A hierarchical graph model of the semantics of programs

by TERRENCE W. PRATT

The University of Texas
Austin, Texas

INTRODUCTION

The problem of developing an adequate formal model for the semantics of programming languages has been under intensive study in recent years. Unlike the area of syntax specification, where adequate models have existed for some time, the area of semantic specification is still in the formative stages. Development of formal semantic models has proceeded along two main lines, lambda-calculus models (e.g., Landin, Strachey) and directed graph models (e.g., Narasimhan, Floyd). This paper describes a model for the semantics of programs based on hierarchies of directed graphs.

A formal model or theory of the semantics of programming languages must provide descriptions on a number of different levels, much as the theory of context-free grammars provides descriptions of the syntax of programming languages on a number of different levels. At the most general level a semantic theory provides a framework for describing a class of programming languages and investigating the mathematical properties of their formal representations in the model. At this general level the context-free grammar model of syntax has been particularly successful, giving rise to an extensive mathematical theory as well as providing characterizations of specialized syntactically-similar classes of languages. One would hope that adequate models of semantics would lead to the same sort of mathematical development and to the classification of semantically-similar programming languages.

At a more specific level, a semantic model must provide descriptions of the semantics of particular programming languages, much as a particular context-free grammar may be used to describe the syntax of a particular programming language. One would expect the formal specification of the semantics of a programming language to allow conciseness and unambiguity in definition and also to lead to definition of new languages which are more "regular" semantically.

At a still more specific level, a semantic model must provide descriptions of particular programs, and these descriptions must be constructable from the representation of the program (the syntax) in a straightforward manner. Thus it should be possible to construct a "compiler" which will translate a program in its written representation into its equivalent in the semantic model. From this representation the formal properties of the model may be used to guide further processing, by producing, for example, a more efficient program which is semantically equivalent or by translating the program into another representation, such as a semantically equivalent program in another language.

At the level of complete specification, a semantic model must provide a representation of the data used by a program and allow execution of the program on the particular data. Thus it should be possible to construct an "interpreter" which will execute a program-data pair in its representation in the model.

The importance of the development of adequate models of semantics stems from the probable gains to be realized by their use, both in the area of definition of programming languages, where the formal definition of semantics should lead to semantically unambiguous as well as more easily intelligible languages, and in the area of processor construction, where it is likely that a better understanding of the semantic features of languages will lead to more powerful and efficient processing techniques for those features.

The model for semantics described in the following sections is based on the use of hierarchies of directed graphs to represent both programs and data. The hierarchical graph or H-graph concept which is basic to the model is defined in the next section, along with the
concepts of “level” and “path” which serve as important structural concepts in the application of H-graphs to the description of programming languages. In a later section the basic semantic model is introduced by structuring further the “atomic units” which lie at the lowest level of the hierarchy in H-graphs. Basic programming concepts such as “program,” “data,” and “execution of a program” are defined also. In the last section examples are given to illustrate informally how a number of semantic features of actual programming languages may be represented in a natural way in the model. Included are complete models of the semantics of particular Turing machine, Lisp, and Fortran programs. Finally, a concluding section attempts to evaluate the model and view possible applications and extensions.

Hierarchical graphs (H-graphs)

In this section a generalization of the mathematical concept of “finite directed graph” is presented which forms the basis for the model of semantics presented in the next section.

Basically, a directed graph (sometimes called a “network”) is composed of a finite set of “nodes,” and a finite set of “edges” connecting pairs of nodes in a “one-way” manner. Commonly, nodes are represented as circles or boxes and edges as arrows connecting nodes. Examples of the use of directed graphs in computer science include flow charts, automata transition diagrams, PERT networks, and representations of list structures and trees.

Three extensions of the basic concept of directed graph are of interest here. The first two lead to a definition of “extended directed graph,” and the third to a definition of “hierarchical directed graph.”

Extended directed graphs

Informally, an extended directed graph is a directed graph with a designated node (called the “entry point”) and with the edges leaving each node uniquely labeled. Formally:

Defn: An extended directed graph is an ordered quadruple (N, L, S, E) where N is a finite non-empty set (of nodes),

L is a finite set (of labels),

S is an element of N (called the entry point), and

E is a partial function from N x L into N, defining the edges. If E(n, p) = q, then there is said to be an edge from node n to node q with label p. Note that there may not be two edges leaving a node with the same label, but two edges leaving a node with different labels may end at the same node; thus “parallel” edges are allowed. Ignoring the contents of the nodes, every flow chart may be considered as an extended directed graph.

