined for each element x of A, and when x is in A the value of f at x is an element of B

we generalise this notation to a list of such domain → range elements:

$$A_1 \to B_1, \ldots, A_n \to B_n$$

so that the assertion:

$$f \in (A_1 \to B_1, \ldots, A_n \to B_n)$$

means:

f is defined for each element of the set $A_1 \cup \ldots \cup A_n$, and when x is in A_i, the value of f at x is in the set B_i, $i = 1, \ldots, n$

For this extended notation to make sense it is of course necessary that the sets $A_1, \ldots A_n$ be *disjoint.*

The syntactic equations for ASCL are intended as a characterization of a family of sets -- the syntactic *classes* (categories and subcategories) of ASCL -- whose union is the set of ASCL *texts* or *program elements.* The information essentially contained in the equations is that each program element is either *simple* (such as an identifier or a primitive command or a primitive expression) or *composite;* and that in the second case it has a unique analysis into *immediate constituents* of given syntactic classes which can be obtained formally by applying the program element (as a *function*) to its selectors (as *argument).* Those who are wedded to traditional *mores* are permitted to view this immediate constituent analysis, by a suitably contrived symbolic pun, as the application of the various selectors (as *functions)* to the text (as *argument).*

So much, for the time being, for the syntax of ASCL and for the several definitional techniques for specifying syntactic structures. We proceed next to its semantics, the heart of the matter.

2.4 Value Domains of ASCL

There are six sets which are specified to be *semantic domains* for ASCL. They are:

T S C D E K

The first two are *primitive domains.*

- T is the domain of *truth values*
- S is the domain of *machine states.*

The other five are *compound domains:*

- C = S → S is the set of *command continuations,* or *state transition functions:*
- D = T + C is the set of *denotations,* i.e., the set of entities which can be denoted by identifiers;
- E = T + C is the set of *expression results,* i.e., the set of entities which can be the results of computing the value of an expression;
- K = D → C is the set of *expression continuations,* i.e., the set of entities which correspond to the possible contexts which can arise in a computation when an expression is evaluated.

In the subsequent parts of the definition of ASCL the following variables (with or without subscripts) will be used to range over the sets shown:

θ	over	C
δ	over	D
δ	over	E
κ	over	K

The sets of *continuations* will be explained shortly. For the present, we simply accept the declaration of six sets, two outright, and four as simple constructs of them. D and E are identical as sets in ASCL; they are declared separately because in general the entities which can be *denoted by identifiers* are not always all of those which can be computed, i.e. which can be *values of expressions*.

In a richer language than ASCL there would be further value domains such as the set of *integers*, the set of *characters*, and the set of *locations*.

2.5 Semantic Functions of ASCL

The set ENV of *environments* is a composite domain whose elements have two constituents: a *binding* and a *resumption*. The *binding* is a function

$$\beta \in (\mathrm{ID} \rightarrow \mathrm{D})$$

which gives, for each identifier ξ in ID, the entity $\beta \blacktriangleright \xi$ to which ξ is *bound*, or which ξ *denotes*. The *resumption* is a function

$$\kappa \in \mathrm{K}$$

which is an *expression continuation* whose significance will be made clear shortly.

One could therefore take (as is done, e.g. in C. Strachey and C. Wadsworth [5]) an element ρ of ENV to be the ordered pair $<\beta\ \kappa>$ of its two immediate constituents. We prefer to use the same technique here as in the case of syntactical constituents: to construe each ρ in ENV as a function with domain

$$\mathrm{ID} \cup \{res\}$$

so that we can write the equation

$$\mathrm{ENV} = (\mathrm{ID} \rightarrow \mathrm{D}),\ \{res\} \rightarrow \mathrm{K}.$$

Two particular functions, $\mathcal{P}$ and $\mathcal{E}$, are declared to have the functionalities:

$$\mathcal{P} \in (\text{CMD} \rightarrow (\text{ENV} \rightarrow (\text{C} \rightarrow \text{C})))$$

$$\mathcal{E} \in (\text{EXP} \rightarrow (\text{ENV} \rightarrow (\text{K} \rightarrow \text{C})))$$

and their detailed definition and explication is the central part of the specification of ASCL. We shall now discuss the ideas underlying the formal definition of ENV, $\mathcal{P}$ and $\mathcal{E}$.

