AUTOMATIC GENERATION OF NEAR-OPTIMAL LINEAR-TIME TRANSLATORS FOR NON-CIRCULAR ATTRIBUTE GRAMMARS

by

Rina Cohen and Eli Harry

Technical Report #120
March 1978
ABSTRACT

Attribute grammars are an extension of context-free grammars devised by Knuth as a formalism for specifying the semantics of a context-free language along with the syntax of the language. The syntactic phase of the translation process has been extensively studied and many techniques are available for automatically generating efficient parsers for context-free grammars. Attribute grammars offer the prospect of similarly automating the implementation of the semantic phase. In this paper we present a general method of constructing, for any non-circular attribute grammar, a deterministic translator which will perform the semantic evaluation of each syntax tree of the grammar in time linear with the size of the tree. Each tree is traversed in a manner particularly suited to the shape of the tree, yielding a near optimal evaluation order for that tree. Basically, the translator consists of a finite set of "Local Control Automata", one for each production; these are ordinary finite-state acyclic automata augmented with some special features, which are used to regulate the evaluation process of each syntax tree. With each node in the tree there will be associated the Local Control Automaton of the production applying at the node. At any given time during the translation process all Local Control Automata are inactive, except for the one associated with the currently processed node, which is responsible for directing the next steps taken by the translator until control is finally passed to a neighbor node, reactivating its Local Control Automaton. The Local Control Automata of neighbor nodes communicate with each other.

The construction of the translator is custom tailored to each individual attribute grammar. The dependencies among the attributes...
occurring in the semantic rules are analyzed to produce a near-optimal evaluation strategy for that grammar. This strategy ensures that during the evaluation process, each time the translator enters some subtree of the syntax tree, at least one new attribute evaluation will occur at each node visited. It is this property which distinguishes the method presented here from previously known methods of generating translators for unrestricted attribute grammars, and which causes the translators to be near-optimal.

1. INTRODUCTION

Attribute grammars are an extension of context free grammars devised by Knuth [K1] for specifying the semantics of languages along with the syntax. Each grammar symbol has an associated set of attributes specifying the various components of its "meaning", and each production is provided with semantic rules, defining the attributes of symbols in the production in terms of other attributes associated with the production. To find the meaning of a string, first we construct its syntax tree and then we determine the values of all the attributes of symbols in the tree, a process which is called semantic evaluation of the tree.
An attribute grammar is non-circular if the system of semantic definitions given by the grammar avoids circularity in all possible instances. The problem of detecting circularity in an attribute grammar, first solved by Knuth in [KL2], was shown in [J&O&R] to be of inherent exponential complexity.

Since their definition, attribute grammars have attracted widespread interest in the area of programming languages. They have been used by investigators in fields such as natural language recognition and question answering systems [P], program optimization ([N&A],[J2]) and the theory of program correctness [G]. But their most important contribution was towards the formal definition of programming languages and automatic design of compilers. Two high-level programming languages, namely SIMULA and PL360, were fully defined and implemented using the attribute grammar model ([W1],[D]). Techniques for the automatic construction of language processing devices based on the model of attribute grammar were discussed in [K3], [AGU2,3],[L&S],[L&R&S1,2]. These applications seem to support Knuth's original content that attribute grammars are a natural, humanly understandable method for language definition.

The compilation process may be divided into a syntactic phase and a semantic phase. While the syntactic phase has been extensively studied and a number of techniques are available for automatically generating efficient
parses. For context free grammars, the semantic phase has not been so far well understood. Knuth's attribute grammars offer the prospect of similarly automating the implementation of the semantic phase.

When an attribute grammar is used to specify the translation and code generation performed by a compiler, the attributes may represent such things as data types of expressions, symbol tables for use in translating identifiers, register usage information or machine code generated for a statement. Defining a programming language and its compilation process by means of an attribute grammar offers the following advantages:

(i) The semantics is given in a declarative, non-procedural, and is independent of any parsing scheme. There is no explicit statement in the definition about the order in which values are assigned to attributes. The semantic rules merely state how the values of the attributes of neighboring nodes relate to each other.

(ii) The semantic description is modular, given on a production by production basis, which makes the definitions more understandable and facilitates the addition and removal of features from the language.

(iii) The context sensitive features of the language can be naturally expressed.

(iv) The description of the language can be checked for consistency and used for automatic compiler generation.

Despite these advantages, attribute grammars have not become a widely used tool for compiler generation because of the difficulty in obtaining implementations efficient enough for practical use. Until now, efficient translation algorithms have been developed for restricted classes of attribute grammars ([L&RS12],[B],[J1],[K&W]).
A general framework for studying deterministic implementation of attribute grammars was developed in [W].

In this paper, using a somewhat similar approach, we present a general scheme for automatically constructing a near-optimal linear-time deterministic translator for any non-circular attribute grammar. The translator will traverse each parse tree of the grammar in a manner which is particularly suited to the shape of the tree, yielding a near-optimal evaluation order for the tree. As a result, our translator, unlike the ones constructed in [W], will never wander around aimlessly in the tree before reaching a "useful" place (where new attributes can be computed). Instead, it is guaranteed that each time the translator enters some subtree of the syntax tree, at least one new attribute evaluation will occur at each of the nodes visited.

The construction of the translator is custom tailored to each individual attribute grammar. The dependencies among the attributes in the grammar are analyzed to produce a near-optimal evaluation strategy for that grammar (in fact, one could search through all near-optimal evaluation strategies to find an optimal one).

Basically, the translator consists of a finite set of "Local Control Automata", one for each production. These are finite-state acyclic automata augmented with some special features, which are used to regulate the translation process of each given parse tree. With each interior node of the tree is associated the Local Control Automaton of the production applying at the node. During the translation process, whenever control reaches the particular node, its Local Control Automaton is reactivated, starting from the state in which it was last suspended. The automaton
is responsible for directing the evaluation process while control is at the node. When finally control is passed to a neighbor node (along with some parameter), the currently active Local Control Automaton is suspended, and the Local Control Automaton associated with the neighbor node will be reactivated. This goes on until eventually the Local Control Automaton associated with the root of the tree enters a final state, at which point the translation process is complete.

Section 2 provides the basic definitions and notation concerning attribute grammars. In Section 3, the main tool for analysing data dependencies within a parse tree — the "characteristic graph" — is defined, and Knuth's circularity test for attribute grammars is presented. Section 4 includes an overview of the translation process. Section 5 introduces the set of Local Control Automata used in the translation process, while Section 6 describes the translation process itself. In Section 7 we present an algorithm for constructing, for every non-circular attribute grammar, its set of Local Control Automata. In Section 8 a comparison is made between our method and previously known methods, and in Section 9 some complexity issues are discussed and concluding remarks are given.
2. ATTRIBUTE GRAMMARS

The idea of syntax directed translation is due to Irons [1], who introduced a model of a context free grammar augmented with a single synthesized attribute for each nonterminal; the meaning of this attribute was essentially the translation of the nonterminal. This model was generalized by Aho and Ullman [AU1, AU2] and by Lewis and Stearns [LS], allowing multiple synthesized attributes for each nonterminal.

Knuth [K1] made an important step forward by introducing the inherited attributes, which depend on the context of the occurrence of the nonterminal. Knuth's model, called attribute grammar, allows for flow of information from the bottom up in the parse tree (by synthesized attributes), and from the top down (by inherited attributes).

Lewis, Rosenkranz and Stearns [LRS1] suggested a slightly different, but equivalent model, called attribute translation grammar. In this paper we shall use Knuth's model of attribute grammar. Many of our definitions and notation are taken directly from [8], and from [K&W].
2.1 Definition of Attribute Grammar

An attribute grammar is a context free grammar augmented with attributes and semantic functions. Formally, an attribute grammar consists of the following:

1. **Underlying grammar**: A context free grammar $G = (V_N, V_T, P, S)$, where $V_N$ is the set of nonterminals, $V_T$ is the set of terminals, $P$ is the set of productions and $S \in V_N$ is the start symbol. We assume that there are no useless nonterminals and that $S$ appears on the left hand side of a unique initial production $P_0$, and does not appear on the right side of any production. Let $V$ stand for $V_N \cup V_T$.

A production $p \in P$ is written in the following form:

$$p: X_0 \rightarrow X_1 X_2 \ldots X_n$$

where $n_p \geq 1$, $X_0 \in V_N$ and $X_k \in V$ for $1 \leq k \leq n_p$.

For convenience, we sometimes write $p[k]$ to mean $X_k$ for $k = 0, 1, \ldots, n_p$; i.e. $p[k]$ denotes the $k$th symbol (from $V$) appearing in production $p$.

2. **Attributes**: For each symbol $X \in V$, there are two finite disjoint sets, $I(X)$ - the **inherited attributes** of $X$, and $S(X)$ - the **synthesized attributes** of $X$. For $X = S$ and for $X \in V_T$, we require that $I(X) = \emptyset$. We write $A(X)$ for $I(X) \cup S(X)$. A production

$$p: X_0 \rightarrow X_1 X_2 \ldots X_n$$

has the attribute occurrence $(a, k)$, $0 \leq k \leq n_p$, if $a \in A(X_k)$.
3. **Semantic functions**: For each production \( p \in P \) there is a finite set of semantic functions as follows: for every synthesized attribute occurrence \((a,k)\) with \(k = 0\), and for every inherited attribute occurrence \((a,k)\) with \(1 \leq k \leq n_p\), there is a semantic function \( f^p_{(a,k)} \), which defines the value of attribute occurrence \((a,k)\) from the values of other attribute occurrences appearing in the same production. The value of an attribute occurrence defined by a semantic function is taken from a given set (possibly infinite) of attribute values. The semantic functions usually involve simple operators like assignment, Boolean operators, numerical sums, etc.

