A HIERARCHICAL MACHINE MODEL FOR THE SEMANTICS OF PROGRAMMING LANGUAGES[1)]

Victor R. Basili
Albert J. Turner

University of Maryland

I. Introduction

A formal definitional facility for specifying the semantics of a programming language should provide a tool that aids in the design, definition, implementation and comparison of programming languages. This paper deals with the development of such a facility. The approach taken is that the metalanguage for defining the semantics of a language should try to represent the semantic structure of that language in a manner similar to the way that phrase structure grammars are used to represent the syntactic structure of languages. Just as we think of the complexity of the syntactic structure of a language as being classified by the complexity of the algorithm required to translate it, we would like to be able to think of the complexity of the semantic structure of a language as being categorized by the complexity of the algorithm required to interpret it.

The purpose of this paper is to demonstrate this formal definitional facility by modeling a specific programming language and to point out the possibilities for using it to relate the semantic complexity of the language with the complexity of the machine required to interpret it. The language defined is SIMPL-X [1], a procedure-oriented, structured programming language. A set of hierarchical directed graphs are defined to model the program and data structures of the language and a set of semantic functions are defined for describing the execution of its programs. The semantic functions are divided into two categories: the control functions and the component functions. The control functions are those that specify program flow and call for the execution of the component functions. The component functions are those that define the execution of the high-level components of the language relative to a particular control mechanism (set of control functions). It is the specification of the control mechanism required to define the execution of a particular language which can be considered as characterizing the complexity of that language.

Informally, the system can be viewed as a 4-tuple (N,O,M,F) where N is a set of nodes, O is a set of objects, M is a set of mappings from N into O and F is a set of transition functions which can vary the selection of the mappings in M. More specifically, in the model defined here, O is partitioned into three disjoint sets (D,G,A) where D represents a set of atomic elements, G represents a set of directed graphs over the nodes in N, and A represents a set of attributes or properties. M consists of all mappings v, h, and a where $v:N \rightarrow D$, $h:N \rightarrow G$, and $a:N \times I^{+} \rightarrow A$ (where I^{+} is the set of positive integers). F contains the three transition functions: setv, seth, seta, which vary the selection of the v, h, and a mappings, respectively.

1) This research was supported in part by the Office of Naval Research under Grant N00014-67-A-0239-0021 (NR-044-431).

It should be noted that many interpretation systems for programming languages, such as the Vienna Definition Language [3], the hierarchical graph system of Pratt [4], and the Contour Model of Johnson [2] can be abstracted to the above 4-tuple.

In the next section we describe the formal definitional facility HGL, a hierarchical graph language, which consists of a set of formally-defined structures called attributed h-graphs, and a set of primitive transition functions acting on these h-graphs and transforming them into other h-graphs. We then define for the specific language SIMPL-X a set of graph structures and their construction and accessing primitives to be used in a model for the language. Using these structures we next define a machine structure for SIMPL-X by defining its syntactic structure in terms of h-graphs.

In Section III the semantic structure of the language is defined by a set of control functions. These control functions are defined in terms of the primitive transition functions, the graph construction and accessing primitives, and some standard mathematical primitives.

II. The Hierarchical Graph Language (HGL)

HGL offers a method of describing the semantics of a programming language relative to a particular graph structure defined over a set of nodes. Each node may be thought of as representing a unit of information. Associated with the node may be some atomic data value (possibly representing program data), some further substructure of information again represented by a graph over the nodes (possibly representing program or data structures), and some set of attributes which might define the characteristics of that unit of information (possibly representing some compile-time properties of that unit). The model defined permits the structuring of both program and data. This syntactic description is accomplished by using an attributed h-graph as the basic syntactic structure.

Let N be a set of nodes, $\mathcal{D}$ be a set of atoms, and A be a set of attributes. Let G be the set of all graphs defined over nodes in N with labels (if any) from $\mathcal{D}$.

Definition: An attributed hierarchical graph (h-graph) over $(N,\mathcal{D},A,G)$ is a 7-tuple (N,D,A,G,v,h,a) where $N \subseteq N$, $D \subseteq \mathcal{D}$, $A \subseteq A$, $G \subseteq G$, and $v:N \to D$, $h:N \to G$, and $a:N \times I_k^+ \to A$ where I_k^+ denotes the first k positive integers.

The graphs of G are used for specifying the structure of the language. The types of graphs (e.g., multigraph, tree, list, pseudograph, etc.) used may vary depending upon the model.

The execution of a program P is defined in the usual manner as a sequence of states. Each state is represented by an h-graph, which in turn represents the program and its data structures at some point in the execution. A state transition is caused by some change in one of the three mappings v, h, or a on the nodes or the inclusion or removal of a node from the set N of nodes. The three functions which change the v, h, a mappings are setv, seth, and seta which are defined below. The two functions that alter the set N are create and delete.