2.6 Oxford Semantics

The ideas underlying the formal definitions of ENV, $\mathcal{P}$ and $\mathcal{E}$ are the heart of ODM. Certain aspects of these ideas will be unfamiliar at first, and require faith and practice for full acceptance. Persistence is strongly recommended: these are ideas whose time has come.

The ODM plan is to explain the meaning of programs and program elements by supplying *semantic functions* (here, for ASCL, they are the functions $\mathcal{P}$ and $\mathcal{E}$; in general they are a set of functions, one for each main syntactic category of the language being defined). *The meaning of a text is then the entity yielded by applying the appropriate semantic function to that text.*

Thus, in the case of ASCL, every text τ is either a *command* γ or an *expression* ε. In the first case we apply $\mathcal{P}$ to γ; in the second case, we apply $\mathcal{E}$ to ε. In either case, the entity yielded *is* the meaning. By referring to the functionality declarations for $\mathcal{P}$ and $\mathcal{E}$:

$$\mathcal{P} \in (\text{CMD} \rightarrow (\text{ENV} \rightarrow (\text{C} \rightarrow \text{C})))$$

$$\mathcal{E} \in (\text{EXP} \rightarrow (\text{ENV} \rightarrow (\text{K} \rightarrow \text{C})))$$

we see that we have:

$$\mathcal{P} \blacktriangleright \gamma \in \text{ENV} \to (\text{C} \to \text{C})$$

$$\mathcal{E} \blacktriangleright \varepsilon \in \text{ENV} \to (\text{K} \to \text{C})$$

that is, the meaning of a command γ or of an expression ε is a function which can be applied to any *environment* ρ. *It is this function which is the meaning of the ASCL text* τ.

Our present task is to grasp the intention behind this. Let us first confine our attention to *commands*. We note that the function $\mathcal{P} \blacktriangleright \gamma$ provides, for each environment ρ in ENV, a function:

$$(\mathcal{P} \blacktriangleright \gamma) \blacktriangleright \rho \in (\text{C} \to \text{C})$$

which, when applied to a *state-transition-function* θ_0 in C, yields another *state-transition-function* θ_1 in C.

Let us recall the distinction between an *open subroutine* and a *closed subroutine*.

An *open* subroutine is a piece of machine code θ which, when executed, causes the machine to make a transition from the state σ_0 *before* the execution of θ, to the state σ_1 *after* the execution of θ. Thus θ is essentially a state-transtion-function θ: $S \to S$ from machine states to machine states.

A *closed* subroutine μ is a piece of machine code containing a "return address" parameter θ which must be specialized to (the address of) an open subroutine before $\mu \blacktriangleright \theta$ becomes an open subroutine. In other words, $\mu \blacktriangleright \theta$ is an open subroutine, for each open subroutine θ; which is to say that the functionality of μ is "open subroutines to open subroutines":

$$\mu \in (S \to S) \to (S \to S)$$

or

$$\mu \in (C \to C).$$

In terms of this distinction, $((\mathcal{P} \blacktriangleright \gamma) \blacktriangleright \rho) = \mu$ is a closed subroutine; in order to run it as part of a program we must provide it with a θ in $C = S \to S$ as a *continuation*, by e.g. giving it as an argument the address of θ as *return address*.

This account has to be slightly complicated for the case that the text is an expression ε instead of a command γ.

In order to run the "closed subroutine" $\pi = (\mathcal{E} \blacktriangleright \varepsilon) \blacktriangleright \rho$ as part of a program we must provide it with (the return address of) a *"special open subroutine"* κ which will yield a simple open subroutine $\pi \blacktriangleright \kappa$ corresponding to the whole program.

However, since in general the meaning of π will yield both a *value* δ (that of the expression ε in the environment ρ) and a new machine state σ (that produced as a side effect of the process of computing δ) the special open subroutine κ must be one which takes both δ and σ as its inputs in order to produce the final state of the whole program. It is convenient to construe κ as a function in the set:

$$E \to (S \to S)$$

that is, we regard the whole program execution as producing the state

$$(\kappa \blacktriangleright \xi) \blacktriangleright \sigma_1$$

from the state σ_o, where δ is the value of ε in ρ, and where σ_1 is the state produced from σ_o by evaluating ε in ρ with the machine initially in σ_o.