H-graphs

A hierarchy is introduced into a set of extended directed graphs (hereafter called simply “graphs”) in the following manner. A universe U of atomic units (distinct symbols) is assumed. A hierarchical graph or H-graph over U is (informally) a graph in which the nodes are considered to be “containers,” each of whose contents is either an atomic unit from U or an H-graph over U. Thus in flow-chart terms, the analogue of an H-graph is a flow chart in which each box contains either an “atom” or another flow chart, whose boxes may in turn contain atoms or flow charts, and so forth, to any depth. Formally:

Defn: If U is a set (of atomic units or atoms), then an H-graph over U is an ordered pair (N, V) where:

1. N is a finite set (of nodes or containers).
2. V (the contents or value function) is a function mapping N into U U {x | x is a graph with nodes from N and labels from U}. If c E N, then V(c) is called the contents or value of c.

Levels

The hierarchical structure of an H-graph is brought out by considering the levels of its nodes.

Defn: The level of a node c is defined recursively as:

a. 1 if V(c) E U (i.e., if c contains an atomic unit).
b. n if V(c) is a graph (N, L, S, E), the level of each node in N is in the set {1, 2, ..., n - 1}, and at least one node in N has level n - 1.

If a node is not of level n for any n, then it is termed a recursive node.

Defn: The node set of a node c in an H-graph is defined to be:

a. if c has level 1, then Λ, the empty set, or
b. if c contains a graph (N, L, S, E), then N U {x | x is a graph with nodes from x}. Thus the node set of a node is composed of all the nodes “reachable” in the hierarchy starting from that node.
Defn: An H-graph \((N, V)\) is recursive if \(N\) contains a recursive node.

Defn: The atom set of a node \(c\) is the set of values of all level 1 nodes of the node set of \(c\). Thus the atom set of a node is the set of atomic units "reachable" in the hierarchy starting from that node.

Paths

Along with the concept of level, the idea of a "path" in an H-graph is important in the development of the model of the next section. The concept of path will be defined first for graphs and then extended to H-graphs. Informally, a path in a graph from one node to another is just a set of edges which may be traversed in going from the first node to the second, passing from node to node along connecting edges. Formally,

Defn: If \(G = (N, L, S, E)\) is a graph, then a path from node \(x\) to node \(y\) of \(G\) is an ordered sequence \((x, k_0, k_1, \ldots, k_m)\) where each \(k_i\) is a label such that \(E(x, k_0) = n_1, E(n_1, k_1) = n_2, \ldots, \) and \(E(n_m, k_m) = y\).

In general, certain nodes of a graph will have no edges leaving them. Of particular importance here are the paths which begin at the entry point of a graph and terminate at such nodes.

Defn: A path beginning at node \(x\) is a path from node \(x\) to a node \(y\) such that \(y\) has no edges leaving it, i.e., such that \(E(y, k)\) is not defined for any \(k \in L\).

Defn: A path in a graph \(G\) is any path beginning at \(s\), where \(s\) is the entry point of \(G\).

Extending this concept to H-graphs, if \(C\) is a node of an H-graph, then (informally) a path through \(C\) is composed by taking a path through the graph contained in \(C\), a path through each of the graphs contained in the nodes encountered in that path, and so forth down the hierarchy until level 1 nodes are reached. Formally,

Defn: If \(G\) is an H-graph and \(C\) is a node of \(G\), then a path \(P\), through node \(C\), is represented by:

1. if \(C\) contains an atomic unit \(A\), then \((C, A)\)
2. if \(C\) contains a graph, then \((C, P)\) where \(P = (P_s, e_0, P_{e_0}, \ldots, e_m, P_{e_m})\) and \((s, e_0, \ldots, e_m)\) is a path in \(G\) passing through nodes \(s, n_0, \ldots, n_m\) and each \(P_{e_i}\) is a path through node \(n_i\).

Defn: The atom sequence of a path is simply the sequence of atomic units encountered in following the path. In the representation given above, the atom sequence is simply the sequence of atomic units (not including edge labels) obtained in scanning the representation of a path from left-to-right ignoring everything except atomic units.

For example, Figure 1 represents an H-graph, where \(C1, C2, \ldots, C7\) are nodes each of whose contents is given by the contents of the correspondingly labeled box. One path through node \(C7\) is represented by: \((C7, ((C6, ((C2, 27), 0, (C1, ABC))), 0, (C5, ((C1, ABC), 0, (C2, 27), 0, (C3, QED), 1, (C4, 10))), 0, (C1, ABC)))\).

