The Vienna Definition Language

ETER WEGNER

Brown University, Providence, Rhode Island 02912*

The Vienna Definition Language (VDL) is a programming language for defining programming languages. It allows us to describe precisely the execution of the set of all programs of a programming language. However, the Vienna Definition Language is important not only as one definition technique among many others but as an illustration of a new information-structure-oriented approach to the study of programming languages. The paper may be regarded as a case study in the information structure modeling of programming languages, as well as an introduction to a specific modeling technique.

Part 1 includes a brief review and comparison of techniques of language definition, and then introduces the basic data structures and data structure manipulation operators that are used to define programming languages. Part 2 considers the definition of sets of data structures. Part 3 introduces the notation for specifying VDL instructions, and illustrates the definition of a simple language for arithmetic expression evaluation by giving the complete sequence of states (snapshots) that arise during the evaluation of an expression. Part 4 defines a simple block structure language in VDL and considers certain basic design issues for such languages in a more general context. Part 5 introduces an alternative definition of the block structure language defined in Part 4, proves the equivalence of the two definitions, and briefly reviews recent work on proving the equivalence of interpreters. An Appendix considers the definition of VDL in VDL.

This paper may be read at a number of different levels. The reader who is interested in a quick introduction to the basic definition techniques of VDL need read only Section 1.3 and Part 3. The reader who is interested in a deeper understanding of block structure languages should emphasize Part 4, while the reader concerned with proofs of equivalence of interpreters will be interested in Part 5.

Key words and phrases: metalanguages, language definition, interpreter-interpreters, semantics, proofs about programs, interpreter equivalence

CR categories: 4.13, 5.24

1. BASIC DATA STRUCTURES

Part 1 introduces the notion of language definition, compares alternative strategies of language definition, reviews models of language definition that have been considered in the literature, and indicates the relation between the Vienna Definition Language (VDL) and other approaches to language definition. The basic data structures and data manipulation operators of VDL are then introduced, and a general data manipulation operator is defined that combines *assignment* to existing components of a tree structure with *allocation* of new components of a tree structure. The relation between VDL and LISP is considered, and an axiomatic definition of the basic data structure manipulation operators of VDL is introduced. The use of VDL data structures for

* Division of Applied Mathematics. This work, originally published as Technical Report 71-47 by the Center for Computer and Information Sciences, Brown University, was supported in part by NSF Grant GJ-28074.

CONTENTS

1. Basic Data Structures 5-20
   1.1 Compiler- and Interpreter-Oriented Language Definition
   1.2 Review of Language Definition Models
   1.3 Selection and Construction Operators
   1.4 The Generalized Assignment Operator
   1.5 Comparison with Lisp
   1.6 An Axiomatic Definition of VDL Data Structures
2. Sets of Data Structures 20-26
   2.1 Sets of Structured Objects
   2.2 Lists
   2.3 Comparison of the BNF and VDL Syntactic Metalanguages
3. Evaluation by Data Structure Transformation 26-32
   3.1 Instruction Execution
   3.2 The Semantics of Expression Evaluation
   3.3 The Evaluation of '$x_1 + x_2 * x_3$'
4. Definition of a Simple Block Structure Language 32-47
   4.1 The Syntax of EPL
   4.2 The Evaluation Strategy of EPL
   4.3 Components of the State
   4.4 The Semantics of EPL
5. Proofs of Interpreter Equivalence 47-58
   5.1 The Local Environment Model
   5.2 The Twin Machine Model
   5.3 The Structure of Twin Machine Proofs
   5.4 Equivalence of Direct Accessing and Chaining Models
   5.5 Proof Techniques for Interpreter Equivalence
Appendix. The Definition of VDL in VDL 58-62
   A.1 The Representation of Control Trees as Composite Objects
   A.2 The Instruction Execution Cycle of VDL
Bibliography 62-63

the syntactic and semantic definitions of programming languages is considered in subsequent sections.

1.1 Compiler- and Interpreter-Oriented Language Definition

In defining a programming language it is convenient to distinguish between *syntactic definition*, which is concerned with the specification of the set of all valid programs of the programming language, and *semantic definition*, which is concerned with specification of the "meaning" of valid programs of the programming language.

The syntax of a programming language may be defined by a *grammar* which specifies the *alphabet* of primitive symbols and *construction rules* for building up program structures from their components. For example, *context-free grammars* [H3] may be characterized by quadruples $G = (N, T, P, S)$, where T is the alphabet of primitive symbols, N is a set of names of syntactic categories used in specifying the way in which complete programs are constructed from their components, P is a set of construction rules (productions), and $S \in N$ is the name of the class of structures that constitute complete programs. A grammar defines a set of rules for *generating* the set of all valid symbol strings of a programming language. However, in many instances, generative grammars may be converted into *recognition algorithms*. Thus, a context-free grammar $G = (N, T, P, S)$ may be mechanically converted into one of a number of alternative recognition algorithms that identify substrings of a symbol string corresponding to instances of T; recognize substructures that are instances of syntactic categories N; and, finally, identify the complete symbol string to be an instance of S.

A syntactic definition defines the class of symbols that constitute valid programs of the programming language, but does not in itself associate *meaning* with the symbols that occur in programs. We shall be concerned with the *operational definition* [W6] of symbols and expressions occurring in program strings in terms of the transformations to which they give rise during execution [W4]. There are two principal ways of defin-

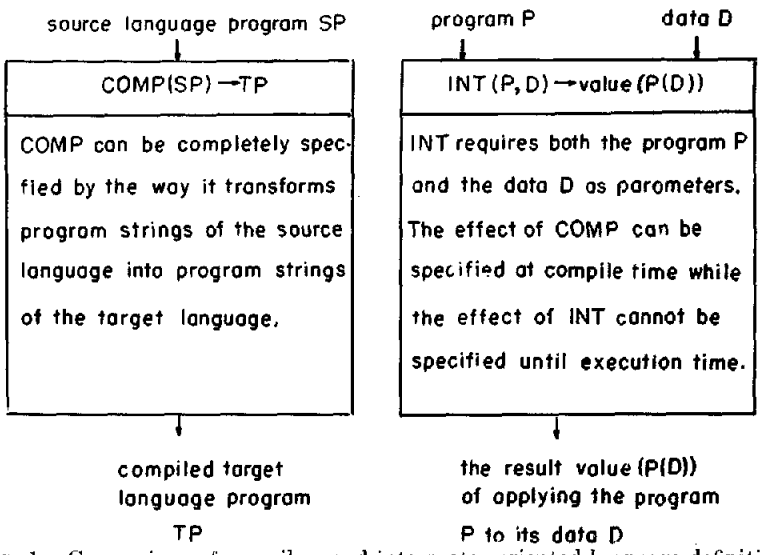


FIG. 1. Comparison of compiler- and interpreter-oriented language definition.

ing the operational meaning of programs of a programming language:

1) by a compiler, COMP, which defines how source programs, SP, may be translated into target programs, TP = COMP(SP), of a target language with known semantics; and
2) by an interpreter, INT, which defines for each program P and its data D an algorithm for computing the value, "value$(P(D))$ = INT(P, D)," which results from executing the program P for the data D.

The relation between compiler-oriented language definitions, COMP, and interpreter-oriented language definitions, INT, is illustrated in Figure 1.

Compilers and interpreters were initially specified in an ad hoc manner by system programs written in assembly language or in a higher-level language. During the 1960s a number of metalanguages for defining compilers were developed. The Vienna Definition Language is a metalanguage for defining interpreters rather than compilers. Before reviewing the development of metalanguages for compilers and interpreters (in Section 1.2), certain conceptual differences between compiler-oriented and interpreter-oriented definitions will be briefly considered.

Both compiler- and interpreter-oriented language definitions are generally *syntax-based* in the sense that the semantics of complete programs are defined by associating semantics with syntactically specified program components. However, there are characteristic differences between compiler-oriented and interpreter-oriented semantics. These differences are illustrated below for the case in which the syntax of the programming language is specified by a context-free grammar.

A *compiler-oriented* language definition associates with each production of the form $X \rightarrow Y_1\ Y_2 \cdots Y_k$ a *compile-time* semantic action $f_X(Y_1, Y_2, \cdots, Y_k)$ that may include the generation of target language code and the updating of compile-time state variables used in controlling the compilation process. At the same time, the symbol string $Y_1\ Y_2 \cdots Y_k$ is replaced by the symbol X. If the recognition algorithm involves backtracking, then a stack is required to keep track of the sequence of partially reduced program states to allow for the possibility of return to a previous state.

An *interpreter-oriented* language definition associates with each production $X \rightarrow Y_1\ Y_2 \cdots Y_k$ a state transformation from the current state I_j to a new state I_{j+1}. States frequently have the form (P, ξ_j), where P is an invariant program component and ξ_j represents the portion of the state that may vary during execution and is referred to as

the *state vector* [M2] or the *record of execution* [J1]. The state vector ξ_j generally contains an instruction pointer component *ip* which is updated to point to the next instruction. In definitions of block structure languages, ξ_j generally contains a stack of partially executed function activations with static links to indicate the static nesting of corresponding function definitions in the program component P [W2].

In a compiler-oriented language definition, the recognition that a substring of a program is generated by a given production gives rise to a *translation step* that may generate a target-language string corresponding to the recognized source-language substring. In an interpreter-oriented language definition the recognition that a substring is an instance of a specific syntactic structure gives rise to the *execution* of a sequence of instructions that generally depends on the current state vector ξ_j. Moreover, a compiler requires a given source-language string to be scanned at most once for each pass of the compiler, while an interpreter may require a given source-language string to be executed an arbitrarily large number of times.

A compiler-oriented language definition avoids the necessity of specifying execution-time semantics by assuming that the execution-time semantics of the target language is given as a primitive in terms of which source languages may be defined. This allows many semantic attributes of compiler-oriented definitions to be characterized by compile-time algorithms that are guaranteed to terminate. However, definition of a programming language by a compiler consists, from a mathematical point of view, of solving a problem A by reducing it to the solution of a previously solved problem B. Such an approach has the following drawbacks:

1) The reduction of a problem A to a problem B assumes that the problem B has been solved. In the case of programming language definitions it is assumed that the target-language interpreter is adequately defined. Thus, a compiler-oriented definition merely postpones the difficult problem of defining the programming language interpreter without eliminating it.
2) If a problem A is solved by reducing it to a problem B, then we generally obtain very little insight into the intrinsic nature of the problem A, only into the nature of the relation between A and B. Thus, the definition of a programming language by its compiler provides insight only into the relation between the source and target languages, and not into the intrinsic execution-time semantics of the source language.

We shall briefly review the history of compiler-oriented language definitions and then consider the history of interpreter-oriented language definitions.

1.2 Review of Language Definition Models

The ALGOL report [N1, L2] constitutes the first attempt to rigorously define a complete programming language. The syntax of ALGOL 60 is defined in terms of Backus-Naur Form (BNF), which corresponds effectively to the notation of context-free languages [H3]. The semantics of ALGOL 60 is defined in [N1] by an informal and somewhat ad hoc verbal specification, because, at the time the report was written, there existed no clear idea of the primitives in terms of which semantics should be defined. The ALGOL report gave rise to a family of compiler-oriented language definitions that defined the syntax of a programming language in a syntactic notation, as did the report, but introduced a number of different styles of rigorous semantic specification for defining the meanings of syntactically specified constructs. Thus, Irons [I1], Brooker and Morris [B4], Wirth and Weber [W7], and Feldman [F1] introduced different mechanisms for associating compiler-oriented "meaning" with syntactically specified constructs. Lewis and Stearns [L3] investigated certain theoretical properties of such syntactic models. Knuth [K2] considered a further mechanism for associating meaning with syntactically specified structures. However, all these models were "compiler-oriented" in the sense that the meaning of a syntactically specified symbol string of a source language S was assumed to be a symbol string in a target language T that depended only on compile-time characteristics of the source-language string.

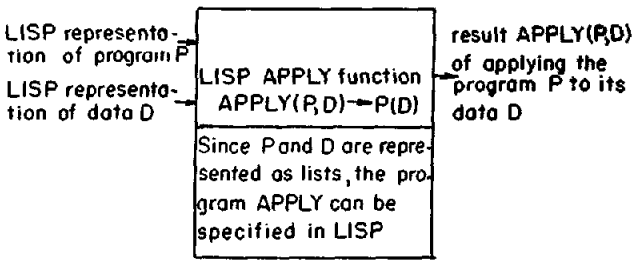


FIG. 2. The LISP APPLY function.

At about the time that ALGOL 60 was being defined, McCarthy developed the programming language LISP [M5] and provided a rigorous, interpreter-oriented definition of LISP that has served as a model for the interpreter-oriented definitions of other programming languages. LISP was defined in terms of a LISP program APPLY (P, D), which, given a list P representing a LISP program and a list D representing a data list to which P is applicable, computes the value APPLY(P, D) that results from applying the LISP program P to its data D. The operation of the LISP APPLY function is illustrated in Figure 2.

This method of defining LISP in terms of itself corresponds theoretically to the definition of a universal Turing machine U, which, given a tape containing the description of an arbitrary Turing machine T together with a description of the initial tape IT on which T is to operate, computes the value $U(T, IT)$ that results from applying the machine T to its data IT. At a practical level the LISP APPLY function demonstrates that the set of all programs, L, of any programming language can be treated as a *data structure* of a specific program $P_{L'}$ in a metalanguage L'. In the case of the LISP APPLY function, the metalanguage L' is the same as the language L. The function "APPLY" is an example of an interpreter for a language L that is written in the language L itself. The technique of language definition illustrated by the LISP APPLY function was an important influence in determining the approach to language definition of VDL.

McCarthy went on to develop an interpreter-oriented theory of computation [M2, M3]. In [M2] he clearly states that "the meaning of a program is defined by its effect on the state vector." McCarthy's model of semantics assumes that execution-time states of a program have the form $I_j = (P, \xi_j)$, where P is an invariant program component and ξ_j a state vector. This model was applied by McCarthy to the description of a subset of ALGOL [M4] and to a proof of correctness of a compiler for a simple class of arithmetic expressions [M6].

One further contribution of McCarthy was his introduction of the notion of *abstract syntax* [M2]. Context-free grammars assume that the set of objects whose syntax is being defined is a set of *strings*, and, therefore, sensitive to the *textual order* in which components appear in a string. However, compilers and interpreters are concerned not with strings written on a sheet of paper but with structures stored in a digital computer. Components of structures in a computer are identified not by a linear textual ordering but by pointers (selectors) that associate a name (address) with each of the components of a structure.

The context-free production $X \rightarrow Y_1\ Y_2 \cdots Y_n$ may be thought of as a one-level subtree whose edges have an implicit order. The corresponding production of an abstract syntax has the form $X \rightarrow (\langle s_1 : Y_1 \rangle, \langle s_2 : Y_2 \rangle, \cdots, \langle s_n : Y_n \rangle)$, where $s_1, s_2, \cdots, s_n$ are selectors. This production is represented by a one-level subtree with labeled, unordered nodes as illustrated in Figure 3.

In defining programming languages we often run into expressions that differ in syntax but have the same semantics. For example, the infix arithmetic expression $a + b$ would be represented in prefix form as $+ab$ and in postfix form as $ab+$. These three forms of syntax would give rise to different productions in an associated context-free

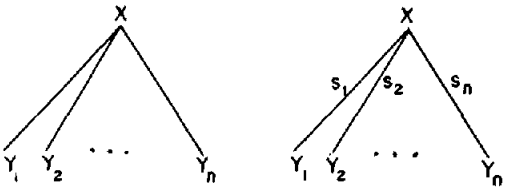


FIG. 3. Comparison of VDL and BNF syntaxes.

grammar. However, the three different linear representations can be represented by a single construction of the abstract syntax having the form (⟨s_1:a⟩, ⟨s_2:b⟩, ⟨op:+⟩), where s_1, s_2 are selector names of structures that represent arguments of the addition operator and op is the selector name of the structure that represents the operator. The abstract syntax represents the essential syntactic relation between components of a syntactic structure in a *normal form* that may easily be mapped onto corresponding storage structures within a computer. Therefore, abstract syntax is more appropriate than context-free syntax or BNF syntax for specifying the structure of programs and execution-time states in the definition of interpreters.

The Vienna Definition Language was inspired by McCarthy's interpreter-oriented definition of LISP, his subsequent definition of programming language interpreters in terms of state vector transformations [M2, M4], and McCarthy-style definitions of interpreters such as the SECD machine [L1]. There is, however, one big conceptual advance in the Vienna approach to language definition that is not present in earlier interpreter-oriented definitions: that is the notion that both the *syntax of programs* and the *syntax of complete states* are described in terms of the *same syntactic notation.*

Although McCarthy's states were, in principle, as general as those of the Vienna group, he had no explicit notation for the representation of states with a high degree of internal structure, but worked with state vectors whose components consisted of simple values. The Vienna group, in using the same syntactic notation to describe program structures and execution-time data structures, replaced McCarthy's state vector by a tree structure. Moreover, the Vienna group introduced some powerful tools for selection of tree components, construction of new trees from their components, and assignment of new values to vertices of trees. The facilities in the Vienna Definition Language for the construction and manipulation of trees are considered below.

1.3 Selection and Construction Operators

In the Vienna Definition Language, states that may occur during the execution of programs of the programming language being defined are described in terms of an underlying class of data structures. State transformations corresponding to the execution of instructions of the defined programming language are described in terms of transformation operators applicable to data structures that represent states. The remainder of Part 1 is devoted to describing the underlying class of data structures of VDL. The use of this class of data structures for programming language definition is then considered in subsequent parts.

There are two classes of data objects in VDL:

1) *elementary objects* (atomic objects), which have *no components* but have data structure transformation attributes when they occur as operators or operands during program execution; and
2) *composite objects*, which may be built up from elementary objects by *construction operators.* Composite objects have *components* that may be selected by *unique selectors.* The components may be either elementary objects or composite objects.

Figure 4 is a representation of a composite object whose three components a, +, and b may be selected by the selectors s_1, s_2, s_3. The association of a selector s with an object

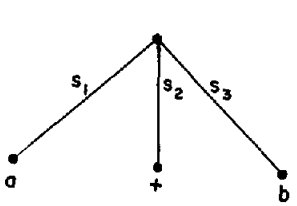


FIG. 4. The composite object μ_0(⟨s_1:a⟩, ⟨s_2:+⟩, ⟨s_3:b⟩).

ob will be represented by the notation ⟨*s*:*ob*⟩ and will be referred to as a *selector-object pair*. The construction of the composite object of Figure 4 from its three components may be accomplished by the application of the *construction operator* μ_0 to the three selector-object pairs ⟨s_1:*a*⟩, ⟨s_2:+⟩, ⟨s_3:*b*⟩:

$$\mu_0(\langle s_1{:}a\rangle, \langle s_2{:}+\rangle, \langle s_3{:}b\rangle).$$

The construction operator μ_0 takes as its arguments a variable number of selector-object pairs of the form ⟨s_i:t_i⟩, where s_i is a unique selector and t_i is an elementary object or a tree-structured composite object. The general form of μ_0 is illustrated in Figure 5.

Figure 6 illustrates a composite object '*a*+*b*∗*c*' with two levels of tree structure.* The composite object of Figure 6 may be constructed from the elementary objects comprising it by nested application of the construction operator μ_0 :

$$\mu_0(\langle s_1{:}a\rangle, \langle s_2{:}+\rangle, \langle s_3{:}\mu_0(\langle s_1{:}b\rangle, \langle s_2{:}*\rangle, \langle s_3{:}c\rangle)\rangle).$$

Selectors may be used as *operators* that, when applied to a structured object, select one of its components.

EXAMPLE: If t is the composite object of Figure 6, then $s_1(t)$ selects the elementary object '*a*', while $s_3(t)$ selects the composite object '*b*∗*c*'.

When the object to which a selector s is applied has no edge labeled "s" emanating from its root vertex, then $s(t)$ is defined as the *null object* Ω. Thus, if t is the composite object of Figure 6, then $s_4(t) = \Omega$.

The object $s_3(t)$ = '*b*∗*c*', which results from the application of the selector s_3 to the composite object t of Figure 6, is itself a composite object whose components may be selected by selectors. For example, the application of the selector s_1 to the object $s_3(t)$ yields the elementary object $s_1(s_3(t))$ = '*b*'. We shall write $s_1(s_3(t))$ as $s_1 \cdot s_3(t)$ and refer to $s_1 \cdot s_3$ as a *composite selector*.

Composite selectors are operators of the same kind as simple selectors in the sense

* Single quotes around a symbol string will be used to denote the composite object associated with that symbol string. Thus '*a*+*b*∗*c*' denotes the not necessarily unique composite object associated with the symbol string *a*+*b*∗*c*.

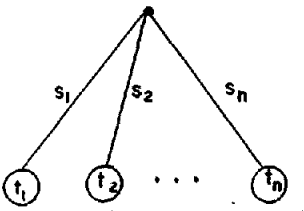


FIG. 5. The composite object $\mu_0(\langle s_1{:}t_1\rangle, \langle s_2{:}t_2\rangle, \cdots, \langle s_n{:}t_n\rangle)$.

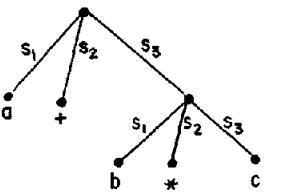


FIG. 6. The composite object $\mu_0(\langle s_1{:}a\rangle, \langle s_2{:}+\rangle, \langle s_3{:}\mu_0(\langle s_1{:}b\rangle, \langle s_2{:}*\rangle, \langle s_3{:}c\rangle)\rangle)$.

that they have the same domain and range. The sequence of individual selectors of a composite selector such as $s_n \cdot s_{n-1} \cdots s_2 \cdot s_1$ is applied to objects in a *right-to-left order* since

$$s_n \cdot s_{n-1} \cdots s_2 \cdot s_1(t) = s_n(s_{n-1}(\cdots(s_2(s_1(t)))\cdots)).$$

Composite selectors will be denoted by the Greek letter χ.

The selector I will be used to denote the identity selector. The identity selector I has the property that $I(t) = t$ for any elementary or composite object t.

The selectors emanating from a given vertex of a composite object are *required* to be different. This uniqueness rule allows the component associated with any tree vertex to be uniquely specified by the composite selector χ, consisting of the *sequence* of selectors required to reach that vertex from the root vertex.

A composite object may be characterized by the set of all pairs of the form ⟨χ:*eo*⟩, where χ is a selector path from the root vertex to a terminal vertex and *eo* is the elementary object at that terminal vertex. For example, the object of Figure 6 can be characterized by the following set of pairs:

$$\{\langle s_1{:}a\rangle, \langle s_2{:}+\rangle, \langle s_1 \cdot s_3{:}b\rangle, \langle s_2 \cdot s_3{:}*\rangle, \langle s_3 \cdot s_3{:}c\rangle\}.$$

This set of pairs is referred to as the *characteristic set* of the object. The characteristic set of a composite object provides a linear representation of its tree structure from

which the tree structure may easily be reconstructed. Transformations of composite objects may be specified by indicating the effect of a given transformation on the characteristic set of the object.

When the object t to which a composite selector $\chi = s_n \cdots s_2 \cdot s_1$ is applied does not have a sequence of edges labeled "s_1, s_2, $\cdots$, s_n" emanating from its root vertex, then $\chi(t)$ is defined as the null object Ω. If $s(t)$ is non-null we say that the object t has an s-component, while if $s(t)$ is null we say that the object t does not have an s-component. Similarly, we say that t has a χ-component if $\chi(t)$ is non-null, and that t does not have a χ-component if $\chi(t)$ is null.

The class of elementary and composite objects introduced above will be referred to as *Vienna objects*. The composite selectors of the characteristic set of a Vienna object determine the *structure* of the object, while the elementary objects at terminal vertices of a Vienna object may be thought of as determining the *content* associated with the structure. It is convenient to distinguish between *structure manipulation operators*, which transform objects in a way that depends only on their structure and not on the content of terminal vertices, and *content manipulation operators*, whose transformational effect may depend on the values of terminal vertices. In the remainder of Part 1 we shall be concerned only with structure manipulation operators.

The class of Vienna objects has been deliberately restricted to tree-structures because it has been found in practice that tree structures are adequate for the representation of programs and structured states. The class of Vienna objects is deliberately more restricted than two other classes of structures studied in the literature:

1) Directed ordered acyclic graphs (DOAGs), which permit paths through a structure to join together but do not permit any loops. In DOAGs having a distinguished root vertex from which all other vertices may be reached, a vertex may generally be reachable by a finite number of different paths, each of which may be characterized by a composite selector. A number of computational models in the literature use DOAGs as the basic data structure, including the models of Dennis [D1, Chapter 9] and Rosenberg [R1].
2) Directed graphs with loops, which permit infinite paths within a data structure.

FIG. 7. A tree structure, a DOAG, and a graph with loops.

Figure 7 gives examples of a tree structure, a DOAG, and a graph with loops. Tree structures are adequate for the representation of symbol strings with nested components. They can be used in representing DOAGs or arbitrarily directed graphs by allowing elementary objects that are composite selectors to occur at terminal vertices of a tree. However, it may well be that tree structures will eventually be replaced by a more general data structure as the underlying data structure of language definition systems.

One big advantage of using trees as the basic data structure is that the class of trees is closed under the operation of substitution of subtrees for subtrees. Moreover, the uniqueness of the accessing of components and the finiteness of the accessing path are preserved under the substitution of subtrees for subtrees. A generalized assignment operator for tree structures that makes use of these properties of tree substitution is introduced in the following section.

1.4 The Generalized Assignment Operator

We have defined a class of data structures, introduced a construction operator for the construction of instances of data structures from their components, and introduced the inverse operation of selection. The selection and construction operators are related by the following identity:

$$s_i(\mu_0(\langle s_1{:}t_1\rangle, \cdots, \langle s_i{:}t_i\rangle, \cdots, \langle s_n{:}t_n\rangle)) = t_i, \text{ for } 1 \leqslant i \leqslant n.$$

In order to be able to update and manipulate data structures, the construction operator μ_0 will be generalized to allow *assignment* of

values to components of data structures and *allocation* of new components in an existing data structure. The assignment operator, denoted by μ, may be defined as follows:

$$\mu(t; \langle\chi : t'\rangle).$$

Assign the value t' to the χ-component of t if t has a χ-component; add a new χ-component t' to t if t does not have a χ-component; delete the χ-component of t if $t' = \Omega$.

EXAMPLE: Let $t = \mu_0(\langle s_1 : x_1\rangle, \langle s_2 : x_2\rangle, \langle s_3 : x_3\rangle)$. The transformations $\mu(t; \langle s_2 : y\rangle)$, $\mu(t; \langle s_4 : y\rangle)$, and $\mu(t; \langle s_2 : \Omega\rangle)$ are illustrated by (b), (c), and (d), respectively, of Figure 8. The operation $\mu(t; \langle s_2 : y\rangle)$ *assigns* a new value to a component of t; the operation $\mu(t; \langle s_4 : y\rangle)$ *allocates* and initializes a new component of t; and the operation $\mu(t; \langle s_2 : \Omega\rangle)$ *deletes* a component of t.