So the functionality of π is "special open subroutines to open subroutines":

$$\pi \in (E \rightarrow (S \rightarrow S)) \rightarrow (S \rightarrow S)$$

or

$$\pi \in (K \rightarrow C).$$

The function κ is also called a *continuation*, since it corresponds to the open subroutine which *continues* the program to completion; but it is called an *expression continuation* in order to distinguish it from the simpler kind of continuations in $(S \rightarrow S)$, which are called *command continuations.*

Let us summarise this discussion of continuations.

A program text, which may be a command γ or an expression ε, is "compiled" into a routine by applying to it the appropriate semantic function, $\mathcal{P}$ or $\mathcal{E}$. The routines thus compiled are *closed.* They require, at "run time", to be supplied with open routines as *continuations*, which will take over from them and continue the computation to completion. Thus the closed routines become open routines when supplied with appropriate open subroutines:

$\mu \blacktriangleright \theta$ is *open:* i.e. is in $S \rightarrow S$

θ is *open:* i.e. is in $S \rightarrow S$

$\therefore$ μ is *closed:* i.e. is in $(S \rightarrow S) \rightarrow (S \rightarrow S)$

where $\mu = (\mathcal{P} \blacktriangleright \gamma) \blacktriangleright \rho$

$\therefore$ $\mathcal{P}$ is in $(CMD \rightarrow (ENV \rightarrow ((S \rightarrow S) \rightarrow (S \rightarrow S))))$

$\pi \blacktriangleright \kappa$ is *open:* i.e. is in $(S \rightarrow S)$

κ is *open:* i.e. is in $(E \rightarrow (S \rightarrow S))$

$\therefore$ π is *closed:* i.e. is in $(E \rightarrow (S \rightarrow S)) \rightarrow (S \rightarrow S)$

where $\pi = (\mathcal{E} \blacktriangleright \varepsilon) \blacktriangleright \rho$

$\therefore$ $\mathcal{E}$ is in $(EXP \rightarrow (ENV \rightarrow ((E \rightarrow (S \rightarrow S)) \rightarrow (S \rightarrow S))$

2.7 Semantic Equations

The detailed specification of $\mathcal{P}$ and $\mathcal{E}$ is given by writing equations. We shall have eight equations, C1 to C8 for $\mathcal{P}$ and five equations E1 to E5 for $\mathcal{E}$.

FN

C1. $\mathcal{P} \blacktriangleright \phi \blacktriangleright \rho$ = some given function μ in $C \to C$, to be associated with ϕ.

A primitive command ϕ in the set FN yields some constant routine μ in each environment ρ

DUM

C2. $\mathcal{P} \blacktriangleright \mathit{skip} \blacktriangleright \rho \blacktriangleright \theta = \theta$

The command *skip* compiles to the identity function in $(C \to C)$

SEQ

C3. $\mathcal{P} \blacktriangleright \{< \mathit{first}\ \gamma_o> \ <\mathit{second}\ \gamma_1>\} \blacktriangleright \rho \blacktriangleright \theta = \mathcal{P} \blacktriangleright \gamma_o \blacktriangleright \rho\ (\mathcal{P} \blacktriangleright \gamma_1 \blacktriangleright \rho\ \theta)$

The sequence of two commands $\gamma_o\ ;\ \gamma_1$ compiles to code which, when given the continuation routine θ, runs the code for γ_o to which it gives the continuation routine $\mathcal{P} \blacktriangleright \gamma_1 \blacktriangleright \rho \blacktriangleright \theta$. This latter routine is the code for γ_1 furnished with the continuation θ. Thus (running $[\gamma_o\ ;\ \gamma_1]$ - followed-by-θ) is the same as (running $[\gamma_o]$-followed-by-(running γ_1-followed-by-θ)).

