Journal of Computer Languages, Vol. 1, pp. 255–273. Pergamon Press, 1975. Printed in Northern Ireland

A STRUCTURED APPROACH TO LANGUAGE DESIGN*

V. R. BASILI
Computer Science Department, University of Maryland, College Park, Maryland 20742, U.S.A.

(*Received* 6 *June* 1974; *and in revised form* 12 *November* 1974)

Abstract—This report is an attempt at systematizing a set of ground rules for high-level language design. It recommends the use of a hierarchical semantic model schema, HGL, in a step by step, top-down approach imposing more and more structure on the language components as the design becomes solidified. The approach is demonstrated by showing the stepwise design of the high-level language, GRAAL. The method recommended is divided into three major phases. The first is an informal one. The second is encoding the language components into a very high-level model. This high-level design allows a redesign of language components before they have been specified at too detailed a level. The third phase is to design the compiler in HGL using the final language design.

High-level language design Design methodology Language modeling Hierarchichal graph models Sets Graphs

1. INTRODUCTION

DESIGNING a high-level programming language is at best a difficult task. The designer must keep the entire design in mind at the top level in order to decide what the various components of the language should be, what they should look like, and what effect they have upon one another. And if he should settle on a design, and then change some aspect of it, he must be concerned with the effect this would have on the rest of the language. These problems are made more difficult if any of the features of the language are 'new' or have never been used in combination with one another.

Unfortunately, as with most new and large projects, there is not much guidance available to the designer. There is little information in the literature on what approach to take. Experience with other language designs is helpful but where does one start? What is really needed is a set of tools to give the designer as much control over the project as possible.

This paper is an attempt at systematizing language design based in part on a design and implementation experience with GRAAL [1–3], a high-level programming language for use in the solution of graph problems primarily arising in applications. A sequence of hierarchical semantic models is recommended for designing the language and its compiler; and the approach is demonstrated by application to the set and graph features of GRAAL.

This sequence of hierarchical semantic models, specified in the hierarchical graph language HGL [4], is in effect a sequence of very high-level machines, each of which is a refinement of its predecessor, with the data and control structures for the new machine being more highly specified. This permits language design to be approached in a top-down manner.

The next section contains a description of the approach. Sections 3, 5 and 6 describe the application of the approach to GRAAL, and Section 4 contains an informal description of the HGL semantic model used in the process.

2. THE METHODOLOGY

The top-down approach recommended here can be broken down into three major phases, each of which is in turn organized in a topdown manner. Topdown in this case

* This work was partially supported by ONR Grant N00014-67-A-0239-0021 (NR-044-431) to the Computer Science Center of the University of Maryland.

means adding more detail or more structure at each step in the process. The first phase is the discussion stage which consists of an informal specification of the language components. The second phase is the formal language design stage which is the formalization of the discussion stage. Phase 3 encompasses the high-level design of the implementation. No phase is ever frozen and work on any one phase may cause changes in the others. Only after several passes through these three phases are completed should the actual language specification (detailed syntax and semantics) and the compiler specification and implementation begin.

The design of a programming language is centered around the basic set of primitives which define the problem area addressed by the language. These primitives form the basic data types and operations of the language. Once discovered and chosen, they are imbedded in a set of control and data structures appropriate for expressing algorithms in the problem area. The role of phase 1 is to specify informally these basic language components and define a full complement of language features necessary to develop a viable programming language design.

The second phase of the language design consists of formalizing the discusson stage by representing the design in an interpretive semantic model. Ideally, the design should be representable on many levels so that the designer can get the 'whole picture' at whatever level of detail is needed and minor changes in the design can be represented by minor changes in the model at the right level. The problem here is that a complete interpretive semantic model, e.g. VDL [5], usually requires too much detail to be of much value at this level. The solution proposed is to initially model the language at a very high level using a model which allows a hierarchical and modular specification of the design. Then in successive passes a remodeling of different aspects of the design can take place, specifying the structures at lower and lower levels. This is similar to the approach taken in writing a structured program.

This remodeling process involves the introduction of more and more structuring into the features of the language. This might involve a design path through a variety of data structures starting at the highest level (level of least specificity) and continuing through the design of the implementation of these features. Aside from the design benefits afforded by a semantic model, it also yields a formal definition of the semantics of the language.

Phase 3 of the design process is the design of the implementation of the language. This has been the traditional use of interpretive semantic models. The model for the compiler design in phase 3 flows naturally from the model of the language design in phase 2 by continuing the remodeling process to the level of an implementation. A formal model for the design of the compiler enhances the clarity of the design and provides a vehicle for proving the correctness of the compiler.

In order to illustrate the approach, assume that a high level language for numerical analysis problems is to be designed. In phase 1 of the language design, informal decisions are made involving the data primitives of the language. Assume that since the primary applications involve the manipulation of matrices, matrices are chosen as data primitives of the language.

Phase 2 includes the modeling of the data primitives at a high level. For example, a matrix could be modeled by a set whose elements are tuples (x, y, z) where x and y denote row and column indices and z is the element value. Matrix operations would then be defined in terms of set operations. Thus the sum of two matrices, represented by sets

A and B, is represented by the set

$$\{(x, y, z) \mid (x, y, z_1) \in A, (x, y, z_2) \in B, \text{ and } z = z_1 + z_2\}.$$

This defines the matrix language primitives without imposing structure.

Pass 2 can be iterated to allow refinements of the model, perhaps introducing more structure visible to the user, depending upon the design goals of the language. Pass 3 would then define the additional structuring needed for an implementation.

The following sections describe this process in more detail as applied to the development of GRAAL.

3. PHASE 1—INFORMAL LANGUAGE SPECIFICATION FOR GRAAL

The first pass in the design of GRAAL consisted of a general specification of the language primitives. It was decided that a strictly set theoretic development of graph theory would allow for considerable flexibility in the selection of storage representations for different graph structures (see [1] for a motivation of this decision). Therefore two primitives that were included in the language were sets and graphs.

Sets, however, were placed in GRAAL mainly for the purpose of defining graphs and their specific design was motivated by this. Their introduction into the language generated the need for several set operations. These include the standard set union ($\cup$), intersection ($\cap$), difference ($\sim$), and symmetric sum (Δ). There is a **subset** operator which constructs the subset of all elements of a set that satisfy a specified Boolean relation. There are some common relational operators such as $=$, $\neq$, and $\subseteq$, which return the value **true** if two sets are equal, not equal, or one is a subset of the other, respectively, and the value **false** otherwise.