Let $S_i = (N_i,D_i,A_i,G_i,v_i,h_i,a_i)$ be the h-graph representation of the program in state S_i and $S_{i+1} = (N_{i+1},D_{i+1},A_{i+1},G_{i+1},v_{i+1},h_{i+1},a_{i+1})$ be the h-graph representation of the program in state S_{i+1}. The state transitions associated with the mappings from the state graph S_i into the state graph S_{i+1} are defined in Table 1. For all states S_i, $D_i = \{d \mid v_i(n) = d, \text{ for some } n \in N_i\}$, $G_i = \{g \mid h(n) = g, \text{ for some } n \in N_i\}$, and $A_i = \{a \mid a_i(n,k) = a, \text{ for some } n \in N_i, k \in I_j^+\}$.

In order to define a model for a specific language L, there is a set of nodes $N_L \subseteq N$, a set of atoms $\mathcal{D}_L \subseteq \mathcal{D}$, a set of attributes $A_L \subseteq A$, and a set of graphs $G_L \subseteq G$, which define the subuniverse pertinent to the model M_L for the language L. M_L may then be defined as a triple (H,T,S). H is the set of all h-graphs defined over $(N_L,\mathcal{D}_L,A_L,G_L)$, i.e., the set of all possible states. T is the translation mapping for any program P into its h-graph syntactic form; it is used to generate the start state s_0 for any program P. S is a set of semantic routines which define execution by specifying the transitions from the start state to a final state. A state is a final state when there are no further transitions as specified by S.

setv: $N_i \times \mathcal{D} \rightarrow \{\underline{true}\}$

where the associated transition $S_i \rightarrow S_{i+1}$ for setv(n,d) is defined by:

$$N_{i+1} = N_i,\ a_{i+1} = a_i,\ h_{i+1} = h_i, \text{ and}$$

$$v_{i+1}(m) = \begin{cases} v_i(m), \forall\ m \in N_i,\ m \neq n \\ d,\ m = n \end{cases}$$

(setv assigns an atomic value to a node)

seth: $N_i \times G \rightarrow \{\underline{true}\}$

where the associated transition $S_i \rightarrow S_{i+1}$ for seth(n,g) is defined by:

$$N_{i+1} = N_i,\ v_{i+1} = v_i,\ a_{i+1} = a_i, \text{ and}$$

$$h_{i+1}(m) = \begin{cases} h_i(m), \forall\ m \in N_i,\ m \neq n \\ g,\ m = n \end{cases}$$

(seth assigns a graph to a node)

seta: $N_i \times I_r^+ \times A \rightarrow \{\underline{true}\}$

where the associated transition $S_i \rightarrow S_{i+1}$ for seta(n,j,p) is defined by:

$$N_{i+1} = N_i,\ v_{i+1} = v_i,\ h_{i+1} = h_i, \text{ and}$$

$$a_{i+1}(m,k) = \begin{cases} a_i(m,k), & \text{for } m \neq n,\ m \in N_i, \\ & \text{and } k \in I_r^+; \text{ and for} \\ & m = n,\ k \neq j, \text{ and} \\ & k \in I_r^+ \\ p,\ m = n,\ k = j \end{cases}$$

(seta assigns an attribute to a node)

create: $\phi \rightarrow N - N_i$

where the associated transition $S_i \rightarrow S_{i+1}$ for create = n is defined by:

$N_{i+1} = N_i \cup \{n\}$; $v_{i+1}(m) = v_i(m)$,
$h_{i+1}(m) = h_i(m)$, and
$a_{i+1}(m,k) = a_i(m,k)$, $\forall\ m \in N_i$, $k \in I_r^+$; and
$h_{i+1}(n) = a_{i+1}(n) = v_{i+1}(n) = \underline{undefined}$

(create adds a new node to the nodeset of the h-graph)

delete: $N_i \rightarrow \{\underline{true}\}$

where the associated transition $S_i \rightarrow S_{i+1}$ for delete(n) is defined by:

$N_{i+1} = N_i - \{n\}$; $v_{i+1}(m) = v_i(m)$,
$h_{i+1}(m) = h_i(m)$, and
$a_{i+1}(m,k) = a_i(m,k)$, $\forall\ m \neq n \in N_i$, $k \in I_r^+$

(delete removes a node from the nodeset of the h-graph)

Table 1. State Transitions

Then, for a specific program P, the model generates the tuple $(\{P_i\},P_0,S)$ where P_0 is the h-graph syntax of the program and $\{P_i\}$ represents the set of all possible states generable by S starting in state P_0.

Let us now consider a model for the programming language SIMPL-X. The following graph definition and graph construction and accessing primitives are used to represent the structures of the language.