IFC

C4. $\mathcal{P} \blacktriangleright \{<\mathit{test}\ \varepsilon> \ <\mathit{positive}\ \gamma_o> \ <\mathit{negative}\ \gamma_1>\} \blacktriangleright \rho \blacktriangleright \theta$

$= (\mathcal{E} \blacktriangleright \varepsilon \blacktriangleright \rho)\ (\mathit{cond}\ (\mathcal{P} \blacktriangleright \gamma_o \blacktriangleright \rho \blacktriangleright \theta,\ \ \mathcal{P} \blacktriangleright \gamma_1 \blacktriangleright \rho \blacktriangleright \theta))$

The conditional command *if* ε *then* γ_o *else* γ_1 compiles to code which, when given the continuation routine θ, carries out either γ_o-followed-by-θ or γ_1-followed-by-θ, according as the expression ε evaluates to *tt* or *ff*. The function *cond* thus must be of the type

$$(C \times C) \to (T \to C)$$

so that the code

$$cond\ (P \blacktriangleright \gamma_o \blacktriangleright \rho \blacktriangleright \theta,\ P \blacktriangleright \gamma_1 \blacktriangleright \rho \blacktriangleright \theta)$$

has the type

$$(T \rightarrow C)$$

of an expression continuation which can be used by the closed routine

$$(E \blacktriangleright \varepsilon \blacktriangleright \rho)$$

as the continuation to which it transfers control after obtaining the value of ε in ρ. The function *cond* is of course defined by :

$$cond\ (\theta_1,\ \theta_2) > tt = \theta_1$$

$$cond\ (\theta_1,\ \theta_2) > ff = \theta_2.$$

WDO C5. $P \blacktriangleright \{\langle condition\ \varepsilon\rangle\ \langle repeat\ \gamma\rangle\} \blacktriangleright \rho \blacktriangleright \theta$

$= Y \blacktriangleright (\lambda\ \theta'.(E \blacktriangleright \varepsilon \blacktriangleright \rho)\ (cond\ (P \blacktriangleright \gamma \blacktriangleright \rho \blacktriangleright \theta',\ \theta)))$

This equation gives the semantics of the command:

while ε *do* γ

which is the normal while-loop which repeatedly evaluates ε, and each time ε is found to be true executes γ; the first time ε is found to be false marks the completion of the event asked for by the command.

The equation says that P produces code for this command which, when run with the continuation routine θ as its follow-on, has the effect of the code displayed on the right-hand side.

Let us examine the right-hand side:

$$Y \blacktriangleright (\lambda\ \theta'\ .\ (E \blacktriangleright \varepsilon \blacktriangleright \rho) \blacktriangleright (cond \blacktriangleright (P \blacktriangleright \gamma \blacktriangleright \rho \blacktriangleright \theta',\ \theta)))$$

and figure out what it does.

The function Y is the *minimal fixed point operator* which obeys the characteristic law:

$$f \blacktriangleright (Y \blacktriangleright f) = Y \blacktriangleright f$$

for every function f in its domain.

Consider the function H, where:

$$H \blacktriangleright \theta' \;=\; (\mathcal{E} \blacktriangleright \varepsilon \blacktriangleright \rho) \blacktriangleright (\mathit{cond} \blacktriangleright (\mathcal{P} \blacktriangleright \gamma \blacktriangleright \rho \blacktriangleright \theta', \theta))$$

For any particular θ', $H \blacktriangleright \theta'$ is a state-transformation whose effect on a state σ is this: evaluate ε in ρ with state σ; let the result be δ and the new state σ'. If δ is *tt*, carry out γ in ρ with initial state σ' *and continuation* θ'; but if δ is *ff*, simply carry out the continuation θ on the state σ'. In this account, if we identify θ' with $H \blacktriangleright \theta'$, we get a spelling out of the *repetitive* nature of the while-loop. But the argument of Y in the right-hand side of equation C5 is the function H:

$$H = \lambda\, \theta' \,.\, (\mathcal{E} \blacktriangleright \varepsilon \blacktriangleright \rho) \blacktriangleright (\mathit{cond} \blacktriangleright (\mathcal{P} \blacktriangleright \gamma \blacktriangleright \rho \blacktriangleright \theta', \theta))$$

and so the right-hand side of C5 describes the minimal solution of the equation

$$\theta' = H \blacktriangleright \theta',$$

which is just what is required.