The above example illustrates the assignment operator μ when its second argument is a simple selector. Operations involving assignment to a composite selector χ are illustrated by the following example.

EXAMPLE: Let $t = \mu_0(\langle s_1 : x_1\rangle, \langle s_2 : \mu_0(\langle s_1 : x_2\rangle, \langle s_2 : x_3\rangle)\rangle)$. The effect of the operations $\mu(t; \langle s_2 : y\rangle)$, $\mu(t; \langle s_1 \cdot s_2 : y\rangle)$, $\mu(t; \langle s_3 \cdot s_2 : y\rangle)$, and $\mu(t; \langle s_4 \cdot s_1 : y\rangle)$ are illustrated in (b), (c), (d), and (e), respectively, of Figure 9 (on next page).

Figure 9(b) illustrates the replacement of a component that is a subtree by a component that is an elementary object; 9(c) illustrates the replacement of an atomic component by a different atomic component; 9(d) illustrates the grafting of a new component at the vertex $s_2(t)$; and 9(e) illustrates replacement of an atomic component by one that is a composite object.

When $t' = \Omega$, the component selected by χ is deleted. The effect of $t' = \Omega$ is illustrated in Figure 10 by the structures that would obtain from $y = \Omega$ in Figure 9. Thus, in Figure 10(b) the selector-object pair $\langle s_2 : \Omega\rangle$ is deleted; in 10(c) the element $\langle s_1 \cdot s_2 : \Omega\rangle$ of the characteristic set is deleted; 10(d) illustrates that the effect of grafting in a new component with value Ω is to leave the tree unaltered; and 10(e) shows that adding a new component with value Ω onto the end of an existing subtree may delete a sequence (chain) of selectors.

The effect of the assignment operator $\mu(t; \langle\chi : t'\rangle)$ on the object t may be rigorously defined by specifying its effect on the char-

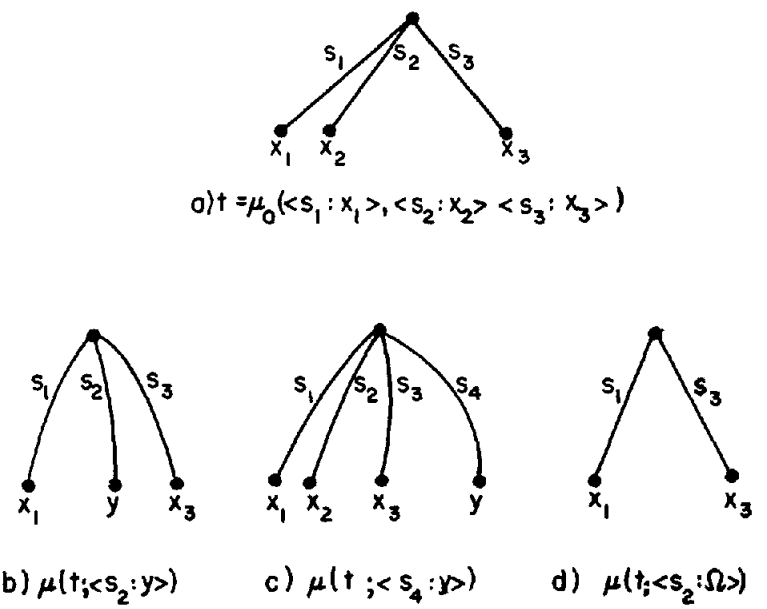


FIG. 8. The transformations $\mu(t; \langle s_2 : y\rangle)$, $\mu(t; \langle s_4 : y\rangle)$, $\mu(t; \langle s_2 : \Omega\rangle)$.

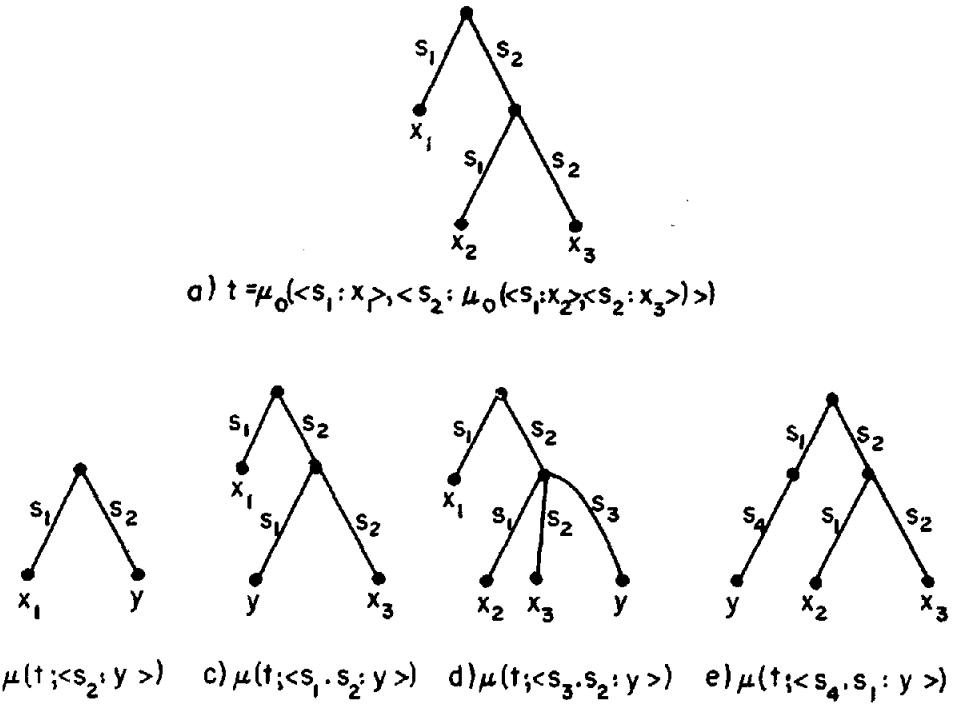


FIG. 9. Examples of applications of the μ-operator.

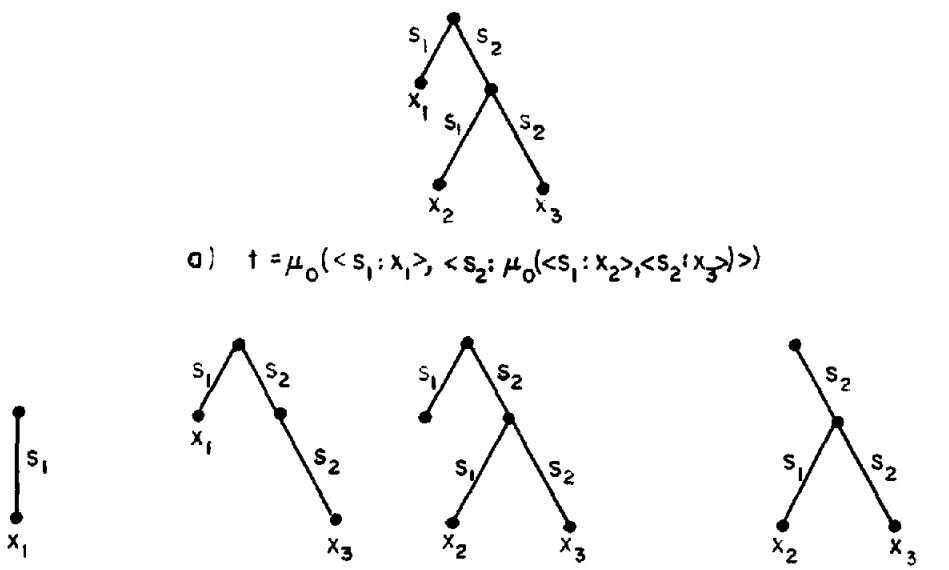


FIG. 10. The effect in Figure 9 when $y = \Omega$.

acteristic set $ch(t)$ of t.* The characteristic set $ch(t)$ consists of the set of all pairs of the form $\langle \chi_i : eo_i \rangle$, where $\chi_i = s_n \cdots s_2 \cdot s_1$ represents the path $s_1, s_2, \cdots, s_n$ from the root vertex of t to the terminal vertex of t containing the elementary object eo_i. The composite selector χ to which a value is assigned by the operation $\mu(t; \langle \chi : t' \rangle)$ may have one of the following relations to composite selectors χ_i of the characteristic set of t:

1) there is a $\chi_i = s_n \cdots s_k \cdots s_2 \cdot s_1$ in the characteristic set of t such that $\chi = s_k \cdots s_2 \cdot s_1$, where $k \leqslant n$; i.e, $\langle \chi_i : eo_i \rangle \in ch(t)$ such that $\chi_i = \chi' \cdot \chi$, where χ' may be any composite selector including the identity selector; or
2) the selector χ is not a rightmost subsequence of any of the composite selectors χ_i of the characteristic set of t.

Intuitively, condition (1) means that χ denotes an existing component of the composite object t, while condition (2) means that χ does not correspond to the composite selector of any of the components of t. This intuitive notion may be formalized as follows:

DEFINITION: A composite selector χ is said to be *dependent* on a composite selector χ_i if χ is a rightmost subsequence of χ_i; i.e., if $\chi_i = \chi' \cdot \chi$, where χ' is a possibly null sequence of selectors. Note: the identity selector is dependent on every composite selector as $\chi \cdot I = \chi$ for all χ.

DEFINITION: Two selectors χ_i and χ_j are said to be *dependent* if χ_i is dependent on χ_j, *or* χ_j is dependent on χ_i; they are said to be *independent* if they are not dependent.

DEFINITION: A set $S = \{\chi_1, \chi_2, \cdots\}$ of composite selectors is said to be *pairwise independent* if and only if for all distinct $\chi_i, \chi_j \in S$, χ_i and χ_j are independent.

THEOREM:** For all characteristic sets $ch(t) = \{\langle \chi_i : eo_i \rangle \mid \chi_i(t) = eo_i \wedge eo_i \text{ is an elementary object}\}$, the set of all first components χ_i is pairwise independent.

PROOF: If $\chi_i(t)$ selects an elementary object, and χ_j is dependent on χ_i and distinct from χ_i, then $\chi_j(t)$ selects a composite object and cannot, therefore, be one of the selectors of the characteristic set.

THEOREM: A selector χ selects a component of a composite object t if and only if it is dependent on a selector χ_i that occurs in the characteristic set $\{\langle \chi_i : eo_i \rangle\}$ of t.

PROOF: The proof follows from the definitions of "characteristic set," "dependence," and "component."

The μ operator may be defined as follows using the notion of dependence:

DEFINITION: The effect of $\mu(t; \langle \chi : t' \rangle)$ may be defined by the following transformation of the characteristic set of t: 1) delete from $ch(t)$ all components $\langle \chi_i : eo_i \rangle$ for which χ_i is dependent on χ, and let the resulting characteristic set be $ch(t^-)$; 2) if $ch(t') = \{\langle \chi_1 : eo_1 \rangle, \cdots, \langle \chi_n : eo_n \rangle\}$, then form the object t'' whose characteristic set is the union of $ch(t^-)$ and $\{\langle \chi_1 \cdot \chi : eo_1 \rangle, \cdots, \langle \chi_n \cdot \chi : eo_n \rangle\}$. It is assumed that $ch(\Omega)$ is the empty set $\{\ \}$, and $ch(eo)$ is the singleton set $\{\langle I : eo \rangle\}$.

The characteristic set $ch(t)$ defines a composite object in terms of the set of all paths from a root vertex to the terminal vertices of the tree structure. This definition may be thought of as a "vertical" characterization of tree structure, which partitions the tree into interrelated vertical paths from the root vertex to terminal vertices. The above definition of the μ operator illustrates that intuitively simple tree concepts, such as replacement of a subtree by some other subtree, become quite complex when expressed in terms of their effect on the set of all (vertical) paths from the root vertex to terminal vertices.

Trees may also be characterized "horizontally" by partitioning vertices into "levels" determined by the number of edges that separate them from their root, and specifying a recursive relation between one

* The reader who is satisfied with an intuitive understanding of the μ operator and is concerned primarily with the use of VDL for language definition may skip directly to Part 2.

** The theorems in this section are not particularly deep, but are included in order to provide a better understanding of the notions of dependence and independence.

level and the next. A horizontal characterization of tree structure leads to the following recursive definition of the class of Vienna objects:

DEFINITION: An object (Vienna object) is either an elementary object *eo* or has the form $\mu_0(\langle s_1:t_1\rangle, \langle s_2:t_2\rangle, \cdots, \langle s_n:t_n\rangle)$, where $s_1, s_2, \cdots, s_n$ are selectors and $t_1, t_2, \cdots, t_n$ are objects.

DEFINITION: An object of the form $\mu_0(\langle s_1:t_1\rangle, \langle s_2:t_2\rangle, \cdots, \langle s_n:t_n\rangle)$ is said to have immediate components $s_1, s_2, \cdots, s_n$, and no other immediate components.

The assignment operator $\mu(t; \langle\chi:t'\rangle)$ may be defined in terms of an *immediate assignment operator* $\mu_1(t; \langle s:t'\rangle)$ whose second parameter s is a *simple* selector:

$$\mu_1(t; \langle s:t'\rangle).$$

Assign the value t' to the s-component of t if t has an s-component; add a new s-component t' to t if t does not have an s-component; delete the s-component of t if $t' = \Omega$.

Let $\chi_n = s_n \cdot s_{n-1} \cdots s_1$ be a composite selector of length n, and let μ_n denote the assignment operator for composite selectors of length n. The assignment $\mu_n(t; \langle\chi_n:t'\rangle)$ can be expressed recursively in terms of the immediate assignment operator μ_1 and the assignment operator μ_{n-1}:

$$\mu_n(t:\langle s_n \cdots s_2 \cdot s_1:t'\rangle) = \mu_1(t; \langle s_1:\mu_{n-1}(s_1(t); \langle s_n \cdots s_2:t'\rangle)\rangle).$$

Assignment of the t' component of t corresponds to immediate assignment to the s_1-component of t of the object obtained by starting with the s_1-component of t, and assigning t' to its $s_n \cdots s_2$-component.

Using the above method of expressing assignment to composite selectors of length n in terms of assignment to composite selectors of length $n - 1$, the assignment operator μ for selectors of arbitrary length may be defined in terms of the assignment operator μ_1 by the following conditional expression:

$$\mu(t;\langle\chi:t'\rangle) =$$

if $\chi = I$ **then** t'

else if $\chi = s$ **then** $\mu_1(t;\langle s:t'\rangle)$

else if $\chi = s_n \cdots s_2 \cdot s_1$ and $n > 1$ **then**

$$\mu_1(t;\langle s_1:\mu(s_1(t);\langle s_n \cdots s_2:t'\rangle)\rangle)$$

This "horizontal" definition of the assignment operator μ in terms of the immediate assignment operator μ_1 differs in style from the definition in terms of characteristic sets. Whether this definition is any simpler is a matter of opinion. However, a comparison of the two styles of definition provides insight into different styles of defining operators over tree structures. The horizontal recursive characterization is closer in style to mathematical notations for expressing functions [K1, M2], and will be used in Section 1.6 for defining an axiom system for assignment and selection operations.

The μ operator can be generalized even further to allow the assignment of a sequence of objects $t_1, t_2, \cdots, t_n$ at vertices of t designated by composite selectors $\chi_1, \chi_2, \cdots, \chi_n$. This generalized operator has the form $\mu(t; \langle\chi_1:t_1\rangle, \langle\chi_2:t_2\rangle, \cdots, \langle\chi_n:t_n\rangle)$, and may be defined as follows:

$$\mu(t; \langle\chi_1:t_1\rangle, \langle\chi_2:t_2\rangle, \cdots, \langle\chi_n:t_n\rangle) = \mu(\mu(t; \langle\chi_1:t_1\rangle); \langle\chi_2:t_2\rangle, \cdots, \langle\chi_n:t_n\rangle).$$

That is, the μ operator for a sequence of assignments $\langle\chi_1:t_1\rangle, \cdots, \langle\chi_n:t_n\rangle$ is defined in terms of the previously defined single assignment $\mu(t; \langle\chi_1:t_1\rangle)$, followed by assignment of the sequence $\langle\chi_2:t_2\rangle, \cdots, \langle\chi_n:t_n\rangle$ to the result.

In general, multiple assignments may be dependent on the order in which they are executed, as illustrated by the following example:

$$\mu(t; \langle s_1:t_1\rangle, \langle s_1:t_2\rangle) \neq \mu(t; \langle s_1:t_2\rangle, \langle s_1:t_1\rangle), \text{ if } t_1 \neq t_2.$$

When assignment is restricted to immediate components of an object, then the order of two assignments $\langle s_i:t_i\rangle$ and $\langle s_j:t_j\rangle$ may be interchanged if and only if $s_i \neq s_j$. The conditions under which the order of two assignments $\langle\chi_i:t_i\rangle$ and $\langle\chi_j:t_j\rangle$ may be interchanged are given by the following theorem.

THEOREM: The order of assignment of $\langle\chi_1:t_1\rangle$ and $\langle\chi_2:t_2\rangle$ to an object t is interchangeable for all t_1, t_2 if and only if χ_1

and χ_2 are independent; i.e., $\mu(t; \langle\chi_1{:}t_1\rangle, \langle\chi_2{:}t_2\rangle) = \mu(t; \langle\chi_2{:}t_2\rangle, \langle\chi_1{:}t_1\rangle)$ for all t_1, t_2 if and only if χ_1 and χ_2 are independent.

PROOF: (a) *If-condition:* If χ_1 and χ_2 are independent, then both orders of assignment will create a new object t' which has a χ_1-component t_1, a χ_2-component t_2, and has the same components as t for all components independent of χ_1 and χ_2. (b) *Only-if condition:* Let χ_1 and χ_2 be dependent, say χ_2 is dependent on χ_1. Then $\mu(\mu(t; \langle\chi_1{:}t_1\rangle), \langle\chi_2{:}t_2\rangle)$ first assigns t_1 to the χ_1-component of t, and then destroys the value t_1 in assigning t_2 to the χ_2-component of t. The assignment $\mu(\mu(t; \langle\chi_2{:}t_2\rangle,) \langle\chi_1{:}t_1\rangle)$ first assigns t_2 to the χ_2-component of t, and then assigns t_1 to the χ_1-component of t. Thus, the former method results in an object t', which does not generally have a χ_1-component with value t_1, while the latter method results in an object t'', which does have a χ_1-component with value t_1. The objects t' and t'' created by the two orders of assignment differ in at least one component and are therefore different.

The construction operator μ_0, which has been defined only for simple selectors, may be generalized to pairwise-independent composite selectors.

DEFINITION: If $\chi_1, \chi_2, \cdots, \chi_n$ are pairwise independent composite selectors, then $\mu_0(\langle\chi_1{:}t_1\rangle, \langle\chi_2{:}t_2\rangle, \cdots, \langle\chi_n{:}t_n\rangle)$ constructs from the objects $t_1, t_2, \cdots, t_n$ a composite object that has a χ_i-component t_i for $i = 1, 2, \cdots, n$, and no other components.

Note that μ_0 could, in principle, be defined for selectors that are not pairwise independent by specifying the order in which components $\langle\chi_i{:}t_i\rangle$ are introduced into the composite object. However, there are advantages in restricting construction operators so that the end result of any primitive construction operation is independent of the order in which components are used in the construction process.

The construction operator μ_0, defined above, may be shown to be a special case of the operator μ.

THEOREM: If $\chi_1, \chi_2, \cdots, \chi_n$ are pairwise independent, then

$$\mu_0(\langle\chi_1{:}t_1\rangle, \langle\chi_2{:}t_2\rangle, \cdots, \langle\chi_n{:}t_n\rangle) = \mu(\Omega; \langle\chi_1{:}t_1\rangle, \langle\chi_2{:}t_2\rangle, \cdots, \langle\chi_n{:}t_n\rangle).$$

PROOF: The left-hand side and the right-hand side give rise to objects with the same characteristic set.

The theorems introduced above are relatively trivial and could, no doubt, be deduced as special cases of a general theory of tree structure or linear dependence. They are, however, significant because of the *interpretation* we associate with the tree structured objects of our universe of discourse.

Composite objects may be regarded as tree structured *computer memories.* Composite selectors may be regarded as *addresses* that select *locations* of the tree-structured memory, where locations may have components that are, themselves, locations. The object $\chi(t)$ selected by the composite selector χ may be regarded as the *value* contained in the location χ of the tree-structured memory t.

In the following section, the relation between VDL and LISP is briefly considered, and the analogy between Vienna objects and computer memories is further explored.

1.5 Comparison with LISP

The class of data structures and the structure manipulation operators of VDL are patterned after those of LISP.

LISP has two classes of data objects:

1) *atoms*, which have no components but which may have attributes specified by an attribute list; and
2) *lists*, whose elements have two components, atoms or lists, which may be selected by the selectors CAR and CDR.

The atoms of LISP correspond to the elementary objects of VDL, and the lists of LISP correspond to the composite objects of VDL. LISP data structures are richer than those of VDL, as VDL is restricted to tree-structures while the lists of LISP may include merging selector chains (common sublists) and even circular selector chains (circular lists). However, the class of tree structures that can be constructed in VDL is richer than the class of tree structures that can be

constructed in LISP, since composite objects may have an arbitrarily finite number of components that may be selected by arbitrary selector names, while list elements have two components that are selected by the two fixed selector names CAR and CDR.

The LISP operator CONS(X, Y) creates a new list whose CAR component is X and whose CDR component is Y; it is similar to the operator $\mu_0(\langle \text{CAR}:t_1\rangle, \langle \text{CDR}:t_2\rangle)$, which creates a new composite object whose CAR-component is t_1 and whose CDR-component is t_2. Selector names need not be explicitly mentioned as arguments of the LISP CONS operator since they are implied by the context. However, selector names must be explicitly mentioned in the case of the VDL construction operator.

The VDL operator μ is a more powerful primitive tree transformation operator than the LISP operator CONS. The operation $\mu(t; \langle\chi:t'\rangle)$ replaces an arbitrary component χ of the tree t by t' if t has a χ-component; allocates a new component χ if t does not have a χ-component; and thus performs in a single operation a task that would require a recursively specified sequence of operations using LISP-like primitives.

The operator μ causes VDL data structures to have some of the attributes of storage structures of a digital computer. The composite selectors, which select non-null components of a composite object, play the role of addresses, while the vertices selected by composite selectors have the attributes of variable-capacity storage cells that may contain elementary or composite objects as their values.

Let S^* be the set of all possible composite selectors, including the identity selector I. Since the operator $\mu(t; \langle\chi:t'\rangle)$ not only assigns values to existing components t, but also allocates new components $\chi(t)$ with value t' for any $\chi \in S^*$, we may think of any composite object t as an infinite, tree-structured storage structure that, at any given time, contains only a finite number of components with non-null values, but allows assignment of a non-null value to any one of the infinite number of components $\chi \in S^*$. Our convention that null components can be deleted allows us to represent infinite storage structures with a finite number of non-null components by finite trees.

The modeling of objects with a finite number of components by infinite objects with a finite number of non-null components (finite support) is a common mathematical device for the representation of elements of sets the numbers of whose components are unbounded. This device is used in representing Turing machines by infinite tapes with a finite number of non-blank components [S1]. It is also used in specifying infinite tables arising in property grammars [L4] by means of tables with a finite number of non-empty entries. In each of these cases, infinite objects may alternatively be represented by finite objects that can grow in size. For example, Shepherdson and Sturgis [S1] have defined Turing machines as objects with finite tapes and operators for splicing extra tape squares onto the end of the tape whenever an operation that moves the reading head beyond the end of the tape is executed.

It is generally more convenient to regard composite objects as finite objects subject to operators that may add new components than to regard them as infinite objects. However, there are probably contexts in which the notion of a composite object as an infinite object with a finite number of non-null components is more convenient.

1.6 An Axiomatic Definition of VDL Data Structures

In order to gain further perspective regarding the structures and operators of VDL, a formal system, due to Standish [S2], that captures certain basic attributes of VDL operators in a representation-independent way is described.

DEFINITION: A *Vienna definition system* V is defined as a sextuple $V = (EO, CO, \Omega, S, \sigma, \mu)$, where EO is a set of objects called elementary objects, CO is a set of objects disjoint from EO called composite objects, $\Omega \in EO$ is a distinguished element of EO called the null element, $S \subset EO$ is a set of elements called selectors (simple selec-

tors), and σ and μ are operators called the selection and assignment operators.

The set $OB = EO \cup CO$ will be called the set of objects (Vienna objects). The selection operator σ is a function from $S \times OB \rightarrow OB$; i.e., $\sigma(s, t)$ has a first argument s that is a selector and a second argument t that is an object, and produces a value that is the effect of applying the selector s to the object t. The expression $\sigma(s, t)$ may be abbreviated by the notation $s(t)$, and the value of $s(t)$ is referred to as the s-component of t. If $s(t) = \Omega$, then t is said to have no s-component.

The (immediate) assignment operator μ is a function from $OB \times S \times OB \rightarrow CO \cup \Omega$; i.e., $\mu(t, s, t')$ has arbitrary objects as its first and third arguments and a selector as its second argument, and produces as its value either a composite object or the object Ω.

We have now introduced the components of a Vienna definition system and the domains and ranges of the operators σ and μ. A complete specification of the effects of the operators σ and μ on the objects EO, S, and CO may be defined by the following axioms:

AXIOM 1: Elementary objects have no components; i.e., every elementary object $eo \in EO$ has the property that for all $s \in S$, $s(eo) = \Omega$.

AXIOM 2: There is a decision procedure for determining for arbitrary elementary objects eo_1 and eo_2 whether $eo_1 = eo_2$.

AXIOM 3: Composite objects are equal only if they have equal components;* i.e., if co_1, $co_2 \in CO$, then $co_1 = co_2$ only if $s(co_1) = s(co_2)$ for all s.

AXIOM 4 (assignment axiom): Assignment to a specific component s of an object t updates its s-component and leaves all other components unaltered; i.e., for any two selectors s and s' and any two objects t and t': if $s' = s$ then $\sigma(s', \mu(t, s, t')) = t'$, and if $s' \neq s$ then $\sigma(s', \mu(t, s, t')) = \sigma(s', t)$.

Axioms 1, 2, and 3 define the notion of equality of composite objects in terms of equality of their characteristic sets. The assignment axiom specifies the effect of the assignment operator μ in terms of the way in which it modifies the selection attributes (components) of the object to which a value is assigned. These axioms capture the relation between the assignment and selection operators more simply than previous verbal definitions. The simplicity of the axiom system suggests that the choice of this particular form of the assignment operator is an appropriate one.

It should be noted that the assignment axiom does not hold for composite selectors, since it does not follow from $\chi' = \chi$ that $\sigma(\chi', \mu(t, \chi, t')) = \sigma(\chi', t')$. However, assignment to composite selectors can be defined in terms of assignment to simple selectors by the following definition:

$$\mu(t, s_n \cdots s_2 \cdot s_1, t') = \mu(t, s_1, \mu(s_1(t), s_n \cdots s_2, t')).$$

It may be verified that, when $\mu(t, s_n \cdots s_2 \cdot s_1, t')$ is interpreted as $\mu(t; \langle s_n \cdots s_2 \cdot s_1 : t' \rangle)$, all the properties defined for the VDL assignment operator μ may be derived from the axioms.

