Semantics of Context-Free Languages

by

DONALD E. KNUTH

California Institute of Technology

ABSTRACT

"Meaning" may be assigned to a string in a context-free language by defining "attributes" of the symbols in a derivation tree for that string. The attributes can be defined by functions associated with each production in the grammar. This paper examines the implications of this process when some of the attributes are "synthesized", i.e., defined solely in terms of attributes of the *descendants* of the corresponding nonterminal symbol, while other attributes are "inherited", i.e., defined in terms of attributes of the *ancestors* of the nonterminal symbol. An algorithm is given which detects when such semantic rules could possibly lead to circular definition of some attributes. An example is given of a simple programming language defined with both inherited and synthesized attributes, and the method of definition is compared to other techniques for formal specification of semantics which have appeared in the literature.

为 上下文无关语法定义的语言
指定含义

A simple technique for specifying the "meaning" of languages defined by context-free grammars is introduced in Section 1 of this paper, and its basic mathematical properties are investigated in Sections 2 and 3. An example which indicates how the technique can be applied to the formal definition of programming languages is described in Section 4, and finally, Section 5 contains a somewhat biased comparison of the present method to other known techniques for semantic definition. The discussion in this paper is oriented primarily towards programming languages, but the same methods appear to be relevant also in the study of natural languages.

1. Introduction. Let us suppose that we want to give a precise definition of binary notation for numbers. This can be done in many ways, and in this section we want to consider a manner of definition which can be generalized so that the meaning of other notations can be expressed in the same way. One such way to define binary notation is to base a definition on

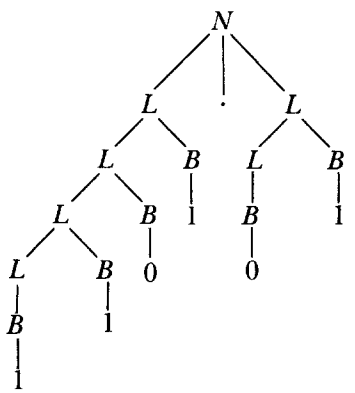
MATHEMATICAL SYSTEMS THEORY, Vol. 2, No. 2
Published by Springer-Verlag New York Inc.

the following context-free grammar:

$$
\begin{aligned}
&B \to 0\\
&B \to 1\\
&L \to B\\
&L \to LB\\
&N \to L\\
&N \to L \cdot L
\end{aligned}
\tag{1.1}
$$

(Here the terminal symbols are ·, 0, and 1; the nonterminal symbols are B, L, and N, standing respectively for bit, list of bits, and number; and a binary number is intended to be any string of terminal symbols which can be obtained from N by application of the above productions.) This grammar says in effect that a binary number is a sequence of one or more 0's and 1's, optionally followed by a radix point and another sequence of one or more 0's and 1's. Furthermore, the grammar assigns a certain *tree structure* to each binary number; for example, the string 1101 · 01 receives the following structure:

(1.2)



It is natural to define the meaning of binary notation (1.1) in a step-by-step manner corresponding to this structure; the meaning of the notation as a whole is built up from meanings of each part. This can be done by assigning *attributes* to the nonterminal symbols, as follows:

Each bit B has a "value" $v(B)$ which is an integer.

Each list of bits L has a "length" $l(L)$ which is an integer.

Each list of bits L has a "value" $v(L)$ which is an integer.

Each number N has a "value" $v(N)$ which is a rational number.

(Note that each L has two attributes; in general we could ascribe any desired number of attributes to each nonterminal symbol.)

The grammar (1.1) may now be augmented so that semantic rules are

given for each rule of the syntax:

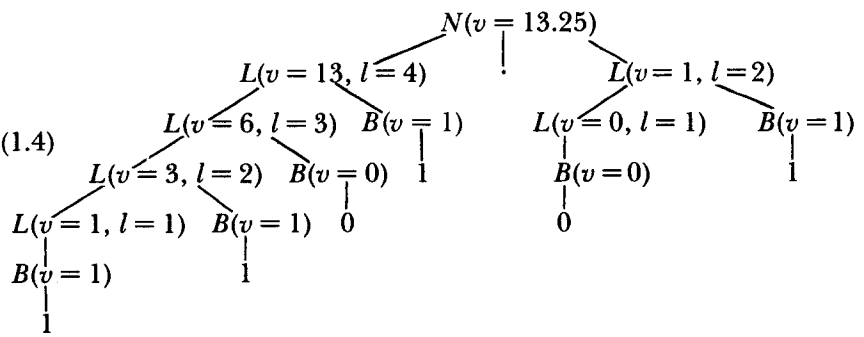
$$
\begin{array}{lll}
B \to 0 & v(B) = 0 & \\
B \to 1 & v(B) = 1 & \\
L \to B & v(L) = v(B), & l(L) = 1 \\
L_1 \to L_2 B & v(L_1) = 2v(L_2) + v(B), & l(L_1) = l(L_2) + 1 \\
N \to L & v(N) = v(L) & \\
N \to L_1 \cdot L_2 & v(N) = v(L_1) + v(L_2)/2^{l(L_2)} &
\end{array} \tag{1.3}
$$

语义规则定义所有属性

(In the fourth and sixth rules subscripts have been used to distinguish between occurrences of like nonterminals.) Here the semantic rules define all of the attributes of a nonterminal in terms of the atrributes of its immediate descendants, so ultimately values are defined for each attribute. The semantic rules are phrased in terms of notations which are assumed to be already understood. Notice for example that the symbol "0" in the semantic rule "$v(B) = 0$" is to be interpreted quite differently from the symbol "0" in the production "$B \to 0$"; the former denotes a mathematical concept, the integer zero, while the latter denotes a written character which has a certain elliptical shape. In a sense it is just coincidence that the two symbols look the same.

The structure (1.2) may be augmented by showing the attributes at each level:

(1.4)



Thus "1101 · 01" means 13.25 (in decimal notation).

新的定义方式

This manner of defining semantics for context-free languages is essentially well known, since it has already been used by several authors. But there is an important way to extend this method, and it is this extension which will be of primary interest to us.

Suppose for example that we want to define the semantics of binary notation in a different way corresponding more closely to the manner in which we usually think of the notation. The leading "1" in "1101 · 01" really denotes 8, although according to (1.4) it is ascribed the value 1. Perhaps therefore it would be better to define the semantics in such a way that

以一种不同的方式定义语义，更加接近我们如何思考这个概念

positional characteristics play a role. We could have the following attributes:

Each B has a "value" $v(B)$ which is a rational number.
Each B has a "scale" $s(B)$ which is an integer.
Each L has a "value" $v(L)$ which is a rational number.
Each L has a "length" $l(L)$ which is an integer.
Each L has a "scale" $s(L)$ which is an integer.
Each N has a "value" $v(N)$ which is a rational number.