**Example 1**

The following is an example of an attribute grammar:

\[
\begin{align*}
V_N &= \{S, A, B\} & V_T &= \{a, b\} \\
I(S) &= \emptyset & S(S) &= \{s_1\} \\
I(A) &= \{i_1\} & S(A) &= \{s_1, s_2\} \\
I(B) &= \{i_1, i_2\} & S(B) &= \{s_1\}
\end{align*}
\]

**Production**

**Semantics**

0: \( S \rightarrow abA \)

\[
\begin{align*}
(s_1, 0) &= (s_1, 3) + (s_2, 3) \\
(i_1, 3) &= (s_1, 3) \\
(s_1, 0) &= (s_1, 2) \\
(s_2, 0) &= (s_2, 2) + (s_1, 4) \\
(i_2, 4) &= (i_1, 0) \\
(i_1, 4) &= 1 \\
(i_1, 2) &= 1
\end{align*}
\]

1: \( A \rightarrow aAbB \)
2: \[ A \rightarrow abb \]

\[
\begin{align*}
(s_1,0) &= 1 \\
(s_2,0) &= 1
\end{align*}
\]

3: \[ A \rightarrow ab \]

\[
\begin{align*}
(s_1,0) &= 1 \\
(s_2,0) &= (1_1,0)
\end{align*}
\]

4: \[ B \rightarrow ab \]

\[
\begin{align*}
(s_1,0) &= (1_1,0)
\end{align*}
\]

5: \[ B \rightarrow ba \]

\[
\begin{align*}
(s_1,0) &= (1_2,0)
\end{align*}
\]

2.2 Evaluation of Attribute Grammars

A parse tree of an attribute grammar is a derivation tree of the underlying context free grammar, i.e. a finite tree whose nodes are labelled with symbols from \( V \). For each interior node \( N \) there is a production \( p \in P \) such that \( N \) is labelled with the symbol \( p[0] \), has \( n_p \) sons, and for each \( k, 1 \leq k \leq n_p \), the \( k \)-th son of \( N \) is labelled with \( p[k] \). In this case we say that production \( p \) applies at node \( N \).

By the \( k \)-th neighbor of node \( N \), for \( 0 \leq k \leq n_p \), we shall mean the father of \( N \) in case \( k = 0 \), or the \( k \)-th son of \( N \) in case \( 1 \leq k \leq n_p \).

A parse tree with root \( X \), \( X \in V_\text{N} \), is a parse tree in which the root is labelled by nonterminal \( X \).

A semantic tree of an attribute grammar is a parse tree augmented with attributes; i.e. each node labelled with \( X \in V \) is a structured variable whose field selectors are the elements of \( A(X) \). In order to determine the "meaning" or "translation" of a string generated by the underlying context free grammar, we first parse the string and build the semantic tree. Then we have to fill in the fields of each node by computing the attribute values according to the semantic functions. This process is called evaluation of the semantic tree.
An algorithm which accepts as input a semantic tree generated by a given attribute grammar $G$ and evaluates it is called a translator or evaluator for $G$.

The only restriction imposed on the translator is that at any point in time during the evaluation process it can only evaluate an attribute occurrence whose semantic function is ready to evaluate; that is, all attribute occurrences which are arguments of the semantic function have been previously evaluated. Thus the translator will move in the tree from node to node, at each node evaluate some attributes which are ready to evaluate, until all attribute occurrences in the tree have been evaluated.

Consider the attribute grammar of Example 1 and the semantic tree of Fig. 1. The fields have been filled in as prescribed by the semantic functions. Note, for example, that $i_1$ of node 11 cannot be evaluated before $s_1$ of the same node is evaluated, which in turn demands pre-evaluation of $s_1$ of node 6.

In later sections we shall describe how to systematically construct an efficient translator for each attribute grammar. The construction of the translator will be carried out once and for all at construction time. Once the translator for the attribute grammar has been constructed, the evaluation process for each semantic tree, as described above, will take place at run time.

2.3 The Dependency Graph

Let $p$ be the production $p: X_o \rightarrow X_1 X_2 \cdots X_n$. A convenient tool for describing production $p$, and its semantic dependencies, is the dependency graph of production $p$, denoted $D_p$. The nodes of $D_p$ are the...
attribute occurrences of \( p \). There is a directed arc from node \((a',k')\) to node \((a,k)\) iff \((a,k)\) depends on \((a',k')\) i.e. the semantic function \( f_p^{(a,k)} \) uses \((a',k')\) as argument. This means that \( D_p \) reflects the dependencies among attribute occurrences imposed by the semantic functions of \( p \). Fig. 2 shows the dependency graph of production 1 of Example 1.

In order to represent the semantic dependencies of a whole semantic tree rather than a single production, we define the (compound) dependency graph of a semantic tree \( T \), denoted \( D(T) \). This graph is constructed by "pasting together" copies of the \( D_p \)'s for the productions occurring in the tree. Each one of the \( D_p \)'s is selected according to the production \( p \) applying at the interior node of the tree. Fig. 3 shows the compound dependency graph of the tree of Fig. 1

2.4 Circularity

The compound dependency graph of a semantic tree reflects the flow of information among the attributes in the tree. In the semantic tree there is a flow of information from the root towards the leaves by the inherited attributes, and from the leaves towards the root by the synthesized attributes. Since information may flow in both directions, a cycle may be created. An attribute grammar is said to be circular if there exists at least one semantic tree whose dependency graph contains a cycle.

Circular grammars are clearly ill-defined in that there is at least one semantic tree which cannot be evaluated.

Knuth [K1,2] has presented an algorithm which tests an attribute grammar for circularity. The time taken by the algorithm is exponential in the size of the grammar. A faster but still exponential algorithm
appears in [J60SR], where it was also established that the circularity problem for attribute grammars is of inherent exponential time complexity.

2.5 A Normal Form for Attribute Grammars

In the general definition of attribute grammar, for each production \( p \), every semantic function \( f^p_{(a,k)} \) may be defined in terms of all other attribute occurrences, excluding \( (a,k) \) itself. Without loss of generality, we may assume that a semantic function \( f^p_{(a,k)} \) does not use as argument an attribute occurrence, which is defined in the same production. This leads to the following normal form [J1].

**Definition:** An attribute grammar is said to be in **normal form** iff it satisfies the following restrictions. For each production \( p: X_0 \rightarrow X_1 \ldots X_n \):

(a) A synthesized attribute occurrence \( (s,0) \), \( s \in S(X_0) \), may depend only on:

1. Inherited attribute occurrences \( (i,0) \), \( i \in I(X_0) \);
2. Synthesized attribute occurrences \( (s,k) \), \( s \in S(X_k) \), \( 1 \leq k \leq n_p \);

(b) An inherited attribute occurrence \( (i,k) \), \( i \in I(X_k) \), \( 1 \leq k \leq n_p \), depend only on:

1. Inherited attribute occurrences \( (i,0) \), \( i \in I(X_0) \);
2. Synthesized attribute occurrences \( (s,k) \), \( s \in S(X_k) \), \( 1 \leq k \leq n_p \).

There is no loss of generality in imposing the above restrictions, because any other attribute occurrence appearing as argument in \( f^p_{(a,k)} \) can be replaced by its own semantic function, which appears in the same production. A series of such replacements will eventually yield the
desired normal form for production \( p \), provided there is no local circularity in \( p \). Local circularity simply means that the dependency graph \( D_p \) of production \( p \) contains a cycle. Local circularity can be easily tested, and its existence clearly implies circularity.

From the above discussion it follows that there is a simple algorithm, which transforms every attribute grammar without local circularity into the above normal form.

3. THE CHARACTERISTIC GRAPHS AND KNUTH'S CIRCULARITY TEST

We now construct for each nonterminal of the attribute grammar a collection of directed graphs, called the characteristic graphs of the nonterminal. These graphs play a major role in Knuth's test for circularity and are essential for the understanding of the evaluation process and translator construction to be described in subsequent sections.

Definition: For each semantic tree \( T_X \) with root \( X \), the root dependency graph of the tree \( T_X \) is derived from the compound dependency graph of \( T_X \) by "projecting" the attribute dependencies on the root \( X \) as follows. The set of nodes of the root dependency graph is \( A(X) \), and there is an arc from inherited attribute \( i \) to synthesized attribute \( s \) iff the compound dependency graph of \( T_X \) has a path from \( i \) to \( s \).

Figure 4 shows the root dependency graph of the subtree rooted at node 11 of the tree in Fig. 1. Observing the compound dependency graph of Fig. 3, one can see that there is a path from \( i_1 \) of the node 4 to \( i_2 \) of node 8, continuing to \( s_1 \) of the same node and ending in \( s_2 \) of node 4. This path yields the arc from \( i_1 \) to \( s_2 \) in Fig. 4.
The root dependency graph of a semantic tree $T_X$ reflects the dependencies among the attribute occurrences of the root $X$ induced by the tree $T_X$. Since the node set is $\mathcal{A}(X)$, there can be only finitely many distinct root dependency graphs for trees with root $X$ (though the number of such trees may be infinite). In the evaluation process of semantic trees to be described in the following sections, each subtree of the semantic tree will be characterized by its root dependency graph.

Define a characteristic graph of nonterminal $X$ to be any graph $C$ with node set $\mathcal{A}(X)$ s.t. $C$ is the root dependency graph of at least one semantic tree $T$ with root $X$. Let $\mathcal{C}_X = \{C_1, C_2, \ldots, C_{n_X}\}$ denote the set of all characteristic graphs of nonterminal $X$.

Before presenting Knuth's construction of the characteristic graphs we need another definition.

**Definition:** Let $D_p$ be the dependency graph for production $p: X_o \rightarrow X_1 X_2 \ldots X_n^p$, and let $G_1, G_2, \ldots, G_{n_p}$ be any given set of directed graphs s.t. for each $k = 1, \ldots, n_p$, the nodes of $G_k$ are the elements of $\mathcal{A}(X_k)$. Then the merged graph, denoted $D_p[G_1, G_2, \ldots, G_{n_p}]$, is the directed graph obtained from $D_p$ by adding an arc from $(a, k)$ to $(a', k')$ whenever the graph $G_k$ has an arc from $a$ to $a'$.

The above definition is illustrated in Fig. 5. Let $D_p$ be the graph of production $A \rightarrow aAbB$ in Fig.2, let $G_2$ and $G_4$ be the graphs shown in Fig.5 (on top) and let $G_1$ and $G_3$ be the empty graphs (corresponding to the terminals $a$ and $b$ with no attributes). The merged graph $D_p[G_1, G_2, G_3, G_4]$ is shown in Fig.5; for instance, the arc from $(i_1,2)$ to $(s_1,2)$ originates from the arc from $i_1$ to $s_1$ in $G_2$. 


Figure 4

$D_p[G_1, G_2, G_3, G_4]$

$(i_1,0) (s_1,0) (s_2,0)$

$(i_1,2) (s_1,2) (s_2,2) (i_1,4) (i_2,4) (s_1,4)$

Figure 5
Algorithm 1 - Construction of the Sets of Characteristic Graphs

(i) Initialization: For each $X \in V_N$ let $C_X$ be the empty set, and for each $X \in V_T$ let $C_X$ contain a single graph whose nodes are $A(X)$ and which has no arcs.