There are a variety of graph structures available in GRAAL distinguished by the family of graph operators provided for building and traversing each specific structure. Here we will only discuss the **undirected graph** in node form.

The operators for building graphs are **assign**, which attaches a node or two nodes and their connecting arc to a graph, and **detach**, which removes a set of nodes from a graph. Both operators return graphs as a result. The operator **nodes** returns the set of all nodes in a graph. The only graph connectivity operator for the undirected graph is the adjacency operator **adj**. It takes as an argument a set s of nodes and a graph g and returns the set of all nodes adjacent to any member of s with respect to g.

The data structures for the language are arrays and lists. These two structures are used for the static and dynamic storage needs of the language.

The control structure is the standard algebraic language control structure since the purpose of the language is to write readable and easily expressed algorithms. In fact, the language could be imbedded in a standard algebraic language format which includes the normal integer and real variables, and integer and real arithmetic.

In a second pass of phase one the set and graph features of the language can be examined more closely. First a GRAAL set is a data type rather than a data structure. It is composed of elements, usually representing nodes or arcs, which are members of some universe of elements, just as the subelements of a string are characters which are members of some universe of characters. The difference is that the size of the universe of characters is usually thought of as fixed while the universe of elements must be highly dynamic. In order to define a new element, it must be dynamically created from this universe of elements.

Each element may have associated with it any number of typed properties. This permits the elements of a set or the nodes and arcs of a graph to have any number of integer, real or string values associated with them.

A graph is also a structured data type, which is composed of undeclared data elements which play the roles of nodes or arcs and are interrelated to define the particular graph structure.

Having made some basic informal language decisions, phase two may begin. This involves the modeling of language features in a formal but high-level manner. The next section defines the modeling facility that will be used.

4. THE MODEL DEFINITIONAL FACILITY

The interpretive semantic modeling facility used in the development of GRAAL is the Hierarchical Graph Language (HGL) [4]. It has the major benefit of supporting a variety of data structures, including graphs, and a variety of control structures. Thus it can easily be used to define models at a variety of levels.

HGL is a definitional facility for specifying the semantics of a programming language by defining an interpretive model for the language. A model defined in HGL is an operational semantic model [6] which is defined over user-specified control and data structures, which are in turn defined over a set of basic HGL primitives. These basic HGL primitives consist of (1) a set of nodes, each with an associated structured value, atomic value, and a set of attributes, and (2) a set of primitive transition functions that can change the structure, atom, or attribute set associated with a node and can define and delete nodes. This hierarchical system permits a relatively modular breakup of the language structures and a variety of levels of description within the individual modules.

More specifically, each node in HGL can be thought of as a high-level memory cell representing a unit of information about the language. The atomic value associated with the node usually represents some nonstructured program or data element. The structured value associated with a node usually represents some language component which must be structured and whose structure is of interest. The attributes associated with a node usually represent characteristics or properties of the unit of information contained in a node. These attributes can be thought of as a symbol table which might contain the compile time properties known about that unit of information.

The data and control structures imposed over the HGL primitives are defined by the user. The data structure is specified by the definition of some particular graph structure over the nodes along with a set of construction and accessing primitives that permit the building and traversing of that graph structure. Thus the data structures defined for a particular model may be sets, lists, trees, directed graphs, etc., as appropriate for the level of specification. Once the data structures to be used in the model have been chosen, an appropriate set of construction and accessing primitives must be defined. If the data structure is represented by a set, the construction and accessing primitives might be the standard set-theoretic operators acting on sets of nodes. If the data structure is represented by a list, then the construction and accessing primitives might be the LISP car, cdr and cons operators acting on lists of nodes.

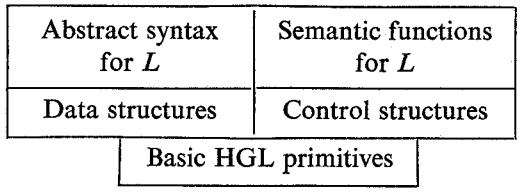
The particular control structure is specified by the definition of a meta-control language defined over the HGL primitive transition functions and the graph accessing and construction primitives. The definition of this meta-control language consists of a specification of

both the syntax of the language and its interpreting algorithm which defines the actual state transitions.

The choice of a control structure is usually dependent upon the data structure used. It may vary from some set-theoretic expressions, if the data structures are sets, to LISP-like recursive functions, if the data structures are lists, to some standard algebraic programming language control structures, if the data structures are arrays or even lists or sets.

The semantic model for a particular language L is then defined over some specific data structure and control structure. This is done by defining the mapping of the strings of L into the data structures (specifying the set of initial states) and defining a set of semantic functions in the meta-control language.

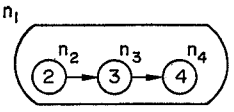
The HGL framework for building a model for a particular language is represented by the following figure, where each block is built using the definitions of the ones below it in conjunction with its adjacent block.



The execution of a program in the model is defined in the usual manner as a sequence of states. Each state specifies a particular snapshot of the program at some point in its execution. The order of the sequence of states is determined by the algorithm that interprets the meta-control language. An actual state transition is specified by one of the HGL primitive transition functions which defines the change that is to occur in any state that transforms it into a new state.

The association between a node and a structure value, atomic value and attributes can be formally defined as mappings. Let $\mathcal{N}$ be a set of nodes, $\mathcal{D}$ be a set of atoms, and $\mathcal{A}$ be a set of attributes. Let $\mathcal{G}$ be a set of graphs defined over elements of $\mathcal{N}$. An *attributed hierarchical graph* (*h-graph*) over $(\mathcal{N}, \mathcal{D}, \mathcal{A}, \mathcal{G})$ is a 4-tuple (N, v, h, a) where $N \subset \mathcal{N}$, $v: N \to \mathcal{D}$, $h: N \to \mathcal{G}$, and $a: N \times I_k^+ \to \mathcal{A}$ where I_k^+ denotes the first k positive integers.

As an illustration of the mappings, consider as a possible representation for an array the model given in Fig. 1. The array represented is a one-dimensional array with three



(type: integer, form: array, id: A)

$h(n_1)$ = graph $g = (V_g, E_g, n_g, e_g) = (\{n_2, n_3, n_4\}, \{(n_2, n_3), (n_3, n_4)\}, n_2, \varphi)$

$v(n_1) = \Omega$
$a(n_1, \text{type}) = \text{integer}$
$a(n_1, \text{from}) = \text{array}$
$a(n_1, \text{id}) = A$

$h(n_2) = \Omega$
$v(n_2) = 2$
etc.