These attributes can be defined as follows:

	Syntactic rules	*Semantic rules*
	$B \to 0$	$v(B) = 0$
	$B \to 1$	$v(B) = 2^{s(B)}$
	$L \to B$	$v(L) = v(B),\ s(B) = s(L),\quad l(L) = 1$
(1.5)	$L_1 \to L_2 B$	$v(L_1) = v(L_2) + v(B),\quad s(B) = s(L_1),$ $s(L_2) = s(L_1) + 1,\quad l(L_1) = l(L_2) + 1$
	$N \to L$	$v(N) = v(L),\quad s(L) = 0$
	$N \to L_1 \cdot L_2$	$v(N) = v(L_1) + v(L_2),\quad s(L_1) = 0,$ $s(L_2) = -l(L_2)$

等式的右端 是左端的定义

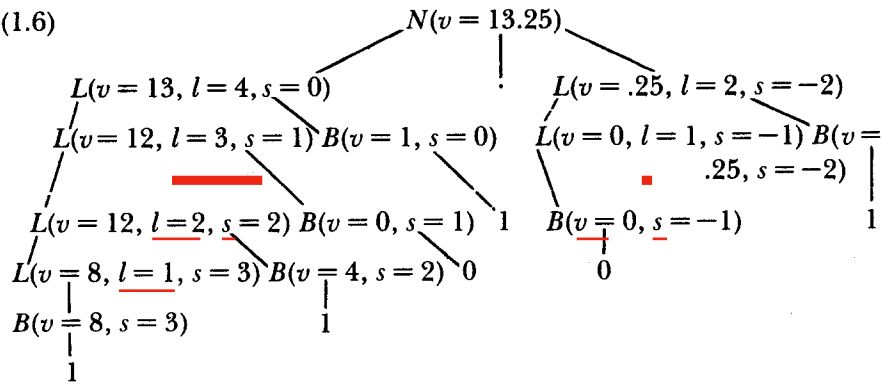
(Here the semantic rules are listed using the convention that the right-hand side of each equation is the definition of the left-hand side; thus, "$s(B) = s(L)$" says that $s(L)$ is to be evaluated first, then $s(B)$ is defined to have this same value.)

文本

The important feature of grammar (1.5) is that some of the attributes are defined for nonterminals which appear on the right side of the corresponding production, while in (1.3) all attributes were defined when the nonterminal appeared on the left side. Here we are using both *synthesized attributes* (which are based on the attributes of the descendants of the nonterminal symbol) and *inherited attributes* (which are based on the attributes of the ancestors). Synthesized attributes are evaluated from the bottom up in the tree structure, while inherited attributes are evaluated from the top down. Grammar (1.5) contains the synthesized attributes $v(B)$, $v(L)$, $l(L)$, $v(N)$ and also the inherited attributes $s(B)$ and $s(L)$, so the evaluation involves going in both directions. The evaluated structure corresponding to the string 1101 · 01 is

文本

(1.6)



Here it can be noted that the "length" attributes of the L's to the right of the radix point must be evaluated from the bottom up before the "scale" attributes can be evaluated (from the top down) and finally the "value" attributes (from the bottom up).

Grammar (1.5) is probably not the "best possible" grammar for binary notation, but it does seem to correspond better to our intuition than grammar (1.3). (A grammar which agrees more exactly with our conventional understanding of binary notation could be based on a different set of production rules which would assign another structure to the string of bits at the right of the radix point; then the "length" attribute, which is not really relevant, would be unnecessary.)

Our interest in grammar (1.5) is not that it is an ideal definition of binary notation, but rather that it shows an interaction between inherited and synthesized attributes. It is not always obvious when semantic rules such as those in (1.5) do not amount to a circular definition, because the attributes are not evaluated in a single direction; an algorithm which tests for circularity appears later in this paper.

The importance of inherited attributes is that they arise naturally in practice and that they are "dual" to synthesized attributes in a straightforward manner. Although binary notation can be formulated using nothing but synthesized attributes, there are many languages for which such a restriction leads to a very awkward and unnatural definition of semantics. Situations which involve a mixture of inherited and synthesized attributes are essentially the same as the cases which have been most difficult to handle in previous formulations of semantic rules.

2. Formal properties. Let us now put the ideas of synthesized and inherited attributes into a more precise and more general setting.

Suppose we have a context-free grammar $\mathscr{G} = (V, N, S, \mathscr{P})$, where V is the (finite) vocabulary of terminal and nonterminal symbols; $N \subseteq V$ is the set of nonterminal symbols; $S \in N$ is the "start symbol", which appears on the right-hand side of no production rule; and $\mathscr{P}$ is the set of production rules. Semantic rules are added to $\mathscr{G}$ in the following manner: To each

symbol $X \in V$ we associate a finite set $A(X)$ of attributes; $A(X)$ is partitioned into two disjoint sets, the synthesized attributes $A_0(X)$ and the inherited attributes $A_1(X)$. We require $A_1(S)$ to be empty (i.e., the start symbol S has no inherited attributes); similarly we require $A_0(X)$ to be empty if X is a terminal symbol. Each attribute α in $A(X)$ has a (possibly infinite) set of possible values V_α, from which one value will be selected (by means of the semantic rules) for each appearance of X in a derivation tree.

Let $\mathscr{P}$ consist of m productions, and let the p-th production be

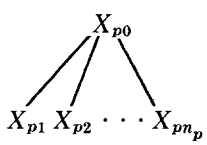
$$X_{p0} \rightarrow X_{p1}X_{p2} \cdots X_{pn_p}, \tag{2.1}$$

where $n_p \geqslant 0$, $X_{p0} \in N$, and $X_{pj} \in V$ for $1 \leqslant j \leqslant n_p$. The semantic rules are functions $f_{pj\alpha}$ defined for all $1 \leqslant p \leqslant m$, $0 \leqslant j \leqslant n_p$, and $\alpha \in A_0(X_{pj})$ if $j = 0$, $\alpha \in A_1(X_{pj})$ if $j > 0$. Each such function is a mapping of $V_{\alpha_1} \times V_{\alpha_2} \times \cdots \times V_{\alpha_t}$ into V_α, for some $t = t(p, j, \alpha) \geqslant 0$, where each $\alpha_i = \alpha_i(p, j, \alpha)$ is an attribute of some X_{pk_i}, for $0 \leqslant k_i = k_i(p, j, \alpha) \leqslant n_p$, $1 \leqslant i \leqslant t$. In other words, each semantic rule maps values of certain attributes of $X_{p0}, X_{p1}, \cdots, X_{pn_p}$ into the value of some attribute of X_{pj}.