(ii) Repeat until no more graphs can be added to any of the sets $C_X$:

For each production $p: X_o \rightarrow X_1X_2 \ldots X_n$ and for every choice of $n_p$ graphs $C_1, \ldots, C_{n_p}$ s.t. $C_k \in C_{X_k}$ for $k = 1, \ldots, n_p$, form the graph $C'$ with node set $A(X_o)$, s.t. $C'$ has an arc from node $i$, $i \in I(X_o)$, to node $s$, $s \in S(X_o)$, whenever the merged graph $D_p[C_1, C_2, \ldots, C_{n_p}]$ has a path from $(i, 0)$ to $(s, 0)$. If $C'$ is not yet in $C_{X_o}$ then add it to $C_{X_o}$. □

It is clear that the above process must ultimately terminate with no more graphs created, since there exist only finitely many such graphs. Note that in Algorithm 1 the first production chosen after the initialization is one whose right side consists only of terminals, because for all $X \in V_N$, $C_X$ is still empty. The production $p$ chosen will yield a graph $C'$ which is added to $C_{X_o}$ even if it contains only nodes and no arcs. The next production chosen can now contain on its right side the nonterminal $X_o$ of the previous production plus all the terminals. As the algorithm proceeds, there may be for every production $p$ multiple choices for $C_k$ for each nonterminal $X_k$ on the right side.

Figure 6 shows the sets of characteristic graphs generated by Algorithm 1 for the attribute grammar of Example 1.

Knuth has shown that for each $X \in V_N$ the set $C_X$ constructed in Algorithm 1 is precisely the set of all characteristic graphs of $X$. 
Thus, for every semantic tree \( T \) with root \( X \), there is in \( C_X \) a corresponding characteristic graph of \( X \) (which coincides with the root dependency graph of \( T \)), and vice-versa, for each graph in \( C_X \) there corresponds at least one semantic tree with root \( X \). From now on, the root dependency graph of a semantic tree \( T \) will be referred to as the characteristic graph of the tree \( T \).

Knuth's circularity test can now be stated.

**Theorem 1:** \([K1,2]\) : An attribute grammar is non-circular iff for each production \( p \), none of the merged graphs of the form \( D_p[C_1,C_2,\ldots,C_n] \), for any choice of \( C_i \in C_{X_i}, i = 1,\ldots,n_p \) contains an oriented cycle.

We shall now define for each nonterminal \( X \) another directed graph, called the "superimposed graph" of \( X \), which is obtained by superimposing all the characteristic graphs in \( C_X \).

**Definition:** For each \( X \in V_N \), the superimposed graph of \( X \), denoted \( S_X \), is a graph with node set \( A(X) \), and an arc from inherited attribute \( i \) to synthesized attribute \( s \) iff there exists a characteristic graph \( C \in C_X \) with one arc from \( i \) to \( s \).

It follows that the superimposed graph \( S_X \) has an arc from \( i \) to \( s \) iff there exists a semantic tree \( T \) in which there is a chain of dependencies from \( i \) to \( s \).

Figure 7 shows the superimposed graph of the nonterminals of the attribute grammar of Example 1.
4. AN OVERVIEW OF THE EVALUATION PROCESS

Given a semantic tree with root $S$ (the start symbol), the evaluation process consists of evaluating the semantic functions of the attribute occurrences in the tree. Sometimes the goal of the evaluation process is defined to be the evaluation of some distinguished attribute of the root $S$. Hence we shall consider evaluation to be complete only when all attribute occurrences in the tree have been evaluated.

The evaluation process is carried out in two phases. The first phase serves as a preparation for the evaluation itself, which takes place in the second phase.

The first phase consists of a depth-first left to right postfix traversal of the semantic tree. The computation in each node takes place after all of its sons have been computed. During this phase, the translator will compute for each node $N_X$ (labelled by $X \in V$) of the semantic tree, the characteristic graph of the subtree rooted at $N_X$. This characteristic graph belongs to the set $C_X$ of characteristic graphs of $X$, and represents the dependencies among the attributes of $N_X$ imposed by the structure of the subtree of $N_X$.

During the second phase, the translator will again process the semantic tree; however, this time the order in which the nodes are processed is not known a priori, but is determined dynamically at runtime according to the individual structure of the tree.

We may view the second phase as a processor with control that always points to some "current" node of the semantic tree. While at a node $N_X$, the processor may perform one of the following types of elementary
actions (or instructions):

(i) **CALL**(*a*, *k*) - call for the evaluation of semantic function \( f^p_{a,k} \), where \( p \) is the production applying at the node \( N_X \).

(ii) **TRANSFER DOWN** - transfer control down to a specific son of the current node \( N_X \).

(iii) **TRANSFER UP** - transfer control up to the father of the current node.

Starting with control pointing to the root of the semantic tree, the translator will process the tree until all attribute occurrences in the tree have been evaluated, at which point control returns.

Let us examine the evaluation process from the viewpoint of an individual node \( N_X \). During evaluation control is moving around the tree, and in the meanwhile the processor computes new attribute values. From the point of view of our node \( N_X \), nothing seems to happen until for the first time control arrives at \( N_X \) from above. Some semantic functions associated with the production applying at \( N_X \) may then be evaluated, and afterwards control leaves \( N_X \) in some direction. Later on control returns from the same direction it left, perhaps some more attribute occurrences are evaluated, and again control leaves the node. This goes on until eventually all attribute occurrences of the production applying at \( N_X \) have been computed and control leaves \( N_X \) going towards the root, never to return. Thus from a local point of view, evaluation consists of comings and goings of control, interspersed with evaluations of attributes performed while at our node. We shall refer to a part of the evaluation process from the point control leaves our node \( N_X \) towards its \( k \)-th neighbour, and until the next time control returns to our node, as a visit to the \( k \)-th neighbour of \( N_X \).
At each stage of the above evaluation process, the choice of the next elementary action to be taken by the evaluator at the particular node \( N_X \) may depend on the following:

(a) The production \( p \) which applies at the node \( N_X \), and its associated semantic functions;

(b) The current "state of affairs" of the evaluation process, as regards our node \( N_X \), i.e., the set of attribute occurrences of the production \( p \) applying at \( N_X \), which are currently available.

(c) The dependencies among the attributes of \( N_X \), which are induced by the subtree rooted at \( N_X \); these dependencies are represented by the characteristic graph of this subtree, which has been computed during the first phase.

To keep track of the sets of available attributes and to regulate the evaluation process at the particular node \( N_X \), the Local Control Automaton, \( \text{LCA} \) for production \( p \) is introduced. The Local Control Automaton is an ordinary finite state automaton augmented with some special features in its transition function. Each state of the Local Control Automaton represents a set of attribute occurrences of production \( p \), which are available upon entering this state. Each transition in the automaton represents an instruction of one of the types CALL\((a,k)\), TRANSFER DOWN or TRANSFER UP.

Associated with each node of the semantic tree is the Local Control Automaton \( A_p \) of the production \( p \) which applies at the node. The LCA \( A_p \) is responsible for directing control when it arrives at the node, and it calls for the evaluation of the semantic functions of production \( p \). Whenever control arrives at a node, it (re-)activates its LCA; if it is the first time control reaches this node, then the LCA starts from its
initial state; otherwise it resumes action from the state in which it left off.

In our translator control must be capable of passing parameters among the various LCA's. This passing of parameters serves as a means of communication between the LCA's associated with the nodes and is used for two main purposes:

(a) When control is transferred down to a son of the current node, the translator has to inform the receiving LCA what activities took place while the latter was inactive. Specifically, the parameter passed down is a set of inherited attributes of the receiving node, which are currently available. These inherited attributes are used by the receiving LCA to determine its next move, and their values may be used by the translator as arguments in computing semantic functions.

(b) When control is transferred up in the tree, the translator has to inform the father's LCA about the activities which took place in the subtree just visited. Specifically, the parameter returned to the father is the set of synthesized attributes of the son just visited, which are currently available.

In both cases, we see that the parameter carried along by control is a set of currently available attributes of the node visited. Such parameters will be called transmitted sets.

The evaluation process described above uses the LCA's as basic building blocks. In order to obtain a translator for a given attribute grammar, a finite set of LCA's, one for each production, will have to be constructed.
5. LOCAL CONTROL AUTOMATA

As mentioned above, an LCA $A_p$ will be associated with each production $p$ of the attribute grammar. At runtime, to each node of the semantic tree there will be attached the LCA of the production applying at the node. Whenever control reaches a particular node, the LCA of the node will be reactivated, and will dictate the next elementary actions to be taken by the translator. When eventually control is transferred to a neighbour node, the LCA will be suspended and will send along with control a parameter - the transmitted set.

Before we can formally describe the LCA's and the way they communicate via the transmitted sets, the following definition is needed.

**Definition:** Consider a production $p: X_0 \rightarrow X_1 X_2 \ldots X_{n_p}$.

Let $A(p)$ denote the set of all attribute occurrences of $p$. For each $k = 0, 1, \ldots, n_p$, define:

(a) For each set of attributes $\mathcal{T} \subseteq A(X_k)$

$$\mathcal{T} \ast k = \{(a, k) \mid a \in \mathcal{T}\}.$$  

(b) For each set of attribute occurrences $\mathcal{A} \subseteq A(p)$

$$\mathcal{A}/k = \begin{cases}  
\{a \mid (a, k) \in \mathcal{A} & \text{if } k = 0 \text{ then } a \in S(X_k) \} \\
\{a \mid (a, k) \in \mathcal{A} & \text{if } 1 \leq k \leq n_p \text{ then } a \in I(X_k) \} 
\end{cases}.$$  

The operator $\ast$ transforms a set of attributes of $X_k$ onto the corresponding set of attribute occurrences of production $p$, while the operator $\mathcal{A}/k$ maps a set $\mathcal{A}$ of attribute occurrences of production $p$ onto a set $\mathcal{A}/k$ of attributes of $X_k$. Note that for $k = 0$, $\mathcal{A}/k$ contains only synthesized
attributes, while for $1 < k < n_p$, $A/k$ contains only inherited attributes. $A/k$ will be called the $k$-th projection of $A$.