GTO | C6. $\mathcal{P} \blacktriangleright \{\langle \mathit{label}\ \varepsilon\rangle\} \blacktriangleright \rho \blacktriangleright \theta = (\mathcal{E} \blacktriangleright \varepsilon \blacktriangleright \rho) \blacktriangleright \mathit{Jump}$

This is the equation which gives the meaning of the *goto* ε command. It says that if the evaluation of ε in ρ with state σ is a continuation θ', and if the evaluation alters the state to σ', the effect of *goto* ε will be

$$\mathcal{P} \blacktriangleright \mathit{goto}\ \varepsilon \blacktriangleright \rho \blacktriangleright \theta \blacktriangleright \sigma \;=\; (\mathit{Jump} \blacktriangleright \theta') \blacktriangleright \sigma'$$

which in turn we want to be simply

$$\theta' \blacktriangleright \sigma'$$

no matter what the continuation θ is. For this to work out, *Jump* must be the *expression continuation*

$$\mathit{Jump}\ \varepsilon\ E \rightarrow (S \rightarrow S)$$

which satisfies the equation

$$(Jump \blacktriangleright \theta') = \theta'$$

for all θ' in E. We recall that

$$\begin{aligned} E &= T + C \\ &= T + (S \to S). \end{aligned}$$

Application of *Jump* to a truth value is a run-time type error; we therefore complete the definition of *Jump* by

$$Jump \blacktriangleright tt = Jump \blacktriangleright ff = \lambda\ \sigma.\sigma_{error}$$

a continuation which will abort any computation in a distinguished error state.

BLK

C7. $$P \blacktriangleright \{\langle entry\ \gamma_o\rangle\ \ \langle tag_1\ \xi_1\rangle\ \ \langle command_1\ \gamma_1\rangle \ldots \langle tag_n\ \xi_n\rangle\ \ \langle command_n\ \gamma_n\rangle\} \blacktriangleright \rho \blacktriangleright \theta = \theta_o$$

where

$$\left.\begin{aligned} \theta_o &= P \blacktriangleright \gamma_o \blacktriangleright \rho' \blacktriangleright \theta_1 \\ \theta_1 &= P \blacktriangleright \gamma_1 \blacktriangleright \rho' \blacktriangleright \theta_2 \\ &\vdots \\ \theta_n &= P \blacktriangleright \gamma_n \blacktriangleright \rho' \blacktriangleright \theta \end{aligned}\right\} \quad (*)$$

and $$\rho' = \rho\ [\theta_1 \ldots \theta_n\ /\ \xi_1 \ldots \xi_n]$$

This is the most elaborate of the semantic equations, but its import is straight forward. It says that the code for a block

$$begin\ \gamma_o\ ;\ \xi_1 : \gamma_1\ ;\ \ldots\ \xi_n : \gamma_n\ end$$

must first set up an extended environment

$$\rho' = \rho\ [\theta_1 \ldots \theta_n\ /\ \xi_1 \ldots \xi_n]$$

in which the identifiers $\xi_1 \ldots \xi_n$ are bound to the continuations $\theta_1 \ldots \theta_n$ described by the equations (*).

The state-transformation for the whole block (with continuation θ) is then θ_o, where (back-substituting the above equations)

$$\theta_o = (P\ \gamma_o\ \rho')\ (P\ \gamma_1\ \rho')\ \ldots\ (P\ \gamma_{n-1}\ \rho')\ (P\ \gamma_n\ \rho')\ \theta.$$

Within the block, occurrences of the labels $\xi_1 \ldots \xi_n$ will be evaluated in the environment ρ' set up at block-entry, and hence will be found to denote the continuations shown, which is the intended meaning.

ESC

C.8	$P \blacktriangleright \{\langle value\ \varepsilon\rangle\} \blacktriangleright \rho \blacktriangleright \theta \;=\; (E \blacktriangleright \varepsilon \blacktriangleright \rho) \blacktriangleright (res \blacktriangleleft \rho)$

This equation describes the meaning of the command

result is ε

which causes control to jump out to the smallest textually surrounding *value of* γ block, with the value of the expression ε. Let us therefore consider this equation together with that for the *value of* γ expression:

CMP

E5.	$E \blacktriangleright \{\langle code\ \gamma\rangle\} \blacktriangleright \rho \blacktriangleright \kappa \;=\; P \blacktriangleright \gamma \blacktriangleright (\rho\ [\kappa/res]) \blacktriangleright Fail$

This equation says that, to carry out the *value of* γ *in* ρ block with continuation κ we set up a modified environment

$$\rho\ [\kappa/res]$$

in which the continuation κ is the resumption component, and run the code for γ with continuation *Fail.* The reason for the failure continuation is that we do not expect the code for γ to continue in this way, but rather to terminate by executing a *result is* ε command and hence continuing with the resumption code denoted by *res,* namely the continuation κ. Thus C.8 can now be seen to describe the intended behavior of the *result is* ε command: evaluation of ε followed by the continuation denoted by the identifier *res.*