The above representation-independent axiomatic definition of Vienna definition systems complements the representation-dependent verbal description given in previous sections. In order to obtain a proper understanding of a class of computational structures, it is necessary to consider both representation-independent axiomatic characterizations and specific models that are realizations of the axiom system.

Standish [S2] demonstrates that this axiom system has interpretations in domains other than those of the Vienna objects, but, at the same time, proves a representation theorem which asserts that any class of objects satisfying the axiom system may always be isomorphically represented by the Vienna objects.

The above definition of Vienna definition systems is interesting in that the definition

* The assertion "composite objects are equal *if* they have equal components" follows from the axioms for equality (substitution of equals for equals). Thus, the alternative axiom, "Composite objects are equal *if and only if* they have equal components" can be weakened by leaving out the "if" part.

may be naturally partitioned into syntactic and semantic parts. Thus, the first five components (EO, CO, Ω, S, σ), together with the first three axioms, define a purely syntactic system, say V_{SYN}, in the sense that V_{SYN} defines a class of syntactic objects for which equality of objects implies identity of structure. The operator μ, together with the assignment axiom, introduces semantic content into the system in the sense that two different expressions containing the operator μ may define the same composite object. The question of whether two arbitrary expressions define the same composite object is, however, decidable for finite objects, since we can, in each case, determine the composite object in a finite number of steps and then compare the composite objects for equality in a finite number of steps.

2. SETS OF DATA STRUCTURES

In Part 1 we introduced a class of data structures and certain operators for manipulating these data structures. Both an individual program of a programming language and a state (snapshot) of a given point of program execution can be expressed as data structures in this class. A programming language is characterized by a *set* of programs which may be represented by a *set* of data structures. In order to describe the set of all states that may occur during the execution of programs of a programming language, a notation is required for talking not only about individual data structures but also about sets of data structures. The notation developed below depends on the idea of adapting the mathematical notion of a predicate to the case in which the sets characterized by the predicate have a high degree of internal structure.

2.1 Sets of Structured Objects

A set S with a finite number of objects may be defined by listing its elements. Thus, $S = \{1, 2, 3\}$ denotes the set whose three elements are 1, 2, 3. A set with an infinite number of elements is generally defined in terms of an attribute P_S having the property that $P_S(x)$ is true if and only if x belongs to the set S.

EXAMPLE: The set of all prime numbers may be defined as follows:

$$S = \{x \mid \text{prime } (x)\}.$$

The attribute "prime" is assumed to be true for all integers x that are prime numbers, and false otherwise. It may either be regarded as a primitive notion or defined in terms of more primitive notions (e.g., by an algorithm for computing whether or or not x is prime).

Attributes such as "prime" above, which are true for arguments x belonging to a specified set and false otherwise, are referred to as *predicates*. Predicates may be thought of as *names* of sets. We shall use the notation $\hat{P}$ to denote the set named by the predicate P. Thus,

$$\widehat{\text{prime}} = \{x \mid \text{prime } (x)\}.$$

Complex sets that are built up in a uniform manner from simpler sets have associated predicates that may be defined in terms of the predicates of simpler sets. For example, if even(x) is a predicate satisfied by the set of all even numbers, and P is the predicate satisfied by the set of all nonnegative integers that are either even or prime, the specification "P = even $\vee$ prime" denotes the predicate that is satisfied by all elements x satisfying the predicate "even" or the predicate "prime." This specification is, in turn, an instance of the following more general specification:

$$P = P_1 \vee P_2 \vee \cdots \vee P_n .$$

The predicate P is defined to be the predicate that is satisfied by all elements x satisfying at least one of the predicates P_1, P_2, $\cdots$, P_n.

The above notational conventions may be used to define the predicate "$P = P_1 \wedge \cdots \wedge P_n$" as the predicate that is satisfied by those elements x satisfying all the predicates P_1, P_2, $\cdots$, P_n; and the predicate $P = \neg P_1$ as the predicate that is satisfied only by elements x not satisfying the predicate P_1.

The sets of structures S that arise in the

specification of programming languages may generally be defined by specifying how instances of S may be built up from instances of component sets, say Y, Z, by construction operations such as concatenation or μ_0.

EXAMPLE: The context-free production $X \rightarrow YZ$ may be regarded as a specification of how instances of the set of structures X may be built up from instances of the set of structures Y and Z. The set X is defined to be the set of all structures that can be generated by concatenating an instance of the set Y with an instance of the set Z. Alternatively, we may interpret X as a predicate and define it to be the predicate that is satisfied by all elements x having the form yz, where y satisfies the predicate Y and z satisfies the predicate Z.

EXAMPLE:

$$\text{expr} \rightarrow \text{var} + \text{var}$$

The instance of a context-free production may be interpreted as a specification that the predicate "expr" be satisfied by strings of the form $x_1x_2x_3$, where x_1 satisfies the predicate "var," x_2 is the symbol $+$, and x_3 satisfies the predicate "var."

When the underlying set of structures is symbol strings, then the basic construction operator is concatenation, and complex predicates are defined in terms of simpler predicates by *extending* the concatenation operator from symbol strings to predicates. The Vienna Definition Language uses a different mechanism for the construction of complex structures out of simpler structures, and, therefore, requires a different mechanism for defining the predicates associated with sets of structured objects in terms of predicates of component structures. The method used to describe the set of objects formed by the construction operator μ_0 is illustrated by the following example:

$$\text{is-expr} = (\langle s_1{:}\text{is-var}\rangle, \langle s_2{:}\text{is-var}\rangle, \langle \text{s-op}{:}\text{is}-+\rangle).$$

An object satisfies the predicate "is-expr" if it has an s_1-component satisfying the predicate "is-var," an s_2-component satisfying the predicate "is-var," an s-op-component satisfying the predicate "is-t," and no other components.

More generally, the predicate X, which is satisfied by all objects that can be constructed from objects $x_1, x_2, \cdots, x_n$ satisfying the predicates $X_1, X_2, \cdots, X_n$ by the construction operator $\mu_0(\langle s_1{:}x_1\rangle, \langle s_2{:}x_2\rangle, \cdots, \langle s_n{:}x_n\rangle)$, may be defined as follows:

$$X = (\langle s_1{:}X_1\rangle, \langle s_2{:}X_2\rangle, \cdots, \langle s_n{:}X_n\rangle).$$

A structured object satisfies the predicate X if it has s_i-components satisfying the predicate X_i for $i = 1, 2, \cdots, n$, and no other components. Using these principles for constructing predicates, a predicate "is-expr," which is satisfied by a class of objects representing a simple class of arithmetic expressions, may be constructed as follows:

$$\text{is-expr} = \text{is-const} \vee \text{is-var} \vee \text{is-binary}$$

$$\text{is-binary} = (\langle s_1{:}\text{is-expr}\rangle, \langle s_2{:}\text{is-expr}\rangle, \langle \text{s-op}{:}\text{is-op}\rangle)$$

$$\text{is-op} = \{+,*\}$$

According to the above definition, the predicate "is-expr" is an elementary object satisfying the primitive predicate "is-const" or "is-var," or it is a composite object satisfying the predicate "is-binary." The predicate "is-binary" is satisfied by three-component objects whose s_1- and s_2-components satisfy the predicate "is-expr" and whose s-op-component satisfies the predicate "is-op." The definition is thus recursive, and is satisfied by an infinite number of composite objects (trees) with an arbitrary number of levels of substructure. For example, the arithmetic expression $a + b * c$ would be modeled in the above syntax by the composite object of Figure 11. This representation of the string $a + b * c$ makes explicit certain semantic attributes that were implicit in the string representation. For example, the precedence of multiplication over addition is explicitly indicated in the tree representation by the specification that the two operands of the addition operator are, respectively, a and $b*c$.

The form of predicate construction $X = (\langle s_1{:}X_1\rangle, \langle s_2{:}X_2\rangle, \cdots, \langle s_n{:}X_n\rangle)$ parallels the

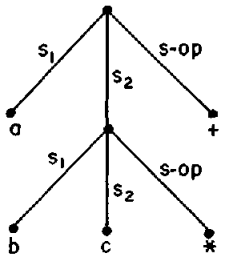


FIG. 11. The structured object $\mu_0(\langle s_1{:}a\rangle, \langle s_2{:}\mu_0(\langle s_1{:}b\rangle, \langle s_2{:}c\rangle, \langle \text{s-op}{:}*\rangle)\rangle, \langle \text{s-op}{:}+\rangle)$.

construction operator μ_0. In order to emphasize this correspondence between construction processes for individual objects and construction processes for predicates, the notation pr_0 will be used generically to indicate this form of predicate construction:

$$pr_0 = (\langle s_1{:}P_1\rangle, \langle s_2{:}P_2\rangle, \cdots, \langle s_n{:}P_n\rangle).$$

The predicate pr_0 is satisfied by all composite objects having an s_1-component satisfying the predicate P_1, an s_2-component satisfying the predicate P_2, an s_n-component satisfying the predicate P_n, and no other components.

VDL has a form of composite predicate construction pr_μ that parallels the construction of composite objects from simpler objects using the μ operator:

$$pr_\mu = (pe_0;\langle\chi_1{:}pe_1\rangle, \langle\chi_2{:}pe_2\rangle, \cdots, \langle\chi_n{:}pe_n\rangle).$$

An object t satisfies the predicate pr_μ if and only if it can be constructed from an object x satisfying the predicate pe_0 by successive assignment of objects satisfying the predicates $pe_1, pe_2, \cdots, pe_n$ to the $\chi_1, \chi_2, \cdots, \chi_n$ components of x. Thus, if x satisfies pe_0, and $y_1, y_2, \cdots, y_n$ satisfy $pe_1, pe_2, \cdots, pe_n$, then $\mu(x; \langle\chi_1{:}y_1\rangle, \langle\chi_2{:}y_2\rangle, \cdots, \langle\chi_n{:}y_n\rangle)$ will satisfy the predicate pr_μ.

A form of predicate construction pr^* will now be introduced which allows the definition of sets of structures having a variable number of immediate components of the form $\langle s{:}x\rangle$, where s must belong to a subset of selectors satisfying a predicate pe_1 and x satisfies a predicate pe_0:

$$pr^* = (\{\langle s{:}pe_0\rangle \parallel pe_1(s)\}).$$

An object t satisfies the predicate pr^* if and only if all its immediate components are of the form $\langle s{:}x\rangle$, where s is a unique selector satisfying the predicate pe_1 and each x is an object satisfying the predicate pe_0; i.e., pr^* defines a finite set of selector-value pairs $\langle s_i{:}y_i\rangle$, where s_i satisfies the predicate pe_1 and y_i satisfies the predicate pe_0.

A common example of a class of objects having a variable number of components is the case of tables containing a variable number of elements. Consider tables whose entries (d_i, v_i) have first components d belonging to a domain whose elements satisfy the predicate "is-domain" and whose second components satisfy the predicate "is-value." A set of composite objects for representing such tables may be represented by the following predicate:

$$\text{is-table} = (\{\langle d{:}\text{is-value}\rangle \parallel \text{is-domain}(d)\}).$$

An object t satisfies the predicate "is-table" if and only if all its entries have the form $\langle d{:}v\rangle$, where d satisfies the predicate "is-domain" and v satisfies the predicate "is-value." The composite object of Figure 12 satisfies the predicate "is-table" and has n components of the form $\langle d_i{:}v_i\rangle$.

When tables are represented as in Figure 12, the operations of table lookup and table updating may be defined by simple operations. Thus, if t satisfies the predicate "is-table," then $d(t)$ will yield the value v if $\langle d{:}v\rangle$ is an entry in the table, and will yield the value Ω if the table t does not contain an entry with first component d. The operation $\mu(t; \langle d_i{:}v_i\rangle)$ updates the table t with the entry $\langle d_i{:}v_i\rangle$, replacing an existing entry for d_i if there is one, and adding a new entry otherwise.

Other common structures arising in programming language implementations may be conveniently specified in VDL. For example, stacks may be represented by entries with a fixed number of components where one of the components has the same syntactic

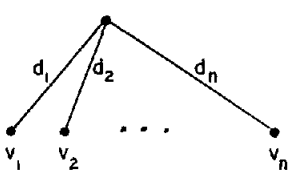


FIG. 12. Representation of a table with entries (d_i, v_i).

structure as the component as a whole:

$$\text{is-stack} = (\langle\text{s-stack:is-stack}\rangle, \langle\text{s-element:is-element}\rangle) \vee \text{is-}\Omega.$$

The predicate "is-stack" is satisfied by objects whose s-stack-component satisfies the same predicate as the complete object, and whose s-element-component satisfies the predicate "is-element" (which defines the structure of individual stack elements).

If x is an object satisfying the predicate "is-stack" and y is an object satisfying the predicate "is-element," then pushdown and popup operations may be defined as follows:

$$\text{pushdown}(x, y) = \mu_0(\langle\text{s-stack:}x\rangle, \langle\text{s-element:}y\rangle)$$

and

$$\text{popup}(x) = \text{s-stack}(x).$$

2.2 Lists

Program structures and states occurring during execution may contain ordered sequences of objects whose elements are retrieved by specifying an index or a successor. For example, a statement or instruction sequence is an example of an ordered sequence of objects in which the order is important. VDL has a mechanism for indicating that a collection of elements is ordered; such sequences are referred to as *lists*.

Ordering of a collection of elements may be accomplished by associating each element with a selector chosen from the distinguished, countable sequence of selectors elem(1), elem(2), $\cdots$. A VDL object is said to be a list of length n if and only if it has n non-null immediate components whose selectors are elem(i), $i = 1, 2, \cdots, n$:

$$(\langle\text{elem}(1):y_1\rangle, \langle\text{elem}(2):y_2\rangle, \cdots, \langle\text{elem}(n):y_n\rangle).$$

Thus, VDL lists are either empty or are composite objects of the form shown in Figure 13.

It should be emphasized that VDL lists are different in structure from lists of list structure languages such as LISP. In fact, the accessing structure of VDL lists is just like that of one-dimensional arrays, with the function "elem" playing the role of an address mapping function. However, we shall see below that VDL permits operators on lists that simulate the list construction operators of LISP.

A list of length 0 is represented by the symbol $\langle\rangle$, and satisfies the predicate "is-$\langle\rangle$." Let is-list(x) be the predicate that is true for precisely elements x (which are lists), and false otherwise. A number of operations applicable to lists are defined below.

The length n of a list may be defined as follows:

$$\text{length}(L) = \text{is-list}(L) \rightarrow \textbf{if } \text{is-}\langle\rangle(L) \textbf{ then } 0 \textbf{ else } (\text{the unique } i \text{ such that } \text{elem}(i)(L) \neq \Omega \text{ and } \text{elem}(i+1)(L) = \Omega).$$

That is, length(L) is defined only for objects L satisfying the predicate "is-list." For such objects, length(L) is defined to be 0 when $L = \langle\rangle$, and is otherwise defined as the maximum i for which elem(i)$L \neq \Omega$. This informal notation for function specification is used below to specify a number of other functions on lists.

An operation "head(x)," which selects the first element of a list, and a second operation "tail(x)," which removes the first element from a list, may be defined as follows:

$$\text{head}(x = \text{is-list}(x) \wedge \neg \text{is-}\langle\rangle(x) \rightarrow \text{elem}(1)(x)$$

and

$$\text{tail}(x) = \text{is-list}(x) \wedge \neg \text{is-}\langle\rangle(x) \rightarrow \mu(\Omega; \{\langle\text{elem}(i):\text{elem}(i+1)(x)\rangle, \mid 1 \leqslant i \leqslant \text{length}(x) - 1\}).$$

The operation "tail(x)" deletes the first element, elem(1), of the list and moves the nth element to elem($n - 1$). It may be compared to the implementation of popping up of a stack by moving each stack element into the next higher stack position.

Deletion of the first element of a list is a

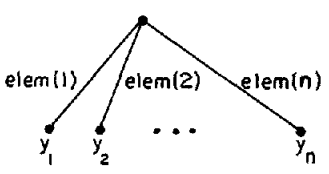


FIG. 13. A VDL "list structure."

relatively complex operation when an n-element list is represented by an ordered set of n immediate components, as in VDL. LISP, by representing an n-element list hierarchically in a stack, allows the first element of the list to be deleted by simply popping up the stack.

The concatenation x_1x_2 of two lists x_1 and x_2 may be obtained by adjoining the elements of x_2, with appropriately renamed selector names, onto the end of x_1:

$$x_1x_2 = \text{is-list}(x_1) \wedge \text{is-list}(x_2) \rightarrow \mu(x_1 ; \{\langle \text{elem}(\text{length}(x_1) + i) : \text{elem}(i)(x_2)\rangle \mid 1 \leqslant i \leqslant \text{length}(x_2)\}).$$

The list $\mu_0\{\langle \text{elem}(1):x_1\rangle, \langle \text{elem}(2):x_2\rangle, \cdots, \langle \text{elem}(n):x_n\rangle\}$ is sometimes written as $\langle x_1, x_2, \cdots, x_n\rangle$ to correspond to the conventional notation for lists.

If $\text{pr}(x)$ is a predicate satisfied by a class of objects x, then the notation pr-list(y) will be used to denote the derived predicate which is satisfied by any list of objects x satisfying the predicate $\text{pr}(x)$. For example, if is-st(x) is a predicate satisfied by the set x of statements of some programming language, then the predicate is-st-list(y) will be satisfied by any object y that is a list of statements of that programming language.

The predicates is-st-list, is-param-list, and is-id-list, which are used in Part 4 to express the syntax of statement lists, parameter lists, and identifier lists, may be defined as follows:

$$\text{is-st-list} = (\langle \text{elem}(1):\text{is-st}\rangle, \langle \text{elem}(2):\text{is-st}\rangle, \cdots, \langle \text{elem}(n):\text{is-st}\rangle) \vee \text{is-}\langle\rangle$$

$$\text{is-param-list} = (\langle \text{elem}(1):\text{is-param}\rangle, \langle \text{elem}(2):n\text{-param}\rangle, \cdots, \langle \text{elem}(n):\text{is-param}\rangle) \vee \text{is-}\langle\rangle$$

$$\text{is-id-list} = (\langle \text{elem}(1):\text{is-id}\rangle, \langle \text{elem}(2):\text{is-id}\rangle, \cdots, \langle \text{elem}(n):\text{is-id}\rangle) \vee \text{is-}\langle\rangle$$

2.3 Comparison of the BNF and VDL Syntactic Metalanguages

There is a similarity between syntactic specifications in VDL and BNF. For example, the previously considered VDL arithmetic expression syntax may be specified by the following BNF syntax:

EXPR::=CONST|VAR|BINARY

BINARY::=EXPR OP EXPR

OP::=+|−

The relation between BNF and VDL is brought out by considering the relation between the second production above and the following VDL production:

is-binary = ($\langle s_1$:is-expr$\rangle$, $\langle s_2$:is-expr$\rangle$, $\langle$s-op:is-op$\rangle$).

The BNF production "BINARY::=EXPR OP EXPR" specifies that a string of the form "BINARY" consists of a string of the form "EXPR" *followed by* a string of the form "OP" *followed by* a string of the form "EXPR." The corresponding VDL production specifies that objects satisfying the predicate "is-binary" are composite objects with an s_1-component satisfying the predicate "is-expr," an s_2-component satisfying the predicate "is-expr," an s-op-component satisfying the predicate "is-op," and no other components.

The BNF metalanguage is designed for specifying sets of *strings*, and extends the operation of concatenation from individual strings to symbols denoting sets of strings to allow sets of strings with a common substructure to be uniformly specified in terms of their components. The VDL metalanguage does not directly specify program strings of a programming language. It specifies sets of *composite objects* whose components may be selected by selectors. Such composite objects are closer in structure to the way in which a program might be represented in computer memory after syntactic analysis, than to the way the program is represented when originally written.

The representation of programs by composite objects and of programming languages by sets of composite objects has a number of advantages over string representation.

1) Composite objects constitute a *representation-independent normal form* for classes of representation-dependent programs. For instance, the arithmetic expression "$a + b$" may also be represented by any one of the strings $+ab$, $+(a, b)$, $ab+$, or plus ab. These representations are different at the level of string representation, and each one would have to be represented by a different BNF syntax. But all

representations consist of three components, two of which are operands and the third an operator, and may, therefore, be represented by the same composite object in the Vienna Definition Language. Composite objects capture the essential structural relations among components of objects without any commitment to a specific sequential representation. The specification of syntax in terms of structural relations among components was first introduced by McCarthy [M2], and was referred to by him as an *abstract syntax.*

2) Composite objects may be easily mapped, into an internal computer representation since selectors may be represented by pointers. In turn, the modeling of syntactic objects in a form that is close to their internal representation in digital computers allows operators on syntactic structures to be defined so that they closely model transformations of actual computer representations.
3) An initial syntactic analysis and translation pass is required to transform program strings into composite objects of a VDL syntax. This analysis and translation pass is an important part of any compilation or interpretation process, but is concerned largely with lexical and syntactic matters that are conveniently handled prior to program execution. Composite objects of VDL exhibit the essential semantic structure of programs after lexical and syntactic features of a particular external representation have been filtered out. Starting with composite objects allows us to emphasize the semantics of program execution with a minimum of syntactic distractions.

Operator precedence is a good example of a syntactically important property of program representations that can be dealt with prior to program execution. Precedence relations are syntactically important because they allow the writing of programs to be simplified by the omission of parentheses. A BNF syntax specification must take explicit account of precedence relations in order to generate a correct unambiguous phrase structure for its subexpressions. For example, the previously given BNF syntax for expressions is ambiguous since the expression "3 + 4 * 5" may be generated either in terms of subexpression components 3 + 4 and 5, or in terms of subexpression components 3 and 4*5.

EXPR → BINARY → EXPR OP EXPR → BINARY * 5 →
EXPR OP EXPR * 5 → 3 + 4 * 5

EXPR → BINARY → EXPR OP EXPR → 3 + BINARY →
3 + EXPR OP EXPR → 3 + 4 * 5

An alternative BNF syntax that properly handles operator precedence is as follows:

EXPR::=TERM|EXPR+TERM

TERM::=FACTOR|TERM*FACTOR

FACTOR::=CONST|VAR

In terms of this BNF syntax, the expression 3 + 4 * 5 may be unambiguously generated as follows:

EXPR → EXPR + TERM → CONST + TERM * FACTOR →
3 + CONST * CONST → 3 + 4 * 5

There is no possibility of generating 3 + 4 * 5 by means of component subexpressions 3 + 4 and 5, since such a generating sequence would have to start with "EXPR → TERM → TERM * FACTOR," and the TERM can never be expanded into subexpressions containing the operator +.

Whereas BNF syntactic specifications must be carefully constructed to reflect operator precedence, the tree representations of VDL represent the operator/operand structure unambiguously. Any precedence relations required in interpreting the original program string must already have been used in constructing the composite-object representation. The precedence relations of the string representation are irrelevant in interpreting the composite object since they are explicitly represented by the hierarchical substructure.

The BNF metalanguage is designed for a different purpose from the VDL syntactic metalanguage. BNF was introduced as a *generative* mechanism to generate the sets of strings occurring in natural languages and programming languages. Its use for the purposes of recognizing whether a given string belongs to the language and for determining

the string structure was secondary. The VDL syntactic metalanguage was specifically designed to facilitate the structural descriptions of objects in terms of the components that are to be manipulated and transformed. The coupling of the VDL syntactic metalanguage with a semantic metalanguage specifying execution-time transformations of syntactic objects is further described below.

3. EVALUATION BY DATA STRUCTURE TRANSFORMATION

Section 3.1 introduces the notion of control trees and gives the basic instruction formats for the specification of instructions that define the semantics of programming languages. Section 3.2 defines the semantics of a simple class of arithmetic expressions and illustrates, in general terms, the significance of the instructions that constitute the definition of this programming language. Section 3.3 illustrates the step-by-step evaluation of a specific arithmetic expression by giving a trace of the evaluation process. The notations developed in this part are used in Part 4 to specify the syntax and semantics of a non-trivial block structure language.

3.1 Instruction Execution

The semantics of a programming language is defined in VDL in terms of the sequences of information structure transformations to which its programs give rise during execution. Every computation starts from an initial representation, say ξ_0, and may be characterized by a sequence of information configurations $\xi_0 \rightarrow \xi_1 \rightarrow \cdots \rightarrow \xi_n$, where for $j = 0, 1, \cdots, n-1$, ξ_{j+1} is obtained from ξ_j by the execution of a primitive instruction. The information configurations ξ_j are referred to as *instantaneous descriptions, snapshots,* or *states.* A state is represented in VDL as a composite object whose immediate components $s(\xi)$ may be selected by selectors s and whose lower-level components may be selected by sequences of selectors. Both instructions and data may be regarded as part of the state ξ. We shall first consider the structure of the state component containing the instructions and then go on to consider the structure of the complete state.

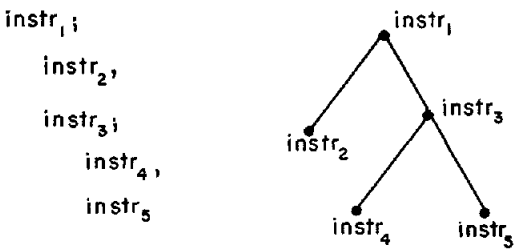


FIG. 14. The structure of control trees.

The instructions of a VDL program are, at any given point of execution, represented by a tree structure called a *control tree.* Figure 14(b) illustrates a control tree containing five instructions. Figure 14(a) illustrates how the control tree of 14(b) can be represented by indentation. Instructions of a control tree may also be represented in linear form by the use of braces to indicate subtrees. Thus, the linear representation "instr_1; {instr_2, instr_3; {instr_4, instr_5}}" unambiguously indicates subtrees by a semicolon followed by braces, and separates successors of a given vertex by commas. The literature of VDL uses both indentation and punctuation to indicate tree structure.

The instruction execution cycle of VDL specifies that, on completion of an instruction, *any terminal vertex* of a control tree may be executed as the next instruction. Thus, in Figure 14 any one of the three instructions instr_2, instr_4, and instr_5 may be executed next.

An instruction in VDL has the following format:

$$\text{instruction-name}(x_1, x_2, \cdots, x_n) =$$
$$p_1 \rightarrow a_1$$
$$p_2 \rightarrow a_2$$
$$\cdots$$
$$p_1 \rightarrow a_n$$

or

$$\text{instruction-name}(x_1, x_2, \cdots, x_n) = a_1$$

The symbols p_i, $i = 1, 2, \cdots, k$, are pred-

icate expressions that select alternative actions a_i. The symbols $x_1, x_2, \cdots, x_n$ are formal parameters that may occur in p_i or a_i, and that are replaced by values before the instruction is executed. The execution of an instance of the above instruction causes the current state to be transformed by the action a_i corresponding to the *first true predicate*, p_i. The transformations determined by actions a_i may be classified in two categories:

1) *macro instructions* (self-replacing instructions), which replace the vertex of the control tree currently being executed by a subtree of instructions and do not modify any other state components; and
2) *value-returning instructions* (assignment instructions), which *delete* the control tree vertex being executed, *pass* computed values to predecessor vertices of the control tree, and may, as a side effect, modify other components of the state.