**Definition:** A Local Control Automaton (LCA) $A_p$ for production $p: X_0 \rightarrow X_1X_2 \ldots X_{n_p}$ is an ordinary finite state automaton augmented with some features in its transition function.

The set of states is divided into two subsets:

(a) **Active states.** Each active state is labelled with a set of attribute occurrences $A \subseteq A(p)$. No two active states are labelled with the same set $A$. An active state will therefore be identified by its label $A$, and will be denoted by $[A]^p$ (the superscript $p$ will be omitted when the production $p$ is understood).

(b) **SUSPEND states.** Each SUSPEND state is labelled by a pair $(k,A)$, where $0 < k \leq n_p$ and $A \subseteq A(p)$ as above. For each pair $(k,A)$ there is at most one SUSPEND state labelled with this pair. We shall denote this state by $\text{SUS}[k,A]^p$ (with the superscript $p$ omitted when $p$ is understood).

During the evaluation process, whenever LCA $A_p$ is found in an active state $[A]$ or enters a SUSPEND state $\text{SUS}[k,A]$ ($0 < k \leq n_p$), then $A$ is precisely the set of attribute occurrences of production $p$ which have been evaluated so far.

There are several types of transitions in an LCA:

Transitions leaving Active states: Each transition leaving an active state corresponds to one of the following three types of instructions:

(i) $\text{CALL}(a,k)$ - call for the evaluation of semantic function $f^p(a,k)$;

(ii) $\text{TRANSFER}_0$ - transfer control up to the 0-th neighbour (i.e. father) of the current node;
(iii) **TRANSFER** \(_k\) \((1 \leq k \leq n_p)\) - transfer control down to the \(k\)-th neighbour (i.e. \(k\)-th son) of the current node.

Every active state \([A]\) may have precisely one of the following three types of exists:

1. **CALL transition** - Exit from \([A]\) leading to another active state \([A']\). Such a transition is labelled by \(\text{CALL}(a,k)\), where \((a,k) \in A(p)-A\) and \(A' = A \cup \{(a,k)\}\). There can be no other transition leaving state \([A]\).

A transition \(\text{CALL}(a,k)\) represents an instruction for evaluating the semantic function \(f^p(a,k)\) and storing the result in the appropriate field of the semantic tree. Such an instruction is executable only if the semantic function \(f^p(a,k)\) is ready to evaluate. Therefore we shall impose the additional requirement that whenever transition \(\text{CALL}(a,k)\) leaves state \([A]^p\), all attribute occurrences \((a',k') \in A_p\) which are used as arguments in \(f^p(a,k)\) are contained in \(A\).

Recall that if \(a \in S(x_k)\) then \(k\) equals 0, and if \(a \in I(x_k)\) then \(1 \leq k \leq n_p\). Figure 8 illustrates a CALL transition; note that the active states are represented by circles.

2. **Unconditional TRANSFER** Transition - A transition from active state \([A]\) to a SUSPEND state \(\text{SUS}_k[A]\) (same \(A\)), where \(0 \leq k \leq n_p\), and \(x_k \in V_N^*\). The transition is labelled with \(\text{TRANSFER}_k\) and represents an instruction to the translator to transfer control to the \(k\)-th neighbour LCA. Again this is the only transition leaving state \([A]\). Fig. 9 illustrates an unconditional TRANSFER transition. Note that the SUSPEND state's are represented by squares.
(iii) Conditional TRANSFER Transitions - A set of \( t+1 \) exists (for some \( 1 \leq t \leq n_p \)) from active state \([A]\), leading to \( t+1 \) distinct SUSPEND states \([A], [A], \ldots, [A], [A]_0 \) (same \( A \) as for the active state), where \( 1 \leq i_1 < i_2 < \ldots < i_t \leq n_p \) and \( X_j \) is a nonterminal for \( j = i_1, i_2, \ldots, i_t \). The transition into state \([A]_j \) is labelled by the pair \(( \text{TRANSFER}_j, c(j) )\), where \( c(j) \) is a subset of \( C_{X_j} \), the set of characteristic graphs of nonterminal \( X_j \). The last transition into state \([A]_0 \) is labelled by \( \text{TRANSFER}_0 \) only. Figure 10 illustrates a set of conditional TRANSFER transition.

The above set of conditional TRANSFER transitions leaving state \([A]\) has the following meaning. During the evaluation process, of a given semantic tree, if LCA \( A_p \) is associated with node \( N_{X_0} \), and \( A_p \) enters state \([A]\), the exit to be taken from \([A]\) depends on the characteristic graphs of the sons of \( N_{X_0} \) in the tree (recall that for each node in the tree, the characteristic graph of the subtree rooted at the node has been computed during the first phase of evaluation).

Specifically, for each \( j = i_1, i_2, \ldots, i_t \), the conditional TRANSFER transition labelled by \(( \text{TRANSFER}_j, c(j) )\) will be called admissible iff the characteristic graph attached to the \( j \)-th son of \( N_{X_0} \) in the semantic tree belongs to \( c(j) \). As we shall see in Section 7, a transition \(( \text{TRANSFER}_j, c(j) )\) is admissible precisely when the corresponding visit to the \( j \)-th son of the current node \( N_{X_0} \) is guaranteed to be
"fruitful", i.e., to produce at least one new synthesized attribute of the j-th son.

Therefore when the LCA of node \( N_{X_0} \) enters state \([A]\), the conditional TRANSFER transitions leaving state \([A]\) will be checked one by one until the first admissible transition is encountered; this will be the transition to be taken. If, however, none of the first \( t \) transitions is admissible, then the TRANSFER transition will be taken. Thus control will be transferred up to the father of \( N_{X_0} \) only if in the current situation, no more attribute occurrences can be evaluated for production \( p \), not even after some more processing of the subtree of \( N_{X_0} \). Before any further evaluation at \( N_{X_0} \) can take place, some new inherited attributes of \( N_{X_0} \) must become available.

Transmitted Sets. Whenever a TRANSFER \(_k\) instruction (either conditional or unconditional) leading into state \( S_{US_k[A]} \) is executed, a transmitted set \( T \) is passed as parameter to the \( k \)-th neighbour LCA along with control. Specifically, this set is \( T = A/k \), the projection of \( A \) on the \( k \)-th neighbour. For \( k = 0 \) this set \( T \) consists of all synthesized attributes of \( X_0 \), which are currently available, and for \( k = 1, 2, \ldots, p \), \( T \) consists of all inherited attributes of \( X_k \), which are currently available.

Transitions Leaving SUSPEND States. Every state \( s = S_{US_k[A]} \) may have several, say \( \ell \geq 1 \), exists, each labelled with a distinct set \( T_i \), \( 1 \leq i \leq \ell \), of attributes of \( X_k \), s.t. \( T_i \times k \subseteq A \). Let these exists be denoted by \( E(s, T_1), E(s, T_2), \ldots, E(s, T_\ell) \). If \( k = 0 \) then \( T_1 \) is
a set of inherited attributes of the d.h.s. \( X_0 \), and if \( k > 0 \) then \( T_i \) is a set of synthesized attributes of the son \( X_k \). The exit labelled by \( E(s, T_i) \) will lead to the active state \([A \cup T_i \ast k]\). Since by assumption \( T_i \ast k \notin A \), no transition leaving state \( SUS_k[A] \) will lead into state \([A]\) again (i.e. a cycle cannot be created). Fig. 11 illustrates the exists from a SUSPEND state.

To explain the meaning of the sets \( T_i \), recall that during runtime, when the LCA \( A_p \) attached to node \( N_{X_0} \) enters state \( SUS_k[A] \), a TRANSFER \( k \) instruction leading into this state is executed. The LCA of node \( N_{X_0} \) is then suspended, and control is transferred to the \( k \)-th neighbour LCA. After some activities take place in other regions of the tree, control finally returns to the node \( N_{X_0} \) from its \( k \)-th neighbour, carrying with it a transmitted set \( T \). At this point the LCA \( A_p \) at \( N_{X_0} \) is reactivated, starting from the same state \( SUS_k[A] \). The exit taken from this state is chosen according to the transmitted set \( T \) received; namely, it will be precisely the exit whose label \( T_j \) coincides with \( T \). In this way, the transmitted set returned by the \( k \)-th neighbour determines the next move of the LCA after it resumes action in state \( SUS_k[A] \). The sets \( T_i \) labelling the exists from \( SUS_k[A] \) represent all possible transmitted sets, which can be returned by any \( k \)-th neighbour LCA in any particular situation.

**Initial and Final States.**

1. Each LCA has precisely one initial state, a state which has no in-arcs, which is a state \( SUS_0[A] \) (for some set \( A \)).

2. Each LCA has at least one terminal state, a state with no out-arcs. Every terminal state is a state \( SUS_0[A] \) (for some set \( A \)).
Figure 8

\[ A' = A \cup \{(a,k)\} \]

Figure 9

Figure 10

\[ A' = A \cup \{(a,k),(b,k)\} \]

\[ A'' = A \cup \{(a,k),(d,k)\} \]

\[ A''' = A \cup \{(b,k),(c,k)\} \]
Let us now summarize the structural properties of LCA's:

1. Each LCA is acyclic. This is because for every path in the LCA, the sets of attributes $A$ labelling both active and SUSPEND states along the path must be ordered by inclusion, and no more than two consecutive states can be labelled by the same set.

2. Every SUSPEND state $\text{SUS}[A]$ (excluding the initial state) has precisely one in-arc (from the active state with same label $[A]$), while there may be several in-arcs for any active state.

Figure 12 illustrates a complete set of LCA's for the attribute grammar of Example 1. Each state is given a number, $C$ is an abbreviation for CALL transitions, the set $A$ of available attribute occurrences and the label $\text{TRANSFER}_k$ were omitted.

6. HOW TO EVALUATE A SEMANTIC TREE WITH A GIVEN SET OF LCA's

Suppose that we are given a translator for some noncircular attribute grammar. The translator is made up of a finite set of LCA's, one for each production. The construction of such LCA's for any given noncircular attribute grammar will be described in detail in Section 7.

In order to carry out the evaluation process, two variables will be associated with each node $N_X$ of the semantic tree:

(i) A variable indicating one of the characteristic graphs $C_i$ of non-terminal $X$, namely the characteristic graph of the subtree rooted at node $N_X$. This variable is computed during the first phase of the evaluation process.