Definition: A directed graph g is a 4-tuple (N_g,E_g,n_g,e_g) where $N_g \subseteq N$ is the set of nodes, denoted by nodes(g); $E_g \subseteq N_g \times N_g$ is the set of edges, denoted by arcs(g), with $(n,n) \notin E_g$ for all $n \in N_g$; $n_g \in N_g$ is a distinguished node called the entry node and denoted by entry(g); and $e_g: E_g \rightarrow \mathcal{D}$ is the edge label mapping.

Let G be a set of directed graphs. In order to construct and traverse the graphs $g \in G$, the graph construction and accessing primitives in Table 2 and Table 3 are defined. A further discussion of these primitives may be found in [5].

attach: $G \times N \times N \times \mathcal{D} \rightarrow G$ If $g' = attach(g,n,m,d)$, then $nodes(g') = nodes(g) \cup \{n,m\}$, $arcs(g') = arcs(g) \cup \{(n,m)\}$, $entry(g') = entry(g)$, and $e_{g'}(p,q) = \begin{cases} e_g(p,q), & \text{for } (p,q) \in arcs(g),\ p \neq n,\ q \neq m \\ d, & \text{for } p = n,\ q = m \end{cases}$ (attach adds a node(s) and any associated arc to a graph)	detach: $G \times N \rightarrow G$ If $g' = detach(g,n)$, then $nodes(g') = nodes(g) - \{n\}$, $arcs(g') = arcs(g) - (\{(n,m) \mid m \in nodes(g)\} \cup \{(m,n) \mid m \in nodes(g)\})$, and $entry(g') = entry(g)$. (Note that the graph is not well-defined if $n = entry(g)$.) (detach removes a node and its associated arcs from a graph)
setentry: $G \times N \rightarrow G$ If $g' = setentry(g,n)$ where $n \in nodes(g)$, then $nodes(g') = nodes(g)$, $arcs(g') = arcs(g)$, $entry(g') = n$, and $e_{g'} = e_g$. (setentry sets the entry node of a graph)	

Table 2. Graph Construction Primitives

padj: $N \times G \rightarrow 2^N$, defined by $padj(n,g) = \{m \in nodes(g) \mid (n,m) \in arcs(g)\}$ (padj(n,g) is the set of nodes that terminate an outgoing arc from node n in graph g)	nadj: $N \times G \rightarrow 2^N$, defined by $nadj(n,g) = \{m \in nodes(g) \mid (m,n) \in arcs(g)\}$ (nadj(n,g) is the set of nodes that begin an incoming arc to node n in graph g)

Table 3. Graph Accessing Primitives

What follows is an example giving the h-graph syntax for a major subset of SIMPL-X. (A subset was chosen for the sake of brevity; the full language has been modeled by the authors.) This syntax defines the specification for the translation mapping T. The notation used is an extension to BNF notation, and the standard regular expression notation.

The following extensions to BNF are defined:

a) <a> ::= [<b>] | [<c>] | [<d>; <e>]

This means that <a> is defined as: [<b>], a node whose h value is given by <b>; or [<c>], a node whose v value is given by <c>; or [<d>; <e>], a node whose h value is given by <d> and whose v value is given by <e>.

b) <f> ::= [<g>]; $att_1 = val_1, \ldots, att_n = val_n$

This means that <f> is defined as a node whose h value is given by <g> and whose a_i value is given by val_i for all attributes att_i given.

c) <h> ::= <i> $\overset{*}{\rightarrow}$ <j> <k>

This means that <h> is defined as a directed graph from an element of class <i> to an element of class <k> with a directed arc from <i> to <k> whose label is an element of class <j>. The entry node of the graph is <i>.

d) <ℓ> ::= [<m_1>,...,<m_i>*,...,<m_n>]

This means that <ℓ> may be defined as a node whose h value is a graph with n disjoint components, from classes <m_1>,<m_2>,...,<m_n>. The entry node of the graph is <m_i>.

The h-graph syntax for expressions is basically given by a two-node graph where the operator is contained in the entry node and the arguments are contained in the other node as nodes in a graph. This syntax is used for all the functions defined in the model.

<expr> ::= <operation> | <variable>

<operation> ::= [<operator>*{→<operand list>}^{1}]; class = 'operation'

<operand list> ::= [<operand>*{→<operand>}]

<operand> ::= <expr>

<operator> ::= [<primitive binary function>]; form = 'binary', type = 'primitive' |
[<primitive unary function>]; form = 'unary', type = 'primitive' |
[<semantic function>]

The <primitive binary function> and <primitive unary function> represent the primitive language operations such as add, subtract, multiply, unary minus, etc. The <semantic function> represents the semantic routines such as assign, ref, convert, call, etc. that are used to define the semantics of the language.

Since this expression graph is used so heavily throughout this paper, the following shorthand notation will be used for the sake of brevity and readability:

[name] → [x → y] will be written as name(x,y)

Data in SIMPL-X is of one type, integer, and the only data structure is a one-dimensional array. However, procedures may be recursive and so local variables and parameter variables are defined as recursive variables, containing their own stacks.