Fig. 1

elements whose values are 2–4 respectively. The model is represented pictorially by a sort of hierarchical flow chart which gives a somewhat intuitive flavor to the array structure. Here the graph structure used is a directed graph which is defined as a 4-tuple (V_g, E_g, n_g, e_g) where $V_g \subset \mathcal{N}$ is the set of vertices, $E_g \subseteq V_g \times V_g$ is a set of edges, $n_g \in V_g$ is a distinguished vertex called the entry vertex, and $e_g: E_g \to \mathcal{D}$ is the edge label mapping.

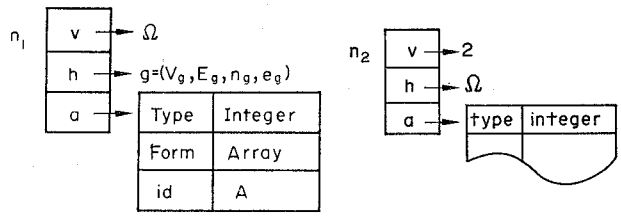


Fig. 2

Figure 2 presents another way of visualizing the same mappings which is more suggestive of an implementation. Here each node is considered as being represented by three pointers: one to its atomic value, one to its structural value, and one to its table of attributes.

As another example, consider the illustration in Fig. 3 as a possible representation for an if-statement. The if-statement takes the form **if** expression **then** statl **else** stat2.

A state in the sequence of states is essentially represented by an *h*-graph, which in turn represents a program and its data structures at some point in the execution. A state transition effects 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*, and *a* mappings are *setv* which assigns an atomic value to a node, *seth* which assigns a graph value to a node, and *seta* which assigns an attribute to a node. The two functions that alter the set *N* are *define* which adds a new node to the nodeset of the *h*-graph, and *delete* which removes a node from the nodeset of the *h*-graph. These five functions are more formally defined in Appendix 1.

HGL has been used in defining the base language SIMPL-X upon which a new version of GRAAL is being built as an extension to the language. Thus the basic control structures for SIMPL-X and GRAAL are similar. A more formal definition of HGL and the control structure for SIMPL-X may be found in [4].

5. PHASE 2—FORMAL LANGUAGE DESIGN FOR GRAAL

We shall now begin a high-level description and design of the set and graph aspects of GRAAL. As the paper progresses, the data and control structures used in defining the various models will vary. At the highest level these data structures are just sets of nodes and the control structure set-theoretic expressions. As the level of the design gets lower and lower, or attempts are made to define the implementation, the data structures become more and more structured and the control structures vary accordingly.

Pass one—Before we begin, let us make some remarks about notation. Each syntactic structure and semantic function definition has been assigned a reference tag which consists of a letter followed by a number. The letter specifies the class of a definition and the number

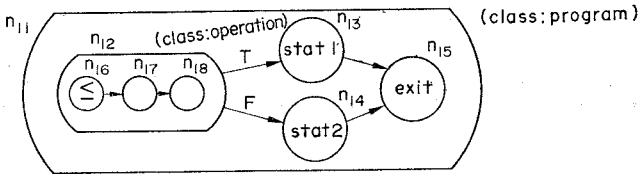


Fig. 3

is used to distinguish among the various definitions within a class. The classes are specified as follows: *S* refers to a syntactic structure definition, *F* and *G* refer to set and graph semantic function definitions, and *B* refers to some basic data-dependent function definition. Refinements of definitions all have the same letter and initial number but with the revision number appended to the reference. Thus F4.1 refers to the first revision of the set semantic function 4.

We must now begin to define the model specifying the data and control structures used in our high-level machine. The data structure used is a set. A *set* is defined as a collection of nodes from *N*.

The *construction primitives* for a set consist of (a) enumerating the elements of the set, e.g. $\{n_1, n_2\}$, (b) defining sets in terms of other sets using the standard set-theoretic operators such as $(\cup)$, $(\cap)$, (—), etc., (c) defining a set by a predicate, i.e. $\{x \mid BE(x)=true\}$ where *BE* is some Boolean expression on the set.

An *accessing primitive* must be given which selects a single element from a set of elements. The primitive *member*(s) is defined to return a randomly chosen element from the set *s*. This choice must be random since there is no ordering on the elements of a set.

The control structures used consist of (a) set expressions using the set construction and accessing primitives along with the five HGL primitive functions, (b) a linear sequence of statements of type (a) which are performed in sequential order. Each statement or subsequence of statements may be preceded by a conditional implying the statement or subsequence is performed only if the conditional is true.

We use these data and control structures to define a model (high-level machine architecture) for the set and graph features of GRAAL. The memory is essentially an associative memory which can be accessed by set operations returning the node or nodes which satisfy some specified requirements. It will also be assumed that each node in memory has a unique name associated with it, called the *id* attribute of that node. This id-attribute will be used to access a node when the node id is known; otherwise some set operation will be used. For example, $h(n)$, where n is a node, may sometimes be written as $h(id(n))$, i.e. h(name) will be shorthand notation for h(member($\{n \mid n \in N \wedge id(n)=$name$\}$)).

Let the universe of elements be defined by a node, with id-attribute *U*, which contains all the elements of the universe. Each element can then be represented as a node of *U* having an id-attribute which distinguishes it from all other element nodes. Assume the *id* of each element node takes the form u_i where i represents a unique integer ordering of the elements. In what follows, an extended BNF notation is used to define the syntactic structures of the model. Its use is explained as the examples proceed. Basically, it uses brackets $[x]$ to represent a node whose v or h mapping is defined by x. The distinction between the v and h mappings is made by using an underscore to designate the v mappings. We will use the term *content* of a node when it is immaterial as to whether we are referring to the v or h mapping of that node. Thus

(S1) $\quad \langle \text{universe} \rangle ::= [\{\langle \text{element} \rangle\}];\ id = U$

defines ⟨universe⟩ to be a node with one attribute *id*, equal to *U*, whose h mapping is a set of ⟨element⟩'*s*, i.e. $h(U)$ = a set of elements.

Properties may then be associated with an element by setting the h mapping of that element node to a set of nodes called property nodes. The content of a property node is the property value, and the attributes represent the name and type of that property. The property may be an atomic value, if the type is integer or real for example, or it may be

a structure if the type is set or graph. Thus we may define.