(ii) A variable which, during the second phase, will store the state of the LCA of $N_X$ at the time it is suspended, while control wanders in other regions of the semantic tree. When control returns to $N_X$, ...
the LCA of $N_x$ will be reactivated, starting from this state. This variable is initialized during the first phase.

**The First Phase** The first phase consists of a depth-first left to right postfix traversal of the semantic tree. During this traversal, with each node of the tree, there will be associated the characteristic graph of the subtree rooted at the node. The characteristic graph associated with a terminal node $N_t$, $t \in V_T$, will be the trivial graph with node set $A(t)$ and no arcs. Due to the postfix manner of traversal, when the translator reaches a nonterminal node $N_x$ to compute its characteristic graph, the characteristic graphs of all sons of $N_x$ have already been computed.

Let us describe the construction of the characteristic graph for node $N_x$, where the production applying at the node is $p: X_o \rightarrow X_1 X_2 \ldots X_p$, and $X = X_o$, (see also Section 3). Let the characteristic graphs associated with nodes $N_{x_i}$, $i = 1, 2, \ldots, n_p$, be $D_i$. Form a graph $C$ whose nodes are $A(X_o)$ such that $C$ has an arc from inherited attribute $i$ to synthesized attribute $s$ whenever the merged graph $D_1 \cup D_2 \cup \ldots \cup D_{n_p}$ has a path from $(i, 0)$ to $(s, 0)$. $C$ is precisely the characteristic graph of node $X_N$.

To avoid the need for storing characteristic graphs as part of the translator, and then at runtime comparing the characteristic graph $C$ constructed above against all graphs in the set $C_{X_0}^X$, some kind of Gödel numbering for graphs can be used. Each graph will be identified by its "Gödel number" (which will constitute the first variable of the node $N_x$) and during runtime comparison will take place only between the Gödel numbers.
An alternative way for obtaining the characteristic graph \( C \) would be to use a look up table, which gives for each production \( p \) and for each set of characteristic graphs \( \{D_1, D_2, \ldots, D_n\} \) as above, the corresponding characteristic graph \( C \). Such a look up table can be prepared once and for all at construction time, while performing the circularity test (see Section 3). When using this method, there is no need for Gödel numbering and each graph will be represented simply by its index in the set \( C_{X_0} \). The graphs themselves need not be kept in memory, but the look-up table will have to be stored as part of the translator.

The above two alternatives for computing the characteristic graph \( C \) constitute an instance of the usual trade-off between time and storage.

After determining the characteristic graph of \( N_X \), and the appropriate LCA \( A_p \) according to the production \( p \) applying at node \( N_X \), the second variable of the node is initialized to the initial state of \( A_p \).

The Second Phase. The manner in which the semantic tree is traversed during the second phase is custom-tailored to the individual structure of the tree.

Evaluation begins by sending control to the root LCA \( A_{p_0} \). Control begins executing the instructions of \( A_{p_0} \), which after a while will transfer control down to one of its sons. Thus control will start wandering in the tree from node to node in an order dictated by the LCA's, and by the characteristic graphs associated with the nodes. At each point in time during the evaluation process, all LCA's of the tree are dormant except for one which is active. The active LCA may call for the evaluation of semantic functions, or it may direct control to one of its neighbour LCA's,
Eventually after some finite number of steps, when all subtrees of the semantic tree have been evaluated, control will return for the last time to the root LCA, which (possibly after executing a few CALL instructions) will enter its final state SUS_0[A], and control will leave the tree from above. Evaluation is then complete.

Algorithm 2 - Evaluation of a Semantic Tree

1. Perform the first phase.

2. Transfer control to the initial state of the root LCA \( A_0 \) along with an empty transmitted set.

3. Repeat until a terminal state of \( A_0 \) is entered:

   Let the currently active LCA \( A_p \) be in state \( s = \text{SUS}_k[A] \) and let \( T \) be the transmitted set received upon reactivation of this LCA.

   (i) Take the exit from \( s \) labelled by \( T \); let \( t \) be the active state entered:

   (ii) Execute the instructions of \( A_p \), starting from state \( t \), until a SUSPEND state, say \( \text{SUS}_{k'}[A'1] \), is reached.

   (iii) Transfer control to the \( k' \)-th neighbour LCA along with the transmitted set \( A'/k' \). Reactivate the \( k' \)-th neighbour LCA, starting from the SUSPEND state stored in the second variable of the \( k' \)-th neighbour node.

\[ \square \]
Figure 13 illustrates the evaluation process of a semantic tree of the grammar of Ex. 1 according to Algorithm 2, using the set of LCA's in Figure 12. Starting from the root, control moves down to node 2 (due to entering state 3 in LCA $A_0$), then down to node 3; for each node visited, the attributes evaluated at the node are indicated in the figure; e.g. at node 3, $s_1$ and $s_2$ are evaluated and then control moves up back to the root. After evaluating the inherited attribute $i_1$ of node 2, control moves down again, this time to node 4; after evaluating $s_1$ for node 4, then $s_2$ of node 2, control returns to the root, and evaluation is complete. We see that every visit to every node in the tree produces at least one new attribute value.

To ensure that step 3(i) in Algorithm 2 can always be carried out, the set of LCA's must satisfy the following condition.

The Closure Condition: Whenever at runtime, control is transferred to an LCA $A_p$ with transmitted set $T$, the resumed SUSPEND state $s$ in $A_p$ must have an exit labelled by $T$.

The above Closure Condition is guaranteed to hold by the translator construction to be presented in Section 7. This condition assures us that Algorithm 2 will always terminate upon entering a final SUS0 state of the root LCA $A_{p_0}$.

Theorem 2: If the set of LCA's for attribute grammar $G$ satisfies the Closure Condition, then Algorithm 2 terminates for each semantic tree $T$ after executing $O(|T|)$ elementary operations (where $|T|$ indicates the size of $T$).
Figure 13

Parse Tree

Evaluation Process
Proof: Every node in the tree $T$ is associated with an acyclic LCA, and each execution of the repeat step (3) causes at least one move in one of the LCA's towards a final state, which is a $SUS_0$ state. By the Closure Condition and the above argument the evaluation will proceed until finally a terminal $SUS_0$ state of the root LCA $A_{Po}$ is reached. Clearly the number of times the repeat loop (3) is executed is proportional to the number of LCA's taking part in the evaluation process, i.e. to the size of the tree, and every execution of the repeat loop takes a constant number of elementary operations (when the evaluation of a semantic function is considered to be an elementary operation). Hence the total number of elementary operations is $O(|T|)$.

The Completion Attribute: To guarantee that the above evaluation algorithm will result in the complete evaluation of every semantic tree, a slight modification of the attribute grammar has to be made. A new dummy synthesized attribute, called the completion attribute, is added to each nonterminal.

For each production $p: X_o \rightarrow X_1 \ldots X_p$, the completion attribute of $X_o$ is defined in terms of all inherited attributes of $X_o$ and of all synthesized attributes (including the completion attribute) of all sons $X_k, 1 \leq k \leq n_p$.

An attribute grammar to which a completion attribute as above has been added for each nonterminal will be called an augmented attribute grammar.

In an augmented grammar, the completion attribute is dummy in the sense that it is not computed at runtime. However, this attribute enables us to construct such translators in which the complete evaluation of every semantic tree will be enforced. For this reason we add to the definition of LCA's the following condition:

Completion Condition: For each terminal state $SUS_0[A]_{Po}$ of the root LCA
the set \( A \) must contain the completion attribute of the start symbol \( S \).

It then follows from the Completion Condition (using simple induction on the levels in the tree) that for any semantic tree, when Algorithm 2 terminates by entering a final state of the root LCA \( A_{p_0} \), all attribute occurrences in the tree have been evaluated. We therefore have:

**Theorem 3:** For every augmented attribute grammar \( G \) and for every set of LCA's for \( G \) which satisfies both the Closure Condition and the Completion Condition, Algorithm 2 fully (and correctly) evaluates each semantic tree of \( G \) in time proportional to the size of the tree.

7. **THE CONSTRUCTION OF A TRANSLATOR FOR A GIVEN ATTRIBUTE GRAMMAR.**

So far we have described the translation process for a given semantic tree assuming that the translator (represented mainly as a set of LCA's) is given. In this section we show how a complete set of LCA's satisfying the Closure Condition and the Completion Condition can be systematically built for each non-circular attribute grammar.

The LCA's are constructed in parallel, state by state and transition by transition, until all of them are complete. The construction is based on simulation of all possible situations that can arise at runtime.

Before starting the actual construction of LCA's, some preliminary computation takes place. For each nonterminal \( X \), the set of characteristic graphs \( C_X \), and also the superimposed graph \( S_X \) are constructed, and Knuth's circularity test is performed. After this preliminary computation the construction process is initialized by defining for each production an LCA consisting of the initial state alone. As the algorithm proceeds the LCA's are expanded.
by adding new states and new transitions according to rules specified below.

In Section 5 we have seen how at runtime, when in some LCA $A_p$ attached to some node in the tree, a TRANSFER$_k$ instruction leading into state SUS$_k[A]$ is executed, the LCA is temporarily suspended and the LCA of the $k$-th neighbour, after receiving the transmitted set $T = A/k$, is reactivated, starting from the SUSPEND state $s'$ in which it was last suspended, and leaves state $s'$ via the exit labelled by $T$. We have to ensure by our construction of LCA's that an appropriate exit labelled by $T$ will always exist in the $k$-th neighbour LCA, whichever it may be (i.e. that the Closure Condition will be satisfied). For this reason, whenever we add to LCA $A_p$ a TRANSFER$_k$ transition leading into state SUS$_k[A]$, we must be able to identify all SUSPEND states in all LCA's, which might be reactivated as a consequence of executing this TRANSFER$_k$ instruction; each of these SUSPEND states must have an exit labelled by the transmitted set $T = A/k$.

**Definition:** For every non-initial SUSPEND state SUS$_k[A]^p$ in LCA $A_p$, define the set $\text{REACTIVATE}(\text{SUS}_k[A]^p)$ to be the set of all SUSPEND states in all LCA's to which, under certain conditions at runtime, control might be transferred as a consequence of suspending LCA $A_p$ in state SUS$_k[A]$. The states in $\text{REACTIVATE}(\text{SUS}_k[A]^p)$ may belong to any LCA $A_p$, which, in some semantic tree, interacts with LCA $A_p$ as its "$k$-th neighbour". Therefore if $k \neq 0$ then $p[k]$ must coincide with $p'[0]$ and if $k = 0$ then $p[0]$ must coincide with one of the sons of $p'$.