Value-returning instructions have the following general format:

$$\text{PASS} \leftarrow e_0$$

$$s\text{-}sc_1 \leftarrow e_1$$

$$s\text{-}sc_2 \leftarrow e_2$$

$$\cdots$$

$$s\text{-}sc_n \leftarrow e_n$$

Evaluate the expressions $e_0, e_1, \cdots, e_n$. Pass the value of e_0 to predecessor vertices to be stored as the value returned by the instruction. Assign the value of e_i to the state component $s\text{-}sc_i$ for $i = 1, 2, \cdots, n$.

If in Figure 14, the instruction instr_4 is a value-returning instruction with the above format, its execution would cause: 1) the vertex instr_4 to be deleted from the control tree; 2) the value of the expression e_0 to be assigned as the value of instruction parameters x_i in certain predecessor vertices of the control tree; and 3) each of the state components $s\text{-}sc_i$, $i = 1, 2, \cdots, n$, to be updated to the value of the corresponding expression e_i. If, on the other hand, the instruction instr_4 were a macro instruction, its execution would cause the control tree vertex instr_4 to be replaced by the instruction subtree which constitutes the macro expansion.

Whereas conventional programs generally consist of a fixed set of instructions whose sequence of execution is dynamically determined by conditional branching instructions, VDL instructions corresponding to control tree vertices are generated by the execution of macro instructions during execution. Macro instructions may be viewed as *syntactic instructions* concerned with realizing a certain potential syntactic structure of programs and with generating "implicit instructions" required to realize higher-level operations, such as block and procedure entry. Value-returning instructions perform the *semantic transformations* which constitute the actual computation at the level of expression evaluation, assignment, and storage allocation.

In order to compare the structure of VDL computations with computations on conventional computers, the principal state components associated with program execution on conventional computers will be briefly characterized. The principal state components for conventional program execution are:

1) a program component P;
2) a processor component Π, which generally includes an instruction pointer, ip, to the next executable instruction of P; and
3) a record of execution R, which contains data and control structures generated during program execution and which may be organized into characteristic subcomponents such as stacks and heaps.

A set of states that are uniformly structured into the above subcomponents could be represented by the predicate, "is-conventional-state," as follows:

is-conventional-state =$_{\Delta}$ (⟨s-P:is-program⟩,

⟨s-Π:is-processor⟩, ⟨s-R:is-record⟩).

That is, the predicate "is-conventional-state" is satisfied by composite objects having three components selected by the selectors s-P, s-Π, and s-R, which respectively satisfy the predicates "is-program," "is-processor," and "is-record."

In defining the set of computations of a given programming language we need a further predicate "is-initial-state," which, for fixed-program interpreters, might be de-

fined as follows:

is-initial-state = (⟨s-P:is-program⟩,
⟨s-Π:is-initial-processor⟩,
⟨s-R:is-initial-record⟩).

The sets of structures satisfying the predicates "is-initial-processor" and "is-initial-record" would typically be subsets of the sets of structures satisfying the predicates "is-processor" and "is-record." For example, in the case of block structured languages like ALGOL 60, the predicate "is-record" is satisfied by the set of all stack structures that can occur during execution, while the predicate "is-initial-record" is satisfied only by the empty stack. The instruction pointer component of an initial processor typically points to the "first" instruction, while the ip-component of a structure satisfying the predicate "is-processor" may point to any instruction.

The structure of VDL states differs from that described above in that the processor has been complicated to allow some nondeterminacy in the selection of the next instruction. This allows accurate modeling of the semantics of programming languages like PL/I which do not always specify completely the order in which instructions are to be executed. The nondeterminacy associated with multitasking may be partially simulated by leaving unspecified the order in which next instructions of multiple tasks are to be executed. However, it should be emphasized that the VDL execution mechanism allows only one instruction at a time to be executed, and cannot, therefore, model the problems of asynchronous execution that arise in real multiprocessing.

In VDL, the notion of a static program component P whose next executable instruction is pointed to by an instruction pointer, is replaced by a component referred to as the control tree, which is a combination of the program and processor components. Nondeterminacy is introduced by adopting the strategy that any instruction at a terminal vertex of the control tree may be chosen as the next instruction to be executed. Thus, there is no analog of the instruction pointer in a VDL computation. The simple next-instruction selection rule of conventional computers is replaced in VDL by a nondeterministic selection rule among terminal vertices of the control tree.

The initial state of a VDL computation generally has a control tree component with a single vertex containing the macro instruction **interpret-program**(t). The execution of this macro instruction gives rise to *replacement* of this single-vertex control tree by a control tree that makes explicit the syntactic structure of the composite object t. A sequence of replacements of macro instructions having composite objects as parameters by control subtrees that make the program explicit eventually gives rise to a control tree whose terminal vertices consist of value-returning instructions, with elementary objects as parameters. Execution of these latter instructions gives rise to the transformations that correspond to conventional program execution. The remainder of Part 3 illustrates how a combination of macro instructions and value-returning instructions may be used to define a simple class of arithmetic expressions.

3.2 The Semantics of Expression Evaluation

The semantics of VDL will be illustrated by defining an interpreter for the class of objects specified by the predicate "is-expr," defined in Section 2.1. The semantics of the simple class of arithmetic expressions satisfying the predicate "is-expr" may be defined by the following instructions:

eval-expr(t) = **print**(a);
a:**value**(t)

The instruction **eval-expr**(t) is a macro instruction which causes itself to be replaced by the two-vertex control tree, **print**(a); a:**value**(t).

value(t) = is-binary(t) → **apply**(a, b, s-op(t))
a:value($s_1(t)$)
b:value($s_2(t)$)
is-var(t) → PASS ← t·s-env(ξ)
is-const(t) → PASS ← t

If the parameter t satisfies the predicate "is-binary," replace **value**(t) by a three-instruction subtree. If t satisfies the predi-

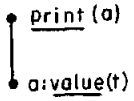
cate "is-var" or "is-const," perform the value-returning actions specified by the second and third alternatives. Note that the instruction **value**(t) behaves like a macro-instruction when t satisfies the predicate "is-binary," and behaves like a value-returning instruction when t satisfies the predicate "is-var" or "is-const."

apply (val_1, val_2, op) = op = '+' → PASS ← $val_1 + val_2$

op = '*' → PASS ← $val_1 * val_2$

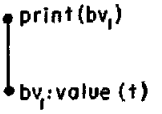
If "op" is the symbol "+," return the sum of the first two parameters as a value. If "op" is the symbol "*," return the product.

Initially, the control tree always consists of a single vertex **eval-expr**(t) so that execution of the first instruction always results in a control tree with the following structure:

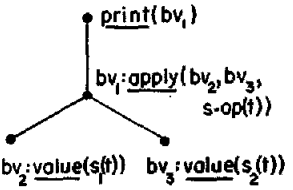


The instruction a:**value**(t) is an example of a "labeled instruction." However, the identifier preceding the colon should be thought of as a bound variable to which the value computed by the instruction is to be *assigned*, rather than as a label. It is useful to think of the colon as an assignment operator, and to think of the instruction a:**value**(t) as though it were written "a: = value(t)." Execution of an instruction a:instr causes the value computed by the instruction to be *copied* into all parameter positions of preceding instructions that contain the parameter a. In the present example, evaluation of the instruction a:**value**(t) will initially generate subtrees by execution of the macro **value**(t), but will eventually result in a value being returned to the parameter position a of the instruction **print**(a) and then being printed out.

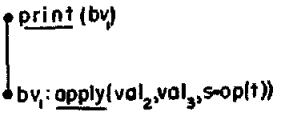
Identifiers in a given macro specification are *bound variables* in the sense that all instances of an identifier within a macro specification may be replaced by some other bound variable without changing the meaning of the macro. In order to avoid conflict of names between identifiers of different macros we may replace each identifier of an instance of expansion of a macro by a new *unique name* bv_i (bound variable i). Thus, the two-instruction control tree obtained on expanding eval-expr(t) can be represented as follows:



If t satisfies the predicate "is-binary," the four-instruction control tree obtained by evaluating the instruction **value**(t) may be represented as follows:



The operands $s_1(t)$ and $s_2(t)$ may, in turn, satisfy the predicates "is-binary," "is-const," or "is-var," and result in further macro expansion. However, evaluation of the subtrees generated by the two terminal vertices of the above control tree will eventually cause the values of the operands to be placed in the parameter position of the apply function. If these values are val_2, val_3, then the control tree at this point may be represented as follows:



The execution of the instruction at the terminal vertex of this control tree causes the application of the operator s-op(t) to the values val_2, val_3 and passing of the resulting value, say val_1, to the parameter position bv_1 of the root vertex. Execution of the instruction "print(val_1)" causes "printing" of the output on the output component s-output(ξ), and yields a null control tree which causes the computation to be terminated.

In order to complete the definition of semantics for expression evaluation, we shall specify the structure of states ξ occurring during a computation. The predicate "is-state," which is satisfied by all states that occur during expression evaluation, may be defined as follows:

is-state = (⟨s-c:is-c⟩, ⟨s-env:is-env⟩,

⟨s-output:is-output⟩).

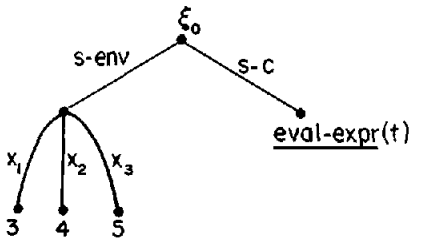


FIG. 15. Initial state for expression evaluation.

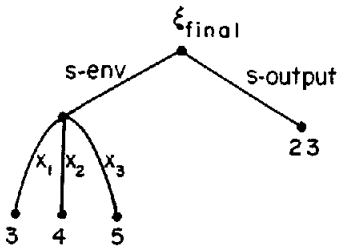


FIG. 16. Final state for an evaluated expression.

That is, states have a control component selected by the selector "s-c" satisfying the predicate "is-c," an environment component selected by the selector "s-env" satisfying the predicate "is-env," and an output component selected by the selector "s-output" satisfying the predicate "is-output."

The initial state has a null output component and a single-vertex control tree with the instruction **eval-expr**(t). It may be defined by the predicate "is-initial-state":

$$\text{is-initial-state} = (\langle \text{s-c}:\textbf{eval-expr}(t)\rangle, \langle \text{s-env}:\text{is-env}\rangle).$$

The environment consists of a set of selector-value pairs $\langle x_i:v_i\rangle$, where x_i is an identifier satisfying the predicate "is-var" and v_i is a value satisfying the predicate "is-const." The predicate "is-env" may be defined as follows:

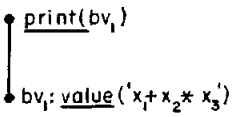
$$\text{is-env} = (\{\langle x:\text{is-const}\rangle \| \text{is-var}(x)\}).$$

We shall consider below the evaluation of the expression '$x_1 + x_2 * x_3$', where the x_1, x_2, and x_3 components of the environment have the values 3, 4, 5, respectively. In this case, the form of the initial state ξ_0 is given in Figure 15. The environment component remains fixed during expression evaluation. The control tree expands to reflect the structure of the expression t, and eventually causes a print instruction to print the value "23" in the output component. The form of the final control tree is given in Figure 16.

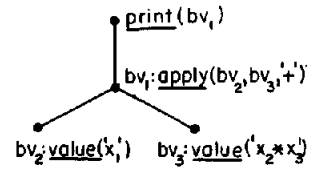
The sequence of intermediate configurations of the control tree during the above computation is considered in detail in the section below.

3.3 The Evaluation of '$x_1 + x_2 * x_3$'

In this section we wish to trace the sequence of states $\xi_0, \xi_1, \cdots$, generated during the evaluation of the expression '$x_1 + x_2 * x_3$'. Since the environment component remains fixed and the output component remains empty until the last step, it is sufficient to indicate the sequence of snapshots of the control tree associated with the successive states $\xi_0, \xi_1, \cdots$. Initially, the control tree consists of the single instruction **eval-expr**(t), where t consists of the structure $\mu_0(\langle s_1:x_1\rangle, \langle s_2:\mu_0(\langle s_1:x_2\rangle, \langle s_2:x_3\rangle, \langle \text{s-op}:x\rangle)\rangle, \langle \text{s-op}:x\rangle)$. Execution of this instruction generates the following control tree:

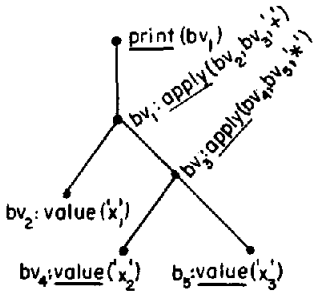


Since the composite object '$x_1 + x_2 * x_3$' satisfies the predicate "is-binary," the following control tree is generated by execution of the instruction value ('$x_1 + x_2 * x_3$'):

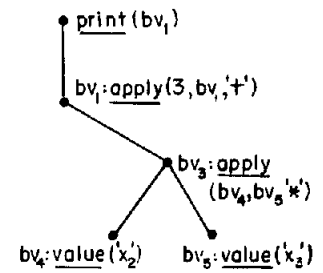


The VDL interpreter may now choose either of the two instructions at terminal vertices of the control tree to execute next. Assume that the instruction at the terminal vertex bv_3 is executed next. The argument of the "value" instruction is the composite object representing $x_2 * x_3$, and it satisfies the predicate "is-binary." Therefore, execu-

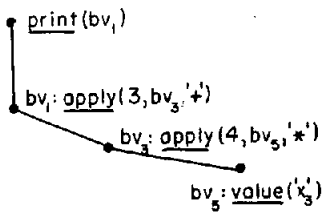
tion of this instruction yields the following control tree:



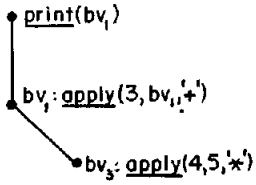
There are now three possible next instructions, corresponding to the vertices labeled bv_2, bv_4, and bv_5. Assume that the instruction at the vertex bv_2 is executed next. The parameter 'x_1' of the instruction **value**(t) satisfies the predicate "is-var," so that the value returning alternative "PASS ← t·s-env(ξ)" of the instruction **value**(t) is executed. Since x_1·s-env(ξ) = 3, the value "3" is returned to the parameter position bv_2 of the instruction labeled bv_1. When execution of this instruction has been completed, the control tree is as follows:



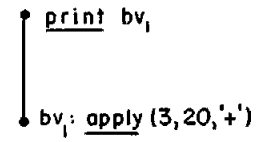
Assume that the instruction at the vertex bv_4 is executed next. The parameter of the instruction **value**(t) again satisfies the predicate "is-var." This time, the value returned to the parameter position bv_4 is "4." When execution of this instruction has been completed, the control tree is as follows:



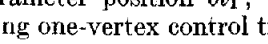
The only executable instruction at this point in the computation is the one at the vertex labeled bv_5. The parameter t of the instruction **value**(t) again satisfies the predicate "is-var," and this time x_3·s-env(ξ) = 5. When execution of this instruction has been completed the control tree is as follows:



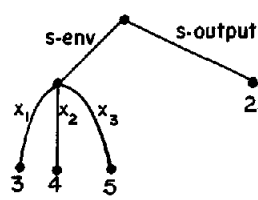
Execution of the instruction bv_3:**apply**(4,5,*) causes the value "20" to be passed back to the parameter position bv_3, and results in the following control tree:



Execution of the instruction at the vertex bv_1 causes the value "23" to be passed back to the parameter position bv_1, and yields the following one-vertex control tree:



Execution of the instruction **print**(23) causes the value "23" to be printed and results in an empty control tree, which, in turn, causes the computation to terminate. Printing may be interpreted as assignment of the printed value to output component s-output(ξ) of the state ξ. The final state thus has the following form:



States ξ of the above computation have three immediate state components: s-env(ξ), s-c(ξ), and s-output(ξ). The environment component s-env(ξ) is a "read-only component" that is never modified during the computation. The control component s-c(ξ) is a constantly changing component which, in

the case of arithmetic expressions, completely characterizes intermediate points of the computation. The output component s-output(ξ) is used for purposes of output just as the output medium of a conventional computer.

When the above technique is applied to languages with assignment and block structure, the environment component becomes updatable, and has a more complex structure reflecting the fact that a given identifier may denote different memory cells at different points of instruction execution. The syntactic and semantic specification techniques are further illustrated below by showing how an interpreter for a simple block structure language may be defined in VDL.

4. DEFINITION OF A SIMPLE BLOCK STRUCTURE LANGUAGE

4.1 The Syntax of EPL

The syntax of a language with nested block structure, called EPL (Elementary Programming Language), will now be described. An EPL program consists of a single block containing a *declaration part* and a *statement part*. The declarations in a block may include variables having *integer* or *logical* values and declarations of *procedures* and *functions*. Statements may be *assignment statements*, *conditional statements*, *procedure statements*, and *blocks*. EPL does *not* have arrays or labels, and restricts actual parameters of procedure and function calls to identifiers. All parameters are called by reference (as in FORTRAN and PL/I).

The principal productions of the abstract syntax of EPL are discussed below. The complete syntax of EPL is specified by the 17 productions given in Table I. The key numbers of the productions below refer to their position in the Table.

An EPL program consists of a single unlabeled block, as indicated by the following production:

is-program = is-block. (A1)

An EPL block, in turn, consists of a declaration part and a statement list, specified by

TABLE I. THE SYNTAX OF EPL

	Syntax of programs and blocks
(A1)	is-program = is-block
(A2)	is-block = (⟨s-decl-part:is-decl-part⟩, ⟨s-st-list:is-st-list⟩)
	Syntax of declarations
(A3)	is-decl-part = ({⟨id:is-attr⟩‖is-id(id)})
(A4)	is-attr = is-var-attr ∨ is-proc-attr ∨ is-funct-attr
(A5)	is-vår-attr = {INT, LOG}
(A6)	is-proc-attr = (⟨s-param-list:is-id-list⟩, ⟨s-st:is-st⟩)
(A7)	is-funct-attr = (⟨s-param-list:is-id-list⟩, ⟨s-st:is-st⟩, ⟨s-expr:is-expr⟩)
	Syntax of statements and expressions
(A8)	is-st = is-assign-st ∨ is-cond-st ∨ is-proc-call ∨ is-block
(A9)	is-assign-st = (⟨s-left-part:is-var⟩, ⟨s-right-part:is-expr⟩)
(A10)	is-expr = is-const ∨ is-var ∨ is-funct-des ∨ is-bin ∨ is-unary
(A11)	is-const = is-log ∨ is-int
(A12)	is-var = is-id
(A13)	is-funct-des = (⟨s-id:is-id⟩, ⟨s-arg-list:is-id-list⟩)
(A14)	is-bin = (⟨s-rd1:is-expr⟩, ⟨s-rd2:is-expr⟩, ⟨s-op:is-binary-rt⟩)
(A15)	is-unary = (⟨s-rd:is-expr⟩, ⟨s-op:is-unary-rt⟩)
(A16)	is-cond-st = (⟨s-expr:is-expr⟩, ⟨s-then-st:is-st⟩, ⟨s-else-st:is-st⟩)
(A17)	is-proc-call = (⟨s-id:is-id⟩, ⟨s-arg-list:is-id-list⟩)

the following syntax:

is-block = (⟨s-decl-part:is-decl-part⟩, ⟨s-st-list:is-st-list⟩). (A2)

Declarations are specified by an *unordered* set of named objects, since the order of declarations within a block head is not relevant to the semantics. Statements are specified as *lists* whose elements satisfy the predicate "is-st," since the order of statements in a given block does affect the semantics. The declaration part consists of a set of declared identifiers paired with attributes. Thus, the declaration **integer** x would be represented in the abstract syntax by the selector-object pair ⟨x:INT⟩. The declaration part may be specified by the following syntax:

is-decl-part = ({⟨id:is-attr⟩ ‖ is-id(id)}). (A3)

Instances of the attribute "is-attr" associated with a declared identifier depend on the *type* of the identifier. EPL permits identifiers to have the types INT (integer), LOG (logical), PROC (statement-type procedure), and FUNCT (function-type procedure). Accordingly, the predicate "is-attr" has different syntactic forms, depending on whether the type of the declared identifier is a data variable, a procedure declaration, or a function declaration. Figure 17 illustrates the abstract syntax of a block with the declarations INT x, LOG y (**logical** y), and three statements S1, S2, S3.

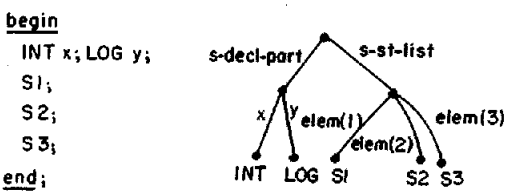


FIG. 17. Syntax of the declaration part and statement list.

The syntax of declaration attributes may be defined as follows:

is-attr = is-var-attr ∨ is-proc-attr ∨ is-funct-attr. (A4)

For data variables, the attribute associated with an identifier is simply the type tag:

is-vâr-attr = {INT, LOG}. (A5)

For procedure declarations, the attribute part of the declaration consists of a formal parameter list and the statement that constitutes the procedure body:

is-proc-attr = (⟨s-param-list:is-id-list⟩, ⟨s-st:is-st⟩). (A6)

A procedure call consists of an identifier selected by the selector s-id and an argument list selected by the selector s-arg-list:

is-proc-call = (⟨s-id:is-id⟩, ⟨s-arg-list:is-id-list⟩).

Figure 18 illustrates the abstract syntax

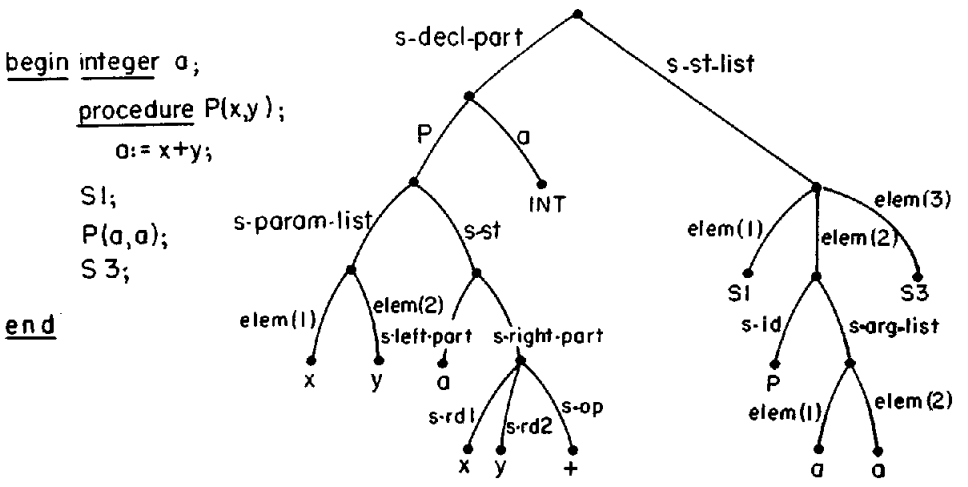


FIG. 18. The syntax of procedure declarations and procedure calls.

of a block containing a declaration of an integer variable a and a two-parameter procedure $P(x, y)$ with body $a:=x+y$, followed by three statements S1, $p(a, a)$, and S3. The attribute component of an identifier declared to be integer or logical always consists of an elementary object (INT or LOG). The attribute component of an identifier declared to be a procedure consists of a complex syntactic structure, even for procedures with relatively simple procedure bodies. Declarations of procedures that return values (function-type procedures) have an expression component specifying the value to be returned in addition to the above procedure attributes:

```
is-funct-attr = (⟨s-param-list:is-id-list⟩,
                 ⟨s-st:is-st⟩, ⟨s-expr:is-expr⟩).        (A7)
```

The above productions specify the abstract syntax of the declaration part of a block. The syntax of the statement part of a block may be specified by giving the syntax of statements in the statement list. EPL has four kinds of statements referred to as *assignment statements*, *conditional statements*, *procedure statements*, and *blocks:*

```
is-st = is-assign-st ∨ is-cond-st ∨
        is-proc-call ∨ is-block.                       (A8)
```

An assignment statement consists of a left part, which is a variable (identifier), and a right part, which is an expression:

```
is-assign-st = (⟨s-left-part:is-var⟩,
                ⟨s-right-part:is-expr⟩).               (A9)
```

The syntax of binary addition and multiplication operators is similar to that specified in Section 3.2. However, components of expressions may include unary operators and function designators (calls of function-type procedures). A function designator consists of a function name and a list of actual parameters that is restricted to being an identifier list:

```
is-funct-des = (⟨s-id:is-id⟩,
                ⟨s-arg-list:is-id-list⟩).              (A13)
```

Conditional statements consist of a logical expression, a **then** component and an **else** component:

```
is-cond-st = (⟨s-expr:is-expr⟩,
              ⟨s-then-st:is-st⟩,                       (A16)
              ⟨s-else-st:is-st⟩).
```

A call of a procedure statement has precisely the same syntax as a function designator, but it occurs in contexts in which it is interpreted as a self-contained statement rather than as an argument of an expression.

4.2 The Evaluation Strategy of EPL

EPL is a block structure language which, at block entry time, creates a new data cell for each identifier declared in the block head, and associates a reference to the newly created cell with instances of execution of the identifier. Identifiers of the source program denote different cells in different instances of activation of the block in which they are declared, and may, in the case of recursively activated procedures, denote multiple, simultaneously-existing cells. The one-to-many correspondence between identifiers and data cells may be implemented in a number of different ways, and we shall prove in Part 5 that two significantly different implementations are equivalent. In the present section, we shall describe, informally, the evaluation strategy of a particular implementation of EPL, so that the reader will have an intuitive basis for understanding the formal specification of this evaluation strategy.

The relation between source language identifiers and their values is modeled by three symbol tables.

1) An *environment table* keeps track of the relation between source language identifiers and the cell names with which they are associated. It is assumed that there is an infinite supply of cells with *unique* cell names, and that these cells are allocated in a specific order determined by a system-defined *unique-name generator*. The unique-name generator is assumed to generate a sequence of integers 1, 2, $\cdots$, which are then used by a system function "un-name" to generate an associated sequence of unique names n_1, n_2, $\cdots$. Entries in the environment table are

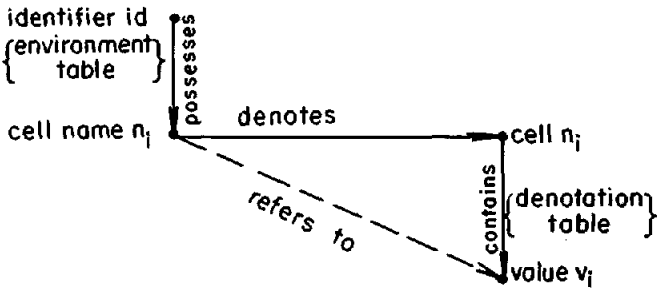


FIG. 19. The "possess," "denotes," "refers to," and "contains" relations.

assumed to have the form ⟨id:n_i⟩, where id is a source language identifier and n_i is the unique name of a cell containing values associated with that identifier.