The atom sequence for this path is: \((27, ABC, ABC, 27, QED, 10, ABC)\).

Further development of the theory of H-graphs is outside the scope of this paper. This brief introduction will suffice for the purposes here.

The basic semantic model

On the basis of the H-graph concept, an elementary model of semantics is developed in this section, including definitions of program, data, and execution of a program.

Program and data H-graphs

The semantic model is based on H-graphs with a more highly structured universe of atomic units. For
the semantic model, the universe $U$ of atomic units is assumed to have the following structure:

$U = D \cup \phi$ (D and $\phi$ not necessarily disjoint) where $D$ is a set of data units and $\phi$ is a set of operator instances.

$D = D_1 \cup D_2 \cup \ldots \cup D_n$ (the $D_i$'s not necessarily disjoint) where each $D_i$ is a set of data units forming a data type class.

$(D_i = \text{e.g., integers, reals, bits, characters, strings})$

Similarly:

$\phi = \phi_1 \cup \phi_2 \cup \ldots \cup \phi_m$ (the $\phi_i$'s not necessarily disjoint) where each $\phi_i$ is a set of operator instances forming an operator type class.

$\phi_i = \text{e.g., “add,” “multiply,” “popup,” “differentiate!”}$

Definitions:

1. An atomic unit $a \in U$ is said to be of type $T$ (where $T$ is a data or operator type class) if $a \in T$. Since type classes are not necessarily disjoint, the type of an atomic unit is not necessarily unique.

2. A node $n$ of an H-graph over $U$ is said to be of type $T$ if the atom set of $n$ is a subset of type class $T$.

3. An H-graph is of type $T$ if each of its nodes is of type $T$.

4. A data node is a node whose atom set is a subset of $D$.

5. A data H-graph is an H-graph whose nodes are all data nodes.

6. A program node is a node whose atom set is a subset of $\phi$.

7. A program H-graph is an H-graph whose nodes are all program nodes.

Operator instances

Although the concept of data type class is familiar from actual programming languages, that of operator instance is not as common.

Defn: An operator instance $Q$ is an ordered triple $Q = (I, 0, f)$, usually written $f(I \cup 0)$, where $I$ is a finite set of (input) nodes, $0$ is a finite set of (output) nodes, and $f$ (the operator) is a function from $X$ into $Y$, where $X$ and $Y$ are sets of H-graphs, each H-graph in $X$ and $Y$ containing the nodes in $I$ and $0$.

A typical example would be the operator instance 

"([A, B], [C], add)" or "add (A, B; C)"

with input nodes $A$ and $B$, output node $C$, and operator "add" which maps any H-graph containing the nodes $A$, $B$, and $C$, and in which $A$ and $B$ have numerical values, into the H-graph identical to the original except with $C$ containing the sum of the values of $A$ and $B$.

Execution of an operator instance is a primitive concept by which is meant the application of the operator to its arguments found in its input nodes, producing values found in its output nodes. When an operator instance is executed, it is understood that the H-graphs defined by its input nodes completely define its arguments, in the sense that given the same input H-graphs, the same values will be produced as output. Similarly, the effect of an operator is assumed to be localized in the H-graphs produced as the values of its output nodes, in the sense that if the value of a node is changed by the operator then that node must be in the node set of one of the output nodes. Thus the set of input nodes of an operator instance in a sense specifies the maximum set of data structures which may influence the effect of that operator instance, and similarly, the output nodes specify the maximum extent of the effects produced.

Note that only the input and output nodes of an operator instance are fixed, not the contents of these nodes, so that different executions of the same operator instance will, in general, produce different results, even though the input and output nodes are the same. Only if the H-graphs defined by the input nodes are identical will execution necessarily produce the same results in the output nodes.

Execution of a program H-graph

Any program node of an H-graph defines a set of possible execution sequences of operator instances as follows:

Defn: If $c$ is a program node, then the sequence $p_1, p_2, \ldots, p_n$ of operator instances is an execution sequence of $c$ if $(p_1, p_2, \ldots, p_n)$ is the atom sequence of some path through $c$.