The next definition enables us to keep track of the states in which the neighbour LCA's were left after the "most recent" visit.
Definition: For each exit $E[s,T]$ from a SUSPEND state $s = SUS_k[A]^P$, define the set $\text{SOURCE}(E[s,T])$ to be the set of all SUSPEND states $SUS_{k'}[A']^P$ in all LCA's, which at runtime may pass control to LCA $A'$ and cause it to be reactivated in state $s$ and take the exit $E[s,T]$. Clearly $\text{SOURCE}(E[s,T])$ will consist only of states of the form $SUS_{k'}[A']^P$ whose transmitted set $A'/k'$ coincides with $T$. Moreover, if $k = 0$, then $p'[k'] = p[0]$, while if $k \neq 0$ then $p[k] = p'[0]$.

During construction we shall be able to compute the sets $\text{REACTIVATE}$ and $\text{SOURCE}$ by examining all possible ways in which two neighbour LCA's may communicate. In particular, to keep track of the "most recent" visit to the $k$-th neighbour LCA, we define for every SUSPEND state $s$, a set of transitions in the same LCA, called the $\text{PREDecessors}$ of state $s$. For each state $s = SUS_k[A]^P$ in $A^P$, let $\text{PRED}(SUS_k[A]^P)$ be the set of all exits $E[s',T]$ in $A^P$ such that $s' = SUS_k[A']^P$ for some $A' \subseteq A$ and the same $k$, and such that there exists a path in $A^P$ starting in $E[s',T]$ and ending in state $s$, which does not go through another $SUS_k$ state with the same $k$. For instance, in the LCA $A_1$ shown in Figure 12, $\text{PRED}(s_{23}) = \{E[s_{13},\{(i_1,0)\}], E[s_0,\{(i_1,0)\}]\}$, while $\text{PRED}(s_9) = \emptyset$, which means that the $\text{TRANSFER}_2$ instruction leading into $s_9$ will cause the first visit to the second son of the node in which production 1 applies.

We are now in a position to introduce the formula for computing the $\text{REACTIVATE}$ set. For each non-initial SUSPEND state $s = SUS_k[A]^P$,

(7.1) $\text{REACTIVATE}(s) = \begin{cases} \text{if } \text{PRED}(s) \neq \emptyset \text{ then } \bigcup_{E[s',T] \in \text{PRED}(s)} \text{SOURCE}(E[s',T]) \bigcup \{s^P \mid s^P \text{ is the initial state in } A^P, \text{ and } p[k] = p'[0]\} \end{cases}$
The first part of the formula takes care of all SUSPEND states in which other LCA's (all possible k-th neighbour's of \( A_p \)) were suspended after the most recent visit made by LCA \( A_p \). However, if (for \( k \leq 0 \)) the TRANSFER\(_k \) instruction leading into state \( \text{SUS}_k[A]^P \) causes the first visit to neighbour \( k \) (i.e. state \( \text{SUS}_k[A]^P \) is the first appearance of a SUS\(_k \) state on the path from the initial state), then the second part of the formula gives all the initial states of the LCA's to which the TRANSFER\(_k \) instruction may branch. It is impossible that the same state \( \text{SUS}_k[A]^P \) (\( k > 0 \)) will be the first SUS\(_k \) state occurring on one path from the initial state, and a non-first occurrence of a SUS\(_k \) state (same \( k \) still) on another such path. This is because in the former case the set \( A \) cannot include any synthesized attribute of the \( k \)-th neighbour, while in the latter case \( A \) must include at least one such attribute. As for \( k = 0 \), a state SUS\(_0[A]^P \) which is not an initial state will always have at least one predecessor, and hence the first part of the formula will be applicable.

For the construction algorithm given below, we still need two more technical definitions. For each active state \( [A]^P \), define the set \( \text{FORWARD}_k([A]^P) \) to be the set of all SUSPEND states \( s = \text{SUS}_k[A'][P] \) (in the same LCA) such that \( A' \supseteq A, k \) is the same as above, and \( s \) is the first SUS\(_k \) state occurring on a path starting at \( [A]^P \). Since there may be several paths starting at \( [A]^P \), the \( \text{FORWARD}_k \) set may include several SUS\(_k[A'][P] \), each with a distinct set \( A' \). Also define for every active state \( [A]^P \) the set:

\[
\text{FORWARD}([A]^P) = \bigcup_{k=0}^{n_p} \text{FORWARD}_k([A]^P).
\]

For instance, in \( A_1 \) of Figure 12, \( \text{FORWARD}_0(s_{11}) = \{ s_{19}, s_{13} \} \) and
FORWARD_4(s_{11}) = \{s_{12}, s_{16}\}, thus FORWARD(s_{11}) = \{s_{19}, s_{13}, s_{12}, s_{16}\}.

Construction Conventions. At any point in time during the construction process, a set of partially constructed LCA's, one for each production, is available. The construction consists of adding new states and new transitions to the LCA's in a specific order, so as to ensure that the Closure Condition will hold.

Each SUSPEND state s is associated with its REACTIVATE set, and each transition E[s, T] leaving a SUSPEND state s is associated with its SOURCE set SOURCE(E[s, T]). These sets are kept in memory and updated during the construction process. The other sets, PRED and FORWARD are computed from time to time but not kept in memory.

A central role in the construction process is played by the queue Q. Q consists of couples of the form: \((E(s^P_p, T), s^{P'}_p)\), where \(s^P\) and \(s^{P'}\) are SUSPEND states in \(A_p\) and \(A_p\), respectively and \(E[s^P, T]\) is an exit from \(s^P\) labelled by T. The meaning of such a couple appearing in Q is that exit \(E[s^P, T]\) has to be created in \(A_p\) (unless already there), and state \(s^{P'}\) must be added to SOURCE(E[s^P, T]). Note that by definition of the SOURCE set it follows that T must be precisely the transmitted set of \(s^{P'}\).

The construction process is based on retrieval and processing of couples from Q, one at a time. The processing of a couple \((E[s^P, T], s^{P'})\) consists of creating the exit \(E[s^P, T]\) and adding \(s^{P'}\) to its SOURCE set. The exit will lead to an appropriate active state \([A U T * k]\) which must also be added if absent. This new active state will then be "developed", which in turn may cause the addition of some new couples to Q. The process goes on until eventually Q remains empty, at which point construction is completed.
Developing an Active State. Recall that an active state \([A]^P\) may have three types of exits: a single CALL\((a,k)\) transition, a single unconditional TRANSFER\(_k\) transition, or a set of conditional TRANSFER transitions.

In developing the active state \([A]^P\), we first try to construct a CALL\((a,k)\) transition out of this state, for some attribute occurrence \((a,k)\) of production \(p\) s.t. \((a,k) \in A\). Such a transition can be created only if the semantic function \(f^P_{(a,k)}\) is ready to evaluate, i.e. depends only on attribute occurrences in \(A\). If such an exit is indeed created, it leads into another active state \([A \cup \{(a,k)\}]\) and we proceed to develop this new state (unless this state already existed before in \(A_p\)).

In case no CALL\((a,k)\) exit out of state \([A]^P\) can be created, we attempt to construct an unconditional TRANSFER\(_k\) exit for some \(k = 1, \ldots, n_p\). If this turns out to be impossible as well, a set of conditional TRANSFER transitions will be constructed. To explain how such transitions are constructed, we need the following important definition.

**Definition:** Let \(A_p\) be the LCA of production \(p: X_0 \rightarrow X_1 X_2 \ldots X_n\); let TRANSFER\(_k[A]^P\) denote TRANSFER\(_k\) transition leaving active state \([A]^P\), for some \(1 \leq k \leq n_p\), and let \(G\) be any directed graph with node \(A(X_k)\) (in fact, \(G\) will be taken to be either one of the characteristic graphs of \(X_k\) or the superimposed graph of \(X_k\)). Define the yield of transition \(\text{TRANSFER}_k[A]\) with respect to graph \(G\) as follows:

\[
\text{YIELD}(\text{TRANSFER}_k[A], G) = \{a : S(X_j) \mid (a,k) \in A, \text{ for all } i \in l(X_k) \text{ with an arc from } i \text{ to } a \text{ in } G, (i,k) \in A\}.
\]

Let \(G = C^{(k)}_j\) be one of the characteristic graphs of nonterminal \(X_k\), \(1 \leq k \leq n_p\). Suppose that during the evaluation of a semantic tree \(T\), LCA \(A_p\) attached to node \(N_{X_0}\), is in active state \([A]\). Furthermore, suppose
that \( c_j^{(k)} \) is precisely the characteristic graph of the subtree \( T_{X_k} \) rooted at the \( k \)-th son of \( X_0 \). Then an arc from inherited attribute \( i \) to synthesized attribute \( a \) exists in \( c_j^{(k)} \) precisely when the dependency graph of the subtree \( T_{X_k} \) contains a path from \( i \) to \( a \). Thus \( \text{YIELD}(\text{TRANSFER}_k[A], c_j^{(k)}) \) contains precisely those not-yet-available synthesized attributes of the \( k \)-th son \( X_k \), which (directly or indirectly) depend only on already available inherited attributes of \( X_k \). Hence the yield is equivalent to the "output" (set of new available synthesized attributes of \( X_k \)) produced by visiting the subtree \( T_{X_k} \). If for some \( k = 1, 2, \ldots, n_p \) the yield of transition \( \text{TRANSFER}_k[A] \) with respect to the characteristic graph of the \( k \)-th son is empty, there is no point in executing a \( \text{TRANSFER}_k \) instruction, because a visit to the \( k \)-th son at this stage will not result in computing new attributes of the \( k \)-th son and will thus be fruitless.

Now let \( G \) be the superimposed graph \( S_{X_k} \) of the \( k \)-th son. If
\[
\text{YIELD}(\text{TRANSFER}_k[A], S_{X_k}) \neq \emptyset
\]
then, regardless of the structure of the \( k \)-th subtree \( T_{X_k} \), a visit to the \( k \)-th son is guaranteed to be fruitful. This is because by definition of \( S_{X_k} \), we have \( \text{YIELD}(\text{TRANSFER}_k[A], c_j^{(k)}) \neq \emptyset \) for each characteristic graph \( c_j^{(k)} \).