2) A *denotation table* keeps track of the relation between cells and their values. Entries in the denotation table have the form ⟨n_i:value⟩, where the form of the value is determined by the type attribute of the identifier with which the cell name n_i is associated.
3) An *attribute table* keeps track of the type attributes of cells. Entries in the attribute table have the form ⟨n_i:typespec⟩ where "typespec" is the type specification of the type attribute of the identifier with which the cell name n_i is associated.

On entry to a block, entries of the form ⟨id:n_i⟩ are created in the environment directory for every identifier declared in that block, and entries of the form ⟨n_i:typespec⟩ are created in the attribute table for every cell n_i created on entry to the block. Uninitialized identifiers of types integer (INT) and logical (LOG) have no entry placed into the denotation directory at block entry time; an entry is created only when a value is assigned to the variable. Identifiers of the type procedure are initialized at block entry time to denotations, which include a specification of the procedure body and of the environment in which nonlocal variables of the procedure are to be interpreted.

The environment table models the notion of a cell being *possessed* by an identifier in the sense of ALGOL 68 [V1], while the denotation table models the relation between cells and the values they contain, as illustrated in Figure 19. The relation between an identifier and the cell name it possesses may vary for different instances of activation of the block in which it is declared, but remains invariant for a given instance of activation. The relation between a cell and the value it contains may vary during the execution of the block in which the cell is created. Assignment of values to cells has a later binding time than "possession" of cell names by identifiers.

The evaluation strategy of EPL will be illustrated by considering the execution of the following simple program:

```
begin integer x, y;
  y:=2;
end
```

In EPL this program would be represented by the composite object shown in Figure 20. Note that in this composite object the object y occurs both as a selector and as an elementary object. Entry to this block will occur when the instruction **interpret-block**(t) (where t is the above composite object) appears at a terminal vertex of the control tree. Execution of this instruction causes the following actions:

1) Push down the current environment

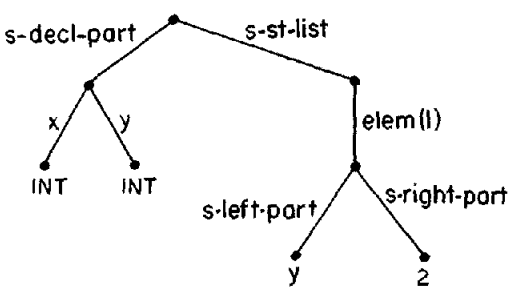


FIG. 20. Syntax of an EPL program.

table and the current control tree so that they can be reinstated when execution of this block has been completed. The state component in which environments and control trees are pushed down is called the *dump*. The structure of the dump is further discussed in the following section.

2) If E is the environment table prior to execution of this instruction, then augment E by the operation $\mu(E; \langle x{:}n_i\rangle, \langle y{:}n_{i+1}\rangle)$, where n_i and n_{i+1} are the unique names of the next cells to be allocated by the unique name generator. Note that this has the effect of *augmenting* the environment table by entries for x and y if none are present, and of *replacing* the entries for x and/or y if they are present. This corresponds to the notion that an identifier of an inner block augments the name space of an identifier if there is no previously-defined instance of that identifier, and replaces a previous instance of the identifier if there is one.
3) Augment the attribute table by the entries $\langle n_i{:}\text{INT}\rangle$ and $\langle n_{i+1}{:}\text{INT}\rangle$. Note that since n_i and n_{i+1} are unique names, there is no question of knocking out previous entries of the attribute table.
4) Make no change in the denotation table, since variables of the type INT are not initialized at block entry time.

When the above block entry transformations have been performed, the body of the block may be executed. In this case, the body consists of the single statement "$y{:}=\ 2$;" which causes the denotation "2" to be associated with the unique name n_{i+1} in the denotation table.

When execution of the block body is completed, the environment and control tree that immediately preceded block entry are reinstated, and a terminal vertex of the reinstated control tree is chosen as the next instruction. The computation terminates when the control tree popped up as a result of block exit is the null object Ω.

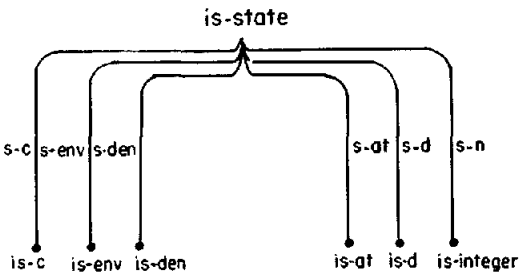


FIG. 21. The principal components of the state.

4.3 Components of the State

States in EPL satisfy the predicate "is-state" and have the following six immediate components:

1) a *control component* selected by a selector "s-c" and satisfying a predicate "is-c";
2) an *environment* component selected by a selector "s-env" and satisfying the predicate "is-env";
3) a *denotation* component selected by the selector "s-den" and satisfying the predicate "is-den";
4) an *attribute* component selected by the selector "s-at" and satisfying the predicate "is-at";
5) a *dump* component (stack) selected by the selector "s-d" and satisfying the predicate "is-d"; and
6) a *unique name generator* component which, at any given moment, specifies the index i of the next unique name n_i to be generated; it is selected by the selector "s-n" and satisfies the predicate "is-integer."

The predicate "is-state" may be specified as follows in terms of the predicates of its immediate components:

$$\text{is-state} = (\langle\text{s-c:is-c}\rangle, \langle\text{s-env:is-env}\rangle, \langle\text{s-den:is-den}\rangle, \langle\text{s-at:is-at}\rangle, \langle\text{s-d:is-d}\rangle, \langle\text{s-n:is-integer}\rangle).$$

The structure of objects satisfying the predicate "is-state" may also be illustrated by the one-level tree in Figure 21.*

At the beginning of an EPL computation the environment, attribute, denotation, and dump directories are empty. Initial states of an EPL computation have the following form:

$$\text{initial-state} = \mu(\langle\text{s-c:}\mathbf{interpret\text{-}program}(t)\rangle, \langle\text{s-n:1}\rangle).$$

* This tree has *predicates* at its terminal vertices. It denotes the set of all Vienna objects having the given set of immediate components satisfying the given predicates.

That is, the control component consists of a tree with a single instruction vertex **interpret-program**(t), where **interpret-program** is a macro instruction whose expansion will give rise to the information structure transformations required to evaluate t, and the unique name component is always initialized to 1.

During the evaluation of a program t, successive instructions at terminal vertices of the control tree are executed until a final state with an empty control tree is reached. States occurring during the computation may have control trees with a large number of vertices, and may have nonempty environment, denotation, attribute, and dump components. The syntax of environment, denotation, attribute, and dump components is described below.

The environment component is a table whose entries have the form $\langle x{:}n_i\rangle$, where x is an identifier and n_i is a unique name. Its syntax may be specified as follows:

$$\text{is-env} = (\{\langle \text{id:is-n}\rangle \parallel \text{is-id(id)}\}).$$

The attribute component is a table whose entries have the form $\langle n_i{:}$is-type$\rangle$, where n_i is a unique name satisfying the predicate "is-n," and "is-type" is satisfied by the literal types INT, LOG, PROC, and FUNCT. Its syntax may be defined as follows:

$$\text{is-at} = (\{\langle n{:}\text{is-type}\rangle \parallel \text{is-n}(n)\}).$$

The entries in the denotation directory have the form $\langle n_i{:}$denotation$\rangle$, where the denotation may be an integer, a logical, a procedure, or a function denotation:

$$\text{is-den} = (\{\langle n{:}\text{is-value} \vee \text{is-proc-den} \vee \text{is-funct-den}\rangle \parallel \text{is-n}(n)\}).$$

Entries in the denotation table are selector-object pairs whose first component is a unique name and whose second component satisfies one of the three predicates "is-value," "is-proc-den," or "is-funct-den."

The set of objects satisfying the predicate "is-value" is the set of representations of integers and logical values. The predicates "is-proc-den" and "is-funct-den" are defined as follows:

$$\text{is-proc-den} = (\langle \text{s-attr:is-proc-attr}\rangle, \langle \text{s-env:is-env}\rangle)$$

and

$$\text{is-funct-den} = (\langle \text{s-attr:is-funct-attr}\rangle, \langle \text{s-env:is-env}\rangle)$$

where is-proc-attr and is-funct-attr are defined by (A6) and (A7) of Table I as:

$$\text{is-proc-attr} = (\langle \text{s-param-list:is-id-list}\rangle, \langle \text{s-st:is-st}\rangle) \tag{A6}$$

and

$$\text{is-funct-attr} = (\langle \text{s-param-list:is-id-list}\rangle, \langle \text{s-st:is-st}\rangle, \langle \text{s-expr:is-expr}\rangle). \tag{A7}$$

Procedure and function denotations contain not only the procedure body and parameter specifications that constitute their syntactic definition, but also a specification of the *environment at the time of declaration.* The environment of declaration of a procedure is required at execution time to determine the denotations of nonlocal procedure parameters, as illustrated by the following example:

```
begin integer a, b;
  procedure P(x, y);
      a := x + y;
  begin integer a;
        a := 1;
        P(a, a);
        print(a);
  end;
  print(a);
end
```

This program consists of an outer block with declarations of two integer variables a, b and a procedure P, and an inner block with a second declaration of a. During execution, the inner instance of a is assigned the value "1," and the call $P(a, a)$ then causes the value "2" to be assigned to the outer instance of a. The print(a) instruction in the inner block then prints the value "1," and the print(a) instruction in the outer block prints the value "2." The procedure P of this program satisfies the attribute "is-proc" and will have its denotation initialized on

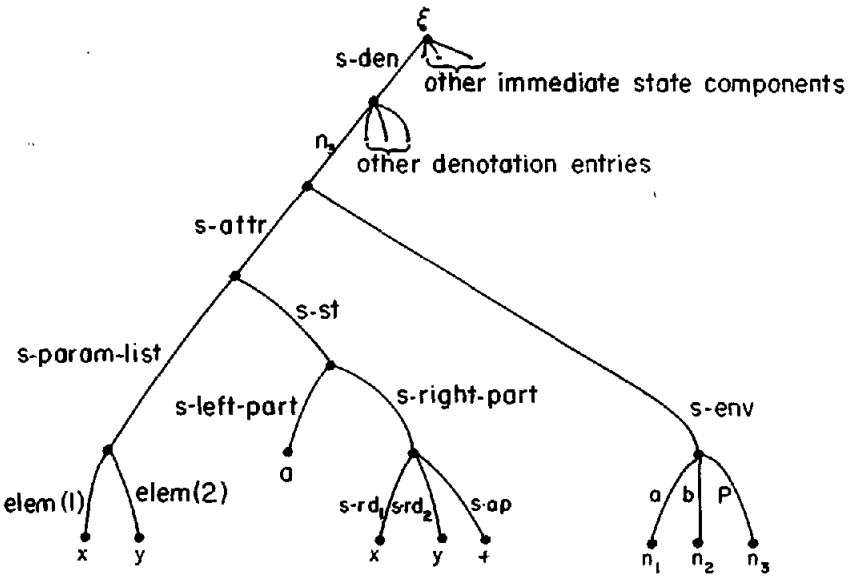


FIG. 22. The denotation of the procedure P.

entry to the outer block according to the structure of Figure 22.

On entry to the outer block the environment component "s-env(ξ)" of the state ξ is assigned the value $\mu_0(\langle a{:}n_1\rangle, \langle b{:}n_2\rangle, \langle P{:}n_3\rangle)$. On entry to the inner block, the environment component "s-env(ξ)" will be updated by the selector-object pair $\langle a{:}n_4\rangle$, resulting in the environment

$$\text{s-env}(\xi) = \mu_0(\langle a{:}n_4\rangle, \langle b{:}n_2\rangle, \langle P{:}n_3\rangle).$$

The assignment $a := 1$ assigns the value "1" to the denotation component n_4 associated with the inner instance of a, and the instances of a in the call $P(a, a)$ likewise refer to the value of n_4.

However, the call $P(a, a)$ causes the "current" environment component $\mu_0(\langle a{:}n_4\rangle, \langle b{:}n_2\rangle, \langle P{:}n_3\rangle)$ to be pushed down, and the environment $\mu_0(\langle a{:}n_1\rangle, \langle b{:}n_2\rangle, \langle P{:}n_3\rangle)$ of the procedure denotation to be installed. This environment is, in turn, updated by entries for x and y, resulting in the following environment during execution of the procedure P:

$$\mu_0(\langle a{:}n_1\rangle, \langle b{:}n_2\rangle, \langle P{:}n_3\rangle, \langle x{:}n_4\rangle, \langle y{:}n_4\rangle).$$

Moreover, the parameters x and y are initialized to their actual parameter values by means of entry $\langle n_4{:}1\rangle$ in the denotation component. Execution of the statement "$a := x + y$;" results in the addition of the component $\langle n_1{:}2\rangle$ to the denotation directory.

Exit from the procedure causes the environment of procedure execution to be replaced by the environment $\mu_0(\langle a{:}n_4\rangle, \langle b{:}n_2\rangle, \langle P{:}n_3\rangle)$ at the time of procedure call. The environment entry $\langle a{:}n_1\rangle$ is inaccessible at this point, and the print(a) instruction of the inner block prints the value "1" of the denotation component n_4. Exit from the inner block causes the environment of the block to be replaced by the environment $\mu_0(\langle a{:}n_1\rangle, \langle b{:}n_2\rangle, \langle P{:}n_3\rangle)$ at the time of block entry. Execution of the second print(a) instruction therefore causes the value "2" of the denotation component n_1 to be printed.

The above discussion of block and procedure execution assumes the existence of a dump component in which environments and initial trees are stacked on entry to blocks and procedures, so that they can be retrieved in a last-in-first-out order on exit. The structure of the dump component "s-d(ξ)" is discussed below. The syntax of the dump component may be defined as follows:

is-d = (⟨s-env:is-env⟩, ⟨s-c:is-c⟩, ⟨s-d:is-d⟩) ∨ is-Ω.

Note that in defining tables we did not have to explicitly list the empty table Ω, since a syntactic specification of an arbitrary number of objects by the use of the double slash is assumed to include zero objects as a special case. However, in the case of the dump, the null object Ω must be explicitly included. In order to illustrate the operations on dump components during entry to and exit from blocks and procedures, the pushdown and popup operations will be briefly illustrated. Pushdown of the current environment and control components in the dump component may be accomplished as follows:

$$\text{s-d} \leftarrow \mu_0(\langle\text{s-env:s-env}(\xi)\rangle, \langle\text{s-c:s-c}(\xi)\rangle, \langle\text{s-d:s-d}(\xi)\rangle).$$

The above operation creates a new component s-d(ξ) which has the old component s-d(ξ) as its s-d-component, and has the current environment and control components of the state as its s-env- and s-c-components. Popup of the environment and control components together with reinstatement of the previous dump can be accomplished by the following instruction:

exit =
s-env ← s-env(s-d(ξ))
s-c ← s-c(s-d(ξ))
s-d ← s-d(s-d(ξ))

The three lines above are an example of a value-returning instruction that has no line containing "PASS," and therefore does not return a value. The effect of this instruction is to modify the s-env-, s-c-, and s-d-components of the state as a side effect (see Section 3.1).

The attribute, denotation, and unique name components contain *global* information that is common to *all* block and procedure activations. The unique name generator must clearly be accessible globally, since unique names generated on block and procedure entry must be globally unique. The attribute and denotation tables are global in the sense that there is only one program-wide copy of each table. However, entries in the attribute and denotation tables are accessible only through the environment table. The environment table itself is local to a given block activation, since it is replaced by a new environment table on entry to a block or procedure, and replaced by a previously existing environment table on exit from a block or procedure. There may be many "local copies" of a given environment table entry in existence, all sharing access to a given, globally-available entry in the attribute or denotation tables.

The structure of the dump component and the proliferation of multiple copies of environment table entries ⟨id:n⟩ in the dump will be illustrated by the following example of a four-level nested block structure.

```
B1: begin integer v, w;
    B2: begin integer x, y;
        B3: begin integer y, z;
            B4: begin integer a;
                statements
                end
            end
        end
    end
```

On entry to the block B1, the dump d_1 and environment env_1 are as follows:

$$d_1 = \Omega$$

and

$$env_1 = \mu(\Omega; \langle v{:}n_1\rangle, \langle w{:}n_2\rangle)$$
$$= \mu_0(\langle v{:}n_1\rangle, \langle w{:}n_2\rangle).$$

On entry to the block B2, the dump d_2 and the environment env_2 each consist of the following structures:

$$d_2 = \mu_0(\langle\text{s-env:env}_1\rangle, \langle\text{s-c:s-c}(\xi)\rangle, \langle\text{s-d:}\Omega\rangle)$$

and

$$env_2 = \mu(env_1; \langle x{:}n_3\rangle, \langle y{:}n_4\rangle)$$
$$= \mu_0(\langle v{:}n_1\rangle, \langle w{:}n_2\rangle, \langle x{:}n_3\rangle, \langle y{:}n_4\rangle).$$

On entry to the block B3, the dump d_3 will consist of the following structure:

$$d_3 = \mu_0(\langle\text{s-env:env}_2\rangle, \langle\text{s-c:s-c}(\xi)\rangle, \langle\text{s-d:}d_2\rangle).$$

The environment env_3 immediately following the entry to B3 will contain five entries:

$$env_3 = \mu(env_2; \langle y{:}n_5\rangle, \langle z{:}n_6\rangle)$$
$$= \mu_0(\langle v{:}n_1\rangle, \langle w{:}n_2\rangle, \langle x{:}n_3\rangle, \langle y{:}n_5\rangle, \langle z{:}n_6\rangle).$$

At this point, there are three copies of the named object ⟨v:n_1⟩, two in the dump d_3

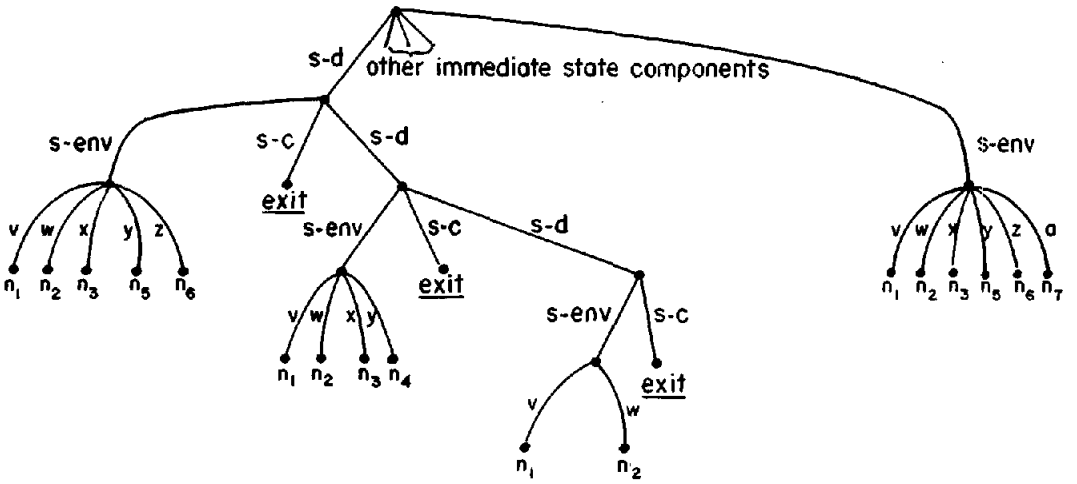


FIG. 23. The dump component after entry to the block B4. There are three copies of the selector-object pair $\langle v{:}n_1\rangle$ in the dump and an additional copy in the current environment.

and one in the current environment env_3. Entry to the block B4 would result in a dump d_4 with three copies of $\langle v{:}n_1\rangle$, and a fourth copy of $\langle v{:}n_1\rangle$ in the environment env_4. However, throughout execution there is at most one entry for n_1 in the attribute and denotation components. The structure of the dump and environment components just after entry to the block B4 is illustrated in Figure 23.

The above discussion of the state components and overall evaluation strategy of EPL provides a basis for the detailed specification of the semantics of EPL in the following section.

4.4 The Semantics of EPL

All EPL computations start with a control tree containing the single vertex **interpret-program**(t), where t is a structure satisfying the predicate "is-program" and a unique name counter initialized to 1. The macro instruction **interpret-program**(t), which causes itself to be replaced by the instruction **interpret-block**(t), may be defined as follows:

interpret-program(t) =
interpret-block(t). (I1)

Thus, execution of the instruction **interpret-program**(t) always yields a new single-vertex control tree containing the instruction **interpret-block**(t). Execution of the instruction **interpret-block**(t) (defined by (I2) below) in turn causes: push-down of the current control tree and environment table in the dump; creation of a new four-vertex control tree that will update the environment, denotation, and attribute directories; execution of the statement list using the updated environment; and, finally, exit from the block. The **interpret-block**(t) instruction is a value-returning instruction that returns a null value and modifies the dump component and control-tree component of the state.*

interpret-block(t) =
s-d ← μ_0(⟨s-env:s-env(ξ)⟩, ⟨s-c:s-c(ξ)⟩,
⟨s-d:s-d(ξ)⟩)
s-c ← **exit**;
int-st-list(s-st-list(t));
int-decl-part(s-decl-part(t));
update-env(s-decl-part(t)) (I2)

* The instruction **interpret-block**(t) may be generated, not only at the beginning of the program from the instruction **interpret-program**(t), but also during execution, whenever an inner block is about to be executed. The state ξ, occurring in the body of the **interpret-block** instruction, is the state immediately prior to execution of this instruction with the currently executed instruction deleted from the control tree (see Appendix). Execution of this instruction for the outer EPL block yields a new component s-d = Ω, since the components s-env(ξ), s-c(ξ), and s-d(ξ) are all Ω.

When execution of this instruction has been completed the control tree of the new state has the form shown in Figure 24. Since the new control tree is linear, its instructions will be executed in a linear sequence starting with the instructions at the terminal vertex. Execution of the instruction **update-env**(s-decl-part(t)) will cause the environment table to be updated with declared identifiers of the current block. Execution of **int-decl-part**(s-decl-part(t)) will cause the denotation and attribute tables to be updated. Execution of **int-st-list**(s-st-list(t)) will cause the statements of the block to be executed. The instruction **exit** will cause the environment prior to execution of this block to be reinstated.

The update-env instruction is defined in terms of an update-identifier instruction for each of the identifiers in the declaration part.

$$\begin{aligned}&\textbf{update-env}(t) =\\ &\quad \textbf{null};\\ &\quad\quad \{\textbf{update-identifier}(\mathrm{id}, n);\\ &\quad\quad\quad n{:}\textbf{un-name} \mid \mathrm{id}(t) \neq \Omega\}\end{aligned} \tag{I3}$$

The instruction **null** in the specification (I3) above passes a null value back to a null set of argument positions, and may be regarded as an abbreviation for the instruction "PASS ← Ω." Execution of the instruction update-env(t) for an argument t that satisfies the predicate "is-decl-part" generates a two-vertex subtree for each identifier declared in the declaration part. For a block with two declared identifiers x, y, the control tree of Figure 25 would be generated.

The instruction **un-name** is a parameterless value-returning instruction that delivers as its value the unique name n_i determined by the current value i = s-n(ξ) of the unique name generator, and increments the unique name generator by 1.

$$\begin{aligned}&\textbf{un-name} =\\ &\quad \text{PASS} \leftarrow n_{\text{s-n}(\xi)}\\ &\quad \text{s-n} \leftarrow \text{s-n}(\xi) + 1\end{aligned}$$

If s-n(ξ) has the value i prior to execution of this instruction, then n_i is passed back as the return information, and the state component s-n(ξ) is incremented to $i + 1$.

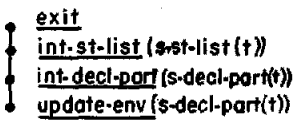


FIG. 24. Control tree after block entry.

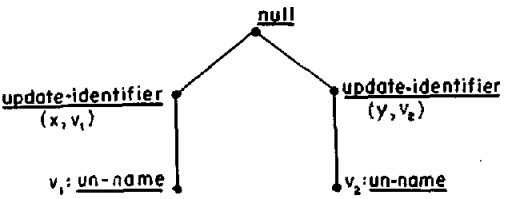


FIG. 25. Control tree for updating environment.

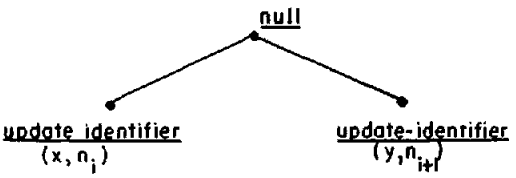


FIG. 26. Partially executed control tree for updating environment.

Execution of the instructions at the vertices labeled v_1 and v_2 (Figure 25) generates two unique names, say n_i and n_{i+1}, and stores them at parameter positions v_1 and v_2, respectively, resulting in the control subtree shown in Figure 26.

The instruction **update-identifier**(id,n) simply updates the environment table by the entry ⟨id:n⟩, and may be defined as follows:

$$\begin{aligned}&\textbf{update-identifier}(\mathrm{id},n) =\\ &\quad \text{s-env} \leftarrow \mu(\text{s-env}(\xi);\ \langle \mathrm{id}{:}n\rangle)\end{aligned} \tag{I4}$$

where id satisfies the predicate "is-id" and n satisfies the predicate "is-n." When execution of each of the **update-identifier** instructions has been completed the control subtree generated by the **update-env** instruction becomes the single vertex **null**.

When execution of the **update-env** instruction has been completed the instruction **int-decl-part**(t) becomes the new terminal vertex of the control tree. This instruction, like the **update-env**(t) instruction, has a parameter t that satisfies the predicate "is-decl-part." The purpose of the **int-decl-part**(t) instruction is to update the attribute

and denotation components, assuming that the environment component has already been updated by the instruction **update-env**(t). In particular, procedure declarations of the current declaration part result in a denotation that has the updated environment as a component, so that nonlocal identifiers within the procedure statement can be properly handled (see example in previous section).

The instruction **int-decl-part**(t) creates a control subtree with a dummy root and one branch for updating the environment and denotation directory of each individual declared variable.

int-decl-part(t) =

null; (I5)

{**int-decl**(id·s-env(ξ), id(t)) | id(t) $\neq \Omega$}

The first parameter "id·s-env(ξ)" of each instruction **int-decl** is the unique name of an identifier, id, of the declaration part, t. The second parameter "id(t)" is the attribute component of the identifier, which is defined by the production (A3) to be INT or LOG for data variables, and by the productions (A6) and (A7) for identifiers of the type PROC or FUNCT (see Table I). Figure 27 illustrates the effect of executing the instruction **int-decl-part**(t) for a declaration part t consisting of an integer declaration for a and a procedure declaration for P.