<variable> ::= <simple var> | <rec var>

<simple var> ::= [<integer>] | [<undefined>]; class = 'data', type = 'int', form = 'simp'

<rec var> ::= [<simple var>*{→<simple var>}]; class = 'data', type = 'int', form = 'rec'

<array> ::= <simple array> | <rec array>

<simple array> ::= [<simple var>*{→<simple var>}]; class = 'data', type = 'array', form = 'simp'

<rec array> ::= [<simple array>*{→<simple array>}]; class = 'data', type = 'array', form = 'rec'

A program in SIMPL-X consists of a set of global variables and a set of procedures, one of which is marked as the start procedure. Procedures may be recursive. There are five types of statements: assignment statement, call statement, if statement, while statement, and case statement.

<program> ::= [{<global variable>},<proc def>*,{<proc def>}]

<global var> ::= <simple var> | <simple array>; scope = 'global'

<proc def> ::= [{<formal parlist>}^{1},[{<loc var>}],<stat list>*]

<formal par list> ::= [<par>*{→<par>}]; class = 'parlist'

<par> ::= <rec var> | <rec array>

<loc var> ::= <rec var> | <rec array>; scope = 'local'

<stat list> ::= [<stat>*{→<stat>}]; class = 'program'

<stat> ::= [<if stat>] | [<while stat>] | [<case stat>] | [<assign stat>] | [<call stat>]

<call stat> ::= <entry>* → [<call operation>] → <exit>

<assign stat> ::= <entry>* → [<assign operation>] → <exit>

<if stat> ::= <entry>* → [<convert operation>] → 1 → <stat list> → <exit>
0 → {<stat list>}^{1} → <exit>

<case stat> ::= <entry>* → <expr> → n_1 → <stat list> → <exit>
n_2 → <stat list> → <exit>
n_n → <stat list> → <exit>
else → {<stat list>}^{1} → <exit>

<while stat> ::= <entry>* → [<convert operation>] → 1 → <stat list> → (back to [<convert operation>])
0 → <exit>

<entry> ::= [no-op]

<exit> ::= [no-op]

<assign operation> ::= assign (<ℓp>,<expr>)

<ℓp> ::= <variable> | <ref operation>

<ref operation> ::= ref (<array>,<expr>)

<call operation> ::= call (<proc def>,{<arg list>})

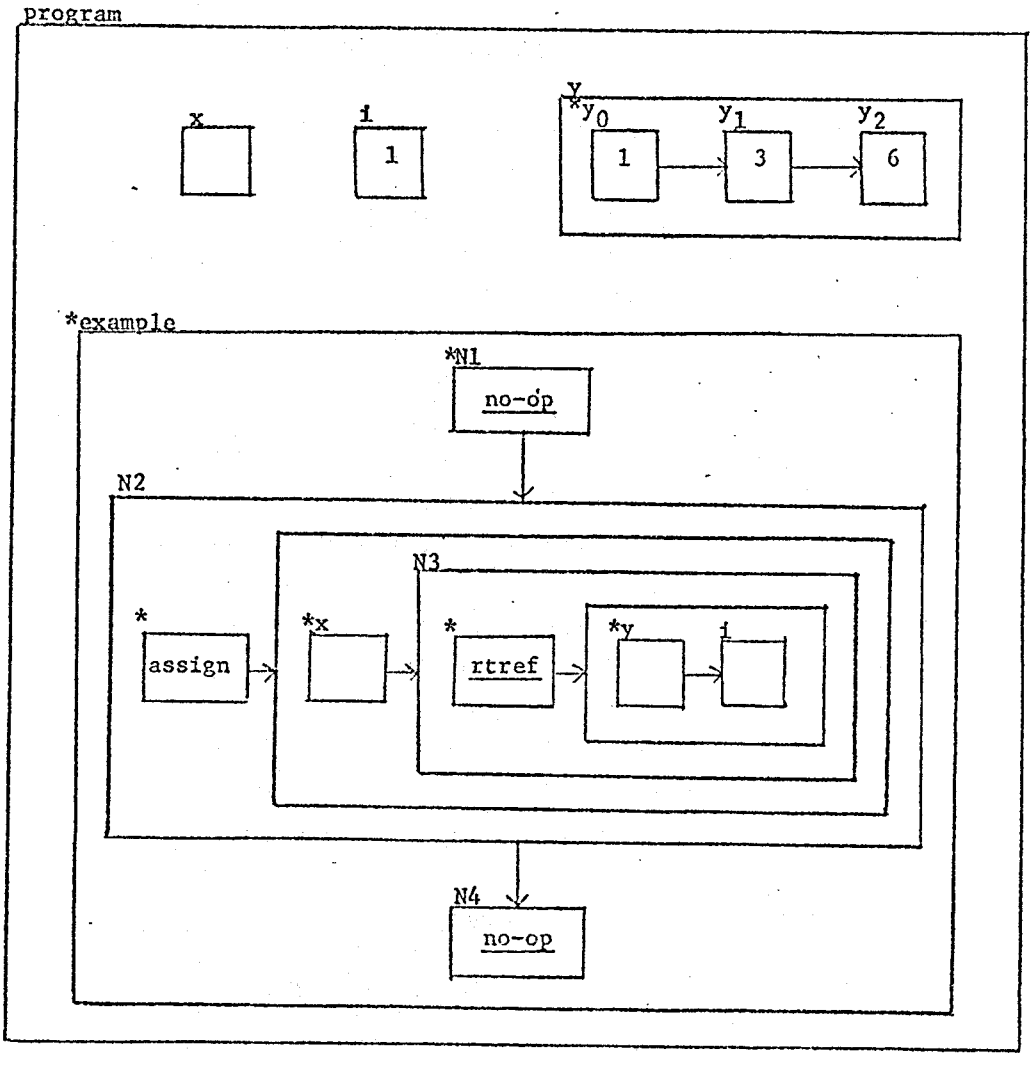
<arg list> ::= [<arg> {*<arg>}]