Every path through a program node defines a possible execution sequence of operator instances. Execution of a program node involves the choice of a path through that node and execution of the operator instances in order from the execution sequence defined by that path. Corresponding to actual execution of programs in a programming language, the process of choosing a path may be represented as a process which in a sense actually traces out a path step by step, executing operator instances as they are encountered in the path, and choosing the next step of the path as a dynamic function of the results of execution of the previous operator instances encountered. Thus the
choice of execution sequence is determined by the "data," that is by the H-graphs in the input node set of the operator instance being executed. The process of execution of a program node $c$ of an H-graph $(N, V)$ may be defined precisely as follows by use of a designated level 1 node $P$ of type "label:"

1. If $c$ is a level 1 node, then execute the operator instance contained in $c$.
2. If $c$ is not a level 1 node and contains the graph $(N, L, S, E)$, then:
   a. Set the current node $= S$.
   b. Execute the current node (saving the name of the node being executed in a stack until execution is complete).
   c. If there is no edge leaving the current node, then stop (returning to the next higher level to continue execution if necessary).
   d. If there is exactly one edge (labeled $k$) leaving the current node, then set current node $= E$(current node, $k$) and go to (b).
   e. If there is more than one edge leaving the current node, then set the current node $= E$(current node, $V(P)$) and go to (b).

Note that at a branch in a program graph the choice of which branch to follow is determined by the label contained in the designated node $P$. Since $P$ may be one of the output nodes of an operator instance, execution of an operator instance may change the value of $P$, and thus the "flow of control" is determined dynamically by the operator instances and the "data." Clearly, a path through node $c$ is determined by this process if the process terminates.

**Correspondence between the model and actual programming languages**

The general model of the previous section serves as a framework for the description, comparison, and classification of the semantics of different programming languages. A model of the semantics of a particular programming language is formed by specification of a particular universe of atomic units together with a set of restrictions or constraints on the types of data and program H-graphs which may be constructed on this universe. A model of a program or a data structure in the language then is a program or data H-graph on the given universe which satisfies the constraints of the language. The classification and comparison of the semantics of languages is based on the classification and comparison of the properties of the universes of atomic units and of the properties of the constraints on H-graphs. Although detailed descriptions of actual programming languages are outside the scope of this paper, an attempt is made in this section to develop in a general way a correspondence between features of actual programming languages and properties of the model.

Particular features of actual programming languages may commonly be represented in more than one way in the model. The examples given below indicate only one possible way that particular constructs may be represented, without precluding the possibility that other representations may be possible and even desirable in certain models of an entire language.

In the examples, nodes are represented by ovals or polygons and edges by arrows. All nodes and edges are labeled. The contents of a node will ordinarily be written inside the oval or rectangle representing it. The entry point node of a graph is indicated by an $*$ next to the node. Thus, for example, Figure 2 represents a node $C$ whose value is a graph on nodes $C_1$, $C_2$, and $C_3$, with entry point $C_1$ and $E(C_1, k) = C_2$, $E(C_2, k) = C_3$, $E(C_3, k) = C_1$. The nodes $C_1$, $C_2$, and $C_3$ have as values the atoms $A$, $B$, and $C$, respectively.

In cases where node names and/or edge labels are not significant, they will ordinarily be omitted. Thus the same graph might be written alternatively as in Figure 3. Operator instances will be represented in the form:

operator name ($i_1, \ldots, i_n; 0_1, 0_2, \ldots, 0_m$)

![Figure 2](image1)

![Figure 3](image2)
where each $i_k$ is the name of an input node and each $o_k$ is the name of an output node.

There complete models of programs and data are given in this section, for Turing machines, LISP, and Fortran. Although no attempt is made to provide complete descriptions of the languages, the models of the particular programs used are constructed so as to indicate how a general model might be constructed.

**Turing machine semantics.** Consider first the representation of the semantics of a particular Turing machine. Intuitively, a Turing machine is very simple semantically. Both the program (the functional matrix) and the data (the tape) are of simple structure. This makes a model of the semantics of Turing machines a useful test case for a semantic theory. The general class of Turing machines that will be modeled have a single tape and read head, and at each move they will (1) read the symbol in the square being scanned, (2) change state, (3) write a symbol, and (4) move left or right one square.

**a. Atomic units.** The universe of atomic units $U$ for Turing machines over a particular alphabet $A$ is composed of:

$$U = A' \cup \phi$$

where $A'$ and $\phi$ are disjoint, $A' = A \cup \{\text{left, right}\}$, $A$ is the alphabet of the Turing machine, $\{\text{left, right}\}$ are labels, and $\phi = \text{READ} \cup \text{WRITE} \cup \text{MOVELEFT} \cup \text{MOVERIGHT} \cup \text{NO-OP}$ where REAL, WRITE, MOVELEFT, MOVERIGHT, and NO-OP are the names of disjoint operator type classes defined below.

