INFORMATION TO USERS

This material was produced from a microfilm copy of the original document. While the most advanced technological means to photograph and reproduce this document have been used, the quality is heavily dependent upon the quality of the original submitted.

The following explanation of techniques is provided to help you understand markings or patterns which may appear on this reproduction.

1. The sign or “target” for pages apparently lacking from the document photographed is “Missing Page(s)”. If it was possible to obtain the missing page(s) or section, they are spliced into the film along with adjacent pages. This may have necessitated cutting thru an image and duplicating adjacent pages to insure you complete continuity.

2. When an image on the film is obliterated with a large round black mark, it is an indication that the photographer suspected that the copy may have moved during exposure and thus cause a blurred image. You will find a good image of the page in the adjacent frame.

3. When a map, drawing or chart, etc., was part of the material being photographed the photographer followed a definite method in “sectioning” the material. It is customary to begin photoing at the upper left hand corner of a large sheet and to continue photoing from left to right in equal sections with a small overlap. If necessary, sectioning is continued again — beginning below the first row and continuing on until complete.

4. The majority of users indicate that the textual content is of greatest value, however, a somewhat higher quality reproduction could be made from “photographs” if essential to the understanding of the dissertation. Silver prints of “photographs” may be ordered at additional charge by writing the Order Department, giving the catalog number, title, author and specific pages you wish reproduced.

5. PLEASE NOTE: Some pages may have indistinct print. Filmed as received.

Xerox University Microfilms
300 North Zeab Road
Ann Arbor, Michigan 48106
WARREN, Scott Kipling, 1950-
THE COROUTINE MODEL OF ATTRIBUTE
GRAMMAR EVALUATION.

Rice University, Ph.D., 1976
Computer Science

Xerox University Microfilms, Ann Arbor, Michigan 48105
RICE UNIVERSITY

THE COROUTINE MODEL
OF
ATTRIBUTE GRAMMAR EVALUATION

by
Scott K. Warren

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

THESIS DIRECTOR'S SIGNATURE:

[Signature]

HOUSTON, TEXAS

APRIL, 1976
ACKNOWLEDGEMENT

My thanks to everyone at Rice who helped me get through, especially to Ken Kennedy and Ed Feustel (who shielded me from the harsh realities of institutional life, although that may not have been their intention).
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0. Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1. Attribute grammars.</td>
<td>4</td>
</tr>
<tr>
<td>2. The coroutine model of attribute grammar evaluation.</td>
<td>17</td>
</tr>
<tr>
<td>3. Verification of coroutine evaluators.</td>
<td>38</td>
</tr>
<tr>
<td>4. Construction of coroutine evaluators.</td>
<td>61</td>
</tr>
<tr>
<td>5. Practical constructions.</td>
<td>78</td>
</tr>
<tr>
<td>6. Uniform evaluation of attribute grammars.</td>
<td>95</td>
</tr>
<tr>
<td>7. Summary and concluding remarks.</td>
<td>104</td>
</tr>
<tr>
<td>References</td>
<td>110</td>
</tr>
</tbody>
</table>
0. INTRODUCTION

This work is a contribution to the art of automatically generating practical translators. Knuth's attribute grammar formalism is a natural, powerful way of specifying mappings such as those involved in compiling programming languages, but until now efficient implementation has been possible only for restrictive classes of attribute grammars. We present methods for mechanically constructing a program which performs the translation described by any given attribute grammar.

The compilation process may be divided into a syntactic phase and a semantic phase. During syntactic analysis a source text is examined to discover whether it is a well-formed program and, if so, what its underlying structure is. Context-free grammars can be used to define the set of syntactically correct programs and their associated structures in a formal yet descriptive and intuitively appealing way. Many techniques are now known for automatically constructing parsers for given CF grammars. As a result, compiler writers are no longer required to implement the syntactic phase manually.