<arg> ::= <expression> | <array>

<convert operation> ::= convert (<expr>)

As an example, consider the following SIMPL-X program and its h-graph representation:

```
int x, i = 1
int array y(2)=(1,3,6)
proc example
  x := y[i]
start example
```



Note that all nodes marked with the same label are the same node with the same v, h, and a mappings.

III. The Semantic Functions

The semantics of a programming language are defined as a set of routines, called semantic functions, which are written relative to a machine whose structure is defined by the hierarchical graph structures chosen for the language and whose instruction set consists of the state transition primitives, the graph structure construction and accessing primitives and some mathematical primitives. The semantic functions fall into two categories: those that specify the sequence of control, called control functions, and those that define the meaning of the constituent components of the language (e.g., procedures) relative to one another, called component functions. The control functions define a control mechanism which permits the execution of the component functions. The component functions are then defined relative to this control mechanism.

For example, the graph structures defined for SIMPL-X in the previous section define the structure of the SIMPL-X machine. Using this machine structure and the instruction set of supplied primitives, a set of control functions for SIMPL-X is defined below. These control functions define a control mechanism which calls for the evaluation of all the semantic functions and uses the results of these evaluations, when necessary, to determine flow of control for a particular program.

Since SIMPL-X is a structured programming language, the semantic functions have been defined as a set of functions which are highly recursive in nature. The definition of the format of these functions is given in Appendix I.

Throughout the semantic function definitions, several primitive notions are used which have not been defined in this paper. Most of these are well-known set-theoretic and mathematical primitives. Those whose meaning may not be clear are given in Appendix II along with some notational shorthand that is also used.

Using the definition of semantic functions in Appendix I, the following control functions are given as the definition of the control mechanism for SIMPL-X. There are three functions, each of which must be understood in terms of the other. An example of their use is given at the end of Section III.

```
execute (node) =
    class(node) = 'operation' ⇒ eval(node)
    class(node) = 'program' ⇒ traverse(*node,node)
traverse (node, gnode) =
    class(node) = 'data' ⇒ v(node)
    true ⇒ traverse(result-node(execute(node),node,gnode),gnode)
result-node (value, node, gnode) =
    value = undefined ⇒ [undefined]
    size({padj(node,h(gnode))})=0 ⇒ [value]
    size({padj(node,h(gnode))})=1 ⇒ padj(node,h(gnode))
    size({x∈padj(node,h(gnode))|e_h(gnode)(node,x)=value})=1 ⇒
       elt{x∈padj(node,h(gnode))|e_h(gnode)(node,x)=value}
    size({x∈padj(node,h(gnode))|e_h(gnode)(node,x)='else'})=1 ⇒
       elt{x∈padj(node,h(gnode))|e_h(gnode)(node,x)='else'}
    true ⇒ [undefined]
```

The recursive function execute executes the single node argument passed to it and returns the value resulting from the execution of that node. The value resulting from the execution of a node is the value of the primitive function contained somewhere in that node which was evaluated last. The function execute executes a node as follows: (1) if the node is an operation node (i.e., contains a single function), then the contents of that node (the function) are evaluated by the function eval, and the result of this evaluation is returned as the value of execute; (2) if the node is a program node (i.e., contains a graph to be executed), then the traverse function is called to traverse the graph contained in that node by passing to it as arguments the entry point of the graph and the graph itself, and the result of

the traverse function (i.e., the value resulting from executing that node) is returned as the value of execute.