Again, it is suitable to define

$$Fail = \lambda\ \sigma.\sigma_{\text{error}}$$

The remaining expression equations are straightforward:

ID E1. $\mathcal{E} \blacktriangleright \xi \blacktriangleright \rho \blacktriangleright \kappa = \kappa \blacktriangleright (\rho \blacktriangleright \xi) = \xi \blacktriangleleft (\rho \blacktriangleleft \kappa)$

This equation says that we evaluate an identifier by first looking its value up in the environment and then applying the given continuation to this value.

YES E2. $\mathcal{E} \blacktriangleright \mathit{true} \blacktriangleright \rho \blacktriangleright \kappa = \kappa \blacktriangleright \mathit{tt}$

NO E3. $\mathcal{E} \blacktriangleright \mathit{false} \blacktriangleright \rho \blacktriangleright \kappa = \kappa \blacktriangleright \mathit{ff}$

E1, E2, E3 say that identifiers, and the constants *true* and *false*, can be evaluated without side-effects: the machine state is the same after their evaluation as before. E.g.

$$(\mathcal{E} \blacktriangleright \mathit{true} \blacktriangleright \rho \blacktriangleright \kappa) \blacktriangleright \sigma = (\kappa \blacktriangleright \mathit{tt}) \blacktriangleright \sigma$$

says that the routine consisting of the work (evaluating *true*)-followed-by-κ has the same effect on σ as that of the routine $(\kappa \blacktriangleright \mathit{tt})$.

Finally, equation E4 describes the semantics of conditional expressions analogously to equation C4:

IFX E4. $\mathcal{E} \blacktriangleright \{\langle \mathit{premiss}\ \varepsilon_0 \rangle \langle \mathit{conclusion}\ \varepsilon_1 \rangle \langle \mathit{alternative}\ \varepsilon_2 \rangle\} \blacktriangleright \rho \blacktriangleright \kappa$

$= (\mathcal{E} \blacktriangleright \varepsilon_0 \blacktriangleright \rho) \blacktriangleright (\mathit{cond} \blacktriangleright (\mathcal{E} \blacktriangleright \varepsilon_1 \blacktriangleright \rho \blacktriangleright \kappa,\ \mathcal{E} \blacktriangleright \varepsilon_2 \blacktriangleright \rho \blacktriangleright \kappa))$

3. The General Case for ODM

There is an important distinction which must be made between the *semantics* (to say of a given program what function it computes) and the *implementation* of a language (to say how a machine is to be organised so as to carry out the actions which are specified by its programs).

> The role of mathematical semantics is to give a precise, unambiguous definition of *what* programs mean, sufficient to determine their outcome, while remaining uncommitted as to the details of *how* this outcome is to be achieved on a (real or abstract) computing machine.
>
> [2], p. 21.

The technique of describing the meanings of programs as functions over certain domains is to give the same sort of "uncommitted" account of semantics as an abstract syntax is able to give of syntax: only the *essential* specifications are explicitly formulated, thereby leaving open *all* questions which do not need to be decided.

By contrast, other defintion methods are to a greater or lesser extent *committed to some form of implementation:* even if this is only to the SECD-machine type of stack-oriented interpreter, or to the reduction-rule rewriting systems of a Markov or Church-Curry kind.

4. SEMANOL vs. ODM

The SEMANOL system for programming language definition is subject to several of the limitations which ODM is intended to avoid. As Strachey remarks:

> Most of the work on syntax and some on semantics has been at the level of symbol manipulation--that is to say it has been concerned with the representations (generally on paper) rather than with the mathematical objects represented.
>
> [2], pp. 2, 3.

4.1 Concrete vs Abstract Syntax

SEMANOL is wedded to concrete syntax. Specifically it assumes that all programming languages will have as their syntactic domains *sets of strings of ASCII characters;* that a context-free grammar will be supplied to characterize these sets.

As a consequence of this assumption a large part of a SEMANOL definition is concerned with a mass of detail which has to be abstracted away again by the user of the definition, in order that the definition be structured intelligibly and its complexity controlled sufficiently for the mind to grasp its essentials.