Attribute grammars offer the prospect of similarly automating the implementation of the semantic phase. The CF grammar specifying syntax is augmented with "attributes" for each grammar symbol and "semantics" for each production. The attributes can be such things as data types of expressions, symbol tables for interpreting variable names, register usage information, or the machine code generated for a statement; the semantic rules determine the relations between these things as a function of syntactic structure. Using an attribute grammar to specify the translation performed by a compiler offers these advantages: the semantics is
given in a descriptive rather than algorithmic notation, independent of any particular parsing scheme; the description is modular, given on a production-by-production basis, so that it is easier to understand and modify; and the notation naturally expresses the idea of context-dependence as well as the relation of a whole to its parts.

Despite these advantages, attribute grammars have not been widely used because of the difficulty of obtaining implementations efficient enough for practical use. Once a parse tree has been constructed, the semantics must be applied to find the values of all attributes of tree nodes. However, attribute grammars leave the order of evaluation of the semantic functions largely unspecified, requiring only that a function's arguments be computed before the function is evaluated. A suitable evaluation order must take into account both the data dependencies of the semantics and the shape of the particular parse tree. Until now, automatic evaluation has been possible only through the use of a nondeterministic algorithm or by severely restricting the class of acceptable attribute grammars.

In this thesis we present a general framework for studying the deterministic implementation of attribute grammars and apply it to yield several methods of constructing, for any attribute grammar, a sequential procedure which performs the specified semantic evaluation. These "coroutine evaluators" are produced by analysing the data dependencies of the attribute grammar and using this information to guide the creation of an acceptable evaluation strategy. The resulting evaluators are efficient in that they make no use of nondeterminism or searching blindly through the parse tree and may be generated as directly executing machine code rather than requiring an interpreter. Our constructions are there-
fore suitable for use in a practical translator-writing system.

Another result of applying our model is the discovery of the **uniform** attribute grammars, a restricted class with considerable practical importance. As well as allowing a simpler implementation, these grammars enforce a healthy discipline on the attribute grammar writer analogous to that imposed on the programmer by "GOTO-less languages".

Our presentation is relatively informal, but rigorous enough to permit proofs of the important results. We do not discuss how to use attribute grammars to specify translations, concentrating instead on how to implement an attribute grammar once it is given. In chapter 1 we lay the groundwork for subsequent exposition, providing some historical background, an attribute grammar example, and the basic definitions and notation that will be needed later. Chapter 2 presents the coroutine model of evaluation and relates it to previous work in this area. In chapter 3 we develop a methodology for verifying the correctness of these evaluators, and in chapter 4 we derive from it a very general algorithm for constructing them. Chapter 5 presents a number of practical constructions based on the general algorithm and demonstrates the existence of some interesting heirarchies of attribute grammars. Chapter 6 defines the uniform attribute grammars, argues that they are the sort that people should be writing, and describes the simplifications they permit in implementation techniques. Finally, chapter 7 contains a summary and some hints for further investigation.
1. ATTRIBUTE GRAMMARS

In this chapter we introduce the basic concepts and terminology that will be used in the rest of the thesis. After a brief survey of the historical development, we discuss attribute grammars informally with the aid of an example. We then give definitions and notation for talking about attribute grammars. Finally, we consider the subject of dependency constraints in an attribute grammar and introduce the notion of "evaluating" an attribute grammar.

**Historical background.**

The twin problems of the formal definition and automatic implementation of programming languages have been the subject of continuing investigation since at least the early Sixties, and many different approaches have been taken [1]. Three general techniques have emerged: the denotational method, in which each program is assigned a mathematical formula expressing its meaning; the operational method, in which a formal description is given of the steps involved in executing a program; and the translational method, in which a mapping is given which takes each program into an equivalent one in another language with "known" semantics. The last of these relates most naturally to the automatic implementation of programming languages, since it provides a model for the compilation process.