The recursive function traverse controls the traversal of the path of execution through a graph g, starting at node n, and returns the value resulting from the last successfully executed node in the path. It calculates the proper path by calling for the execution of the node n (via the execute function) and using the value returned by the execution of n to calculate the next node, m, in the graph to be executed (via the result-node function). If m is a program node or an operation node, traverse calls itself recursively to execute m and calculate the next node in the path. If m is a data node, then n was the last executable node in the path of execution and the contents of m is the value returned by the execution of n (by the definition of result-node). At this point the function terminates, returning the value of m.

The function result-node, given a value v, a node n, and the graph g containing node n, returns either (1) the next node m in the path of execution, possibly using the value v to determine that node, or (2) a node containing the value v. It determines what node to return by calculating the number of edges that leave the given node n relative to the given graph g. If there are no edges leaving the node n, then result-node returns a node containing the value v. If there is one edge (n,m) leaving node n, then result-node returns the node m. If there is more than one edge leaving the node, then result-node returns the node m for which the edge (n,m) is labeled by the value v. If, however, the given value v is the special constant undefined or none of the above possibilities hold, a node containing the special constant undefined is returned. This has the effect of aborting the execution.

Having defined the control mechanism for SIMPL-X, we now complete the definition of the semantics of the language by defining the component functions relative to this control mechanism.

The first function, eval, calls for the evaluation of all data nodes, primitive functions, and semantic functions. It does this by determining what kind of node it is evaluating and then returning the appropriate evaluation. If the node to be evaluated is a data node, eval returns the value contained in that node or the entry node of the graph contained in it. If the node contains a primitive operation of the language (e.g., add, mult, etc.), eval assumes that operation to be a defined primitive and evaluates it. If the node is a semantic function, eval calls for its evaluation.

```
eval (node) =
    class (node) = 'data' ⇒
        form (node) = 'simp' ⇒ v(node)
        T                    ⇒ v(*node)
    class (node) = 'operation' ⇒
        form (node ) = 'bin' ⇒ v(*node)(eval(arg1(node)),eval(arg2(node)))
        form (node ) = 'un' ⇒ v(*node)(eval(arg1(node)))
    v(*node) = 'assign' ⇒ assign(arg1(node),arg2(node))
    v(*node) = 'convert' ⇒ convert(arg1(node))
    v(*node) = 'call' ⇒ call(arg1(node),arg2(node),pars(arg1(node)))
    v(*node) = 'rtref' ⇒ rtref(arg1(node),arg2(node))
    v(*node) = 'installp' ⇒ installp(arg1(node),arg2(node),arg3(node))
                    .
                    .
                    .
    T → undef
```

The other semantic functions are given without any detailed explanation in Appendix III. It is only noted that in SIMPL-X procedure calls, arrays are passed by reference and integer variables are passed by value.

As an example, consider the execute function operating on the node program in the example of Section II. E denotes the node example and P denotes the node program.

```
execute(P) = traverse(E,P)
           = traverse(result-node(execute(E),E,P),P)
           =                  (traverse(N1,E),E,P),P)
           =                  (traverse(result-node(execute(N1),N1,E),E,P),P)
           =                      .       .      (eval(N1),N1,E),E,P),P)
           =                      .       .      (no-op,N1,E),E,P),P)
           =                  (traverse(result-node(true,N1,E),E),E,P),P)
           =                  (traverse(N2,E),E,P),P)
           =                  (traverse(result-node(execute(N2),N2,E),E)E,P),P)
           =                                    (eval(N2),N2,E),E,P),P)
           =                                    (assign(x,N2),N2,E),E,P),P)
           =                                    (setv(x,eval(N3)),N2,E),E,P),P)
           =                                    (setv(x,rtref(y,i)),N2,E),E,P),P)
           =                                    (setv(x,eval(ref(y,i))),N2,E),E,P),P)
           =                                    (setv(x,eval(item(eval(i),y))),N2,E),E,P),P)
           =                                    (setv(x,eval(item(1,y))),N2,E),E,P),P)
           =                                    (setv(x,eval(y1)),N2,E),E,P),P)
           =                                    (setv(x,3),N2,E),E,P),P)
           =                  (traverse(result-node(true,N2,E),E),E,P),P)
           =                  (traverse(N4,E),E,P),P)
           =                  (traverse(result-node(execute(N4),N4,E),E),E,P),P)
           =                      .       .      (eval(N4),N4,E),E),E,P),P)
           =                      .       .      (no-op,N4,E),E),E,P),P)
           =                  (traverse(result-node(true,N4,E),E),E,P),P)
           =                  (traverse([true],E),E,P),P)
           = traverse(result-node(true,E,P),P)
           = traverse([true],P)
           = true
```

IV. Conclusion

To reiterate briefly, this paper presents a method for modeling programming languages by supplying a hierarchical, structured framework for defining program and data structures. A particular language, or class of languages, may be defined relative to a particular graph structure, or set of graph structures, that best describes the actual structures of the language. Each graph structure has associated with it a set of construction and accessing primitives, thus defining the basic machine upon which the semantic functions are defined.