(S2) $\langle \text{element} \rangle ::= [\{\langle \text{property} \rangle\}]$; $id = \langle \text{unique name } u_i \rangle$

(S3) $\langle \text{property} \rangle ::= [\langle \text{atomorstructure} \rangle]$; $id = \langle \text{identifier} \rangle$, $type = \langle \text{type} \rangle$

(S4) $\langle \text{atomorstructure} \rangle ::= \langle \underline{\text{value}} \rangle \mid \langle \text{structure} \rangle$

(S5) $\langle \text{structure} \rangle ::= \langle \text{set} \rangle \mid \langle \text{graph} \rangle \mid \ldots$

(S6) $\langle \text{value} \rangle ::= \langle \text{identifier} \rangle \mid \text{number} \rangle \mid \ldots$

A GRAAL set may then be modeled by a node whose content is a set of element nodes $\{u_i, \ldots, u_k\}$. It should be noted that the nodes in the set are not copies of the element nodes but the actual element nodes. (In terms of an implementation, a set can be thought of as containing pointers to the actual elements in the universe).

(S7) $\langle \text{set} \rangle ::= [\{\langle \text{element} \rangle\}]$; $id = \langle \text{identifier} \rangle$, $type =$ **set**

For convenience let us assume that there is a node with *id*-attribute SETS, such that $h(\text{SETS}) =$ the nodes defining all the sets in the language, i.e.

(S8) $\langle \text{sets} \rangle ::= [\{\langle \text{set} \rangle\}]$; $id = \text{SETS}$

The GRAAL set operators can now be defined using the above definitions of the set and universe data structures and the set construction and accessing operators. The GRAAL operators are in bold type to distinguish them from the set-theoretic ones:

create which creates a single element set, called an *atomic set*, from a newly specified element in the universe

subset which creates a set of elements from elements in the universe that satisfy some Boolean condition

elt which creates an atomic set by randomly selecting an element from a designated set

$\cup$ the set union operator

$\cap$ the set intersection operator

$\sim$ the set difference operator

Δ the symmetric sum operator

$=$ the equivalence predicate

$\neq$ the nonequivalence predicate

$\subseteq$ the subset predicate

These operators may be defined in terms of the model as follows:

(F1) $\textbf{create} = \{\text{member}(\{n \in h(U) \mid v(n)=h(n)=a(n)=\Omega \wedge n \notin h(S), \text{ for all } S \in h(\text{SETS})\})\}$

where Ω represents an undefined value

(F2) $\textbf{subset}(x, BE(x)) = \{n \mid BE(n)=true\}$,

where *BE* is any valid Boolean expression in the language

(F3) $$\textbf{elt}(S) = \begin{cases} \{\text{member}(S)\} & \text{if } S \neq \phi \\ \phi & \text{if } S = \phi \end{cases}$$

(F4) $\cup(S, T) = h(S) \cup h(T)$

(F5) $\cap(S, T) = h(S) \cap h(T)$

(F6) $\sim(S, T) = h(S) \sim h(T)$

(F7) $$\Delta(S, T) = (h(S) - h(T)) \cup (h(T) - h(S))$$

(F8) $$= (S, T) = \begin{cases} \textit{true} \text{ if } h(S) = h(T) \\ \textit{false} \text{ otherwise} \end{cases}$$

(F9) $$\neq (S, T) = \begin{cases} \textit{true} \text{ if } h(S) \neq h(T) \\ \textit{false} \text{ otherwise} \end{cases}$$

(F10) $$\subseteq (S, T) = \begin{cases} \textit{true} \text{ if } h(S) \subseteq h(T) \\ \textit{false} \text{ otherwise} \end{cases}$$

It is also necessary to define a function for accessing a property of an element node of a set. This is done by the two-argument function *access* whose first argument is the *id* of a property node and whose second argument is an atomic set containing the element whose property is desired. The function returns the property node itself.

(F11) $$\textit{access}(\text{prop,set}) = \begin{cases} \text{member}(\{x \mid x \in h(\text{member(set)}) \wedge id(x) = \text{prop}\}) \\ \qquad \text{if } \mid \text{set} \mid = 1 \wedge \text{member(set)} \in h(U) \\ \textit{undefined} \text{ otherwise} \end{cases}$$

In order to make what follows more concise and at the same time to separate the data structure dependent primitives from the general algorithm, consider the following auxiliary functions: insert, add, and sub. Assume p is a property name, a is an atomic set, s is a set and n is a node.

(B1) insert(p, a) = *seth*(member(a), h(member(a))$\cup$\{*define*($m \mid id(m) = p \wedge type(m) =$ **set**)\})
/* insert adds to the element in a, a property node with $id = p$ and $type =$ **set** */

(B2) add(n, s) = *seth*(n, $h(n) \cup s$)
/* add unions the set of nodes s and the contents of n */

(B3) sub(n, s) = *seth*(n, $h(n) - s$)
/* sub removes the set of nodes s from the contents of n */

It was stated in Section 2 that in GRAAL a graph is defined by its operators. A graph may then be modeled by the set of nodes which compose it. The actual graph structure of a graph g may be modeled by letting each of its component elements contain as property nodes the sets of nodes or arcs defined by the relevant graph operators on that component, relative to the graph g. For example, an undirected graph in node form would be modeled by a node whose type attribute is **undirected graph** and whose id-attribute is the name of the graph, say g. This node would contain a single node representing the nodes of g and whose id would be nodes $\cdot g$ (a convenient way of chaining unique names). Then each of the elements in the node called nodes $\cdot g$ would contain one property node of type **set** containing the set of nodes which form the adjacency of that node in g, whose id-attribute is $a \cdot g$. This approach permits the definition of graphs without imposing any specific data structure on them. In what follows g will represent an undirected graph, a_i atomic sets, and s_i any set.

(S9) ⟨undirected graph in node form⟩ ::= [⟨nodes⟩]; *id* = ⟨identifier⟩,
type = **undirected graph**

(S10) ⟨nodes⟩ ::= [{⟨element⟩}]; id = nodes · ⟨identifier⟩

For each **undirected graph** g, the graph structure will be defined as follows: for all elements $x \in h(\text{nodes} \cdot g)$, there will be a property node $n \in h(x)$ such that $id(n) = a \cdot g$, $type(n) =$ **set**. This node represents the set of adjacency nodes for that element x and g. This node is defined by the appropriate graph construction operators.