In developing an active state \([A]^p\), before creating a \( \text{TRANSFER}_k[A]^p \) exit (for some \( k \neq 0 \) s.t. \( X_k \) is a nonterminal), we have to ensure that under all circumstances at runtime, the corresponding visit to the \( k \)-th son will be fruitful, i.e. will produce some new attribute values for the \( k \)-th son. We start by checking, for each \( k = 1, \ldots, n_p \), the set \( \text{YIELD}(\text{TRANSFER}_k[A], S_{X_k}) \). If for some \( 1 \leq k \leq n_p \) this yield is non-empty, then at runtime, when \( \text{LCA} \cdot A_{p} \) of node \( N_{X_0} \) is in state \( A \), a visit to the \( k \)-th son of \( N_{X_0} \) is guaranteed to be fruitful, regardless of the characteristic graph of the subtree of the \( k \)-th son. In this case, an unconditional \( \text{TRANSFER}_k \) exit
from state $[A]$ will be constructed.

If, however, for each $k = 1, \ldots, n_p$ s.t. $X_k$ is a nonterminal, the yield of $\text{TRANSFER}_k[A]$ w.r.t. $S_{X_k}$ is empty, then a set of conditional $\text{TRANSFER}_k$ exits from state $[A]$ will be constructed. For each $k = 1, \ldots, n_p$, s.t. $X_k$ is a nonterminal, the yield of $\text{TRANS}\text{FER}_k[A]$ w.r.t. each characteristic graph $C_j(k) \in C_{X_k}$ will have to be checked for emptiness.

Let $C(k) \subseteq C_{X_k}$ be the set of all characteristic graphs $C_j(k)$ of $X_k$ for which $\text{YIELD}(\text{TRANS}\text{FER}_k[A], C_j(k)) \neq \emptyset$. For each $1 \leq k \leq n_p$ such that $C(k) \neq \emptyset$ there will be constructed a conditional $\text{TRANS}\text{FER}_k$ transition into state $\text{SUS}_k[A]$; this transition will be labelled by the pair $\langle \text{TRANS}\text{FER}_k, C(k) \rangle$. Thus at runtime, the above transition will be taken only if it is admissible (see Section 5), that is, only if the characteristic graph $C_j(k)$ of the $k$-th son of $N_{X_k}$ belongs to $C(k)$. But this happens precisely when the yield with respect to the characteristic graph $C_j(k)$ is non-empty. Hence at runtime, an $\text{TRANS}\text{FER}_k$ instruction will be executed only if the corresponding visit to the $k$-th son is guaranteed to be fruitful. If however, none of the above conditional $\text{TRANS}\text{FER}_k$ transitions is admissible, then the (last) $\text{TRANS}\text{FER}_0$ exit will be taken, and control will move to the father. The efficiency of our translator stems from the above observation.

**Subroutine DEVELOP([A]$^p$).** Let $p: X_1 \rightarrow X_2 \ldots X_n$;

if there exists a semantic function $f_{(a,k)}^p, (a,k) \in A$ which is ready to evaluate,

then do: (a) create a transition labelled `CALL(a,k)` from state $[A]^p$ into active state $[A \cup \{(a,k)\}]^p$; if the latter active state is not yet included in $A_p$ then create it and call DEVELOP([A \cup \{(a,k)\}]^p);

b) stop.
else for each \( k = 1, \ldots, n \) s.t. \( x_k \in V_N \) do:

if \( \text{YIELD} \left( \text{TRANSFER} \_k \left( A \right)^P, x_k \right) \neq \emptyset \)

then do:

(a) create an unconditional \( \text{TRANSFER}_k \) transition from state \( [A]^P \) into state \( \text{SUS}_k [A]^P \);

(b) stop,

for each \( k = 1, \ldots, n \) s.t. \( x_k \in V_N \), do:

(a) compute the set

\[ C^{(k)} = \{ c_j \in C^k | \text{YIELD} \left( \text{TRANSFER}_k \left( A \right)^P, c_j \right) \neq \emptyset \}; \]

(b) if \( C^{(k)} \neq \emptyset \) then create a transition from state \( [A]^P \) into state \( \text{SUS}_k [A]^P \) labelled with \( (\text{TRANSFER}_k, C^{(k)}) \).

create a transition from state \( [A]^P \) to state \( \text{SUS}_0 [A]^P \) labelled with \( \text{TRANSFER}_0 \).

end DEVELOP

Processing a New SUSPEND state. When a new SUSPEND state \( s \) is created, it is not "developed" in the usual sense, i.e. no exits from it are constructed right away. Instead, the REACTIVATE set of \( s \) is computed, giving rise to a set of couples to be entered into the queue \( Q \). These couples represent the need to construct new exits (labelled with the transmitted set of \( s \)) from all SUSPEND states in \( \text{REACTIVATE}(s) \).

It follows that exits from SUSPEND states are created only via retrieving and processing couples from the queue \( Q \), and different exits from the same SUSPEND state are created at different times.

At the time the REACTIVATE set of a new SUSPEND state is computed (according to formula (7.1)) some states may still be missing in it because they arise from as yet unconstruced parts of the LCA's. The full REACTIVATE set is obtained only upon termination of the construction algorithm. This matter will be discussed after the algorithm is presented.
Initialization of the Construction Algorithm. The initialization consists of defining the initial state for each LCA and initializing the queue Q. The following steps are taken:

1. For each LCA $A_p$, $p \in \mathcal{P}$, construct an initial state $\text{SUS}_0^i[A]^P$ (for short $s^p_0$) where $A = \{(a,k) \mid 1 \leq k \leq n_p \land p[k] \in \mathcal{VT} \land a \in S(p[k])\}$.

2. Set the queue Q with the first element $(E[s^0_0,\mathcal{P}])$ (with the second component undefined). The couple $(E[s^0_0,\mathcal{P}])$ corresponds to the root LCA $A_p$ receiving control from outside at the start of the evaluation process of a semantic tree.

Algorithm 3' - Construction of a Set of LCAs for an Attribute Grammar $G$.

We now present the full construction algorithm.

1. Construct for each nonterminal $X$ the set of characteristic graphs $C_X$ and the superimposed graph $S_X$.

2. Perform Knuth’s circularity test (Algorithm 1) for $G$; if $G$ is circular stop.

3. Carry out the initialization steps described above.

4. While Q is not empty:
   4.1. Retrieve a couple $(E[s^P,T],s^P)$ from Q. Let $s^P = \text{SUS}_k[A]^P$.
   
     if $T \ast k \subseteq A$ then goto (4)
   
   4.2. If the state $s^P$ already has an exit labelled by $T$ then do:

     (a) $\text{SOURCE}(E[s^P,T]) = \text{SOURCE}(E[s^P,T]) \cup \{s^P\}$.

     (b) for every state $s' = \text{SUS}_k[A'] \in \text{FORWARD}_k([A \cup T \ast k]^P)$ such that $s^P \in \text{REACTIVATE}(s')$ (where $\text{REACTIVATE}(s')$ is the set stored in memory with the state) do:
1. \text{REACTIVATE}(s') = \text{REACTIVATE}(s') \cup \{s^p\}

2. Add the couple \((E[s^p],A'/k],s')\) to \(Q\):
   
   \text{else (}s^p\text{ does not have an exit labelled by } T\text{) do:}
   
   (i) Create an exit \(E[s^p],T\) leading into the active state
   
   \([A \cup T^k]^P\) (if absent, create this active state as well).
   
   (ii): \text{SOURCE}(E[s^p,T]) = \{s^p\},
   
   (iii) if state \([A \cup T^k]^P\) was created in (i) above,
   
   \text{then call DEPLOY([A \cup T^k]^P);} \quad \text{fi.}
   
   (iv). For every new SUSPEND state \(SUS_k[A]^P\) created in (iii)
   
   if any do:
   
   (a) compute and store the set \text{REACTIVATE}(SUS_k[A]^P).
   
   (b) for every SUSPEND state \(s''\) \in \text{REACTIVATE}(SUS_k[A]^P)
   
   .add the couple \((E[s''],A'/k'],SUS_k[A]^P)\) to \(Q\).
   
   (v) if in (i) or (iii) above a new transition leading into an
   
   existing active state \([\bar{A}]^P\) was created,
   
   then for every state \(SUS_k[A'']^P \in \text{FORWARD}([\bar{A}]^P)\) do:
   
   (a) for every exit \(E[s',T'] \in \text{PRED}(SUS_k[A'']^P)\) do:
   
   for every SUSPEND state \(s''\) \in \text{SOURCE}(E[s',T']) such
   
   that \(s''\) \in \text{REACTIVATE}(SUS_k[A'']^P) do:
   
   1. \(\text{REACTIVATE}(SUS_k[A'']^P) = \text{REACTIVATE}(SUS_k[A'']^P) \cup \{s''\}\).
   
   2. Add the couple \((E[s''],A''/k''],SUS_k[A'']^P)\) to \(Q\).
   
   fi (v)
   
   fi (4.2)

(5) \text{END}
Later Additions to the REACTIVATE Set. As mentioned above, for each newly created SUSPEND state $s$, REACTIVATE($s$) is computed and then "processed", that is, couples representing exits from the states in REACTIVATE($s$), each labelled with the transmitted set of states, are generated and entered into the queue $Q$. However, the set REACTIVATE($s$) computed in step (4.2) (iv) (a) after the creation of state $s$, might be a proper subset of the same set REACTIVATE($s$), if it were computed after construction of the LCA's has been completed. This is because the SOURCE and PRED sets may still grow after computing REACTIVATE($s$) (as a result of expansion of the LCA's), yielding some new contributions to REACTIVATE($s$). The newly added states in REACTIVATE($s$) must also be processed.

Specifically, REACTIVATE($s$) may expand, after it has been computed and processed, in two ways:

1. For some transition $E = E[s',T] \in$ PRED($s$), the set SOURCE($E$) may grow (by one element at a time) by executing step 4.2(a), yielding a new element in REACTIVATE($s$). Step 4.2(b) computes (with the aid of FORWARD) the set of all SUSPEND states whose REACTIVATE set may grow as a result of updating SOURCE($E$). For each of these SUSPEND states, the newly added states in its REACTIVATE set are processed by generating the corresponding couples in $Q$.