**b. Data structures.**

**Level 1 nodes.** Except for the special node $P$ and the nodes containing the alphabet symbols, all level 1 nodes correspond to tape squares containing a symbol from the alphabet $A$. $P$ is the node mentioned in the execution algorithm for program $H$-graphs, and for each alphabet symbol a node is required containing that symbol (for the WRITE operator).

**Level 2 nodes.** Only two nodes are needed, TAPE and HEAD. TAPE contains a two-way list (of level 1 nodes) with edges labeled "left" and "right." HEAD contains a single level 1 node which corresponds to the tape square being scanned.

**c. Program structures.**

**Level 1 nodes.** Contain single operator instances.

**Level 2 nodes.** Only one node $\text{PROG}$ is needed, which contains a graph representing the functional matrix of the Turing machine.

**d. Operator type classes.** See Table I. ($H = \text{HEAD}$, $T = \text{TAPE}$, $V$ is the function which, given a node, returns the value of that node.)

**e. Example.** For the 2-state 3-symbol Turing machine given in Figure 4 an $H$-graph representation is given in Figure 5.

Execution of the program $H$-graph $\text{PROG}$ simulates the operation of the Turing machine. Execution of $\text{PROG}$ involves simply following a path from the entry point (node $P_9$) of the graph in $\text{PROG}$ to the (unique) node $P_5$ of the graph which has no exiting edges. This path is chosen according to the algorithm of the

---

**Table I—Turing machine operator instances**

<table>
<thead>
<tr>
<th>Class</th>
<th>Form of Operator Instances</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ</td>
<td>READ($H; P$)</td>
<td>Transfers $V(V(H))$ to $P$ (i.e., copies the symbol being scanned into $P$)</td>
</tr>
<tr>
<td>WRITE</td>
<td>WRITE($\alpha, H; H$)</td>
<td>Sets the value of the node $V(H) = V(\alpha)$</td>
</tr>
<tr>
<td>MOVELEFT</td>
<td>MOVELEFT($H, T; H, T$)</td>
<td>Let $\alpha = V(H)$, set $V(H) = E(\alpha, \text{left})$ where $E$ is the edge function of the graph in $T$. If $E(\alpha, \text{left})$ is not defined, create a new node with a unique name $\beta$. Set $V(H) = \beta$. Define $E(\alpha, \text{left}) = \beta$ and $E(\beta, \text{right}) = \alpha$ and set $V(\beta) = $ (empty tape square symbol of $A$)</td>
</tr>
<tr>
<td>MOVERIGHT</td>
<td>MOVERIGHT($H, T; H, T$)</td>
<td>Same definition as MOVELEFT, replacing &quot;left&quot; by &quot;right&quot; and &quot;right&quot; by &quot;left.&quot;</td>
</tr>
<tr>
<td>NO-OP</td>
<td>NO-OP($\epsilon$)</td>
<td>The identity operation (does nothing)</td>
</tr>
</tbody>
</table>
Hierarchical Graph Model of Semantics of Programs

Matrix

<table>
<thead>
<tr>
<th>Symbol</th>
<th>0</th>
<th>1</th>
<th>#</th>
</tr>
</thead>
<tbody>
<tr>
<td>State</td>
<td>S₀</td>
<td>S₁</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0₄,₀₄</td>
<td>0₄,₀₄</td>
<td>0₄,₀₄</td>
</tr>
<tr>
<td></td>
<td>L₄,₁₄</td>
<td>L₄,₁₄</td>
<td>L₄,₁₄</td>
</tr>
</tbody>
</table>

Input Tape

<table>
<thead>
<tr>
<th></th>
<th>#</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>#</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial head position</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 4—Turing machine initial configuration

preceding section. Thus beginning at P₀, the operator instance in P₀ is executed (corresponding to reading the tape square under the read head and storing the symbol read in node P). Then the edge leaving P₀ is chosen which is labeled with the symbol contained in P (in this case 0). This edge leads to P₀, which becomes the current node. The operator instance in P₀ is executed (writing a 1 on the tape square under the read head). The single edge leaving P₀ is followed to P₃, the operator instance contained in P₃ is executed, etc. Execution continues in this manner until node P₅ is reached.