The instruction **int-decl**(n,attr) has a first parameter n, which is the unique name of a declared identifier, and a second parameter "attr," which is the attribute of that identifier. Its effect is to update the attribute table for all unique names n_i generated on block entry, and, in the case of procedure declarations, to initialize the denotation table to the procedure denotation.

int-decl(n, attr) =

is-var-attr(attr) → s-at ← μ(s-at(ξ); ⟨n:attr⟩)

is-proc-attr(attr) → s-at ← μ(s-at(ξ); ⟨n:PROC⟩)

s-den ← μ(s-den(ξ);

⟨n:$\mu_\emptyset$(⟨s-attr:attr⟩, (I6)

⟨s-env:s-env(ξ)⟩)))

is-funct-attr(attr) → s-at ← μ(s-at(ξ); ⟨n:FUNCT⟩)

s-den ← μ(s-den(ξ);

⟨n:$\mu_\emptyset$(⟨s-attr:attr⟩,

⟨s-env:s-env(ξ)⟩)))

Thus, if the attribute parameter is variable attribute (A5) from Table I, only the attribute directory is updated because the denotation component is initially null. If the attribute parameter satisfies either predicate "is-proc-attr" or "is-funct-attr," both the attribute and denotation components are updated. Moreover, the environment component of the procedure denotation is the environment of the block in which it is declared. Since the **int-decl** instruction is executed *after* the **update-env** instruction for the current block, the environment of a procedure denotation includes identifiers declared in the same block as the procedure.

When the **int-decl** instruction has been

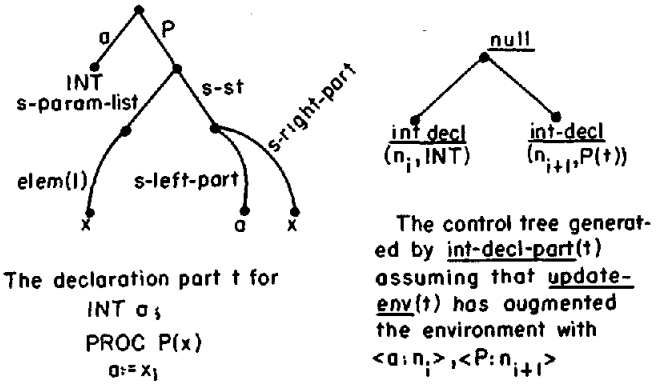


FIG. 27. The instruction **int-decl-part**(t).

executed for all identifiers declared in the block head, and the dummy instruction "PASS ← Ω" has been executed, we are left with a control tree whose terminal vertex is the instruction **int-st-list**(s-st-list(*t*)). This instruction specifies the execution of the sequence of statements of the associated block in the order of their occurrence in the statement list.

int-st-list(*t*) =

  is-<>(*t*) → PASS ← Ω

    T → **int-st-list**(tail(*t*)); (I7)

      interpret-statement

        (head(*t*))

Here, *t* satisfies the predicate "is-st-list," and the list functions "head(*t*)" and "tail(*t*)" are as defined in Section 2.2. This instruction causes execution of the first statement of the statement list, followed by recursive execution of the instruction **int-st-list** for the remaining statements of the statement list until no more statements are left in the statement list.

We are now ready to consider the basic computational instruction **interpret-statement.** A statement may be an assignment statement, a conditional statement, a call of a statement-type procedure, or a block. The instruction **interpret-statement** has a conditional action for each of these alternatives:

interpret-statement(*t*) =

  is-assign-st(*t*) → **int-assign-st**(*t*)

  is-cond-st(*t*) → **int-cond-st**(*t*) (I8)

  is-proc-call(*t*) ∧ (at_t = PROC) →

    int-proc-call(*t*)*

  is-block(*t*) → **int-block**(*t*)

Execution of an assignment statement involves evaluation of the expression that constitutes the right part, and assignment of the resulting value as the denotation of the unique name associated with the left part. Execution of a conditional statement involves evaluating a logical expression and using the truth value of that expression to choose between execution of the "then statement" and the "else statement." Execution of a procedure call involves pushdown of the current environment and execution of the procedure in the environment of the procedure declaration, augmented by the environment of actual parameters. Execution of a block involves recursive activation of the **interpret-block** instruction.

The EPL production (A9) of Table I specifies that an assignment statement has the following syntax:

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

    <s-right-part:is-expr>).

In defining the instruction **int-assign-st**(*t*), we shall make use of the abbreviation n_t to denote the unique name associated with the left part of the assignment statement. Thus, n_t = s-left-part(*t*)(s-env(ξ)). Using this abbreviation, we may define the instruction **int-assign-st** as follows:

int-assign-st(*t*) =

  is-var-attr(n_t(s-at(ξ))) → (I9)

    assign(n_t, *v*);

      v:**int-expr**(s-right-part(*t*))

  T → error

Thus, an assignment statement is legitimate only if the left part is of type INT or LOG. Its execution involves executing the instruction **int-expr** using the right part as a parameter and assigning the result to the unique name n_t. Discussion of the instruction **int-expr** is deferred until later. The **assign** instruction updates the denotation entry of its first parameter, and may be defined as follows:

assign(*n*, *v*) = s-den ← μ(s-den(ξ); (I10)

    <*n*:convert(*v*, *n*(s-at(ξ)))>).

The parameters *n* and *v* are assumed to satisfy the predicates "is-n" and "is-value,"

* The expression "at_t" is an abbreviation for the expression "s-id(*t*) (s-env(ξ)) (s-at(ξ))," which selects the state component that specifies the type of the procedure identifier. In this expression, "s-id(*t*)" selects the procedure identifier (see (A17), Table I). The subexpression "s-id(*t*)(s-env(ξ))" selects the unique name, say n_t, associated with that identifier in the current environment table "s-env(ξ)." Application of the unique name n_t to the attribute table "s-at(ξ)" in turn yields the attribute n_t(s-at(ξ)) associated with the procedure identifier.

respectively. The convert function* is assumed to be a semantic primitive that converts the value v to the type specified by the attribute table entry for the first parameter n. The values v resulting from the application of **int-expr** must be of the type INT or LOG. It is assumed that conversion between integer and logical values is possible in both directions.

The instruction **int-cond-st** is defined in terms of an instruction **branch**(v, t_1, t_2), which chooses between the execution of the statements t_1 and t_2, depending on the truth value of v.

int-cond-st(t) =

branch(v, s-then-st(t), s-else-st(t)); (I11)

v:**int-expr**(s-expr(t))

The instruction **branch** may, in turn, be defined as follows:†

branch(v, t_1, t_2) =

convert(v, LOG) → **int-st**(t_1) (I12)

T → **int-st**(t_2)

It is assumed above that t satisfies the predicate "is-cond-st," t_1 and t_2 satisfy the predicate "is-st," and v satisfies the predicate "is-value."

The instructions **int-proc-call** and the similarly structured **int-funct-call** are probably the most subtle instructions of the EPL definition, since they involve pushing down one environment, installing a second environment, and updating the new environment with an environment of formal/actual parameter correspondences. The parameter t of the instruction **int-proc-call**(t) satisfies the predicate "is-proc-call(t)," and, according to (A17) of Table I, has the following syntax:

is-proc-call = (⟨s-id:is-id⟩,

⟨s-arg-list:is-arg-list⟩).

The identifier selected by s-id has a complex denotation whose components are used in specifying the action of **int-proc-call**(t). It is convenient to introduce the following abbreviations in referring to components of the parameter t:

1) "n_t = s-id(t) (s-env(ξ))" denotes the unique name associated with the procedure identifier of the calling procedure;
2) "$\text{den}_t = n_t$(s-den(ξ))" selects the procedure denotation that has the syntactic form (⟨s-env:is-env⟩, ⟨s-attr:is-proc-attr⟩);
3) "p-list_t = s-param-list·s-attr(den_t)" selects the list of formal parameters of the procedure attribute;
4) "env_t = s-env(den_t)" selects the environment of the procedure attribute;
5) "arglist_t = s-arg-list(t)" selects the actual parameter list of the procedure call;
6) "st_t = s-st·s-attr(den_t)" selects the statement component of the procedure attribute (see definition of "is-proc-den" in the previous section and (A6) in Table I); and
7) "length(list)" is a function which computes the length of the list that constitutes its argument.

The construction "int-proc-call(t)" first checks that the length of the list of actual parameters equals the length of the list of formal parameters. If this check is successful the current environment and control are pushed down in the dump, the environment component "env_t" of the called procedure is installed and modified by actual/formal parameter correspondences, and the control component is set to execute the statement "st_t," which constitutes the procedure body.*

* VDL distinguishes between *instructions*, which have the syntax and semantics described in Section 3.1, and *functions*, which may be regarded as primitive data structure manipulation operators as are the μ operator and selection operators. VDL instructions are always defined in terms of lower-level primitives, while functions are, themselves, primitives. Following [L8], we have adopted the convention that instruction names are in boldface, while function names are in the regular typeface. Thus, in the definition (I10) above the instruction "assign" is in boldface, while the function "convert" is in the regular typeface.

† The instruction **interpret-statement** is abbreviated below as **int-st**.

* Note that ⟨elem(i)(p-list_t):elem(i)(arg-list_t) (s-env(ξ))⟩ is a pair of the form ⟨x:n_i⟩, where x is a formal parameter of the procedure and n_i is the unique name of the corresponding actual parameter. Passing of the unique name as the actual parameter corresponds precisely to call by reference [W2]. During execution of the procedure,

```
int-proc-call(t) =
  (length(arg-list_t) = length(p-list_t)) →
    s-d ← μ_0(⟨s-env:s-env(ξ)⟩,
      ⟨s-c:s-c(ξ)⟩, ⟨s-d:s-d(ξ)⟩)
    s-env ← μ(env_t:{⟨elem(i)                    (I13)
      (p-list_t):elem(i)(arg-list_t)
      (s-env(ξ))⟩ | 1 ⩽ i
      ⩽ length(p-list_t)})
    s-c ← exit;
          int-st(st_t)
  T → error
```

When execution of this macro instruction is completed, the new control tree has the form shown in Figure 28.

The statement st_t, which constitutes the procedure body, may, in general, be a block, and is executed in the new environment. When execution of the procedure body is completed the instruction **exit** is executed. This instruction simply restores the environment control and dump components prior to the procedure call, or, in the case of exit from a block, prior to entry into the block.

```
exit =
  s-env ← s-env(s-d(ξ))
  s-c ← s-c(s-d(ξ))                              (I14)
  s-d ← s-d(s-d(ξ))
```

The above instructions complete the specification of semantics for *statement* execution. In the remainder of this section, we will specify the semantics of *expression* execution.

The basic instruction for expression execution is **int-expr**(t). This instruction specifies actions for binary operations, variables, and constants, which are similar to those given in Section 3.2 above. However, expressions of EPL may contain unary operators with their operands and function designators with their arguments. The conditional instruction that defines **int-expr**(t) has actions corresponding to the alternatives "binary operator," "unary operator," "function designator," "variable," "constant," and "error." The abbreviation n_t used in the alternative "is-var(t)" below stands for "t·s-env(ξ)," and denotes the unique name associated with the variable t.

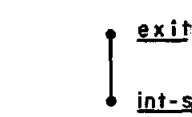


FIG. 28. Control tree for a procedure call.

```
int-expr(t) =
  is-bin(t) → int-bin-op(s-op(t), a, b);
        a:int-expr(s-rd1(t)),
        b:int-expr(s-rd2(t))
  is-unary(t) → int-un-op(s-op(t), a);
        a:int-expr(s-rd(t))
  is-funct-des(t) ∧ (at_t = FUNCT) →             (I15)
    pass-value(n);
      int-funct-call(t, n);
        n:un-name
  is-var(t) ∧ is-var-attr(n_t(s-at(ξ))) →
    PASS ← n_t(s-den(ξ))
  is-const(t) → PASS ← val(t)
  T → error
```

The actions for "is-bin," "is-unary," "is-var," and "is-const" are similar to those described in the expression evaluator of Part 3. However, the action when the parameter t is a function designator is more complex and will be discussed in greater detail below.

If t is a function designator then, according to (A13) of Table I, it has the following syntax:

```
is-funct-des = (⟨s-id:is-id⟩,
                ⟨s-arg-list:is-id-list⟩).
```

The unique name associated with the function identifier will have type specification "FUNCT" in the attribute table and a denotation, constructed at the time of function declaration, which has the following

the denotation of the actual parameter may be accessed by the operation "n_i(s-den(ξ))," while assignment to n_i of the value val_i can be performed by the operation "den ← μ(s-den(ξ); ⟨n_i:val_i⟩)."

syntax:

$$\text{den}_t = (\langle\text{s-env:is-env}\rangle, \langle\text{s-attr:is-funct-attr}\rangle).$$

The environment component of den_t will be a copy of the environment at declaration time. The attribute component of den_t will, according to (A7) of Table I, have the following syntax:

$$\text{is-funct-attr} = (\langle\text{s-param-list:is-id-list}\rangle, \langle\text{s-st:is-st}\rangle, \langle\text{s-expr:is-expr}\rangle).$$

If the parameter t satisfies the expression "is-funct-des," and if the attribute at_t of the function identifier is "FUNCT," then the control subtree shown in Figure 29 is generated. The label v_i is the label of the instruction **int-expr**(t) which caused this control subtree to be generated.

Execution of the terminal vertex of this control tree generates a unique name and stores it as a parameter value in the two preceding vertices. This unique name will be used by the instruction **int-funct-call** to assign the value of the function to the denotation of the parameter of the **pass-value** instruction. The **pass-value** instruction will then pass this denotation to parameter positions specified by its label parameter:

pass-value(n) =
PASS ← n(s-den(ξ)). (I16)

The instruction **int-funct-call**(t,n) is similar in structure to the instruction **int-proc-call**(t): it checks the equality of lengths of actual and formal parameter lists; pushes down copies of the current environment and control trees in the dump; installs the environment of the function declaration modified by the list of formal/actual parameter correspondences; and installs a control tree for executing the statement that constitutes the function body and the expression that constitutes the function value. The abbreviations used in **int-funct-call** are the same as those used in **int-proc-call.**

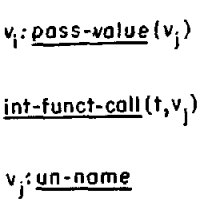


FIG. 29. Control subtree for a function designator.

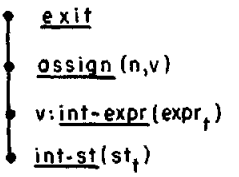


FIG. 30. Control tree for a function call.

int-funct-call(t, n) =
(length(arg-list$_t$) = length(p-list$_t$)) →
s-d ← μ_0(⟨s-env:s-env(ξ)⟩,
⟨s-c:s-c(ξ)⟩, ⟨s-d:s-d(ξ)⟩)
s-env ← μ(env$_t$; {⟨elem(i) (p-list$_t$):
elem(i) (arg-list$_t$) (s-env(ξ))⟩ (I17)
| 1 ≤ i ≤ length(p-list$_t$)})
s-c ← **exit**;
assign(n, v);
v:**int-expr**(expr$_t$);
int-st(st$_t$)
T → error

The control subtree created by the instruction **int-funct-call**(t,n) has the form shown in Figure 30. Execution of this control subtree results in execution of the statement "st$_t$," evaluation of the expression "expr$_t$" which specifies the value of the function, and storage of the result as the value of the parameter v of the **assign** instruction. The **assign** instruction assigns the value of v as the denotation of the unique name n (see (I10)). The **exit** instruction then reestablishes the environment in existence prior to the function call.

The above 17 instructions, (I1) through (I17), constitute a complete definition of EPL in the Vienna Definition Language. This definition assumes that programs are represented by an abstract syntax in an

"intermediate language" that is independent of a specific linear representation but exhibits the operator/operand structure of expressions. The semantics is defined by specifying the information structure transformations to which source programs give rise when they are executed.

It is hoped that the above definition of EPL gives some insight, not only into the definition techniques of VDL, but also into the mechanisms whereby block structure language may be implemented. VDL is an important contribution to the programming language field, not only because it is a tool for the definition of new languages, but also because partial definitions of language features in VDL allow implementors to describe their implementations with precision for the purpose of communication with others interested in the precise characteristics of an implementation. VDL is a user-oriented language-definition language that emphasizes language description for the benefit of users rather than for the benefit of machines.

5. PROOFS OF INTERPRETER EQUIVALENCE

We will now introduce a number of alternative implementations of identifier accessing and develop a proof of equivalence of two alternative methods of identifier accessing using a proof technique known as the *twin machine technique*. Part 5 will conclude with a section briefly surveying techniques for proving assertions about programs in general and about interpreters in particular.

5.1 The Local Environment Model

The identifier accessing mechanism of Part 4 ensures that accessible identifiers "id" can always be accessed by the operation "id·s-env(ξ)." This model of identifier accessing will be called the *complete environment model*, since the environment table contains entries $\langle id{:}n_i\rangle$ for the *complete* set of identifiers, id, accessible at the current point of execution.

The complete environment model leads to the pushdown of *multiple copies* of correspondences between identifiers and unique names in the dump (see Section 4.3 above), and is not generally used in *practical* implementations of block structure languages. An alternative model that corresponds more closely to practical implementations of identifier accessing is considered below.

Identifiers declared in a given block are said to be *local* to that block, and formal parameters of a procedure are said to be *local* to that procedure. An environment table consisting of all entries $\langle id{:}n_i\rangle$ local to a given block or procedure is said to be a *local environment table*. The complete environment at any given point of execution may always be defined in terms of local environments of blocks and procedures that *statically enclose* the current point of execution. This fact may be used to define *local environment models* which simulate complete environment tables by linked lists of local environment tables. The simulation of the complete environment by a linked list of local environments is illustrated in Figure 31 (next page) for the four-level nested block structure of Figure 23 (p. 40).

The complete environment model requires pushdown of the complete environment in the dump on block and procedure entry, and results in duplication of environment table entries in successive dump components. The local environment model requires pushdown of only the local environment component in the dump on entry to blocks and procedures, and thereby avoids duplication of environment table entries.

The link from a given local environment to a local environment of the statically enclosing block or procedure is referred to as a *static link*. When the program consists of a set of nested blocks with no procedure declarations or calls, then the set of statically linked local environments always constitutes a simple chain, and the static link of a given local environment always points to the local environment from which the block was entered. However, when the program contains procedure declarations and calls, the set of statically linked environments is generally a tree structure. The relation between tree and stack representations of local environments of a local environment model is illustrated schematically in Figure 32. It is assumed that the procedure P was called

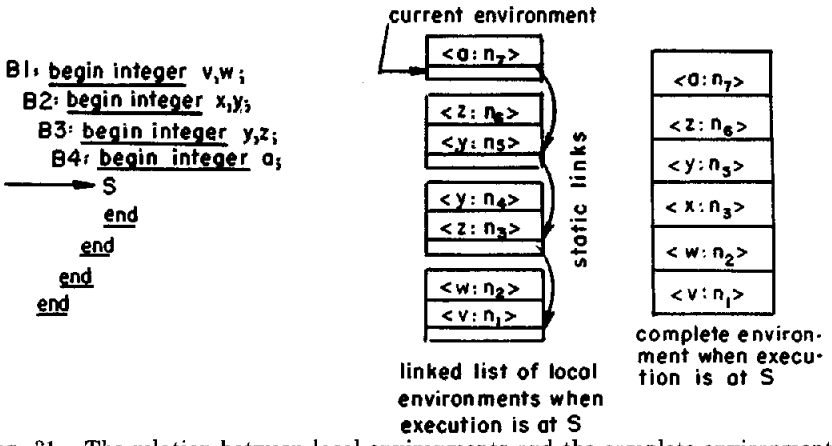


FIG. 31. The relation between local environments and the complete environment.

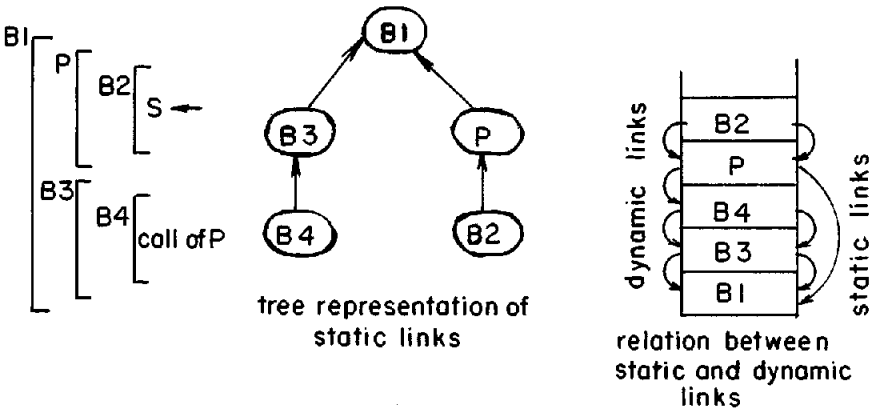


FIG. 32. Tree and stack representations of local environments.

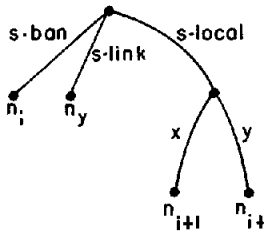


FIG. 33. VDL representation of a local environment.

from the block B4 and that execution is currently at the statement S of the block B2.

Following Lucas [L6], we shall represent a local environment by three state components:

1) a block activation name (ban) selected by a selector "s-ban";
2) a static link selected by a selector "s-link"; and
3) a local environment selected by a selector "s-local."

Figure 33 illustrates a local environment containing two identifiers, x and y. On entry to the block or procedure in which the environment of Figure 33 is created, unique names n_i, n_{i+1}, and n_{i+2} are generated for the block activation name and for identifiers x and y, while the s-link-component is initialized to the block activation name of a statically enclosing local environment. Note that different instances of entry to a block associate different block activation names with that block and different unique names with identifiers x and y. This reflects the fact that block structure languages uni-

formly associate new cells with new instances of activation of local identifiers. The components "s-ban," "s-link," and "s-local" of the "current" local environment are represented by Lucas [L6] as immediate components of the state, and must be pushed down in the dump on entry to a new block or procedure.

The local environment model avoids the duplication of components $\langle id{:}n_i \rangle$ in successive complete environments of the dump, but requires a more complex accessing function for accessing nonlocal identifiers. If an identifier, id, is not in the current local environment, then the local environment whose block activation name corresponds to the s-link-component of the current environment must be found in the dump, and successive local environments determined in this way must be searched for occurrences of the identifier, id. This corresponds to looking for occurrences of id in successive, textually enclosing local environments, starting with the innermost environment and working outwards. The process of searching for an identifier in successive, textually enclosing environments is formally defined in Section 5.3, below.

5.2 The Twin Machine Model

In the remainder of Part 5, we will define a local environment implementation of EPL, and prove that this local environment model is equivalent to the complete environment model of Part 4.

The local environment model could, in principle, be defined by a set of VDL instructions that parallel those given in Section 4.4, but the instructions differ at the points where creation, deletion, and use of identifier accessing structures are involved. However, we wish to define the local environment model so that it is in a form in which its equivalence to the complete environment model may easily be proved. Following Lucas [L6], we shall define a *twin machine model* that contains the identifier accessing mechanisms of *both* the complete environment and the local environment models. As illustrated in Figure 34, the twin machine model contains certain state components that are common to both the local and complete environment implementations of block structure languages, and certain state components concerned with identifier accessing that are different in each implementation.

On entry to blocks and procedures the instructions of the twin machine construct the identifier accessing structure for both the local environment and the complete environment model. On exit from blocks and procedures the instructions of the twin machine update the identifier accessing structure of both implementations. Thus, at any point of execution, identifier accessing may be performed either by the accessing mechanism of the complete environment model or by the accessing mechanism of the local environment model.

Let $\mathrm{FIND}_C(\mathrm{id},\xi)$ be the identifier accessing function of the *complete* environment model, which, given an identifier, id, and a state, ξ, of the twin machine, determines the value of the denotation component currently associated with id, using the complete environment accessing mechanism. Let $\mathrm{FIND}_L(\mathrm{id},\xi)$ be the identifier accessing function of the *local* environment model, which, given an identifier, id, and a state, ξ, of the twin machine, determines the value of the denotation component currently associated with id, using the local environment accessing mechanism.

The proof of equivalence of the complete and local environment models involves proving the functional equivalence of FIND_C and FIND_L for all identifiers id and states ξ of the twin machine:

$$(\forall\xi)(\forall\mathrm{id})(\mathrm{FIND}_C(\mathrm{id},\xi) = \mathrm{FIND}_L(\mathrm{id},\xi)).$$

In order to prove this equivalence we must define the structure of states ξ and the in-

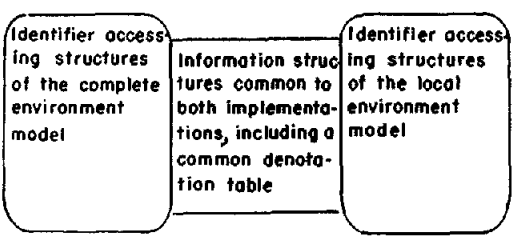


FIG. 34. Information structures of the twin machine model.

structions of the twin machine. The above equivalence can then be proved by showing: 1) that it holds for all initial states ξ_0 of the twin machine; 2) that equivalence is preserved by block and procedure entry; and 3) that exit from a block or procedure results in a state for which the equivalence has already been proved.

The state structure and instructions of the twin machine are defined in the remainder of this section; the structure of twin machine proofs is further considered in the following section; and the proof of equivalence of the complete environment and local environment models of identifier accessing is then given in Section 5.4.

The state ξ of the twin machine has control, environment, denotation, attribute, dump, and unique name generator components that play the same role as the corresponding components of the complete environment model. In addition, it has a block activation name (ban) component which is a unique name, a link component which is a unique name, and a local environment component which has the same syntactic structure as the complete environment component (although it generally has fewer entries at any given point of execution). The syntax of the complete state may be defined as follows:

is-twin-state = (⟨s-c:is-c⟩, ⟨s-env:is-env⟩, ⟨s-den:is-den⟩, ⟨s-at:is-at⟩, ⟨s-d:is-d⟩, ⟨s-n:is-integer⟩, ⟨s-ban:is-n⟩, ⟨s-link:is-n⟩, ⟨s-local:is-env⟩).

The structure of twin machine states is illustrated in Figure 35.

On entry to a block or procedure during the execution of a program in the twin machine, information is stored in the dump to allow both the local environment and the complete environment to be restored on exit from the block or procedure. In order to accomplish this the dump of the twin machine has the six components s-ban(ξ), s-link(ξ), s-local(ξ), s-env(ξ), s-c(ξ), and s-d(ξ). Pushdown in the dump can be defined in terms of the following function:

stack(ξ) = μ_0(⟨s-ban:s-ban(ξ)⟩, ⟨s-link:s-link(ξ)⟩, ⟨s-local:s-local(ξ)⟩, ⟨s-env:s-env(ξ)⟩, ⟨s-c:s-c(ξ)⟩, ⟨s-d:s-d(ξ)⟩).