The graph operators may then be defined as follows:

(G1) **nodes**$(g) = \{n \mid n \in h(\text{nodes} \cdot g)\}$

(G2) **assign**$(g, a_1, a_2) =$
$a_1 \not\subseteq$ **nodes**$(g) \rightarrow$ add(nodes · g, a_1)
insert($a \cdot g$, a_1)
$a_2 \not\subseteq$ **nodes**$(g) \rightarrow$ add(nodes · g, a_2)
insert($a \cdot g$, a_2)
add($access(a \cdot g, a_1), a_2$)
add($access(a \cdot g, a_2), a_1$)

(G3) **assign**(g, a_1)
$a_1 \not\subseteq$ **nodes**$(g) \rightarrow$ add(nodes · g, a_1)
insert($a \cdot g$, a_1)

(G4) **detach**$(g, s) =$
sub(nodes · g, s)
sub($access(a \cdot g, \{x\})$, s), $\forall\, x \in$ **nodes**(g)
sub(y, $\{a \cdot g\}$), $\forall\, y \in s$

(G5) **detach**$(g, s_1, s_2) =$
sub($access(a \cdot g, \{x\}), s_1$), $\forall\, x \in s_2$
sub($access(a \cdot g, \{x\}), s_2$), $\forall\, x \in s_1$

(G6) **adj**$(s, g) = \bigcup_{x \in s} h(access(a \cdot g, \{x\}))$

Pass two—A high level definition of the set and graph features of GRAAL has now been given. It assumed that the user had no knowledge of the data structure used for the sets and graphs other than the set structure given. However, a data structure was then imposed on sets and graphs by the language designers in order to give the user more control over the system and help him attain more efficient programs.

The data structure imposed (from the user's point of view, though not necessarily imposed on the implementation) was a list structure. This allowed the user to have control over the creation of new elements in the universe in a very practical way, and it permitted the user, as well as the system, to make use of the fact that sets could now have an internal ordering. This permitted the user an effective method for sequencing through sets and guaranteed reproducible results when elements of sets were accessed.

In order to impose a new data structure on GRAAL in the HGL model, a definition of the new data structure and control structure must be given, along with an appropriate set of construction and accessing primitives. Since LISP 1.5 [7] is a syntactically and semantically well-defined language for list processing, the *list* data structure used here will be defined as a simple LISP list in which the elements of the list are only atoms (nodes in our case) and never sublists, and the control structures will be defined as the recursive functions of the LISP metalanguage.

The *construction primitive* is the "cons" operator which concatenates a node onto the front of a list and the *accessing primitives* are the "car" and "cdr" operators which return the first element of its list argument and the list consisting of its list argument minus its first element, respectively.

The control structure is the recursive function definition given by the LISP metalanguage and which includes the conditional expression. The two predicates "eq" and "atom" are also assumed. "eq" returns a *true* if its arguments are both the same node and a *false* otherwise. The predicate "atom" returns a *true* if its argument is a node and a *false* otherwise.

Using the LISP metalanguage and primitives, a set of functions will now be defined which will be used as the basis for redefining GRAAL over the list structure. The formal definition of these functions is found in Appendix 2:

find (i, l) returns the ith element of list l
place$(n, l, 1)$ returns the index of element n in list l
choose(name, l) returns the element in list l with id = name
append(l, n) adds node n onto the end of list l
is-member(n, l) returns a *true* if n is an element of list l and *false* otherwise
union(x, y) returns a list, the set of whose elements is the union of the set of elements in the list x and the set of elements in the list y
difference (x, y) returns a list, the set of whose elements is the difference of the set of elements in the list x and the set of elements in the list y
return list(name, l) returns the union of all lists contained in nodes of l with id = name.

Using the list data structure, syntax definition (S1) may be replaced by

(S1.1) ⟨universe⟩ ::= [⟨element⟩{→⟨element⟩}] | [ ]; $id = U$

where the righthand side of S1.1 defines a node whose content is a list of elements or empty. Again the *id*-attribute associated with the node is U.

Since the universe is now a list, a new element may be created by defining a new node and adding it onto the end of the list. The **create** function (F1) may be redefined as:

(F1.1) **create** = append$(h(U),$ *define*$)$

This list structure was imposed on the language design primarily for the purpose of giving the user some control over both the universe and sets by knowing that their elements were in fact ordered in some way. To aid in this control a function **atom** was made available which allows the user to index into the universe. **atom** (i) returns the ith element of the universe. It is defined as follows:

(F12.1) **atom**(i) = find$(i, h(U))$

Imposing the list structure upon sets changes syntax definition (S7) to

(S7.1) ⟨set⟩ ::= [⟨element⟩{→⟨element⟩}] | [ ]; id = ⟨identifier⟩, *type* = **set**

The set operations such as union, etc., may then be redefined, e.g.

(F4.1) **U**(S, T) = union$(h(S), h(T))$

The major effect of defining GRAAL sets over a list data structure is that it provides the user with the ability to access set elements in a non-random way. The **elt** function (F3)

which returned a random element from a set may now be redefined to make use of the structuring. By adding an index argument, the user may sequence through the list by specifying any one of the elements according to its internal ordering. This guarantees both the duplication of a choice of an element of a set and the ability to access a sequence of unique elements by subsequent calls on the **elt** function. The **elt** function may be redefined as

(F3.1) $$\textbf{elt}(i, S) = \text{find}(i, h(S))$$

Conversely, given a particular set, a user may want the index of a particular element in the set. The **index** function has been supplied for this purpose. It takes two arguments; the first is an atomic set containing the element whose index is required in the set given as the second argument. Obviously, this function has no analogue in the definition of GRAAL over a set structure.

(F13.1) $$\textbf{index}(a, S) = \text{place}(h(a), h(S), 1)$$

Defining properties in a list structure changes the definition of ⟨element⟩ in (S2) and the *access* function (F11).

(S2.1) $$\langle\text{element}\rangle ::= [\langle\text{property}\rangle\{\rightarrow\langle\text{property}\rangle\}] \mid [\ \ ]$$

(F11.1) $$\textit{access}(\text{prop}, S) = \text{choose}(\text{prop}, h(S))$$

Changing the definition of node set to be a list alters definition (S10).

(S10.1) $$\langle\text{nodes}\rangle ::= [\langle\text{element}\rangle\{\rightarrow\langle\text{element}\rangle\}] \mid [\ \ ];\ id = \text{nodes} \cdot \langle\text{identifier}\rangle$$

Finally definition (S5) must be changed since all structures in the language are now lists.

(S5.1) $$\langle\text{structure}\rangle ::= [\langle\text{atomorstructure}\rangle]\{\rightarrow[\langle\text{atomorstructure}\rangle]\}$$

The GRAAL graph-connectivity operator **adj** must also be redefined.

(G6.1) $$\textbf{adj}(S, g) = \textit{define}(m \mid \textit{type}(m)=\textbf{set} \wedge h(m)=\text{returnlist}(a \cdot g, h(S))).$$

The data structure dependent functions, insert, add, and sub, may be redefined in order to use the previous definitions of the GRAAL operators **assign, detach**, etc.