In the Turing machine representation both program and data have only two levels, giving a hierarchically simple structure. The given representation readily extends to arbitrary Turing machines and initial configurations.

Lisp semantics. As a second example, consider the problem of representing the semantics of the following Lisp function with its argument:

\[
\text{(LABEL (LAST (LAMBDA (X) (COND ((NULL (CDR X)) (CAR X)) \(-(T (LAST (CDR X)))))) ((A (B C) D E))) (LABEL (LAST (LAMBDA (X) (COND ((NULL (CDR X)) (CAR X)) \(-(T (LAST (CDR X)))))) ((A (B C) D E))}}
\]

An H-graph model may be constructed for this function in such a way that extension of the model to the remainder of the Lisp language is possible.

In representing the Lisp function two major problems arise:

1. The sequence of operations specified implicitly through function composition and the order of evaluation conventions of Lisp must be made explicit in the H-graph representation.
2. The pairing of formal parameters and actual parameters, and the transmission of evaluated arguments to the functions using them must be handled by explicit use of versions of the standard Lisp interpreter A-list and pushdown list. This is due to the fact that the H-graph model contains no built-in provision for argument transmission to subprograms.

A representation of the semantics of the above Lisp function may be constructed as follows:

a. Atomic units. Let the universe U of atomic units be:

\[U = A \cup \{T, NIL\} \cup \emptyset\]

where A is the set of Lisp atoms and \(\emptyset = \text{CAR} \cup \text{CDR} \cup \text{NULL} \cup \text{PAIR} \cup \text{VALUE} \cup \text{POPOP} \text{U}
\]

where these data type classes are defined below.

b. Data structures. Lisp list structures are represented as hierarchical data graphs, where sublists are represented by lower levels in the hierarchy. A list of n elements is represented by a graph of n nodes, connected in sequence. If a list element is an atom, its corresponding graph node contains that atom as its value. If a list element is a sublist of m elements, its corresponding graph node contains a graph of m nodes representing the sublist. Thus the list:

\[(A, (B, C, D), ((E, F), G))\]

is represented by the H-graph in Figure 6. To handle argument transmission, certain auxiliary data structures will be necessary:

1. OP, a node which contains a list, used as a stack to hold operands;
2. ALIST, a node which contains a list, used as a stack to hold paired formal parameters and their values.

c. Program structures. Like all Lisp program structures not using PROG, the program graph for the above function is a tree. Since the function contains a recursive function call, the program graph is recursive.

d. Operator type classes. See Table II. (V is the
function which, given a node, returns the value of that node.)
e. *Example.* The Lisp function:

(LABEL(LAST(LAMBDA(X)(COND
((NULL(CDR X)) (CAR X))
(T (LAST (CDR X))))))) ((A (B C) D E))

is represented by the H-graph of Figure 7.

Note that the sequence of operations performed during execution of the node LAST corresponds closely to the sequence executed by an ordinary Lisp interpreter: first the formal parameter, X, and the actual parameter, the list in C1, are paired on the ALIST, then X is evaluated, CDR of its value is taken, and

Figure 5—Turing machine H-graph representation
### Table II—LISP operator instances

<table>
<thead>
<tr>
<th>Class</th>
<th>Name</th>
<th>Operator Instance Form</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAR</td>
<td>CAR(OP; OP)</td>
<td></td>
<td>Let α = entry point of V(OP). Let β = entry point of V(α). Set V(α) = V(β).</td>
</tr>
<tr>
<td>CDR</td>
<td>CDR(OP; OP)</td>
<td></td>
<td>Let α = entry point of V(OP). Let β = entry point of V(α). Set V(α) = a graph obtained from V(α) by setting the entry point to E(β, t) and deleting node β. If E(β, t) is not defined, set V(α) = NIL.</td>
</tr>
<tr>
<td>NULL</td>
<td>NULL(OP; OP, P)</td>
<td></td>
<td>Let α = entry point of V(OP). Set V(α) = V(P) = T if V(α) = NIL and = NIL otherwise.</td>
</tr>
<tr>
<td>PAIR</td>
<td>PAIR(γ, OP; ALIST, OP)</td>
<td></td>
<td>Let α = entry point of V(OP) and β = entry point of V(ALIST). (1) Pushdown V(ALIST), let C = new entry point (i.e., replace the graph in ALIST by *(C β → ... where C is a new node and *(β → ... was the old graph). (2) Set V(C) = the graph *(γ → α. (3) Popup OP (i.e., replace V(OP) = the graph *(α → α' → ... by the graph *(α' → ... .</td>
</tr>
<tr>
<td>VALUE</td>
<td>VALUE(γ, ALIST; OP)</td>
<td></td>
<td>(1) Find first node Q on list V(ALIST) such that γ = entry point of V(Q). Then V(Q) is a graph *(γ → α. Let C be a new node. (2) Pushdown C onto list V(OP) (i.e., replace *(β → ... by *(C → β → ... . (3) Set V(C) = V(α).</td>
</tr>
<tr>
<td>POPUP</td>
<td>POPUP(α; α)</td>
<td></td>
<td>Popup list V(α) (i.e., replace *(β → γ ... by *(γ → ... .</td>
</tr>
</tbody>
</table>