For example, (1.5) is the grammar $\mathscr{G} = (\{0, 1, \cdot, B, L, N\}, \{B, L, N\}, N, \{B \rightarrow 0, B \rightarrow 1, L \rightarrow B, L \rightarrow LB, N \rightarrow L, N \rightarrow L \cdot L\})$. The attributes are $A_0(B) = \{v\}$, $A_1(B) = \{s\}$, $A_0(L) = \{v, l\}$, $A_1(L) = \{s\}$, $A_0(N) = \{v\}$, $A_1(N) = \varnothing$, and $A_0(x) = A_1(x) = \varnothing$ for $x \in \{0, 1, \cdot\}$. The attribute value sets are $V_v = \{\text{rational numbers}\}$, $V_s = V_l = \{\text{integers}\}$. A typical production rule is the fourth production $X_{40} \rightarrow X_{41}X_{42}$, where $n_4 = 2$, $X_{40} = X_{41} = L$, $X_{42} = B$. A typical semantic rule corresponding to this production is f_{40v}, which defines $v(X_{40})$ in terms of other attributes; in this case f_{40v} maps $V_v \times V_v$ into V_v, and it is the mapping $f_{40v}(x, y) = x + y$. (This is the rule "$v(L_1) = v(L_2) + v(B)$" of (1.5); in terms of the rather cumbersome notation of the preceding paragraph we have $t(4, 0, v) = 2$, $\alpha_1(4, 0, v) = \alpha_2(4, 0, v) = v$, $k_1(4, 0, v) = 1$, $k_2(4, 0, v) = 2$.)

The semantic rules may be used to assign a "meaning" to strings of the context-free language, in the following way. For any derivation of a terminal string t from S by a sequence of productions, construct the derivation tree in the usual way: The root of this tree is S, and each node is labeled either with a terminal symbol, or with a nonterminal symbol X_{p0} corresponding to an application of the p-th production, for some p; in the latter case the node has n_p immediate descendants,

(2.2)



(cf. (1.2)). Now let X be the label of a node of the tree and let $\alpha \in A(X)$ be an attribute of X. If $\alpha \in A_0(X)$ then $X = X_{p0}$ for some p, while if $\alpha \in A_1(X)$ then $X = X_{pj}$ for some j and p, $1 \leqslant j \leqslant n_p$, where in either case the tree in the neighborhood of this node has the form (2.2). The attribute α is defined to

have the value v at this node if, in the corresponding semantic rule

$$f_{pj\alpha}: V_{\alpha_1} \times \cdots \times V_{\alpha_t} \to V_\alpha \tag{2.3}$$

all of the attributes $\alpha_1, \cdots, \alpha_t$ have previously been defined to have the respective values $v_1, \cdots, v_t$ at the respective nodes labeled $X_{pk_1}, \cdots, X_{pk_t}$, and $v = f_{pj\alpha}(v_1, \cdots, v_t)$. This process of attribute definition is to be applied throughout the tree until no more attribute values can be defined, and then the defined attributes at the root of the tree constitute the "meaning" corresponding to the derivation tree (cf. (1.6)).

It is natural to require that the semantic rules are formulated in such a way that all attributes can always be defined at all nodes, in any conceivable derivation tree. Let us say the semantic rules are *well defined* if this condition holds. Since there are in general infinitely many derivation trees, it is important to be able to decide if a given grammar has well defined semantic rules or not. An algorithm for testing this condition is presented in Section 3.

Let us note that this method of semantic definition is as powerful as any conceivable method could be, in the sense that the value of any attribute of any node of a derivation tree may depend in any desired way on the entire tree. For example, suppose we ascribe two inherited attributes l ("location") and t ("tree") to each symbol except S in a context-free grammar, and one synthesized attribute s ("subtree") to each nonterminal symbol. Here l ranges over finite sequences of positive integers $\{a_1 \cdot a_2 \cdot \cdots \cdot a_k\}$ which specify the location of tree nodes in a familiar index or "Dewey decimal" notation (see [8, p. 310]); t and s consist of sets of ordered pairs (l, X), where l is a node location and X is a symbol of the grammar denoting the label of the node at location l. The semantic rules, for each production (2.1), are:

$$\begin{aligned}
l(X_{pj}) &= \begin{cases} l(X_{p0}) \cdot j & \text{if } X_{p0} \neq S; \\ j & \text{if } X_{p0} = S; \end{cases} \\
t(X_{pj}) &= \begin{cases} t(X_{p0}) & \text{if } X_{p0} \neq S; \\ s(X_{p0}) & \text{if } X_{p0} = S; \end{cases} \qquad (2.4) \\
s(X_{p0}) &= \{(l(X_{p0}), X_{p0}) \mid X_{p0} \neq S\} \cup \bigcup_{j=1}^{n_p} \{s(X_{pj}) \mid X_{pj} \in N\}.
\end{aligned}$$

Thus, for example, in the tree (1.2) we have

$$\begin{aligned}
s(N) = \{&(1, L), (2, \cdot)\ (3, L), (1.1, L), (1.2, B), (3.1, L), (3.2, B), \\
&(1.1.1, L), (1.1.2, B), (1.2.1, 1), (3.1.1, B), (3.2.1, 1), \\
&(1.1.1.1, L), (1.1.1.2, B), (1.1.2.1, 0), (3.1.1.1, 0), \\
&(1.1.1.1.1, B), (1.1.1.2.1, 1), (1.1.1.1.2.1, 1)\}.
\end{aligned}$$

This clearly contains all the information of the entire derivation tree. The semantic rules (2.4) define the attribute t on all nodes (except the root) to be the set representing the entire derivation tree, while l is the location

of that node. It is therefore evident that any conceivable function of the derivation tree can be an attribute of any node, since such a function is $f(t, l)$ for some f.