Restoring the state on exit from a block or procedure is then accomplished by the following instruction.

unstack = s-ban ← s-ban·s-d(ξ)
s-link ← s-link·s-d(ξ)
s-local ← s-local·s-d(ξ)
s-env ← s-env·s-d(ξ)
s-c ← s-c·s-d(ξ)
s-d ← s-d·s-d(ξ)

The instruction of the twin machine corresponding to the instruction **interpret-block**(t) of the EPL machine may be defined in terms of the above functions as follows.

interpret-block(t) =
s-d ← stack(ξ)
s-c ← **unstack**; **int-st-list**(s-st-list(t)); **int-decl-part**(s-decl-part(t)); **update-env**(s-decl-part(t)) (I2′)

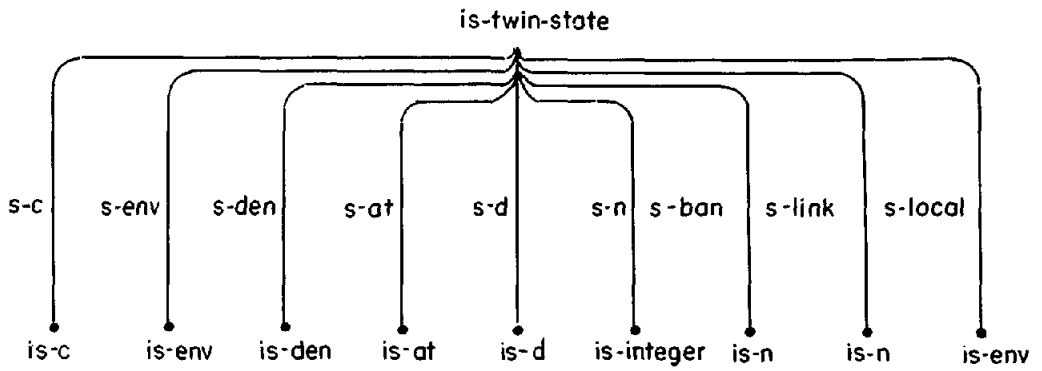


FIG. 35. Immediate state components of the twin machine.

The entries s-ban(ξ), s-link(ξ), and s-local(ξ) associated with a given block activation are created during execution of the **update-env** instruction. This instruction may be defined in terms of an instruction **update-env-1** which essentially generates unique names for all declared identifiers of the block and for the block activation name.*

update-env(declaration-part) =
  update-env-1(v_1, v_2);
    {id(v_1):**un-name** |
    id(declaration-part) $\neq \Omega$}
    $\cup$ {v_2:**un-name**} (I3′)

The value parameter v_1 will be replaced on execution by a structure with one component for every identifier declared in the block. If k identifiers $\mathrm{id}_1, \mathrm{id}_2, \cdots, \mathrm{id}_k$ are declared in the block and the associated unique names $n_{i+1}, n_{i+2}, \cdots, n_{i+k}$, respectively, are generated for these parameters, then the parameter v_1 will be replaced by the structure shown in Figure 36. The parameter v_2 of the **update-env-1** instruction will be replaced by a unique name, say n_{i+k+1}, which will subsequently be used as the block activation name of the block that is being entered.

The **update-env-1** instruction may be defined as follows.

update-env-1(env-0, n) =
  s-env ← μ(s-env(ξ);
    {⟨id:id(env-0)⟩ | id(env-0) $\neq \Omega$})
  s-local ← env-0
  s-ban ← n
s-link ← s-ban(ξ)

Thus, the instruction **update-env-1** performs the following functions: updates the s-env-component of the state just as in the EPL machine; initializes the s-local-component to the local environment "env-0" created by the value-returning instructions executed immediately prior to this instruction; initializes the s-ban-component to the unique block activation name; and initializes the s-link-component to the block activation name of the dynamically enclosing block activation.

* The instruction "id(v_1):**un-name**" causes the unique name that constitutes the value of this instruction to be passed back to the id-component of v_1.

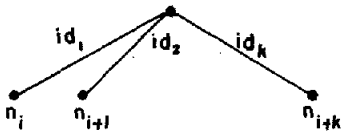


FIG. 36. The evaluated parameter v_1 of **update-env-1**.

The instruction **int-decl** of the twin machine, which initializes the attribute and denotation tables at declaration time, differs slightly from the corresponding instruction of the EPL machine because procedure denotations of the EPL machine must contain the block activation name of the statically enclosing block. Thus, **int-decl**(n,attr) must be redefined as follows for the twin machine.

int-decl(n, attr) =
  is-var-attr(attr) →
    s-at ← μ(s-at(ξ); ⟨n:attr⟩)
  is-proc-attr(attr) →
    s-at ← μ(s-at(ξ); ⟨n:PROC⟩)
    s-den ← μ(s-den(ξ);
      ⟨n:μ_0(⟨s-attr:attr⟩,
      ⟨s-env:s-env(ξ)⟩, (I6′)
      ⟨s-ban:s-ban(ξ)⟩)⟩)
  is-funct-attr(attr) →
    s-at ← μ(s-at(ξ); ⟨n:FUNCT⟩)
    s-den ← μ(s-den(ξ);
      ⟨n:μ_0(⟨s-attr:attr⟩,
      ⟨s-env:s-env(ξ)⟩,
      ⟨s-ban:s-ban(ξ)⟩)⟩)

The principal remaining difference between the twin machine and the EPL machine lies in the instructions for procedure and function call. The parameter t of the twin machine instruction **int-proc-call**(t) has the form (⟨s-id:is-id⟩, ⟨s-arg-list:is-arg-list⟩), just as for the corresponding instruction (I13) of the EPL machine. The list of formal/actual parameter correspondences constructed by the complete-environment and local-environment accessing rules will be referred to as "complete$_t$" and

"local$_t$," respectively. Using the abbreviations preceding the definition of (I13) in Section 4.4, "complete$_t$" and "local$_t$" are each defined as follows:

$$\text{complete}_t = \mu_0(\{\langle \text{elem}(i)\ (\text{p-list}_t):\text{elem}(i)\ (\text{arg-list}_t)\ (\text{s-env}(\xi))\rangle \mid 1 \leqslant i \leqslant \text{length}(\text{p-list}_t)\})$$

and

$$\text{local}_t = \mu_0(\{\langle \text{elem}(i)\ (\text{p-list}_t):\text{find}(\text{elem}(i)\ (\text{arg-list}_t), \xi)\rangle \mid 1 \leqslant i \leqslant \text{length}(\text{p-list}_t)\}).$$

The instruction **int-proc-call**(t) may now be defined as follows.*

int-proc-call(t) =
(length(arg-list$_t$) =
length(p-list$_t$)) →
s-d ← stack(ξ)
s-local ← local$_t$
s-env ← μ(env$_t$; {⟨elem(i)
(p-list$_t$):elem(i)
(arg-list$_t$)(s-env(ξ))⟩ | (I13′)
I ⩽ i ⩽ length(p-list$_t$)})
s-link ← s-ban(den$_t$)
s-c ← **exit**
int-st(st$_t$)
update-ban(n)
n:**un-name**
T → error

where **update-ban**(n) = s-ban ← n.

The instruction **int-funct-call**(t) may be constructed in a corresponding manner from (I17) of Section 4.4.

* Note that we cannot update the complete environment component by the specification "s-env ← μ(env$_t$; complete$_t$)" since the second parameter of the μ instruction must be a sequence of selector-object pairs rather than a constructed object. Note also that we cannot assign a new unique name to the s-ban-component by the specification "s-ban ← **un-name**" since this would assign the unexecuted symbol **un-name** to s-ban(ξ). The only way of generating a new unique name is to execute the **un-name** instruction in the control tree.

5.3 The Structure of Twin Machine Proofs

The twin machine technique is concerned with proving the equivalence of two programming language implementations A and B, where both A and B are defined in terms of the sequences of states (information structures) that they generate during program execution. In order to describe the structure of the proof technique it is convenient to introduce the notion of an information structure model [W3, W5].

DEFINITION: An *information structure model* is a triple $M = (I, I^0, F)$, where I is a countable set of information structures (structured states), $I^0 \subset I$ is a set of initial states, and F is a finitely representable set of unary operations (primitive instructions) whose domain and range are a subset of I.

DEFINITION: A *computation* in an information structure model $M = (I, I^0, F)$ is a sequence $I_0, I_1, \cdots$ of elements of I such that $I_0 \in I^0$, and for each $j = 0, 1, \cdots$ there is an $f_j \in F$ such that $I_{j+1} = f_j(I_j)$.

DEFINITION: A *terminating computation* is a computation that, for some integer n, generates an I_n to which no element $f \in F$ is applicable.

The definition of EPL in Part 4 may be viewed as an information structure model whose set I consists of all composite objects that can occur as states, whose set I^0 consists of the set of all composite objects that can occur as initial states, and whose set F consists of the set of VDL instructions (I1) through (I17). The local environment and twin machine models discussed in the first two sections of Part 5 may similarly be defined as information structure models.

A proof of equivalence of two implementations may be viewed as a proof of equivalence of the two information structure models that constitute the definition of the implementations. Let $M = (I, I^0, F)$ and $M' = (I', I^{0\prime}, F')$ be the information structure models corresponding to the two implementations being proved equivalent. The twin machine proof technique requires a twin machine model $M'' = (I'', I^{0\prime\prime}, F'')$ to

be defined whose information structures I'' contain components corresponding to both I and I', and whose instructions F'' combine the effects of execution of corresponding instructions in F and F'.

The twin machine requires that the computations of the information structure models M and M' can be executed by a common control mechanism. A twin machine can be constructed only for two implementations whose control may be synchronized. However, two implementations of a given programming language that generate different information structures for identifier accessing but execute the same re-entrant program representation will, in general, execute corresponding instruction sequences for corresponding computations, and may, therefore, be embedded in a twin machine.

The equivalence of two implementations M and M' of identifier accessing may be defined in terms of the condition that, for any instance of accessing of an identifier in M'', the cell accessed by the identifier-accessing mechanisms in M'' corresponding to M and M' is the same. Let $\text{find}_1(\text{id},I_j'')$ be an accessing function that computes the cell name n_i for identifiers id accessible in the state I_j'' of M'' by means of the accessing mechanism of M, and be undefined otherwise. Let $\text{find}_2(\text{id},I_j'')$ be the corresponding definition using the accessing mechanism in M'' corresponding to M'. Then the equivalence of M and M' can be defined in terms of the functional equivalence of find_1 and find_2 in M''.

DEFINITION OF INTERPRETER EQUIVALENCE: M is equivalent to M' if and only if for all identifiers id and states I_j'' of M'':

$$\text{find}_1(\text{id},I_j'') = \text{find}_2(\text{id},I_j'').$$

When M is the complete environment model of Part 4, then

$$\text{find}_1(\text{id},\xi) = \text{id}\cdot\text{s-env}(\xi).$$

The function $\text{find}_2(\text{id},\xi)$ associated with the local environment model will be written as "find(id,ξ)." The function find(id,ξ) searches the sequence of local environments statically linked to the current local environment for an entry ⟨id:n⟩ whose first component is "id." If there is no such entry it delivers the null value. The definition of find(id,ξ) requires the definition of an auxiliary function find-d(BAN, DUMP), which searches the dump component for an instance of the block activation "BAN."

$$\text{find}(\text{id},\xi) = \text{s-ban}(\xi) = \Omega \rightarrow \Omega$$

$$\text{id}(\text{s-local}(\xi)) \neq \Omega \rightarrow \text{id}(\text{s-local}(\xi))$$

$$\text{T} \rightarrow \text{find}(\text{id},\text{find-d}(\text{s-link}(\xi),\xi)$$

If $\text{s-ban}(\xi) = \Omega$, then "id" is not declared in the textually enclosing environment, and the null value is returned. If $\text{id}(\text{s-local}(\xi)) \neq \Omega$, then the identifier is declared in the given local environment, and the unique name n_i allocated at declaration time is returned as a value. If neither the first nor the second condition is met then the value of "find(id,ξ)" reduces to the value of "find(id, find-d(s-link(ξ),ξ))," where "find-d(s-link(ξ), ξ)" delivers as its value a new state ξ' which is obtained from ξ by chasing down the dump component until the local environment whose block activation name is "s-link(ξ)" is reached. The function "find-d" may be defined as follows:

$$\text{find-d}(\text{BAN}, \text{DUMP}) =$$

$$\text{s-ban}(\text{DUMP}) = \Omega \rightarrow \Omega$$

$$\text{s-ban}(\text{DUMP}) = \text{BAN} \rightarrow \text{DUMP}$$

$$\text{T} \rightarrow \text{find-d}(\text{BAN}, \text{s-d}(\text{DUMP})).$$

If the block activation name of DUMP is Ω, then the block activation name "BAN" does not occur in the dump and the value "Ω" is returned. If the block activation name of DUMP is "BAN," then we have reached the local environment defined by the first parameter, and we return the resulting dump stack as a value to the calling function "find." If neither of the first two conditions is satisfied we evaluate find-d recursively using the dump component of the current state as the new second parameter. Since the dump is a stack with a finite number of entries, find-d will, in a finite number of steps, return either a dump component with block activation name "BAN" or the value "Ω."

5.4 Equivalence of Direct Accessing and Chaining Models

We wish to prove that for all states ξ of the twin machine and all identifiers id, the value of "find(id,ξ)" is the same as the value of "id·s-env(ξ)." That is, we wish to prove

$$\forall\xi\forall(\mathrm{id})[\mathrm{id}\cdot\text{s-env}(\xi) = \mathrm{find}(\mathrm{id},\xi)]. \quad (0)$$

The proof will be by induction. We shall show that this condition is trivially satisfied by the initial state, and that any relevant state modification preserves the truth of this condition.

Initially, the following state components have the value Ω:

$$\text{s-env}(\xi) = \Omega$$
$$\text{s-ban}(\xi) = \Omega$$
$$\text{s-link}(\xi) = \Omega$$
$$\text{s-local}(\xi) = \Omega$$

From the definition of find(id,ξ), it follows that, since s-ban(ξ) = Ω, find(id,ξ) = Ω for all id. Since s-env(ξ) = Ω, it follows that id·s-env(ξ) = Ω for all id. Thus, find(id,ξ) = id·s-env(ξ) = Ω for all id and for all initial states.

Instead of proving that *all* state transitions preserve equivalence, we note that the environment associated with a state ξ may be modified only on entry to a block, on call of a procedure, or on exit from a block or procedure. Therefore, it is sufficient to prove that equivalence of identifier accessing is preserved by block and procedure entry, and that block and procedure exit recreate an accessing environment for which the equivalence has already been proved.

The proof makes use of an auxiliary function "update," which updates a complete environment "env" by a local environment "env-0" and may be defined as follows.

DEFINITION:

$$\text{update(env, env-0)} = \mu(\text{env};\{\langle \text{id}: \text{id(env-0)}\rangle \mid \text{id(env-0)} \neq \Omega\}). \quad (1)$$

The auxiliary function "update" satisfies the following auxiliary lemma.

AUXILIARY LEMMA:

$$\text{s-env}(\xi) = \text{update}(\text{s-env}(\xi_{\text{link}}), \text{s-local}(\xi)). \quad (2)$$

where ξ_{link} is the dump component pointed to by the static link of the current environment and may be defined as follows:

$$\xi_{\text{link}} = \text{find-d}(\text{s-link}(\xi), \xi). \quad (3)$$

This auxiliary lemma states the seemingly obvious fact that a total environment s-env(ξ) is identically equal to the environment s-env(ξ_{link}) of the textually enclosing activation, updated by the local environment of the current block activation. This relation is used in [L6] as an auxiliary lemma in proving the main result. The proof of this lemma is by induction on entry to blocks and procedures. It is shown that condition (2) is true initially, and that it is preserved on entry to a block or on call of a procedure. Since the proof is tedious, we shall omit the details and prove the main result, assuming the truth of the lemma.

The proof of the main result will be by induction on the number of block and procedure activations that have been entered but not exited at a given point of execution. This number will be called the *dynamic nesting depth*. For the initial state ξ_0 the dynamic nesting depth is zero, and the following identity is satisfied:

$$\mathrm{id}\cdot\text{s-env}(\xi_0) = \mathrm{find}(\mathrm{id}, \xi_0) = \Omega.$$

Now, assume that the condition (0) is satisfied for all states having a dynamic nesting depth less than n. That is, assume that for all states ξ with dynamic nesting depth less than n the following condition is satisfied.

INDUCTIVE HYPOTHESIS:

$$\mathrm{id}\cdot\text{s-env}(\xi) = \mathrm{find}(\mathrm{id},\xi). \quad (4)$$

The identity will be proved for states ξ with dynamic nesting depth n by first considering the case of identifiers local to the nth nesting level and then considering nonlocal identifiers. For local identifiers, the following condition is satisfied.

ASSUMPTION (local id):

$$\text{id}\cdot\text{s-local}(\xi) \neq \Omega \tag{5}$$

Using the auxiliary lemma (2) and the definition of the update function (1), we have the following identity:

$$\text{s-env}(\xi) = \text{update}(\text{s-env}(\xi_{\text{link}}), \text{s-local}(\xi))$$

by (2); and

$$\begin{aligned}\text{s-env}(\xi) = \mu(&\text{s-env}(\xi_{\text{link}});\\ &\{\langle \text{id}:\text{id}\cdot\text{s-local}(\xi)\rangle\\ &\mid \text{id}\cdot\text{s-local}(\xi) \neq \Omega\})\end{aligned} \tag{6}$$

by (1). For all local identifiers satisfying condition (5), we have the following identity:

$$\text{id}\cdot\text{s-env}(\xi) = \text{id}\cdot\text{s-local}(\xi) \tag{7}$$

by (5), (6), and properties of the μ operator. From the definition of find(id,ξ), it now follows that:

$$\text{find}(\text{id},\xi) = \text{id}\cdot\text{s-local}(\xi) \tag{8}$$

by (5) and the definition of "find." From (7) and (8) it follows that:

$$\text{id}\cdot\text{s-env}(\xi) = \text{find}(\text{id},\xi) \tag{9}$$

by (7) and (8). The lemma is thus satisfied for identifiers declared in the local environment.

It remains to prove the lemma for identifiers declared in the nonlocal environment. For such identifiers, the following identity is satisfied.

ASSUMPTION (nonlocal id):

$$\text{id}\cdot\text{s-local}(\xi) = \Omega \tag{10}$$

Using the auxiliary lemma and the definition of the update function as in (6) above, we have:

$$\text{id}\cdot\text{s-env}(\xi) = \text{id}\cdot\text{s-env}(\xi_{\text{link}}) \tag{11}$$

by (6). For identifiers in the nonlocal environment we have:

$$\text{find}(\text{id},\xi) = \text{find}(\text{id},\xi_{\text{link}}) \tag{12}$$

by (10), (3), and the definition of "find." Now, by assumption, ξ_{link} has a nesting depth less than n, and the inductive hypothesis is satisfied for ξ_{link}:

$$\text{id}\cdot\text{s-env}(\xi_{\text{link}}) = \text{find}(\text{id},\xi_{\text{link}}) \tag{13}$$

by (4). It therefore follows from (11), (12), and (13) that the inductive hypothesis is satisfied for ξ:

$$\text{id}\cdot\text{s-env}(\xi) = \text{find}(\text{id},\xi). \tag{14}$$

Thus, the inductive hypothesis is satisfied both for identifiers in the local environment and for identifiers in the nonlocal environment, and hence for all identifiers in the environment.

We have shown that in the twin machine, identifier accessing by static links is equivalent to direct accessing. The direct accessing mechanism makes use of the complete environment component s-env(ξ), but does not use the components s-ban(ξ), s-link(ξ), or s-local(ξ). The linking mechanism makes use of the components s-ban(ξ), s-link(ξ), and s-local(ξ), but does not make use of the component s-env(ξ).

If only the direct accessing mechanism is used, all instruction components in the twin machine that modify s-ban(ξ), s-link(ξ), and s-local(ξ), and all syntactic references to these components in the abstract syntax can be removed without affecting the working of the complete environment model. In this way, the EPL machine of Section 4.4 is obtained. Let this machine be denoted by EPL_{COMP}. Now, if only the static link mechanism is used for accessing, all instruction components that modify s-env(ξ) and all references to s-env(ξ) in the abstract syntax can be removed without affecting the operation of the local environment model. In this way, the "local environment model for EPL" is obtained. Let this machine be denoted by EPL_{LOC}.

Let EPL_{TWIN} denote the twin machine. For the twin machine we have proved:

$$\text{id}\cdot\text{s-env}(\xi) = \text{find}(\text{id},\xi) \tag{15}$$

in EPL_{TWIN}. Also, we have

$$[\text{id}\cdot\text{s-env}(\xi)] \text{ in } \text{EPL}_{\text{TWIN}} = [\text{id}\cdot\text{s-env}(\xi)] \text{ in } \text{EPL}_{\text{COMP}} \tag{16}$$

and

$$[\text{find}(\text{id},\xi)] \text{ in } \text{EPL}_{\text{TWIN}} = [\text{find}(\text{id},\xi)] \text{ in } \text{EPL}_{\text{LOC}}. \tag{17}$$

From (15), (16), and (17) it follows that:

$$[\text{id}\cdot\text{s-env}(\xi)] \text{ in } \text{EPL}_{\text{COMP}} = [\text{find}(\text{id},\xi)] \text{ in } \text{EPL}_{\text{LOC}}. \quad (18)$$

Thus, direct accessing of environments in the complete environment machine always yields the same unique name as accessing by static links in the local environment model. A proof of equivalence for independent accessing mechanisms in the twin machine thus leads to the equivalence of two different machines with different accessing mechanisms.

The twin-machine proof technique is a technique for proving the equivalence of two interpreters, say INT_1 and INT_2, which process corresponding sequences of instructions and differ only in the way in which a given subcomputation, say f, is performed. The relation between the states of the twin machine, INT_{TWIN}, and the two component machines, INT_1, and INT_2, is illustrated in Figure 37. At any given point in the execution of a computation of INT_{TWIN} the instantaneous description consists of a component $I^{(1,2)}$ common to INT_1 and INT_2; a component $I^{(1)}$, which occurs only in computations of INT_1; and a component $I^{(2)}$, which occurs only in computations of INT_2. There is a subcomputation f in INT_{TWIN} that may be implemented by an instruction f_1, corresponding to the way in which f is computed in INT_1, or by an instruction f_2, corresponding to the way in which f is computed in INT_2. When f is implemented by f_1, the allocation of the information structure $I^{(2)}$ becomes redundant since it is not accessed except for purposes of creation and deletion. Similarly, when f is implemented by f_2, the allocation of the information structure $I^{(1)}$ becomes redundant.

The process of removing all references to an information component from instructions of the interpreter and of removing all instances of that component from the state

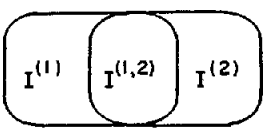


FIG. 37. The relation between states of the complete and local environment models.

syntax may be referred to as *projection*. The syntax and semantics of a twin machine, INT_{TWIN}, are defined so that implementation of the subcomputation f by f_1 allows projection of the states of INT_{TWIN} onto the states of INT_1, and implementation of f by f_2 allows projection onto the states of INT_2. A sufficient set of conditions under which projection is permitted may be summarized by a *projection theorem*.

PROJECTION THEOREM: If we can partition the states I of an information structure model into two components, $I^{(1)}$ and $I^{(2)}$, such that components of $I^{(2)}$ are never used in updating components of $I^{(1)}$ during computations of the interpreter, and if the "value" of the computation is completely determined by the $I^{(1)}$-component, then semantic actions that perform allocation of, or assignment to, components of $I^{(2)}$ can be deleted from instructions of the information structure model, and syntactic components of $I^{(2)}$ can be deleted from the specification of the state.

The notion of projection is related to the notion of garbage collection. Garbage collection may be justified by the following *garbage collection theorem* [W4, W6].

GARBAGE COLLECTION THEOREM: Cells that become inaccessible during execution of a program may be deleted without changing the logical effect of the program.

Projection imposes a weaker condition than inaccessibility as a sufficient condition for the deletion of information structures. However, the projection theorem imposes a *static* condition that must hold throughout execution, while the garbage collection theorem imposes a *dynamic* condition. Structures that play an active role in the initial part of a computation and then become inaccessible can be deleted by appeal to the garbage collection theorem, but not to the projection theorem. The following *deletion theorem* defines a more powerful criterion for deletion; it allows both the garbage collection and projection theorems to be deduced as special cases.

DELETION THEOREM: If the set of states, I,

of a computation can be partitioned into two components, $I^{(1)}$ and $I^{(2)}$, such that, after a certain point in the computation, components in $I^{(2)}$ are never used for updating components of $I^{(1)}$, and if the "value" of the computation is completely determined by the $I^{(1)}$-component, then the $I^{(2)}$-components may be deleted, and semantic actions that perform allocation, assignment, or other computations on elements of $I^{(2)}$ may be replaced by null operations.

Since inaccessible objects satisfy the conditions imposed on structures $I^{(2)}$ of the above theorem, the garbage collection theorem is a special case of the above deletion theorem. The projection theorem is obtained from the above theorem by imposing the following additional restrictions on structures $I^{(2)}$:

1) the point at which structures $I^{(2)}$ become "inactive" is always the beginning of the computation; and
2) structures $I^{(2)}$ must be inactive not only for a single computation but for all computations of the information structure model.

The deletion theorem is very powerful. For example, it may be used to justify the fact that, after the last instruction of a computation has been executed, all structures other than the output component may be deleted.

While the above discussion of projection, garbage collection, and deletion is, perhaps, somewhat of a digression, it has been included because it provides insight into certain principles underlying the twin machine proof.

5.5 Proof Techniques for Interpreter Equivalence

The twin machine proof technique was first developed by Lucas in [L6], and was used by him to prove the equivalence of the complete environment and local environment models. It was used by Henhapl and Jones [H1] to prove the equivalence of a whole spectrum of identifier accessing models.

A proof of equivalence of two implementations A and B becomes a proof of correctness if one of the two implementations, say A, is known to be correct. In order to prove that an equivalence class of implementations is correct, it is necessary to establish that a representative element in the equivalence class is correct or, alternatively, to choose one of the implementations in the equivalence class as a definition of correctness. In order to establish the correctness of the equivalence class of identifier accessing mechanisms proved correct by Lucas and Henhapl and Jones, it is necessary to establish the correctness of one of the mechanisms. This may be done by choosing one of the mechanisms as a definition of correctness. In principle, any one of the mechanisms could be chosen as the definition of correctness. However, in practice, it is appropriate to choose a definition that is close to the original definition of semantics in the ALGOL report [N1] and that is, at the same time, based on the mathematical definition of substitution for languages like the lambda calculus. The defining mechanism for identifier accessing will be referred to as *the copy rule model*. The copy rule model is used as a defining model in Henhapl and Jones [H1]. Slightly different versions of the copy rule model are defined in [B2], [W4], and [W6].

In the copy rule model, block entry gives rise to a new copy of the block being entered, with declared identifiers of the source program replaced by the unique names of cells allocated at block entry time. Procedures give rise to a new copy of the procedure statement, with formal parameters replaced by the unique names of cells which are initialized to actual parameters at procedure entry time. Identifier accessing mechanisms, which may be proved equivalent to the literal substitution model, may be viewed as mechanisms that *simulate* the literal substitution of unique names in a new copy of the block or procedure being entered.

The identifier accessing mechanisms proved equivalent to the copy rule model by Henhapl and Jones included the mechanism used in the implementation of the PL/I F interpreter. In the case of the PL/I F interpreter, the attempt to prove its correctness led to the discovery of an error

in its implementation and to a proof of correctness of a modified implementation. Proofs of correctness of an implementation require the implementation to be scrutinized very carefully, and may lead to the discovery of errors in the implementation or to the discovery of ways of improving or simplifying the implementation.