(B1.1) $$\text{insert}(p, a) = \textit{seth}(\textbf{elt}(1,a), \text{union}(h(\textbf{elt}(1,a)), \textit{define}(m \mid \textit{id}(m)=p, \textit{type}(m)=\textbf{set})))$$

(B2.1) $$\text{add}(n, s) = \textit{seth}(n, \text{union}(h(n), s))$$

(B3.1) $$\text{sub}(n, s) = \textit{seth}(n, \text{difference}(h(n), s))$$

This completes the second pass of the design of GRAAL. It is, in fact, a formal definition of the semantics of the set and graph-related components as they exist in the language.

It should be noted that if the design ended at pass one, pass two would be a first pass on the implementation of the language. It does in fact give strong suggestions for the actual implementation of the language as will be seen in Section 6.

6. PHASE 3—IMPLEMENTATION DESIGN FOR GRAAL

This last phase, which is concerned with the design of the implementation of the language, is really separate from the two previous phases which were concerned with the design of the language itself. However, having a design for the language in a hierarchical operational

semantic model, there is a natural bridge to defining a high-level design of an implementation of that language. This can be done within the framework of that same semantic model by further refining the data and control structures. This process would permit the designer to test a variety of data structures for implementation of the various features of the language with a minimum of cost. It also supports a structured implementation design which uses the modularized language design given in the previous phase. This high-level implementation specification can also provide a good guide to implementations of the language on a variety of machines. As a further aid to this end, the control structures used at this level might be those of a machine-independent compiler–writing language, e.g. SIMPL-T [8].

Another benefit of developing the implementation design from the language design within the same operational semantic model facility is that this process lends itself to attempts at proving the correctness of the compiler design by certifying its equivalence to the language design. Aspects of this equivalence may be performed by applying the twin-machine proof techniques [9–11] or any of the other mapping equivalence techniques [12, 6]. The modularization and structuring of the design techniques used here make these mapping proof techniques quite fruitful.

In order to define the implementation design for GRAAL, let us first consider a redefinition of the data structures. A natural data structure for the universe of elements would be a dynamic array-like structure which would permit an easy and quick ordering and accessing scheme for the elements. This can be done by using the index of the element in the array as its unique (integer) name. This would lead to a relatively efficient implementation of the universe and of sets.

Sets should still use a list-type data structure because of their dynamically varying size. These lists can be ordered, however, according to the unique integer name associated with each element in the universe. This intrinsic ordering permits a great improvement in the efficiency of the set operations [13].

Properties, like sets, should also be defined by a list structure since there are a variable number of properties associated with each element. This tends to be efficient with respect to space but inefficient with respect to accessing speed.

The new control structure is an iterative structure similar to the standard algebraic language control structure. The statement structures used here will be the assignment statement, the if-then-else statement and the while statement. Each function will be written as a function procedure which will include both parameters and local variables.

The two data structures used in the implementation are lists and arrays. Lists were used in pass two of the design phase, but we will look at them in a new way since the control structure is no longer a recursive function structure but an iterative one. A new definition for a list will be given which is more in line with the new control structures and the accessing and construction primitives will be redefined in terms of this new definition.

A *list* is now defined to be a directed graph $l = (N, E, n_f, n_l)$ where N is the set of nodes, $n_f, n_l \in N$ are distinguished nodes called the first and last node respectively, and E is the set of edges defined by the node pairs (n, m), $n, m \in N$ such that each node in N, except n_l, appears as the first element of a node pair defining an edge exactly once, and each node in N, except n_f, appears as the second element of a node pair defining an edge exactly once.

The list *accessing primitives* defined for list $l = (N, E, n_f, n_l)$ are

$$\text{first } (l) = \begin{cases} n_f, \text{ if } l \text{ is not empty} \\ \textit{errornode}, \text{ if } l \text{ is empty} \end{cases}$$

where *errornode* represents a special node (element) that can be checked for as an error with $id = 0$

$$\text{next}(n, l) = \begin{cases} m \text{ such that } (n, m) \in E, \text{ if } n \neq n_l \\ errornode, \text{ if } n = n_l \end{cases}$$

The list *construction primitive* defined for list $l = (N, E, n_f, n_l)$ is append (l, n) which returns the list l' such that $l' = (N \cup \{n\}, E_g \cup \{(n_l, n)\}, n_f, n)$.

We will also use the primitive size$(l) = |N|$, i.e. the number of nodes in the list.

Thus the syntax for those language components that retain the list structure remain as they were defined in pass two of phase two.

An array may be defined in any of several ways. Two possible graph structures for an array are given in Fig. 4. The graph in Fig. 4(a) has an entry node which does not represent an element of the array and a set of edges emanating from that entry node leading to nodes representing the elements of the array. Each edge is labeled with the appropriate index of the array element. Figure 4(b) defines some additional edges on a list of nodes representing array elements, the entry node being the first element of the list and the node pointed to by the edge with label index 1. In either case a relevant set of construction and accessing primitives must be defined.

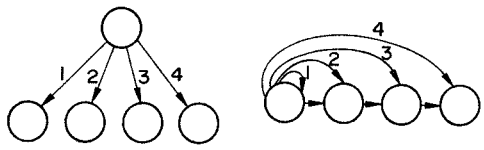


Fig. 4

Another way of defining an array data structure would be to define the structure of the array to be exactly the same as the list defined earlier but to define a more powerful set of accessing and construction primitives. For the purposes of this paper, this option would allow the syntactic definition for all language components to remain as they were defined in the last section.

Thus an *array* is a list $l = (N, E, n_f, n_l)$. The *accessing primitive* is arrayelement (l, i) which returns the ith element in the list l. This primitive may be defined in terms of the list primitives as follows:

```
node function  arrayelement(list l, integer i)
                  integer j node m
                  if size(l) ≠ 0 ∧ size(l) ≥ i
                    then m := first (l)
                      j := 1
                      while i ≠ j
                           do m := next(m, l)
                              j := j+1
                           end
                      return(m)
                           else return(errornode)
                     end
```

The *construction primitive* may then be the same as the list construction primitives, and we can think of the array as dynamic in the same sense as a list.

The universe is now a type array list where a new element may be created by defining

a new node and adding it on to the end of the list. The unique name associated with each newly created element can be its index in the array. This has the effect of mapping the universe of nodes into the positive integers to give them their unique names. The **create** function would then look as follows:

(F1.2) **create** = *define*(*m* | *type*(*m*) = **set**, *h*(*m*)=arrayelement(*seth*(*U*, append(*h*(*U*), *define*(*n* | *id*(*n*)=size(*h*(*U*) + 1)))), size(*h*(*U*)))

The **atom** primitive of GRAAL would be defined

(F12.2) **atom**(*i*)=arrayelement(*h*(*U*), *i*).