Similarly we can show that synthesized attributes alone are sufficient to define the meaning associated with any derivation tree, since the synthesized attribute w defined by the rule

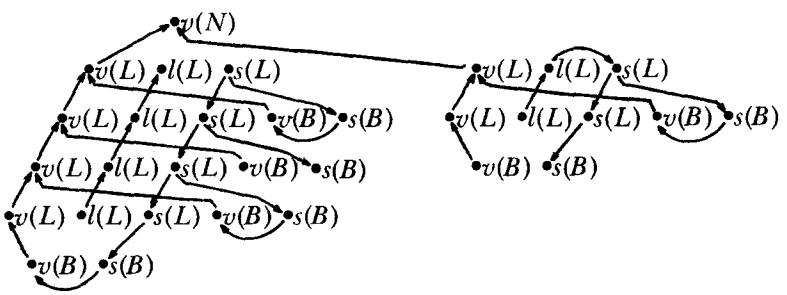
$$w(X_{p0}) = \{(0, X_{p0})\} \cup \bigcup_{j=1}^{n_p} \{(j \cdot \alpha, X) \mid (\alpha, X) \in w(X_{pj}), X_{pj} \in N\} \tag{2.5}$$

evaluated at the root specifies the entire tree. Any semantic rules definable by the method of this section can be considered to be a function of this attribute w, and therefore the method is inherently no more powerful than a method which uses no inherited attributes. But this statement is very misleading, since semantic rules which do not use inherited attributes are often considerably more complicated (and more difficult to understand and to manipulate) than semantic rules which allow both kinds of attributes. The ability to let the whole tree influence the attributes of each node of the tree often leads to rules of semantics which are much simpler and which correspond to the way in which we actually understand the meanings involved.

3. Testing for circularity. Now let us consider an algorithm which determines whether or not a collection of semantic rules, as described in the previous section, is well defined; in other words, we want to know when the semantic rules will always lead to definitions of all attributes at all nodes of all derivation trees. We may assume that the grammar contains no "useless" productions, i.e., that each production of $\mathscr{P}$ appears in the derivation of at least one terminal string.

Let $\mathscr{T}$ be any derivation tree obtainable in the grammar, having only terminal symbols as labels of its terminal nodes, but allowed to have any symbol of V (not only the start symbol S) as the label of the root. Then we can define a directed graph $D(\mathscr{T})$ corresponding to $\mathscr{T}$ by taking the ordered pairs (X, α) as vertices, where X is a node of $\mathscr{T}$ and α is an attribute of the symbol which is the label of node X. The arcs of $D(\mathscr{T})$ go from (X_1, α_1) to (X_2, α_2) if and only if the semantic rule for the value of attribute α_2 depends directly on the value of attribute α_1. For example, if $\mathscr{T}$ is the tree (1.2) and if the semantic rules are given by (1.5), then $D(\mathscr{T})$ is the directed graph

(3.1)

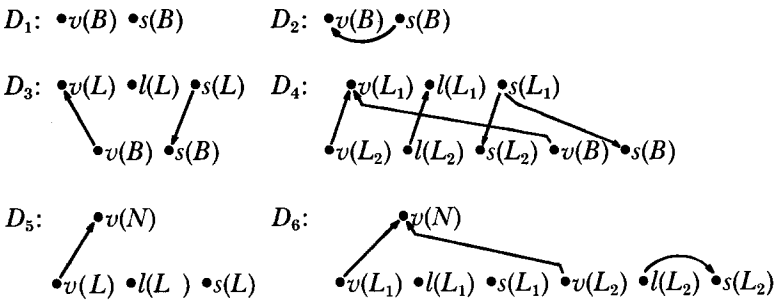


In other words, the vertices of $D(\mathscr{T})$ are the attribute values which must be determined, and the arcs specify the dependency relations which imply that certain attribute values must be computed before others. (Cf. (1.6).)

It is clear that the semantic rules are well defined if and only if no directed graph $D(\mathscr{T})$ contains an oriented cycle. For if there are no oriented cycles, there is a well-known procedure which assigns values to each attribute (see [8, p. 258]). And if there is an oriented cycle in some $D(\mathscr{T})$, the fact that the grammar contains no useless productions implies that there is an oriented cycle in some $D(\mathscr{T})$ in which the root of $\mathscr{T}$ has the label S; this $\mathscr{T}$ is a derivation tree of the language for which it is impossible to evaluate all of the attributes. Therefore the problem, "Are the semantic rules well-defined?" reduces to the problem, "Do the directed graphs $D(\mathscr{T})$ contain any oriented cycles?"

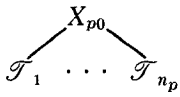
Each directed graph $D(T)$ may be regarded as the superposition of smaller directed graphs D_p corresponding to each of the productions $X_{p0} \to X_{p1} \cdots X_{pn_p}$ of the grammar, $1 \leq p \leq m$. In the notation of Section 2, the directed graph D_p has vertices (X_{pj}, α), for $0 \leq j \leq n_p$, $\alpha \in A(X_{pj})$, and arcs from (X_{pk_i}, α_i) to (X_{pj}, α) for $0 \leq j \leq n_p$, $\alpha \in A_0(X_{pj})$ if $j = 0$, $\alpha \in A_1(X_{pj})$ if $j > 0$, $k_i = k_i(p, j, \alpha)$, $\alpha_i = \alpha_i(p, j, \alpha)$, $1 \leq i \leq t(p, j, \alpha)$. In other words, D_p reflects the dependencies of all the semantic rules associated with the p-th production. For example the six productions of grammar (1.5) correspond to six directed graphs, namely

(3.2)



The directed graph (3.1) is obtained by "pasting together" various subgraphs having these forms. In general if $\mathscr{T}$ has a terminal symbol as the label of the root, $D(\mathscr{T})$ has no arcs; if the root of $\mathscr{T}$ is labeled with a nonterminal symbol, $\mathscr{T}$ has the form

(3.3)



for some p, where $\mathscr{T}_j$ is a derivation tree with X_{pj} as the label of the root, for $1 \leq j \leq n_p$. In the former case we will say $\mathscr{T}$ is a derivation tree of type 0, and in the latter case we will say $\mathscr{T}$ is a *derivation tree of type p*; according to the definition, $D(\mathscr{T})$ is obtained in this case from D_p, $D(\mathscr{T}_1)$, $\cdots$, $D(\mathscr{T}_{n_p})$ by identifying the vertices for attributes of X_{pj} with the corresponding vertices for the attributes of the root of $\mathscr{T}_j$ in $D(\mathscr{T}_j)$, $1 \leq j \leq n_p$.