The semantic functions define the language by describing the effect of execution of the various components of the language upon the basic machine. This is done by first defining a control mechanism via set of control functions, and then defining a set of component functions that describe execution of the language components relative to that control mechanism.

In this paper, the authors have attempted to demonstrate the important role played by the control mechanism in characterizing the complexity of a language for the purposes of comparison. SIMPL-X is a structured programming language with a rather straightforward runtime environment. Hence, the kind of control mechanism definable for it can be a simple, highly recursive path-traversing mechanism. Language like ALGOL, which has a more complex runtime environment, requires a more complex control mechanism. It is the authors' desire to examine a variety of control mechanisms on the same basic

machine in order to analyze and compare the complexity of languages and language components.

As a tool in language design, the hierarchical modular framework of HGL provides a good basic structure for a top-down approach, permitting the examination of language components at a variety of levels of detail. As a definitional facility, HGL permits the choice of structures and semantic function format that are most "natural" for the language. For example, the SIMPL-X program structure can be represented as a flowchart-type directed graph; LISP data structures could be represented using actual lists; and lambda expressions could be used as the basis for the semantic function format for LISP. As an implementation tool, HGL permits the analysis of a variety of data structures and associated primitives for the representation of the different language components. Finally, HGL permits the choice of the most convenient data and control structures for use in the comparison of programming languages.

HGL is quite similar in many respects to other semantic models [6]. The flexibility of the model and the high level of permissible structures make it more general than the more widely used models (such as VDL [3]). The authors have derived much benefit from using HGL as a tool in language design, as a definitional facility, and as an implementation model.

Bibliography

[1] Basili, V. R., SIMPL-X, A Language for Writing Structured Programs, University of Maryland, Computer Science Center, Technical Report TR-223, 1973.

[2] Johnston, J. B., The Contour Model of Block Structured Processes, Symposium on Data Structures in Programming Languages, SIGPLAN Notices, February 1971.

[3] Lucas, P., Lauer, P., and Stigleitner, H., Method and Notation for the Formal Definition of Programming Languages, IBM Laboratory Vienna, Technical Report TR 25.087, 1968.

[4] Pratt, T. W., A Hierarchical Graph Model of the Semantics of Programs, SJCC 34, 1969, 813-825.

[5] Rheinboldt, W. C., Basili, V. R., and Mesztenyi, C. K., On a Programming Language for Graph Algorithms, BIT 12, 1972, 220-241.

[6] Wegner, P., Operational Semantics of Programming Languages, SIGPLAN Notices, January 1972.

Appendix I. Semantic Function Syntax and Semantics

Semantic Function Syntax

<semantic fn> ::= <fn name>{(<formal parlist>)}^{1} = <semantic fn body>

<semantic fn body> ::= <cond def list> | <fn call> | <value>

<cond def list> ::= {<cond def list>,}^{1}<cond> = <semantic fn body>

<cond> ::= <Boolean expression>

<fn call> ::= <semantic fn call> | <primitive fn call>

<semantic fn call> ::= <semantic fn name>{(<actual parlist>)}^{1}

<primitive fn call> ::= <primitive fn name>{(<actual parlist>)}^{1}

<actual parlist> ::= {<actual parlist>,}^{1}<actual par>

<actual par> ::= <fn call> | <graph form> | <value>

<graph form> ::= [<node>* {→<node>}]; class = 'program'

<node> ::= [<semantic fn>]; class = 'semantic fn' | [<primitive fn>]; class = 'primitive fn' |
[<graph form>]; class = 'program' | [<value>]; class = 'data' |
<variable> | <expression> | <program> | <segment def> | <stat list>

<value> ::= true | false | undefined | <number>

<formal parlist> ::= {<formal parlist>,}^{1}<identifier>

<primitive fn name> ::= add | mult | ...

Semantic Function Semantics

Let a function be called evaluable if none of its arguments are functions and it is not contained in a <graph form>. To evaluate a semantic function, expand the function by

1. a) substituting the actual parameters given in the <semantic fn call>, for the corresponding parameters given in the <semantic fn> (this is equivalent to a call by name macro expansion) and
   b) evaluating all evaluable primitive functions, or if there are none
   c) expanding all evaluable semantic functions
2. repeat step 1 until all functions are evaluated or expanded and a single value is returned as the result of the original semantic function call.