4.2 Interpreter-based Semantics

SEMANOL is itself a programming language; and SEMANOL definitions are programs written in it. The meaning of a program P in a language L is to be found in the process which ensues when a pair (P, D) consisting of P together with an input datum D is given as input to the definition DEF. The output

$$DEF\ (P,\ D)$$

of DEF for this pair as input, is declared to be the output of P for the input D. Thus DEF is a "universal machine" in the sense of Turing: we must program it in SEMANOL, and our understanding of L is then one and the same with our understanding of the program DEF.

For example DEF for the language JOVIAL (73) is neither concise nor clear. It is a 321-page book containing over 2,000 program elements. Its complexity is very large and one is left, having struggled through the details, with less than a firm grasp on the essentials.

In general in order that a SEMANOL program DEF shall serve as a specification of a programming language L one must have a clear comprehension of the language SEMANOL itself. In the nature of the interpreter-oriented approach this means that we have to study *the SEMANOL interpreter* in order to find out what SEMANOL programs, i.e. programming language definitions, really are saying. There are two sources available for this: a Reference Manual for SEMANOL, and a booklet documenting the Interpreter Program (which is written in FORTRAN).

The Reference Manual is

> ... not a tutorial; it is terse, and presupposes considerable familiarity with the underlying ideas of SEMANOL (73). The presentation is in a "top-down" sequence -- many structures are defined in terms of structures not yet defined; thus it is intended for use by experienced (or at least well versed) SEMANOL (73) programmers.
>
> [SEMANOL (73) Reference Manual, p. 1.]

Now it must be admitted that ODM requires the user to be "well-versed" in its own intellectual and notational apparatus: but in the case of ODM this apparatus is no more than the universal, traditional mathematical framework of set-theoretic conventions supplemented by the formalism of the lambda-calculus and the theory of continuous functions devised by Scott.

ODM definitions are concise and precise. Those of SEMANOL are prolix and are exact only by arbitrary fiat (in that a SEMANOL program DEF does what the FORTRAN interpreter program makes it do: one cannot deny that this kind of explanation of semantics does resolve ambiguities). In Tennant [1] an ODM definition is given of Reynolds' enormously powerful and general language GEDANKEN. The definition occupies just over two pages of the Communications of the A.C.M. and involves 24 semantic equations most of which are very short.

The ODM approach is a rigorous, mathematical technique for the specification of programming languages. It offers at last a general notation for semantic specification which is based on a language-independent framework of semantical concepts. By comparison, previous methods (such as that embodied in the SEMANOL system) involve a relatively superficial semantic, and in some cases even syntactic, explication of the language under definition. We close with a quotation summing up the spirit of ODM:

> 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.
>
> D. Scott and C. Strachey [6]

References:

[1] R. D. Tennant. The Denotational Semantics of Programming Languages. C.A.C.M. 19 (1976), pp. 437-453.

[2] C. Strachey. Varieties of Programming Language. Technical Monograph PRG-10, Oxford University Computing Laboratory, 1972.

[3] F. L. Morris. The Next 700 Programming Language Descriptions. (unpublished note), 1970.

[4] J. C. Reynolds. GEDANKEN - A Simple Typeless Language Based on the Principle of Completeness and the Reference Concept. C.A.C.M. 13 (1970), pp. 308-319.

[5] C. Strachey and C. Wadsworth. Continuations, a Mathematical Semantics for Handling Full Jumps. Technical Monograph PRG-11, Oxford University Computing Laboratory, 1974.

[6] D. Scott and C. Strachey. Towards a Mathematical Semantics for Computer Languages. Technical Monograph PRG-6, Oxford University Computing Laboratory, 1971.

[7] P. J. Landin. The Mechanical Evaluation of Expressions. Computer Journal 6 (1964), pp. 308-320.

[8] P. J. Landin. The Next 700 Programming Languages. C.A.C.M. 9 (1966), pp. 157-164.

[9] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. C.A.C.M. 12 (1969), pp. 576-580; 583.

[10] A. Tarski. The Concept of Truth in Formalized Languages. In Logic, Semantics, Metamathematics, Oxford, 1956, pp. 152-278.

[11] J. C. Reynolds. Definitional Interpreters for Higher Order Programming Languages. Proceedings of the 25th ACM National Conference, pp. 717-740.