![Figure 6—List structure H-graph representation](image)

NULL of that value is taken. The value of NULL, T or NIL, is then used to control a branch. If T, the execution ends after the CAR of the value of X is taken. If NIL, then the CDR of the value of X is taken and execution descends a level at the recursive node N, which contains a graph of the single node LAST. The stack OP is used throughout to communicate results between operator instances. Execution of the above program H-graph LAST will result in the node OP containing a graph whose entry point node contains the value of the function; thus OP: *(E X N) is the final state of the node OP, where E is the value of the function LAST for the given argument.

**Fortran semantics.** As a final example, consider the problem of representing the semantics of the following Fortran program:

```
PROGRAM EXAMPLE
  DIMENSION M(2,5)
  D0 2 I = 1,5
  M(1, I) = I
  2 M(2, I) = IFACT(I)
  END
  FUNCTION IFACT(N)
    K = 1
```

From the collection of the Computer History Museum (www.computerhistory.org)
As with Lisp, it is necessary to provide explicitly a mechanism for argument transmission to subprograms in the model of Fortran.

The H-graph representation may be constructed as follows:

a. **Atomic units.**

\[ U = I \cup \{-, \neq\} \cup \emptyset \]

where \( I = \) set of integers

\[ \{\neq, \neq\} = \) set of labels

and

\[ \emptyset = \) the union of the operator type classes defined below.

b. **Data structures.** Simple variables are represented as level 1 data nodes. Two-dimensional arrays are represented by a graph (of level 1 nodes) with edges labeled 1 and 2 indicating rows and columns, respectively, as \( M \) in Figure 8.

c. **Program structures.** The program structures correspond roughly to flow charts of the Fortran programs. Subprograms are represented by separate levels in the hierarchy of program graphs.

d. **Operator type classes.** See Table III. \((V = \) function which, given a node, returns the value of that node.\)

e. **Example.** The program may be represented as the H-graph of Figure 8. Note that the loops implied by the two \( D \) statements are made explicit in the H-graph representation, and the operations of array referencing and argument transmission which are implicit in the Fortran program become explicit in the model as the operations \( \text{REF2} \) and \( \text{SETADDR-SET} \), respectively. Execution of the H-graph \( \text{EXAMPLE} \) results in the first row of the array \( M \) being filled with the integers 1-5 and the second row with the integers 1!-5!.

Figure 7—Lisp function H-graph representation
Hierarchical Graph Model of Semantics of Programs

Data:

\[ M: \]

\[ N: 0 \quad N_1: 1 \quad N_2: 2 \quad N_5: 5 \]

\[ I: 0 \quad T: 0 \quad P: 0 \]

\[ ARG1: 0 \quad K: 0 \quad J: 0 \]

Program:

EXAMPLE:

\[ \text{ADD}(N_1, I; I) \]

\[ \text{REF2}(N, N_1, I; T) \]

\[ \text{IASSIGN}(I; T) \]

\[ \text{REF2}(N, N_2, I; T) \]

\[ \text{SETADDR}(I; ARG1) \]

\[ \text{*IFACT} \]

\[ \text{IASSIGN}(ARG1; T) \]

\[ \text{ISEQ}(N_5, I; P) \]

\[ \text{NO-OP()} \]

\[ \text{IFACT:} \]

\[ \text{SET}(ARG1; N) \]

\[ \text{ASSIGN}(N_1; K) \]

\[ \text{ASSIGN}(N_1; J) \]

\[ \text{MULT}(K, J; K) \]

\[ \text{ADD}(N_1, J; J) \]

\[ \text{ISEQ}(J, N; P) \]

\[ \text{ASSIGN}(K; ARG1) \]