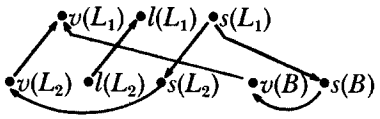
In order to test whether $D(\mathscr{T})$ contains oriented cycles, one further concept is useful. Let p be the number of a production, and for $1 \leq j \leq n_p$ suppose G_j is any directed graph whose vertices are a subset of $A(X_{pj})$, the attributes of X_{pj}; then let

$$D_p[G_1, \cdots, G_{n_p}] \tag{3.4}$$

be the directed graph obtained from D_p by adding an arc from (X_{pj}, α) to (X_{pj}, α') whenever there is an arc from α to α' in G_j. For example, if we have

$$G_1 = \overset{v\ \ l\ \ s}{\bullet\ \bullet\ \bullet}, \qquad G_2 = \overset{v\ \ s}{\bullet\ \bullet}$$

and if D_4 is the directed graph appearing in (3.2), then $D_4[G_1, G_2]$ is



The following algorithm may now be used: "For $1 \leq p \leq m$ let D'_p be the directed graph with vertices $A(X_{p0})$, and with an arc from α to α' if and only if there is an oriented path from (X_{p0}, α) to (X_{p0}, α') in D_p. Let D'_0 be the empty directed graph having no vertices. Now add further arcs to $D'_1, \cdots, D'_m$ by the following procedure until no further arcs can be added: Choose an integer p, with $1 \leq p \leq m$, and for $1 \leq j \leq n_p$ let $q(j) = 0$ if X_{pj} is terminal, or choose an integer $q(j)$ such that X_{pj} is the left-hand side of the $q(j)$-th production, i.e., $X_{q(j)0} = X_{pj}$. Then if there is an oriented path from (X_{p0}, α) to (X_{p0}, α') in the directed graph

$$D_p[D'_{q(1)}, \cdots, D'_{q(n_p)}], \tag{3.5}$$

there should be an arc from α to α' in D'_p." It is clear that this process must ultimately terminate with no more arcs added, since only finitely many arcs are possible in all.

In the case of grammar (1.5), this algorithm begins with

$$D'_1 = \overset{v\ \ s}{\bullet\ \bullet} \qquad D'_2 = \overset{v\ \ s}{\bullet\ \bullet} \qquad D'_3 = \overset{v\ \ l\ \ s}{\bullet\ \bullet\ \bullet}$$

$$D'_4 = \overset{v\ \ l\ \ s}{\bullet\ \bullet\ \bullet} \qquad D'_5 = \overset{v}{\bullet} \qquad D'_6 = \overset{v}{\bullet}$$

and adds arcs until finally we have

$$D'_1 = \overset{v\ \ s}{\bullet\ \bullet} \qquad D'_2 = \overset{v\ \ s}{\bullet\ \bullet} \qquad D'_3 = \overset{v\ \ l\ \ s}{\bullet\ \bullet\ \bullet}$$

$$D'_4 = \overset{v\ \ l\ \ s}{\bullet\ \bullet\ \bullet} \qquad D'_5 = \overset{v}{\bullet} \qquad D'_6 = \overset{v}{\bullet}$$

After the above algorithm terminates, we can prove that *there is an oriented path from* (X, α) *to* (X, α') *in some* $D(\mathscr{T})$, *where* $\mathscr{T}$ *is a derivation tree of type* p *with root* X, *if and only if there is an arc from* α *to* α' *in* D'_p. For the construction does not add any arc from α to α' unless such a $D(\mathscr{T})$ exists; the algorithm could readily be extended so that it would in fact print out an

appropriate derivation tree $\mathscr{T}$ for each arc in $D'_1, \cdots, D'_m$. Conversely, suppose $\mathscr{T}$ is a derivation tree with root X, for which $D(\mathscr{T})$ contains an oriented path from (X, α) to (X, α'); we can prove by induction on the number of nodes of $\mathscr{T}$ that there is an arc from α to α' in D'_p, where $\mathscr{T}$ is of type p: Since $D(\mathscr{T})$ contains at least one arc, $\mathscr{T}$ must be of the form (3.3), and $D(\mathscr{T})$ is "pasted together" from $D_p, D(\mathscr{T}_1), \cdots, D(\mathscr{T}_{n_p})$. By induction and the fact that no arcs run from $D(\mathscr{T}_j)$ to $D(\mathscr{T}_{j'})$ for $j \neq j'$, any arcs of the assumed path which appear in $D(\mathscr{T}_1), \cdots, D(\mathscr{T}_{n_p})$ may be replaced by appropriate arcs in $D_p[D'_{q(1)}, \cdots, D'_{q(n_p)}]$, where $\mathscr{T}_j$ is of type $q(j)$, $1 \leqslant j \leqslant n_p$; and we have an oriented path from (X_{p0}, α') in this directed graph, hence there is an arc from α to α' in D'_p.

The above algorithm now affords a solution to the problem posed in this section:

THEOREM. *Semantic rules added to a grammar as described in Section 2 are well defined if and only if none of the directed graphs* (3.5), *for any admissible choice of* p, $q(1), \cdots, q(n_p)$ *as specified in the above algorithm, contains an oriented cycle.*

Proof. If (3.5) contains an oriented cycle, the remarks just made prove that some $D(\mathscr{T})$ contains an oriented cycle. Conversely, if $\mathscr{T}$ is a tree with the fewest possible nodes, such that $D(\mathscr{T})$ contains an oriented cycle, then $\mathscr{T}$ must be of the form (3.3) and $D(\mathscr{T})$ is "pasted together" from D_p, $D(\mathscr{T}_1), \cdots, D(\mathscr{T}_{n_p})$. By the minimality of $\mathscr{T}$, the oriented cycle involves at least one arc of D_p, and therefore we may argue as above that any arcs of the cycle which are within $D(\mathscr{T}_1), \cdots, D(\mathscr{T}_{n_p})$ may be replaced by arcs of (3.5) when $\mathscr{T}_j$ is of the type $q(j)$.

4. A simple programming language. Now let us consider an example of how the above techniques of semantic definition can be applied to programming languages. For simplicity let us study a formal definition of a little language that describes Turing machine programs.

A Turing machine (in the classical sense) processes an infinite tape which may be thought of as divided into squares; the machine can read or write characters from a finite alphabet on the tape in the square which is currently being scanned, and it can move the scanning position to the left or right. The following program, for example, adds unity to an integer expressed in binary notation and prints a radix point at the right of this number, assuming that the square just to the right of the number is to be scanned at the beginning and end of the program:

```
        tape alphabet is  blank, one, zero, point;
        print "point";
        go to carry;
        test:  if the tape symbol is "one" then
(4.1)              {print "zero"; carry: move left one square; go to test};
        print "one";
        realign: move right one square;
            if the tape symbol is "zero" then go to realign.
```

(It is hoped that the reader will find this programming language sufficiently self-explanatory that he understands it before any formal definition of the language is given, although of course this is not necessary. The above program is not intended as an example of good programming, rather as an example of the features of the simple language considered in this section.)