Appendix II. Semantic Function Primitives and Abbreviations

elt (set) returns any element from a set

item (k,node) returns the kth node in the graph h (node) where the graph is a list

size (set) returns the number of elements in the set

nsize (graph) = size (nodes(g))

*node = entry (h(node)) (i.e., the entry node of the graph associated with a node)

argl (node) = *elt(padj(*node,h(node))) (i.e., the first argument node of an argument list node)

arg2 (node) = elt(padj(argl(node),h(elt(padj(*node,h(node))))))

arg3 (node) = etc.

pars (proc) = elt({x|x=h(*node,node)) and class (x)='parlist'}) (i.e., the node whose h map is the graph of parameters)

locs (proc) = the node whose h map is the graph of locals

locsandpars (proc) = the node whose h map is the graph of locals and parameters

```
assign (ℓp,rp) =
     type (ℓp) = 'integer' ⇒
          form (ℓp) = 'simp' ⇒ setv(ℓp,eval(rp))
          form (ℓp) = 'rec' ⇒ setv(*ℓp,eval(rp))
     class (ℓp) = 'operation' ∧ v(*ℓp) = 'ref' ⇒ setv(ref(arg1(ℓp),arg2(ℓp)),eval(rp))
     T ⇒ undef
ref (array,expr) =
     form (array) = 'rec' ⇒ item(eval(expr)+1,*array)
     T                    ⇒ item(eval(expr)+1,array)
rtref (array,expr) = eval(ref(array,expr))
convert (node) =
     eval (node) = 0 ⇒ 0
     T               ⇒ 1
call (proc,args,pars) = execute([[installp(proc,args,pars)] → proc → [return(proc)]]
installp (proc,args,pars) =
     execute([[evalargs(args,pars)] → [installpargs(args,pars)] → [installplocs(locs(proc))]]
evalargs (args,pars) =
     nsize(h(args)) = nsize(h(pars)) ≠ 0 ⇒ evalarg(*args,args,*pars,pars)
     nsize(h(args)) = nsize(h(pars) = 0 ⇒ true
     T                                  ⇒ undef
evalarg (arg,args,par,pars) = execute([[setv(par,eval(arg))] → [nextevalarg(arg,args,par,pars)]])
nextevalarg (arg,args,par,pars) =
     size(padj(arg;h(args))) = 0 ⇒ true
     T ⇒ evalarg(elt(padj(arg,h(args))),args,elt(padj(par,h(pars))),pars)
installpargs (args,pars) =
     nsize(h(args)) ≠ 0 ⇒ installparg(*h(args),args,*h(pars),pars)
     T                  ⇒ true
installparg (arg,args,par,pars) =
     type(par) = type(arg) = 'integer' ⇒
          execute([[setv(push(create,par),v(par))] → [nextparg(arg,args,par,pars)]])
     type(par) = type(arg) = 'array' ⇒
          form(arg) = 'simp' ⇒ execute([[seth(push(create,par),h(arg)] → [nextparg(arg,par,pars)]])
          T                  ⇒ execute([[seth(push(create,par),h(*arg)] → [nextparg(arg,par,pars)]])
     T                           ⇒ undefined
nextparg (arg,args,par,pars) =
     size(padj(arg,h(args))) = 0 ⇒ true
     T ⇒ installparg(elt(padj(arg,h(args))),args,elt(padj(par,h(pars))),pars)
installplocs (locs) =
     nsize(h(locs)) = 0 ⇒ true
     T                  ⇒ installploc(1,locs)
```

```
installploc (i,locs) =
    i = nsize(h(locs)) ⇒
        type(item(i,locs)) = 'int' ⇒ push(create,item(i,locs))
        T                          ⇒ installplocarray(create,nodesize(*item(i,locs)),item(i,locs))
    T                  ⇒
        type(item(i,locs)) = 'int' ⇒ execute([[push(create,item(i,locs))] → [installploc(i+1,locs)]])
        T ⇒ execute([[installplocarray(create,nodesize(*item(i,locs)),item(i,locs))] →
                                                [installploc(i+1,locs)]])
installplocarray (node,n,array) =
    n = 0 ⇒ truef(push(node,array))
    T ⇒ execute([[truef(push(create,node))] → [installplocarray(node,n-1,array)]])
return (proc) = release(locsandpars(proc))
release (locsandpars) =
    nsize(h(locsandpars)) = 0 ⇒ true
    T                         ⇒ releaseloc(1,locsandpars)
releaseloc (i,locsandpars) =
    i = nsize(h(locsandpars)) ⇒ truef(pop(item(i,locsandpars)))
    T ⇒ execute([[truef(pop(item(i,locsandpars)))] → [releaseloc(i+1,locsandpars)]])
truef (node) = true
push (node1,node2) =
    nsize(h(node2)) = 0 ⇒ seth(node2,setentry(node2,node1))
    T                   ⇒ seth(node2,setentry(attach(h(node2),node1,*node2),node1))
pop (node) =
    nsize(h(node)) = 1 ⇒ execute([[seth(node,[])] → [pass(*node)]])
    T ⇒ execute([[seth(node,detach(setentry(h(node),elt(padj(*node,h(node))),*node))] → [pass(*node)]])
pass (node) = node
```