Sets may be internally ordered by their integer value unique names. This allows the definition of set operations and relations to be very efficient. For example, **union** may be redefined as follows:

(F4.2) **union**(*S*, *T*)=*seth*(*define*(*n* | *id*(*n*)=temp, *type*(*n*)=**set**), union(*h*(S), *h*(*T*))).

with the union of two lists defined as:

```
list function union(listS, listT)
  list R = nil node x, y integer Ssize, Tsize
    Ssize := size(S)
    Tsize := size(T)
    /* initialize x and y to the first elements of S and T */
    x := first(S)
    y := first(T)
    while Ssize ≠ 0 ∨ Tsize ≠ 0
      do /* build union of S and T in R */
        if Tsize = 0 ∨ (id(x) < id(y) ∧ Ssize ≠ 0)
          then /* S contains next element */
            R := append(R, x)
            Ssize := Ssize − 1
            x := next(x, S)
          else if Ssize = 0 ∨ id(x) ≠ id(y)
              then /* T contains next element */
                R := append(R, y)
                Tsize := Tsize − 1
                y := next(y, T)
              else /* S and T contain the next element */
                R := append(R, x)
                Ssize := Ssize − 1
                Tsize := Tsize − 1
                x := next(x, S)
                y := next(y, T)
              end
            end
        end  /* while */
      return(R)
```

All the other set operations and relations may be redefined in an analogous way to make use of the internal ordering of the elements. They will not be redefined here for the sake of brevity.

Once again the insert, add, and sub functions may be redefined to make the GRAAL graph building operators, e.g. **assign**, **detach**, etc., correct as they were in pass one of the design phase. The redefinition may be accomplished by substituting the union function as defined in this section for the union function used in (B1.1) and (B2.1) and by defining a new difference function and substituting it for the difference function in (B3.1).

This completes pass one of the implementation design. It may in fact be the only pass or it may be refined all the way down to the actual compiler program. For the purposes of demonstrating the design technique, however, we will stop here.

7. CONCLUSION

This paper presented an attempt at systematizing language design based on design and implementation experience with GRAAL. The methodology recommended was broken down into three phases: the informal language design, the formal language design and the implementation design.

The informal language design phase is basically a discussion stage where the various language constructs are chosen and their designs considered in an informal way. This phase is essentially a bottom-up approach of specifying what is needed in the language.

In the formal language design phase, the language is modeled by a sequence of hierarchical semantic models written in the hierarchical graph language HGL; each of these models is a variation of the one before with the data and control structures of the new model being more highly specified. This forces an essentially top-down approach to the formal design of the language, permitting the designer a tool at the top level for keeping the entire design in mind at one time. This allows the designer a framework for specifying the language constructs more formally, for examining in a relatively modular way the effect the various constructs have upon one another, and for estimating what a design change will cost and how large a change would be necessary to accomplish certain new goals. It provides him with a cross-section look at the design of his language at the top level and at successively lower levels.

The design of the implementation or the compiler within this framework is just the continued refinement of the language design phase using lower-level data and control structures. The designer can use this scheme to carry his design below the language level and lay out the design of the compiler for himself or other implementors. It permits the examination and definition of a variety of implementation designs.

The model schema used here, HGL, is particularly well-suited to this kind of design scheme since it supports an open-ended set of data and control structures. Because of the highly modularized nature of the models built in HGL, it is easy to flow from one design to the next and various language components at the same design level may use different structures in their definition. Thus building one design model from another can be accomplished in well-defined intermediate steps, giving the designer real control over all aspects of the process. Hopefully this was shown in this paper by the examples of the design models given for the graph and set aspects of GRAAL.

The main result of this approach to language design is (hopefully) a better language because of the better design tools. However the resulting formal definitions of both the language and the implementation in a formal semantic model are important side effects.

These formal definitions can be of great use to the implementor of the language as well as any user of the language to help explain what a particular construct actually does. It is also a great help to the designer in guaranteeing that the language does what he originally intended it to do. The fact that both the language and compiler design are in the same model provides us with some powerful tools in trying to prove the equivalence of the various language designs as well as the equivalence of the language and its implementation.

SUMMARY

This paper is an attempt at systematizing language design based on design and implementation experience with GRAAL, a high-level programming language for use in the solution of graph problems primarily arising in applications. A sequence of hierarchical semantic models is recommended for designing the language and its compiler, and the approach is demonstrated by applying it to the set and graph features of GRAAL.

This sequence of hierarchical semantic models, specified in the hierarchical graph language HGL, is in effect a sequence of very high-level machines, each of which is a refinement of the one before with the data and control structures for the new machine being more highly specified. This permits language design to be approached in a top-down manner.

Essentially HGL is a definitional facility for specifying models which may be used to describe the semantics of programming languages. These models are operational semantic models which are defined over user-specified control and data structures, which are in turn defined over a set of basic HGL primitives. These basic HGL primitives consists of (1) a set of nodes, each with an associated structured value, atomic value, and a set of attributes, and (2) a set of primitive transition functions that can change the structure, atom, or an attribute associated with a node and can define and delete nodes. This hierarchical system permits a relatively modular breakup of the language structures and a variety of levels of description within the individual modules. More specifically, each node in HGL may be thought of as a high-level memory cell representing a unit of information about the language. The atomic value associated with the node usually represents some nonstructured program or data element. The structured value associated with a node usually represents some language component which must be structured and whose structure is of interest. The attributes associated with a node usually represent characteristics or properties of the unit of information contained in a node. These attributes can be thought of as a symbol table which might contain the compile time properties known about that unit of information.

The data and control structures imposed over the HGL primitives are defined by the user. The data structure is specified by the definition of some particular graph structure over the nodes along with a set of construction and accessing primitives that permit the building and traversing of that graph structure. The particular control structure is specified by the definition of a meta-control language defined over the HGL primitive transition functions and the graph accessing and construction primitives. The definition of this meta-control language consists of a specification of both the syntax of the language and its interpreting algorithm which defines the actual state transitions.

The topdown approach recommended here can be broken down into three major phases, each of which is in turn organized in a top-down manner. Top down in this case means adding more detail or more structure at each step in the process. The first phase is the discussion stage which consists of an informal specification of the language components.