Since every programming language must have a name, let us call the language Turingol. Any well-formed Turingol program defines a program for a Turing machine; let us say a Turing machine program consists of

a set Q of "states";
a set Σ of "symbols";
an "initial state" $q_0 \in Q$;
a "final state" $q_\infty \in Q$;

and a "transition function" δ which maps $(Q - \{q_\infty\}) \times \Sigma$ into $\Sigma \times \{-1, 0, +1\} \times Q$. If $\delta(q, s) = (s', k, q')$ we may say informally that, if the machine is in state q and scanning symbol s, it will print symbol s', move k spaces to the right (meaning one space to the left if $k = -1$), and go into state q'. More formally, a Turing machine program defines a computation on any "initial tape contents", i.e., on any doubly infinite sequence

$$\cdots, a_{-3}, a_{-2}, a_{-1}, a_0, a_1, a_2, a_3, \cdots \tag{4.2}$$

of elements of Σ, as follows: At any moment of the computation there is a "current state" $q \in Q$ and an integer-valued "tape position" p; initially $q = q_0$ and $p = 0$. If $q \neq q_\infty$, and if $\delta(q, a_p) = (s', k, q')$, the computation proceeds by replacing the value of a_p by s', then by replacing p by $p + k$ and q by q'. If $q = q_\infty$, the computation terminates. (The computation might not terminate; for program (4.1) this happens if and only if $a_j =$ "*one*" for all $j < 0$.)

Now that we have a precise definition of Turing machine programs, we wish to define the Turing machine program corresponding to any given Turingol program (and at the same time to define the syntax of Turingol). For this purpose it is convenient to introduce a few abbreviation conventions.

(1) The semantic rule "**include** x **in** B" associated with a production will mean that x is to be a member of set B, where B is an attribute of the start symbol S of the grammar. The value of B will be the set of all x for which such a semantic rule has appeared corresponding to each appearance of the production in the derivation tree. (This rule may be regarded as an abbreviation for the semantic rule

$$B(X_{p0}) = \bigcup_{j=1}^{n_p} B(X_{pj}) \cup \{x \mid \text{"}\textbf{include } x \textbf{ in } B\text{" is associated with the } p\text{-th production}\} \tag{4.3}$$

added to each production, with a set B added as a synthesized attribute of each nonterminal symbol, and $B(x)$ the empty set for each terminal symbol. These rules clearly make $B(S)$ the desired set.)

(2) The semantic rule "**define** $f(x) = y$" associated with a production will mean that y is to be the value of the function f evaluated at x, where f is an attribute of the start symbol S of the grammar. If two rules occur defining $f(x)$ for the *same* value of x, this is an error condition, and any derivation tree which allows this condition to occur may be said to be *malformed.* Furthermore, f may be used as a function in any other semantic rules, with the proviso that $f(x)$ may only appear when f has been defined at x; any derivation tree which calls for an undefined value of $f(x)$ is *malformed.* (This type of rule is important, for example, to ensure that there is agreement between the declaration and use of identifiers. In the example below this convention implies that programs are malformed if the same identifier is used twice as a label or if a **go to** statement specifies an identifier which is not a statement label. The rule may essentially be thought of as "**include** (x, y) **in** f", as in (1), if f is regarded as a set of ordered pairs; additional checks for malformedness are also included. We may regard "well-formed or malformed" as an attribute of S; appropriate semantic rules analogous to (4.3) which completely specify this "**define** $f(x) = y$" convention are readily constructed and left to the reader.)

(3) The function "**newsymbol**" appearing in any semantic rule will have, as its value, an abstract element which for each evaluation of "**newsymbol**" is different from the abstract element produced by other evaluations of **newsymbol.** (This convention can readily be expressed in terms of other semantic rules, e.g., by making use of the l attributes of (2.3) which has a different value at each node of a tree. The function **newsymbol** serves as a convenient source of "raw material" for constructing sets.)

We have observed that conventions (1), (2), (3) can be replaced by other constructions of semantic rules which do not use these conventions, so they are not "primitives" for semantics. But they are of fairly wide utility, since they correspond to concepts which are often needed, so they may be regarded as fundamental aspects of the techniques for semantic definition presented in this paper. The effect of using these conventions is to reduce the number of attributes that are explicitly mentioned and to avoid unnecessarily long rules.

Now it is a simple matter to present a formal definition of the syntax and semantics of Turingol.

Nonterminal symbols: P (program), S (statement), L (list of statements), I (identifier), O (orientation), A (alphabetic character), D (declaration).

Terminal symbols: $a\ b\ c\ d\ e\ f\ g\ h\ i\ j\ k\ l\ m\ n\ o\ p\ q\ r\ s\ t\ u\ v\ w\ x\ y\ z\ .\ ,\ :\ ;\ "\ "\ \{\ \}$ **tape alphabet is print go to if the symbol then move left right one square**

Start symbol: p

Attributes:

Name of attribute	*Type of value*	*Purpose*
Q	Set	States of the program
Σ	Set	Symbols of the program
q_0	Element of Q	Initial state
q_∞	Element of Q	Final state
δ	Function from $(Q-q_\infty)\times\Sigma$ into $\Sigma\times\{-1,0,+1\}\times Q$	Transition function
label	Function from strings of letters into elements of Q	State table for statement labels
symbol	Function from strings of letters into elements of S	Symbol table to tape symbols
follow	Element of Q	State immediately following statement or list of statements
d	± 1	Direction
text	String of letters	Identifier
start	Element of Q	State at the beginning of a statement or list of statements (an inherited attribute).

Productions and semantics: See Table 1.

Notice that two states correspond to each statement S: start (S) is the state corresponding to the first instruction of the statement (if any), and it is an inherited attribute of S; follow (S) is the state which "follows" the statement, the state which is normally reached after the statement is executed. In the case of a "go statement", however, the program does not transfer to follow (S), since the action of the statement is to change control to another place; follow (S) may be said to follow statement S "statically" or "textually", not "dynamically" during a run of the program.

In Table 1, follow (S) is a synthesized attribute; it is possible to give similar semantic rules in which follow (S) is inherited, although a less efficient program would be obtained for null statements (see Rule 4.4). Similarly, both start (S) and follow (S) could be synthesized attributes, but at the expense of additional instructions in the Turing machine program for statement lists (Rule 6.2).