Figure 8—Fortran program H-graph representation
Table III—Fortran operator instances

<table>
<thead>
<tr>
<th>Class</th>
<th>Operator Instance Form</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>ADD(α; β; γ)</td>
<td>Sets V(γ) = V(α) + V(β)</td>
</tr>
<tr>
<td>MULTIPLY</td>
<td>MULTIPLY(α; β; γ)</td>
<td>Sets V(γ) = V(α) * V(β)</td>
</tr>
<tr>
<td>ASSIGN</td>
<td>ASSIGN(α; β)</td>
<td>Sets V(β) = V(α)</td>
</tr>
<tr>
<td>IASSIGN</td>
<td>IASSIGN(α; β)</td>
<td>Sets V(V(β)) = V(α) (where V(β) is a single node graph)</td>
</tr>
<tr>
<td>SETADDR</td>
<td>SETADDR(α; β)</td>
<td>Sets V(β) = the single node graph: *α</td>
</tr>
<tr>
<td>SET</td>
<td>SET(α; β)</td>
<td>Sets V(β) = V(α') where α' = entry point node of the graph V(α)</td>
</tr>
<tr>
<td>ISEQ</td>
<td>ISEQ(α; β; P)</td>
<td>Sets V(P) = {</td>
</tr>
<tr>
<td></td>
<td></td>
<td>= if V(α) = V(β)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>≠ if V(α) ≠ V(β)</td>
</tr>
<tr>
<td>REF2</td>
<td>REF2(α, β, γ; δ)</td>
<td>Sets V(δ) = node corresponding to α(β, γ) in 2-dimensional array α</td>
</tr>
<tr>
<td>NO-OP</td>
<td>NO-OP( )</td>
<td>No-operation</td>
</tr>
</tbody>
</table>

CONCLUSION

In this paper the general outline of a hierarchical directed graph model of the semantics of programs has been sketched, and a number of examples of its use in modeling particular features of programming languages have been given. Many interesting questions remain concerning such models. Preliminary indications are that the model provides both mathematical tractability and intuitively appealing characterizations of the semantics of programming languages. It remains to be shown that these indications will remain valid as more extensive development is undertaken. At present, work is proceeding along two main lines. On the one hand, attempts are being made to exploit the formal nature of the model. One would like to prove theorems concerning the model which would allow one to derive properties of particular program-data pairs given their representation in the model. On the other hand, models of various programming languages are being constructed, and the problem of constructing processors based on the model is being investigated. One would like to be able to translate from actual programming languages into the model and vice versa. Given such processors, one could then manipulate the H-graph representation of a particular program according to the formal theory to produce semantically equivalent programs having particularly desirable properties.

Any model of the semantics of programming languages emphasizes in its structure certain semantic features at the expense of others. The model based on hierarchies of directed graphs presented here is no exception. Other models of semantics have tended to emphasize flow of control in programs and argument transmission to subprograms. Landin's lambda-calculus model of ALGOL, for example, emphasizes argument transmission, restricts flow of control to function composition with recursion, and does not consider data structure. The directed-graph model of Narasimhan emphasizes argument transmission and flow of control, representing the latter in a manner somewhat similar to that used here. The model of this paper emphasizes the structure of data and the hierarchical aspects of flow of control. It subordinates argument transmission by incorporating no built-in argument transmission method.

It is clear that the development and study of formal models of the syntax of programming languages has helped greatly to clarify the basic concepts in that area and also has led to the development of better methods for syntactic analysis in processors. It may reasonably be expected that similar benefits will be derived from the development of adequate formal models of the semantics of programming languages.

REFERENCES

1 P. Landin
   A correspondence between ALGOL 60 and Church's lambda-notation
   Communications of the Association for Computing Machinery 1965 8 2 and 3
<table>
<thead>
<tr>
<th><strong>2 T B STEEL</strong> (editor)</th>
<th><strong>4 R W FLOYD</strong></th>
</tr>
</thead>
</table>
| *Formal language description languages for computer programming* | *Assigning meanings to programs*
| **3 R NARASIMHAN** | **5 J McCARTHY et al.** |
| *Programming languages and computers: A unified metatheory* | *Lisp 1.5 programmers manual* |
| Advances in Computers Vol 8 F L Alt editor 1967 | Massachusetts Institute of Technology 1962 |