The second phase is the formal language design stage which is the formalization of the discussion stage. Phase three encompasses the high-level design of the implementation. No phase is ever frozen and work on any one phase may cause changes in the others. Only after several passes through these three phases are completed should the actual language specification (detailed syntax and semantics) and the compiler specification and implementation begin.

APPENDIX I—HGL STATE TRANSITION PRIMITIVES

Let $S_i = (N_i, v_i, h_i, a_i)$ be the h-graph representation of the program in state S_i and $(N_{i+1}, v_{i+1}, h_{i+1}, h_{i+1}, a_{i+1})$ be the h-graph representation of the program in state S_{i+1}. The primitives associated with state transition from S_i to S_{i+1} are defined below:

setv

The execution of *setv*(n, d) for $n \in N_i$ and $d \in D$ returns the node n and generates the state S_{i+1} whose only new component is v_{i+1} defined by

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

(*setv* assigns an atomic value to a node and returns the node)

seth

The execution of *seth*(n, g) for $n \in N_i$ and $g \in G$ returns the node n and generates the state S_{i+1} whose only new component is h_{i+1} defined by

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

(*seth* assigns a graph structure to a node and returns the node)

seta

The execution of *seta*(n, j, p) for $n \in N_i$, $j \in I_k^+$, and $p \in A$ returns the node n and generates the state S_{i+1} whose only new component is a_{i+1} defined by

$$a_{i+1}(m, l) = \begin{cases} p, & m = n,\ l = j \\ a_i(m, l), & \text{otherwise} \end{cases}$$

(*seta* assigns an attribute to a node and returns the node)

define

The execution of *define* returns a node $n \in N - N_i$ and generates the state S_{i+1} which is defined in terms of S_i as follows:

$$N_{i+1} = N_i \cup \{n\}$$

$$v_{i+1}(m) = \begin{cases} v_i(m), & m \in N_i \\ \textit{undefined}, & m = n \end{cases}$$

$$h_{i+1}(m) = \begin{cases} h_i(m), & m \in N_i \\ \textit{undefined}, & m = n \end{cases}$$

$$a_{i+1}(m, j) = \begin{cases} a_{i+1}(m, j), & m \in N_i, j \in I^+ \\ \textit{undefined}, & m = n \end{cases}$$

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

Note—define, as given here takes no arguments. However in the text of this report, another form is used:

$$\textit{define}(n \mid v(n)=d, h(n)=g, a(n, 1)=a_1, \ldots, a(n, r)=a_r)$$

which combines the additon of the new node to the h-graph along with its associated v, h and a mappings.

delete

The execution of *delete*(n) returns a node $n \in N_i$ and generates the state S_{i+1} which is defined in terms of S_i as follows:

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

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

APPENDIX 2—LIST STRUCTURE SUPPORT FUNCTIONS

(L1) find [*i*;*l*] =
[*i*≤0 → *nil*; null[*l*] → *nil*; *i*=1 → car[*l*];
true → find[*i*−1; cdr[*l*]]]

(L2) place[*n*;*l*;*i*] =
[null[*l*] → Ω;
eq[car[*l*]; *n*] → *i*;
true → place[*n*; cdr[*l*]; *i*+1]]

(L3) choose[name;*l*] =
[null[*l*] → *nil*;
id(car[*l*]) = name → car[*l*];
true → choose[name; cdr[*l*]]]

(L4) append[*l*;*n*] =
[null[*l*] → cons[*n*; *nil*];
true → cons[car[*l*]; append[cdr[*l*]; *n*]]]

(L5) is-member[*n*;*x*] =
[null[*x*] → *false*;
eq[*n*; car[*x*]] → *true*;
true → is-member[*n*; cdr[*x*]]]

(L6) union[*x*;*y*] =
[null[*x*] → *y*;
is-member[car[*x*]; *y*] → union[cdr[*x*]; *y*];
true → cons[car[*x*]; union[cdr[*x*]; *y*]]]

(L7) difference [*x*;*y*] =
[null[*y*] → *x*;
null[*x*] → *nil*;
is-member[car[*x*]; *y*] → difference [cdr[*x*]; *y*]
true → cons[car[*x*]; difference [cdr[*x*]; *y*]]]

(L8) returnlist[name,*l*] =
[null[*l*] → *nil*;
true → union[cons[choose[name; cons [car[*l*]; *nil*]]; *nil*]; returnlist[name,cdr[*l*]]]]

REFERENCES

1. W. C. Rheinboldt, V. R. Basili and C. K. Mesztenyi, On a programming language for graph algorithms, *BIT* **12**, 220 (1972).
2. V. R. Basili, C. K. Mesztenyi, and W. C. Rheinboldt, *FGRAAL—FORTRAN extended graph algorithmic language*, University of Maryland, Computer Science Center, Technical Rept. TR-179 (1972).
3. C. K. Mesztenyi, H. Breitenlohner and J. C. Yeh, *FGRAAL technical documentation*, University of Maryland, Computer Science Center, Technical Rept. TR-200 (1972).
4. V. R. Basili and A. J. Turner, A hierarchical machine model for the semantics of programming languages, *Proceedings, Symposium on High Level Language Computer Architecture*, ACM (November 1973).
5. P. Lucas *et al.*, Method and notation for the formal definition of programming languages, IBM Laboratory, Vienna, TR 25.087 (1968).
6. P. Wegner, Operational semantics of programming languages, *Proceedings, Conference on Proving Assertions about Programs*, ACM (January 1972).
7. J. McCarthy *et al.*, *LISP 1.5 Programmer's Manual*. MIT Press, Cambridge, MA. (1969).
8. V. R. Basili and A. J. Turner, *SIMPL-T: a Structured Programming Language*. University of Maryland, Computer Science Center, Computer Note CN-14 (1974).
9. W. Henhapl and C. B. Jones, *The Block Structure Concept and Some Possible Implementations with Proofs of Equivalence*. TR 25.104, IBM Laboratory, Vienna (April 1970).
10. C. B. Jones and P. Lucas, Proving correctness of implementation techniques, *Symposium on Semantics of Algorithmic Languages, Lecture Notes in Mathematics 188*. Springer, New York (1971).
11. P. Lucas, *Two Constructive Realizations of the Block Concept and Their Equivalence*. TR 25.087, IBM Laboratory, Vienna (1968).
12. C. McGowan, An inductive proof technique for interpreter equivalence, *Courant Institute Symposium on Formal Semantics of Programming Languages*. Prentice-Hall, Englewood Cliffs, N. J. (1971).
13. S. Shapiro, The list set generator, *CACM* **13**, 741 (1970).