This example would be somewhat simpler if we had used a less standard definition of Turing machine instructions. The definition we have used requires reading, printing, and shifting in each instruction, and also makes the Turing machine into a kind of "one-plus-one-address computer" in which each instruction specifies the location (state) of the next instruction.

Table 1.

Description	*No.*	*Syntactic Rule*	*Example*	*Semantic Rules*
Letters	1.1	$A \rightarrow a$	*a*	text $(A) = a$.
	. . . . .	. . . . .	(similarly for all letters)	
	1.26	$A \rightarrow z$	*z*	text $(A) = z$.
Identifiers	2.1	$I \rightarrow A$	*m*	text (I) = text (A).
	2.2	$I \rightarrow IA$	*marilyn*	text (I) = text (I) text (A).
Declarations	3.1	$D \rightarrow$ **tape alphabet is** I	**tape alphabet is** *marilyn*	**define** symbol (text (I)) = **newsymbol**; **include** symbol (text (I)) **in** S.
	3.2	$D \rightarrow D, I$	**tape alphabet is** *marilyn, jayne, birgitta*	**define** symbol (text (I)) = **newsymbol**; **include** symbol (text (I)) **in** S.
Print statement	4.1	$S \rightarrow$ **print** "I"	**print** "*jayne*"	**define** δ (start $(S), s$) = (symbol (text (I)), 0, follow (S)) for all $s \in \Sigma$; follow (S) = **newsymbol**; **include** follow (S) in Q.
Move statement	4.2	$S \rightarrow$ **move** O **one square**	**move left one square**	**define** δ (start $(S), s$) = (s, d(O), follow (S)) for all $s \in \Sigma$; follow (S) = **newsymbol**; **include** follow (S) **in** Q.
	4.2.1	$O \rightarrow$ **left**	**left**	d(O) = −1.
	4.2.2	$O \rightarrow$ **right**	**right**	d(O) = +1.
Go statement	4.3	$S \rightarrow$ **go to** I	**go to** *boston*	**define** δ (start $(S), s$) = (s, 0, label (text (I))) for all $s \in \Sigma$; follow (S) = **newsymbol**; **include** follow (S) **in** Q.
Null statement	4.4	$S \rightarrow$		follow (S) = start (S).
Conditional statement	5.1	$S_1 \rightarrow$ **if the tape symbol is** "I" **then** S_2	**if the tape symbol is** "*marilyn*" **then print** "*jayne*"	**define** δ (start $(S_1), s$) = (s, 0, follow (S_2)) for all $s \in \Sigma$ − symbol (text (I))); **define** δ (start $(S_1), s$) = (s, 0, start (S_2)) for s = symbol (text (I))); start (S_2) = **newsymbol**; follow (S_1) = follow (S_2); **include** start (S_2) **in** Q.
Labeled statement	5.2	$S_1 \rightarrow I : S_2$	*boston:* **move left one square**	**define** label (text (I)) = start (S_1); start (S_2) = start (S_1); follow (S_1) = follow (S_2).
Compound statement	5.3	$S \rightarrow \{L\}$	{**print** "*jayne*";; **go to** *boston*}	start (L) = start (S); follow (S) = follow (L).
List of statements	6.1	$L \rightarrow S$	**print** "*jayne*"	start (S) = start (L); follow (L) = follow (S).
	6.2	$L_1 \rightarrow L_2; S$	**print** "*jayne*";; **go to** *boston*	start (L_2) = start (L_1); follow (L_2) = **newsymbol**; **include** follow (L_2) **in** Q; start (S) = follow (L_2); follow (L_1) = follow (S).
Program	7	$P \rightarrow D; L.$	**tape alphabet is** *marilyn, jayne, birgitta;* **print** "*jayne*".	q_0 = **newsymbol**; **include** q_0 **in** Q; start $(L) = q_0$; q_∞ = follow (L).

The method of defining semantic rules in this example, with an inherited "first (S)" and a synthesized "follow (S)" attribute, lends itself readily also to computers or automata in which the $(n + 1)$st instruction normally is performed after the n-th. Then (follow (S) − start (S)) would be the number of instructions "compiled" for statement S.

This definition of Turingol seems to approach the desirable goal of stating almost exactly the same things which would appear in an informal programmer's manual explaining the language, except that the description is completely formal and unambiguous. In other words, this definition perhaps corresponds to the way we actually understand the language in our minds. The Definition 4.1 of a print statement, for example, might be freely rendered in English as follows:

"A statement may have the form

print "I"

where I is an identifier. This means that, whenever this statement is executed, the tape symbol on the currently scanned square will be replaced by the symbol denoted by I, regardless of what symbol was being scanned; afterwards the program will continue with a new instruction, which is defined (by other rules) to be the instruction following this statement."

5. Discussion. The idea of defining semantics by associating synthesized attributes with each nonterminal symbol, and associating corresponding semantic rules with each production, is due to Irons [6, 7]. Originally each nonterminal symbol was given exactly one attribute, its "translation". This idea was applied by Irons and later authors, notably McClure [14], in the design of "syntax-directed compilers" which translate programming languages into machine instructions.

As we have observed in Section 2, synthesized attributes alone are (in principle) sufficient to define any function of a derivation tree. But in practice, the inclusion of inherited attributes as well as synthesized attributes, as described in this paper, leads to important simplifications. The definition of Turingol, for example, shows that the agreement between declaration and use of symbols, and the association of labels to statements, may be easily treated. "Block structure" is another common aspect of programming languages whose definition is greatly facilitated by the use of inherited attributes. In general, inherited attributes are useful when part of the meaning of some construction is determined by the context in which that construction appears. The method of Section 2 shows how both inherited and synthesized attributes can be treated formally, and Section 3 shows that it is possible to rule out problems of circularity (which are potential sources of difficulty when both inherited and synthesized attributes are mixed).

The principal contributions to formal semantic definition of programming languages, at least those known to the author at the time of writing, are de Bakker's definition of ALGOL 60 by means of a growing Markovian

algorithm [1]; Landin's definition of ALGOL 60 by means of the λ-calculus [9, 10, 11] (see also Böhm [2, 3]); McCarthy's definition of Micro-ALGOL by means of recursive functions applied to the program and to "state vectors" [12] (see also McCarthy and Painter [13]); Wirth and Weber's definition of Euler, by means of semantic rules applied as a program is parsed [16]; and the IBM Vienna Laboratory's definition of PL/I [15] based on the work of McCarthy, Landin, and abstract machines defined by Elgot [4, 5].