2. The set PRED($s$) may grow as a result of adding a new transition $E$ into an existing active state (as can happen in steps 4.2(ii) and 4.2(iii)), thus creating new paths leading into state $s$. This in turn leads to the expansion of REACTIVATE($s$). In step 4.2(v), all SUSPEND states whose PRED sets may have grown as a result of adding the new transition $E$ are found (with the aid of the FORWARD set), and all newly added states in the REACTIVATE sets are processed. This again results in entering new couples into the queue.
We conclude that our construction algorithm takes care of all possible later expansion of the REACTIVATE sets, and provides for the necessary adjustments. Specifically, for every SUSPEND state \( s \), each state in the full set REACTIVATE\( (s) \) is processed precisely once, either at the time REACTIVATE\( (s) \) is computed (step 4.2(iv)) or later on by step 4.2(b) or 4.2(v). This observation is useful in proving termination of the construction algorithm.

**Proposition 1:** For each input attribute grammar, Algorithm 3 terminates after a finite number of steps.

**Proof:** The termination of steps (1) - (3) is obvious. It remains to show that step (4) is repeated only finitely many times. Each execution of step (4) involves retrieval of a couple from \( Q \). Clearly the number of distinct couples that can appear in \( Q \) is finite, and by the above observation, every couple is entered into \( Q \) only once.

We shall now proceed to verify that our construction algorithm produces a set of LCA's satisfying all the requirements mentioned in Sections 5 and 6.

**Proposition 2:** The set of LCA's constructed by Algorithm 3 satisfies the Closure Condition.

**Proof:** Clearly the processing of the REACTIVATE set of a SUS state \( s \) takes care of all possible transfers of control from state \( s \), and creates couples in \( Q \) representing all exits from states in REACTIVATE\( (s) \) that might be taken at runtime as a result of entering state \( s \). Every exit represented by a couple in \( Q \) is added to the corresponding LCA (unless it is already there) by step 4.1, except for exits which contribute nothing to the set of available attribute occurrences (i.e. \( T^* \subseteq A \) in step 4.1).
We claim that such exits will never actually be taken at runtime. This is because during the evaluation process, when the LCA $A_p$ of some node $N_{X_0}$ is active in state $[A]^p$, a $\text{TRANSFER}_k$ instruction for $k > 0$ is carried out only if the yield with respect to the characteristic graph of the $k$-th son is non-empty, which guarantees that the visit of the $k$-th son will contribute at least one new synthesized attribute value for $N_{X_0}$.

As for $k = 0$, control is transferred to the father of $N_{X_0}$ only in case no semantic function of the production applying at $N_{X_0}$ is ready to evaluate, and no visit to a son of $N_{X_0}$ can produce new attribute values for the production applying at $N_{X_0}$. The father will later on return control to its son $N_{X_0}$ only if the corresponding visit to $N_{X_0}$ is guaranteed to be fruitful, which can happen only provided that the father has returned at least one new inherited attribute value of $N_{X_0}$.

**Proposition 3:** For every augmented non-circular attribute grammar $G$, the set of LCAs constructed by Algorithm 3 satisfies the Completion Condition.

**Proof:** Suppose the proposition is not true. Then there is a semantic tree $T$ whose evaluation has terminated without computing the completion attribute of the root $S$. It follows that at least one synthesized attribute of a son $N_X$ of $S$ (where $X$ appears on the right side of production $p_o$) was not computed. Hence there exists, in the dependency graph of the tree $T$, a chain of dependencies leading from this attribute to an inherited attribute of $N_X$ (because otherwise the chain of dependencies would terminate with a loop inside the subtree rooted at $N_X$, which contradicts the non-circularity of $G$). This inherited attribute must depend on another synthesized attribute of one of the sons of $S$ (perhaps the same son $N_X$), and so on. But as there are only finitely many synthesized and inherited attributes on
the right side of production \( p_0 \), a circular dependency chain should arise. But this again contradicts the fact that the attribute grammar is non-circular, and hence the proposition is true. \( \square \)

From the above propositions, we obtain:

**Theorem 4:** For every augmented non-circular attribute grammar, Algorithm 3 terminates with a set of LCA’s, which have the structure described in Section 5, and which satisfy both the Closure Condition and the Completion Condition.

**Proof:** It follows directly from the algorithm that every LCA is acyclic, and has one initial state. Since a (conditional or unconditional) \( \text{TRANSFER}_k \) transition for \( k > 0 \) is created only if it will be fruitful for at least one semantic tree (i.e. the set of available attributes will grow when control returns), there must be at least one exit from every \( \text{SUS}_k \) state for \( k > 0 \). Also, by our construction, an active state will always have at least one exit. It follows that the LCA’s will have all their terminal states labelled with \( \text{SUS}_0 \). \( \square \)

Combining the above theorem with Theorems 2, 3 of Section 6, we have:

**Main Theorem:** For every augmented non-circular attribute grammar, Algorithm 3 constructs a translator (based on a set of LCA’s), which will perform the complete evaluation of each semantic tree (using Algorithm 2) in time proportional to the size of the tree.
8. RELATION TO PREVIOUS WORK

The first implementation of attribute grammars is due to Fang [F]. He uses parallel processes, one for each semantic function. A process is passivated when it tries to use a not-yet-available attribute, and is later on reactivated when the attribute becomes available. Such a non-deterministic approach is clearly far from efficient.

A deterministic approach was first developed by Lewis, Rosenkrantz and Stearns [L&R&S1] and by Bochman [B], who introduced an algorithm which traverses the tree in a depth-first left-to-right fashion, performing evaluation of all attributes in a single pass. Because of this restriction, the class of attribute grammars for which this method applies (named "L-attributed" in [L&R&S1,2]) is rather limited. To increase the class of attribute grammars that can be efficiently evaluated, Bochman also proposed to allow evaluation to occur in several left-to-right passes such that on each pass the attributes evaluated by previous passes can also be used. He gave an algorithm which determines whether an attribute grammar can be evaluated by a fixed number of left-to-right passes, and if so, which attributes should be evaluated on each pass.

Jazayeri [J1], observing that not all programming language features are amenable to evaluation from left to right, extended Bochman's method by introducing the "Alternating Semantic Evaluator" that alternately makes left-to-right, then right-to-left passes. Jazayeri showed that certain left-recursive situations could be evaluated by a single right-to-left pass, even though no fixed number of left-to-right passes was sufficient.
As was later on noted by Kennedy and Warren [K&W], Jazayeri's extension still leaves many attribute grammars which cannot be evaluated by any fixed number of alternating passes. They exhibited an example of an attribute grammar with a left-recursive rule $B \rightarrow Bb$, such that the first visit to a $B$-node son cannot be made until during the second visit to its $B$-node father. No method of evaluation in passes can handle such a grammar, for which "nested passes" are required.

Kennedy and Warren were the first to develop a deterministic approach in which the traversal order is not determined a priori, but is tailored to given attribute grammars by analyzing their dependency constraints. Their "treewalk evaluator" works like a recursive routine with a tree node to visit as parameter; while at a node, the evaluator may evaluate semantic functions or call itself recursively to visit sons. Unfortunately, their construction works only for a restricted class of attribute grammars — the "absolutely noncircular" grammars. For a grammar in this class, the evaluator's action at a node need not depend on the structure of the node's subtrees. In our terminology, the absolutely noncircular attribute grammars are precisely the ones for which our translator construction will yield LCA's without any conditional transfer instructions. For such grammars the evaluation algorithm (in Section 6) can be significantly simplified by eliminating altogether the first phase (i.e. the depth-first traversal), because in this case the characteristic graphs of the subtrees need not be computed.

In [Wa] Warren introduced a general model for deterministic evaluation of attribute grammars, called the "coroutine evaluator", and developed general methods for constructing such evaluators. This type of
Evaluator has the advantage over the "treewalk evaluator" in that it can send information upward in the tree and not only downward.

However, both the "treewalk evaluator" and the "coroutine evaluator", are not near-optimal, because they may wander in the tree making a lot of "fruitless" visits to subtrees before finally reaching a node where a new attribute value can be produced.

In this paper, following the deterministic approach suggested in [K&W] and [W], we have presented a general construction of a near-optimal translator for any non-circular attribute grammar. By introducing the characteristic graph as the main tool for analyzing (and representing) dependencies among attribute occurrences in a semantic tree, we have been able to obtain a near-optimal evaluation strategy which takes into account the structure of the subtrees of the node being processed. The translators introduced in this paper will usually be smaller than the ones constructed in [W], due to some reduction techniques used here (implicitly) which produce minimal LCA's (as opposed to the tree shaped evaluators in [W] which tend to be rather redundant). Because of this and due to their being near optimal, our translators will be by far more efficient.

3. COMPLEXITY ISSUES

The translators constructed in this paper all work in linear time w.r.t. the size of the parse tree, provided that one unit of time is charged for the evaluation of a semantic function, and assuming a random access memory. In fact, as was noted in [L&S&S], one could always produce a linear-time evaluation strategy for each individual parse tree by analyzing the dependencies in the tree at runtime; one would then
construct the compound dependency graph of the tree and perform a topological sort on that graph. By the method presented here the dependency analysis is done once and for all for each grammar during the translator construction, thus saving us a considerable runtime overhead.

What may seem somewhat peculiar is that so far no syntax analysis method has been found which parses all context free grammars in linear time (and probably none will be found), while (theoretically at least) the semantic evaluation phase takes linear time for all non-circular attribute grammars. This contrasts our intuitive feeling that the translation and code generation phase is usually considered more complex than the syntactic analysis phase.

As for the time complexity of the translator construction, admittedly, it may grow exponentially with the size of the grammar. In fact, this is unavoidable in view of the inherent exponential complexity of the circularity problem \cite{J&S&R}, as our construction will also detect circularity. The size of the translators may also be exponential, as the set of characteristic graphs may be of exponential cardinality (which is precisely what accounts for the exponentiality of Knuth's circularity test).

A similar situation occurs with respect to parser construction. For instance, it is a well known fact that LR(k) parsers can have number of states which is exponential with the size of the grammar. Moreover, the problem of determining whether an arbitrary context free grammar is LR(k) (with k unspecified) was shown to be \textit{NP}-complete when k is expressed in unary, and complete for non-deterministic exponential time when k is expressed in binary \cite{H&S&U}.

Nevertheless, both the parser and the translator may be worth constructing once and for all for each attribute grammar, to be later on
jointly used for the efficient implementation of the entire compilation process. Furthermore, both can be generated automatically when an attribute grammar specifying a programming language is given as input to a compiler compiler. In a future paper we shall discuss some techniques for making the translators constructed here more practical to be used in compiler generating systems.
REFERENCES


REFERENCES (cont'd):