The twin machine technique for proving the equivalence of interpreters is one of a growing number of techniques for proving assertions about programs. Other such techniques have been developed by McCarthy [M2], Floyd [F2], Manna [M1], Hoare [H2], London [L5], Paterson [P1, L9], and others. However, these techniques have generally been concerned with proving the correctness of algorithms for specific tasks, while an interpreter correctness proof is concerned with the correctness of a method of computing a large class of algorithms.

Other work on the equivalence of interpreters has included the studies of equivalence of lambda calculus interpreters by Wegner [W2] and the proofs of correctness of lambda calculus interpreters by McGowan [M7]. McGowan developed a *mapping technique* for proving interpreter equivalence which may be used to prove the equivalence of two information structure models, $M = (I, I^0, F)$ and $M' = (I', I^{0'}, F')$, by mapping intermediate states of computations of M onto corresponding points of computations of M'. Berry [B2] has used the McGowan mapping technique to prove the equivalence of a literal substitution model of identifier accessing [H1, W4] and the contour model [J1]. Wegner [W4, W6] has developed a classification of the degree of difference between interpreters in terms of the mapping function, ϕ, from states of M to corresponding states of M'. Four kinds of interpreter equivalence may be distinguished and characterized in terms of attributes of the mapping function ϕ:

1) *fixed program equivalence*, in which the two interpreters being proved equivalent execute corresponding instructions of the same fixed program and differ only in the information structures generated during program execution;
2) *one-to-one equivalence*, in which there is a one-to-one correspondence between executed instructions of corresponding computations, but the program representations may be different;
3) *linearly related interpreters*, in which subcomputations are performed in the same order, but there may be a many-to-one or one-to-many correspondence between instructions required to perform specific subtasks; and
4) *segment-wise equivalence*, where there is a level of modularity at which there is a one-to-one correspondence of computational segments of the two interpreters, but where computations within segments may perform equivalent subcomputations in radically different ways.

The development of proof techniques for proving assertions about interpreters is still in its infancy and is likely to be a fruitful area of research during the next decade. This topic is further discussed in [W4, W6].

ACKNOWLEDGMENTS

Dan Berry, Barry Jacobs, and John Kelly contributed greatly to the quality of this paper by relentlessly pinpointing obscurities and errors in previous drafts of the text.

APPENDIX. THE DEFINITION OF VDL IN VDL

A.1 The Representation of Control Trees as Composite Objects

Since VDL is both a language-definition language and a programming language, it is appropriate to consider defining VDL in terms of itself. The language LISP has been defined in terms of itself by means of a LISP function, APPLY(P,D), which, given a LISP representation of a program P and its data D, computes the LISP representation of the value $P(D)$ of applying the program P to its data D.

Similarly, VDL may be defined in terms of itself by defining in VDL the state transformations that result from the execution of VDL instructions. We can define a function APPLY$_{\text{VDL}}(L, (P_L, D))$, which, given a VDL program L that specifies a programming language and a program-data pair (P_L, D) that is an initial representation of a program in the language L, specifies the se-

quence of state transformations that realizes the application of the program P_L to its data D.

In order to define $\text{APPLY}_{\text{VDL}}$, the set of all states that can occur during computations of the VDL interpreter must be specified as VDL data structures. In the body of this paper, all components of the state other than control trees have been represented as VDL data objects. However, control trees have VDL instructions at both terminal and nonterminal vertices, and cannot be treated as VDL data objects by the VDL interpreter.

Instructions are objects with a complex internal structure which are not appropriate for inclusion in the class of elementary objects, but which may be defined as composite objects of VDL. We shall consider a set of rules for converting instructions into composite objects, indicate how control trees may be defined as composite objects which have composite objects defining individual instructions as components, and indicate how the "instruction execution cycle" of VDL may be defined in terms of VDL instructions specifying transformations of the control tree and other state components.

Any given vertex of a control tree is associated both with an *instruction*, which resides at that vertex, and with *successor edges*, which select instructions at successor vertices. The composite object associated with that vertex will contain components that define the instruction and one component for each successor vertex. Instructions have three immediate components, referred to as: the *instruction name* component "s-in," the *argument list* component "s-al," and the *return information* component "s-ri." Successor vertices are always selected by the fixed selector names "succ_1, succ_2, $\cdots$." * Thus, a control tree vertex with n successor vertices is represented by a composite object with $n + 3$ immediate components, as indicated in Figure A1. The instruction name is an elementary object. The argument list is a composite object with as many components elem(1), elem(2), $\cdots$ as there are arguments.

An instruction has a return information component, ri, if and only if the instruction at the associated control tree vertex has a label parameter, v_i. The return information specifies the set of all positions in the control tree to which information is to be returned by *relative tree addresses* $\langle i, j \rangle$, where i specifies the ith predecessor of the instruction vertex and j specifies that information is to be returned to the jth element of the argument list of the ith predecessor vertex. Since values may generally be returned to a *component* of the jth argument of the ith predecessor vertex, the return information consists of a structure with two components selected by selectors "s-comp" and "s-addr," as indicated in Fig. A2 (next page).

In order to illustrate the composite-object representation of control trees, we will consider the representation of the following control tree.

apply$(a,$ s-rator$(t))$

s1(a):**value**(s-rand-1(t))

s2(a):**value**(s-rand-2(t))

This instruction is a variant of the definition of the macro associated with is-binary(t) in the arithmetic expression interpreter of Section 3.2. The second and third lines have structured labels and pass values to the s1- and s2-components of the parameter a. The **apply** instruction then assumes that its first parameter is a composite object with s1- and s2-components.

The control subtree associated with the above macro would be represented in our higher-level notation for control trees by the three-vertex control tree of Figure A3. At the "machine-language level" this control tree would be represented by the composite object of Figure A4.

The vertex of Figure A3 associated with the **apply** instruction is represented in Figure A4 by an s-in-component with the literal "apply"; an s-a1-component with an empty (uninitialized)

* The set of all sequences of composite selectors χ that can be constructed from the set SUCC = {succ_1 succ_2, $\cdots$} will be denoted by "SUCC*." Each vertex of the original control tree is associated with a unique element of SUCC*. The root vertex of the control tree is associated with the identity selector I, which may be thought of as the composite selector in SUCC* consisting of zero component selectors.

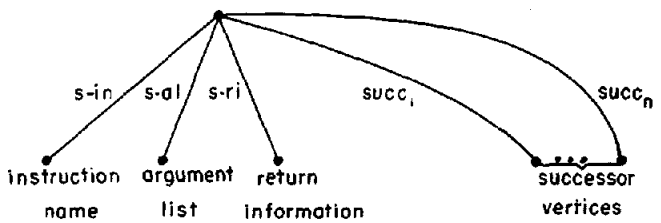
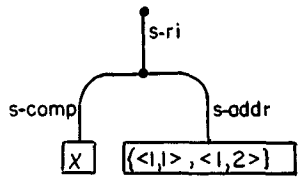


FIG. A1. Immediate components of an instruction vertex.



This return information component "s-ri" specifies that the instruction will return a value to the X-components of the first and second arguments <1,1>, <1,2> of the first predecessor instruction.

FIG. A2. Structure of the return information.

elem(1)-component and an elem(2)-component "s-rator(t)"; an empty s-ri-component (since there is no label parameter); and two successor components. The elem(1)-component of s-a1 will be constructed by the successor instructions, and will eventually have the structure shown in Figure A5.

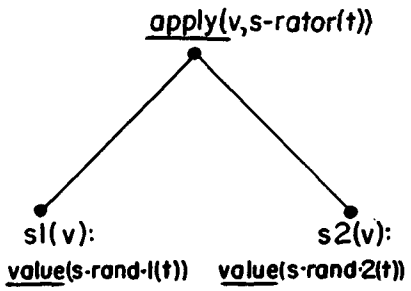


FIG. A3. "High-level representation" of a control tree.

The representation in Figure A4 of the vertex labeled "s1(v_i)" in Figure A3 has an s-in-component with the literal "value," an s-a1-component with the value "s-rand-1(t)," and an s-ri-component with two components s-comp and s-addr. The s-comp-component specifies that the value is to be returned to the s1-component of the argument. The s-addr-component specifies the set of argument positions to which the value is to be returned by a set of relative tree addresses. In this case, the return information specifies return to a single argument position, which is the s1-component of the first argument of the first predecessor. When the label parameter is a simple name v_i, then the s-comp-component of the return information specifies the identity selector I. The representation in Figure A4 of the vertex labeled "s2(v_i)" in Figure A3 is patterned after the representation of the vertex labeled "s1(v_i)."

Now that the instruction tree of the VDL

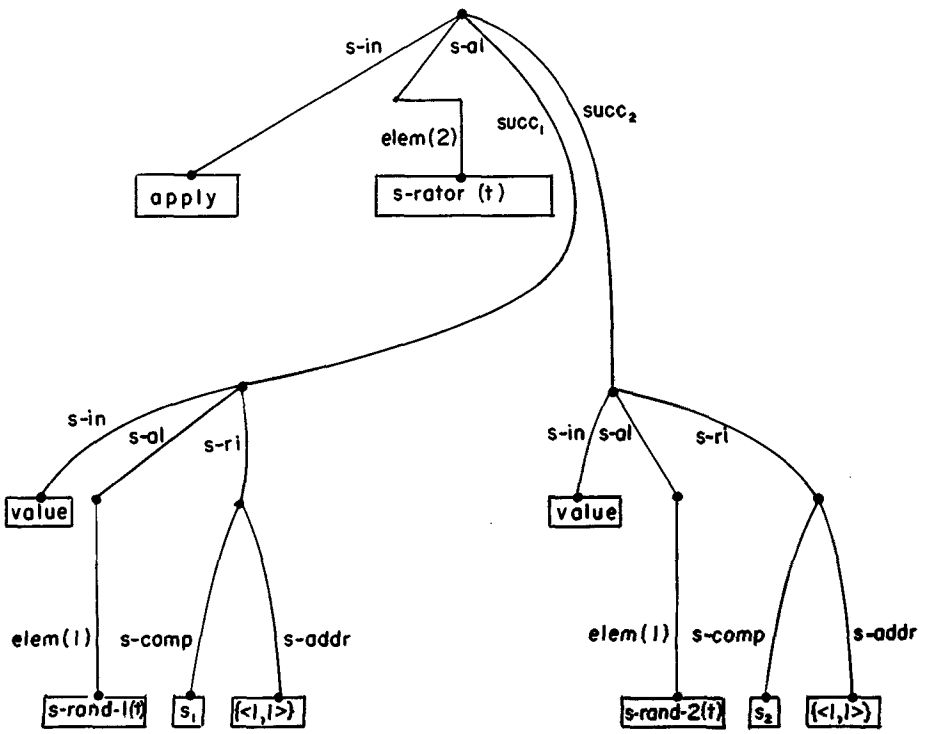
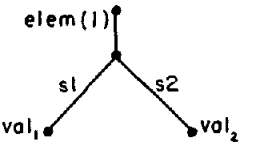


FIG. A4. The composite object for the control tree of Figure A3.



where val_1 is the value returned by "**value** (s-rand-1(t))" and val_2 is the value returned by "**value** (s-rand-2(t))."

FIG. A5. Structured value of the argument list.

machine has been specified as an abstract syntactic structure, the instruction execution cycle may be defined in VDL in terms of transformations of the control tree and of other components of the state. The details of the VDL execution algorithm are given in the following section.

A.2 The Instruction Execution Cycle of VDL

In VDL, each of the instructions associated with terminal vertices of the control tree of a state ξ may give rise to a different next state. The set of next states associated with a given state ξ will be denoted by $\Lambda(\xi)$. The operator Λ is effectively the next state function of a nondeterministic automaton associated with the information structure model.

The instruction execution cycle of a VDL machine may be defined by defining the function Λ, which may then be defined in terms of the set of state transition functions associated with terminal vertices of the control tree. Let $tn(ct)$ be the set of all selector paths $\chi \in SUCC^*$ associated with terminal vertices of the control tree ct, and let $\psi(\xi, \chi)$ be the state transition that results from executing the instruction at the terminal vertex χ of the state ξ. The next state function $\Lambda(\xi)$ may be defined as follows:

$$\Lambda(\xi) = \{\psi(\xi, \chi) \mid \chi \in tn(\text{s-c}(\xi))\}.$$

The instruction execution cycle may be defined by specifying a selection rule for choosing an element of $\Lambda(\xi)$, together with a state transition function for individual instructions $\psi(\xi, \chi)$. The selection rule is left unspecified in VDL. The state transition function $\psi(\xi, \chi)$ is defined in terms of the substructure associated with the vertex ξ described in the previous section and illustrated in Figure A4.

Each vertex has an instruction component, in, selected by "s-in·χ·s-c(ξ)"; an argument list, al, selected by "s-al·χ·s-c(ξ)"; and a return information component, ri, selected by "s-ri·χ·s-c(ξ)." The argument list for an instruction with n_{in} arguments has n_{in} components selected by elem(1)(al), $\cdots$, elem(n_{in})(al), respectively.

The state transition function $\psi(\xi, \chi)$ may be described in terms of: a function ϕ_{in}, which depends on the n_{in} arguments of the instruction; the state ξ with the current instruction vertex deleted; the selector χ; and the return information component "ri = s-ri·χ·s-c(ξ)." Thus,

$$\psi(\xi, \chi) = \phi_{in}(\text{elem}(1)(\text{al}), \cdots, \text{elem}(n_{in})(\text{al}), \delta(\xi; \chi\cdot\text{s-c}), \chi, \text{ri}).^*$$

The state transition determined by ϕ_{in} depends on the specific instruction at the vertex χ of the control tree. Let the instruction at the vertex χ have the following specification:

$$\begin{aligned}&\textbf{inst}(x_1, x_2, \cdots, x_n) =\\ &\quad p_1(x_1, x_2, \cdots, x_n, \xi) \rightarrow a_1(x_1, x_2, \cdots, x_n, \xi)\\ &\quad \cdots\\ &\quad p_m(x_1, x_2, \cdots, x_n, \xi) \rightarrow a_m(x_1, x_2, \cdots, x_n, \xi)\end{aligned}$$

where $p_1, \cdots, p_m$ are predicates and $a_1, \cdots, a_m$ are actions associated with the respective predicates. The state transition associated with the above instruction may be defined as follows:

$$\begin{aligned}&\phi_{in}(x_1, x_2, \cdots, x_n, \xi, \chi, \text{ri}) =\\ &\quad p_1(x_1, x_2, \cdots, x_n, \xi) \rightarrow a_1^*(x_1, x_2, \cdots, x_n, \xi, \chi, \text{ri})\\ &\quad \cdots\\ &\quad p_m(x_1, x_2, \cdots, x_n, \xi) \rightarrow a_m^*(x_1, x_2, \cdots, x_n, \xi, \chi, \text{ri})\end{aligned}$$

where for $i = 1, \cdots, m$, the state transition function a_i^* is determined by the action a_i specified by the programmer.

The state transitions that result from the execution of VDL instructions may be defined by specifying the state transitions a_i^* corresponding to all actions a_i specified by the programmer. First, we shall consider the case in which a_i is a macro instruction, and then the case in which a_i is a value-returning instruction.

When a_i is a macro-instruction, then the χ-component of the control tree is replaced by the composite object a_i, obtained from a_i by the rules given in the previous section:

$$\xi_{new} = a_i^* = \mu(\xi; \langle\chi\cdot\text{s-c}:a_i\rangle).$$

Now, consider the case in which a_i is a value-returning alternative of the following form.

$$\text{PASS} \leftarrow e_0(x_1, x_2, \cdots, x_n, \xi)$$

* The "deletion operator" $\delta(t, \chi)$ deletes the χ-component of the object t, and is an abbreviation for "$\mu(t; \langle\chi:\Omega\rangle)$."

$$\text{s-sc}_1 \leftarrow e_1(x_1, x_2, \cdots, x_n, \xi)$$
$$\cdots$$
$$\text{s-sc}_r \leftarrow e_r(x_1, x_2, \cdots, x_n, \xi)$$

The state transition a_i* involves evaluation of e_j for $j = 0, 1, \cdots, r$; substitution of val(e_0) in all return addresses specified by ri; and substitution of val(e_j) for each component s-sc$_j$. If new$_{ct}$ represents the new control tree after the substitution of val(e_0) in return addresses, then the new state after execution of a value-returning instruction may be defined as follows:

$$\xi_{\text{new}} = a_i* = \mu(\xi; \langle\text{s-c:new}_{ct}\rangle, \langle\text{s-sc:val}(e_1)\rangle, \cdots, \langle\text{s-sc}_r\text{:val}(e_r)\rangle).$$

In order to complete the specification of the state transformation on execution of a value-returning instruction, it remains to specify the form of the new control tree, new$_{ct}$. For this purpose it is convenient to introduce a function "pred," which operates on composite selectors $\chi = s_1 \cdot s_2 \cdots s_k$ and removes the rightmost selector. Thus, pred$(s_1 \cdot s_2 \cdots s_k) = s_2 \cdots s_k$.

The function pred^i will be defined as the i-fold composition of pred. Thus, pred$^i(s_1 \cdot s_2 \cdots s_k) = s_{i+1} \cdots s_k$.

If $\langle i, j\rangle$ is a relative tree address selected by s-addr(ri) (see Figure A4), it specifies that the value of e_0 is to be returned to the jth argument of pred$^i(\chi)$. The control tree position to which val(e_0) is to be returned is defined by the following selector path:

$$\text{s-comp(ri)} \cdot \text{elem}(j) \cdot \text{s-al} \cdot \text{pred}^i(\chi) \cdot \text{s-c}(\xi).$$

The above sequence of selection steps will be explained by discussing successive selection steps individually:

1) "s-c(ξ)" selects the control tree of ξ;
2) "$\chi \cdot$s-c(ξ)" points to the deleted current instruction vertex;
3) "pred$^i(\chi) \cdot$s-c(ξ)" selects the ith predecessor of the current instruction vertex [note that pred^i can be regarded as a function that chops off the final i selectors of χ *before* it is applied to s-c(ξ)];
4) "s-al$\cdot$pred$^i(\chi) \cdot$s-c(ξ)" selects the argument list of the ith predecessor of χ;
5) "elem$(j) \cdot$s-al$\cdot$pred$^i(\chi) \cdot$s-c(ξ)" selects the jth element of the argument list of the ith predecessor of χ; and
6) "s-comp(ri)$\cdot$elem$(j) \cdot$s-al$\cdot$pred$^i(\chi) \cdot$s-c(ξ)": the value of s-comp(ri) is a selector that specifies the component of the jth element of the argument list of the ith predecessor into which the value is to be returned; if s-comp(ri) is the identity selector, I, then the returned value is the complete argument.

Since there may be more than one return address $\langle i, j\rangle$ in the return information, ri, the transformation of the control tree due to substitution of return information val(e_0) into predecessor vertices of the current instruction may be defined as follows:

$$\text{new}_{ct} = \mu(\delta(\text{s-c}(\xi); \chi); \{\langle\text{s-comp(ri)} \cdot \text{elem}(j) \cdot \text{s-al} \cdot \text{pred}^i(\chi) : e_0\rangle \mid \langle i, j\rangle \in \text{s-addr(ri)}\}).$$

The complete state new$_\xi$ which results from a value-returning instruction that passes a value val(e_0) and replaces immediate state components s-sc$_1$, s-sc$_2$, $\cdots$, s-sc$_n$ by the value of $e_1, e_2, \cdots, e_n$, is now defined in terms of the new control tree, new$_{ct}$, as follows:

$$\text{new}_\xi = \mu(\xi; \langle\text{s-c:new}_{ct}\rangle, \langle\text{s-sc}_1\text{:}e_1\rangle, \cdots, \langle\text{s-sc}_n\text{:}e_n\rangle).$$

The above description indicates that the instruction execution cycle of a VDL interpreter can be defined in VDL. A complete definition would require a specification of predicates p_i of a VDL instruction by VDL predicates, a test of whether an action a_i is a value-returning or macro action, and a specification such as that above of the information structure transformations associated with value-returning and macro actions.

BIBLIOGRAPHY

B1. Beech, D. "A structural view of PL/I." *Computing Surveys* 2, 1 (March 1970), 33–64.

B2. Berry, D. M. "Block structure: retention or deletion?" In *Proc. 3rd Annual ACM Symposium on Theory of Computing*, ACM, New York, 1971, 86–100.

B3. Berry, D. M. "Definition of the contour model in the Vienna Definition Language." TR 71-40, Center for Computer and Information Sciences, Brown University, Providence, R. I., April 1971.

B4. Brooker, R. A.; and D. Morris. "A general translation program for phrase structure languages." *J. ACM* 9, 1 (Jan. 1962), 1–10.

D1. Dennis, J. B.; and S. Patil, "Computation structures." Notes for MIT course 6.232, Sept. 1970.

F1. Feldman, J. A. "A formal semantics for computer languages and its application in a compiler-compiler." *Comm. ACM* 9, 1 (Jan. 1966), 3–9.

F2. Floyd, R. W. "Assigning meanings to programs." In *Proc. Symposium on Appl. Math.*, Vol. 19, Amer. Math. Soc., 1967.

H1. Henhapl, W.; and C. B. Jones. "The block structure concept and some possible imple-

mentations with proofs of equivalence." TR 25.104, IBM Lab. Vienna, April 1970.

H2. HOARE, C. A. R. "Proof of a program: FIND." *Comm. ACM* **14**, 1 (Jan. 1971), 39–45.

H3. HOPCROFT, J.; AND J. ULLMAN. *Formal languages and their relation to automata.* Addison-Wesley Publ. Co., Reading, Mass., 1969.

I1. IRONS, E. T. "A syntax directed compiler for ALGOL 60." *Comm. ACM* **4**, 1 (Jan. 1961), 51–55.

J1. JOHNSTON, J. B. "The contour model of block structured processes." In *Proc. Symposium on Data Structures in Programming Languages*, J. T. Tou & P. Wegner (Eds.), *SIGPLAN Notices* **6**, 2 (Feb. 1971), 55–82.

K1. KLEENE, S. C. *Introduction to mathematics.* Van Nostrand, New York, 1952.

K2. KNUTH, D. E. "The semantics of context-free languages." *Mathematical System Theory* **2**, 2 (1968).

L1. LANDIN, P. J. "The mechanical evaluation of expressions." *Computer J.* **6**, 4 (Jan. 1964), 308–320.

L2. LAUER, P. "Formal definition of ALGOL 60." TR 25.088, IBM Lab. Vienna, Dec. 1968.

L3. LEWIS, P. M.; AND R. E. STEARNS. "Syntax-directed transduction." *J. ACM* **15**, 3 (July 1968), 465–488.

L4. LEWIS, P. M.; AND R. E. STEARNS. "Property grammars and table machines." *Information and Control* **14**, 6 (June 1969), 524–549.

L5. LONDON, R. L. "Proving programs correct: some techniques and examples." *BIT* **10**, 2 (1970), 168–182.

L6. LUCAS, P. "Two constructive realizations of the block concept and their equivalence." TR 25.085, IBM Lab. Vienna, Dec. 1968.

L7. LUCAS, P.; ET AL. "Method and notation for the formal definition of programming languages." TR 25.087, IBM Lab. Vienna, 1968.

L8. LUCAS, P.; AND K. WALK. "On the formal description of PL/I." *Annual Reviews of Automatic Programming* **6**, 3 (1969).

L9. LUCKHAM, D. C.; D. M. R. PARK; AND M. S. PATERSON. "On formalized computer programs." *J. Computer and System Sciences* **4**, 3 (June 1970), 220–249.

M1. MANNA, Z. "Properties of programs and the first-order predicate calculus." *J. ACM* **16**, 2 (April 1969), 244–255.

M2. MCCARTHY, J. "Towards a mathematical science of computation." In *Proc. IFIP Cong. 1962*, North-Holland Publ. Co., Amsterdam, 1963.

M3. MCCARTHY, J. "A basis for a mathematical science of computation." In *Formal programming languages*, Braffort & Hirschberg (Eds.), North-Holland Publ. Co., Amsterdam, 1963.

M4. MCCARTHY, J. "A formal description of a subset of ALGOL." In *Formal language description languages for computer programming*, T. B. Steel, Jr. (Ed.), North-Holland Publ. Co., Amsterdam, 1966, 1–12.

M5. MCCARTHY, J.; ET AL. *The LISP 1.5 programming manual.* MIT Press, Cambridge, Mass., 1965.

M6. MCCARTHY, J.; AND J. PAINTER. "Correctness of a compiler for arithmetic expressions." In *Proc. Symposium on Appl. Math.*, Vol. 19, Amer. Math. Soc., 1967.

M7. MCGOWAN, C. "Correctness results for lambda calculus interpreters." PhD Thesis, Cornell University, 1971; available as TR 71-34, Center for Computer and Information Sciences, Brown University, Providence, R. I., Feb. 1971.

N1. NAUR, P. (ED.) "Revised report on the algorithmic language ALGOL 60." *Comm. ACM* **6**, 1 (Jan. 1963), 1–17.

P1. PATERSON, M. S. "Equivalence problems in a model of computation." PhD Thesis, Cambridge University, 1967; available as MIT Artificial Intelligence Laboratory Memo. No. 1, Cambridge, Mass., Nov. 1970.

R1. ROSENBERG, A. L. "Addressable data graphs." In *Proc. 3rd Annual ACM Symposium on Theory of Computing*, ACM, New York, 1971, 138–150.

S1. SHEPHERDSON, J. C.; AND H. E. STURGIS. "Computability of recursive functions." *J. ACM* **10**, 2 (April 1963), 217–255.

S2. STANDISH, T. "Data structures: an axiomatic approach." Unpublished report, Feb. 1971.

V1. VAN WIJNGAARDEN; ET AL. "Report on the algorithmic language ALGOL 68." *Numerische Mathematik* **14**, 2 (1969), 84–218.

W1. WALK, K.; ET AL. "Abstract syntax and interpretation of PL/I, Version III." TR 25.098, IBM Lab. Vienna, April 1969.

W2. WEGNER, P. *Programming languages, information structures and machine organization.* McGraw-Hill Book Co., New York, 1968.

W3. WEGNER, P. "Three computer cultures." In *Advances in computers*, Vol. 10, F. L. Alt & M. Rubinoff (Eds.), Academic Press, New York, 1970, 7–78.

W4. WEGNER, P. "Programming language semantics." In *Courant Inst. Symposium on Formal Semantics*, Prentice-Hall, Inc., Englewood Cliffs, N. J., 1971.

W5. WEGNER, P. "Data structure models in programming languages." In *Proc. Symposium on Data Structures in Programming Languages*, J. T. Tou & P. Wegner (Eds.), *SIGPLAN Notices* **6**, 2 (Feb. 1971), 1–54.

W6. WEGNER, P. "Operational semantics of programming languages." In *Proc. ACM Conf. on Proving Assertions about Programs*, a joint publication of SIGPLAN and SIGACT, *SIGPLAN Notices* **7**, 1 (Jan. 1972), 128–141.

W7. WIRTH, N.; AND H. WEBER. "EULER: a generalization of ALGOL and its formal definition: part I." *Comm. ACM* **9**, 1 (Jan. 1966), 13–23; "Part II." *Comm. ACM* **9**, 2 (Feb. 1966), 89–99.