The most striking difference between the previous methods and the definition of Turingol in Table 1 is that the other definitions are processes which are defined on programs as a whole in a rather intricate manner; it may be said that a person must understand an entire compiler for the language before he can understand the definition of the language. This difficulty is most pronounced in the work of de Bakker, who defines a machine having approximately 800 instructions, analogous to Markov algorithms but somewhat more complicated; at each stage of this machine's computations we are to execute the last applicable instruction, so we cannot verify that instruction number 100 will be performed until we can prove to ourselves that the 700 subsequent instructions are inapplicable; furthermore, additional instructions are added to the list by the actions of the machine. It is clearly very difficult for a reader to understand the workings of such a machine, or to give formal proofs of its important properties. By contrast, the above definition of Turingol defines each construction of the language only in terms of its "immediate environment", minimizing to a large extent the interconnections between the definitions of different parts of the language. The definition of compound statements and go statements, etc., does not influence the definition of print statements in a substantial way; for example, any of Rules 4.1, 4.2, 4.3, 4.4, 5.1, 5.3 could be deleted and we would obtain a valid definition of another language. This localization and partitioning of the semantic rules tends to make the definition easier to understand and more concise.

Although the other authors cited above do not make use of such an intricately interwoven definition as de Bakker's, the relatively complex interdependence is still present. For example, consider the formal definition of Euler given by Wirth and Weber [16, pp. 94–98]; this is a concise definition of a very sophisticated language, and so it is certainly one of the most successful formal definitions ever devised. Yet even though Wirth and Weber tested their definition by means of extensive computer simulation, it is quite probable that their language contains some features which would surprise its authors. The following Euler program is syntactically and semantically well-formed, although the label L is never followed by a colon:

```
⊥ begin label L; new A; A ← 0;
  if false then go to L else L;
  out 1; L; A ← A + 1; out 2;
  if false then go to L else
  if A < 2 then go to L else out 3; L end ⊥
```

The output of this program is 1, 2, 2, 3! Oversights such as this are not unexpected when an algorithmic definition of a language is constructed; they are less likely to occur when the methods of Section 4 are employed.

It appears to be reasonable to assert that none of the previous schemes for formal definition of semantics could produce a definition of Turingol that is as brief or as easy to comprehend as the definition given above; and (although the details have not of course been worked out) it also appears that ALGOL 60, Euler, Micro-ALGOL, and PL/I can be defined using the methods of Section 4 in a manner which has advantages over the definitions previously given. But of course the author cannot judge these things impartially, and more experience is needed before these claims can be substantiated.

Notice that semantic rules as given in this paper do not depend on any particular form of syntactic analysis. In fact, they need not even be tied down to specific forms of the syntax: All that the semantic rules depend on is the name of the nonterminal symbol on the left of a production and the names of the nonterminals on the right. Particular punctuation marks, and the order in which the nonterminals appear on the right-hand side of any production, are immaterial as far as the semantic rules are concerned. Thus, the method of semantics considered here blends well with McCarthy's idea [12, 13] of "abstract syntax".

When a syntax is ambiguous, in the sense that some strings of the language have more than one derivation tree, the semantic rules give us one "meaning" for each derivation tree. For example, suppose the rules

$$L_1 \rightarrow BL_2 \qquad v(L_1) = 2^{l(L_2)}v(B) + v(L_2), \qquad l(L_1) = l(L_2) + 1$$

are added to grammar (1.3). Then the grammar becomes syntactically ambiguous; but it still is semantically unambiguous since the attribute $v(N)$ has the same value over all derivation trees. On the other hand, if we were to change production 5.2 of Turingol from $S \rightarrow I\colon S$ to $S \rightarrow S\colon I$, the grammar would become syntactically *and* semantically ambiguous.

REFERENCES

[1] J. W. DE BAKKER, *Formal definition of programming languages, with an application to the definition of ALGOL 60*, Math Cent. Tracts **16**, Mathematisch Centrum, Amsterdam, 1967.

[2] C. BÖHM, The CUCH as a formal and description language, *Formal Language Description Languages for Computer Programming*, pp. 266–294, Proc. IFIP Working Conf., Vienna (1964), North Holland, 1966.

[3] CORRADO BÖHM and WOLF GROSS, "Introduction to the CUCH," *Automata Theory* (ed. by E. R. Caianiello), pp. 35–65, Academic Press, 1966.

[4] C. C. ELGOT, "Machine species and their computation languages," *Formal Language Description Languages for Computer Programming*, pp. 160–179, Proc. IFIP Working Conf., Vienna (1964), North Holland, 1966.

[5] C. C. ELGOT and A. ROBINSON, "Random-access, stored program machines, an approach to programming languages," *J. ACM* **11** (1964), 365–399.

[6] EDGAR T. IRONS, A syntax directed compiler for ALGOL 60, *Comm. ACM* **4** (1961), 51–55.

[7] EDGAR T. IRONS, Towards more versatile mechanical translators, Proc. Sympos. Appl. Math., Vol. 15, pp. 41–50, Amer. Math. Soc., Providence, R. I., 1963.

[8] DONALD E. KNUTH, *The Art of Computer Programming*, I, Addison-Wesley, 1968.

[9] P. J. LANDIN, "The mechanical evaluation of expressions," *Comp. J.* **6** (1964), 308–320.

[10] P. J. LANDIN, A formal description of ALGOL 60, *Formal Language Description Languages for Computer Programming*, pp. 266–294, Proc. IFIP Working Conf., Vienna, (1964), North Holland, 1966.

[11] P. J. LANDIN, A correspondence between ALGOL 60 and Church's lambda notation, *Comm. ACM* **8** (1965), 89–101, 158–165.

[12] JOHN MCCARTHY, A formal definition of a subset of ALGOL, *Formal Language Description Languages for Computer Programming*, pp. 1–12, Proc. IFIP Working Conf., Vienna (1964), North Holland, 1966.

[13] JOHN MCCARTHY and JAMES PAINTER, Correctness of a compiler for arithmetic expressions, Proc. Sympos. Appl. Math., Vol. 17, to appear, Amer. Math. Soc., Providence, R. I., 1967.

[14] ROBERT M. MCCLURE, TMG—A syntax directed compiler, *Proc. ACM Nat. Conf.* **20** (1965), 262–274.

[15] PL/I Definition Group of the Vienna Laboratory, *Formal definition of PL/I*, IBM Technical Report TR 25.071 (1966).

[16] NIKLAUS WIRTH and HELMUT WEBER, Euler: A generalization of ALGOL, and its formal definition, *Comm. ACM* **9** (1966), 11–23, 89–99, 878.

(Received 15 November 1967)