It is natural to base such translations on the syntactic structure of the source programs, and much has been made of this connection with the well-understood context-free grammars. A variety of techniques have been used to introduce the idea of semantics into CFGs. Many of these
were attempts to represent context-dependency within the framework of
CFGs, such as indexed grammars [2], property grammars [3], programmed
grammars [4], two-level grammars [5], and scattered-context grammars [6].
These efforts were concerned primarily with limiting the set of terminal
strings generated by a grammar, and only secondarily with specifying the
meanings of those strings.

Irons [7] introduced the idea of syntax-directed translation; his
scheme essentially allowed a single synthesized attribute to be associated
with each nonterminal. Generalizations of this idea to allow multiple
synthesized attributes were studied extensively by Lewis and Stearns [8]
and by Aho and Ullman [9,10]. Knuth [11] extended this formalism in an
important way by allowing attributes to depend on the context surrounding
a grammar symbol as well as on its constituents. Such "inherited" attrib-
utes pass information down the parse tree from a node to its sons, in
contrast to Irons' synthesized attributes which convey information from
the bottom up. Knuth's "attribute grammars" have attracted widespread
interest [12-19] as a means of specifying syntax-directed translations
because of their naturalness, "declarative" nature, and expressive power.

Shortly after Knuth's 1968 paper, Culik [20] discussed a general-
ization of attribute grammars in which rules to compute attributes were
not given, but only relations which must hold between them. In 1971
Koster [21] described "affix grammars" in an attempt to reformulate van
Wijngaarden's two-level grammars in a way more suited to practical use.
Affix grammars are essentially attribute grammars in which the attribute
values themselves have a syntactic structure. Crowe [22] has shown how
to build recognizers for a limited class of affix grammars using modified
Floyd-Evans productions.
An attribute grammar example.

An attribute grammar ("ag" or just "grammar" for short) is an ordinary context-free grammar extended to specify the "meaning" of each string in the language. Each grammar symbol has an associated set of "attributes", and each production rule is provided with a corresponding semantic rule expressing the relationships between the attributes of symbols occurring in the production. To find the meaning of a string, we first find its parse tree and then determine the values of all the attributes of symbols in the tree.

Figure 1.1 gives an attribute grammar specifying the translation of binary constants into the numeric values they denote [15]. Observe that the notation \( X_k.a \) stands for the 'a' attribute of nonterminal \( X_k \); if the nonterminal \( X_k \) occurs only once in the production, we drop the subscript \( k \). To represent the desired translation we have invented the attribute 'val' for the start symbol \( N \). 'val' is also an attribute of the bit-lists (L) and of the individual bits (B). However, in positional notation the value of a 1-bit depends on how many places to the left it appears; the attribute 'pos' is introduced to express this. For instance, the semantics for production 7 says that the value of a 1-bit is the power of two corresponding to the bit's position. On the other hand, rule 6 says that the value of a 0-bit is always zero. In this attribute grammar 'val' is a synthesized attribute carrying information up the tree toward the start symbol, while 'pos' is an inherited attribute carrying information down the tree from above.

We can think of each node in the parse tree as a structured variable whose fields are its attributes. Figure 1.2 shows the parse tree for the string "-101" drawn to indicate this. The fields have been
\( V_N = \{N, S, L, B\} \)
\( V_T = \{+, -, 0, 1\} \)

\[
\begin{align*}
I(N) &= \emptyset \\
S(N) &= \{val\} \\
I(S) &= \emptyset \\
S(S) &= \{neg\} \\
I(L) &= \{pos\} \\
S(L) &= \{val\} \\
I(B) &= \{pos\} \\
S(B) &= \{val\}
\end{align*}
\]

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: ( N \rightarrow S , L )</td>
<td>( L.\text{pos} \leftarrow 0, ) ( N.\text{val} \leftarrow \text{if } S.\text{neg} ) ( \text{then } -L.\text{val} ) ( \text{else } L.\text{val} ) ( \text{fi} )</td>
</tr>
<tr>
<td>2: ( S \rightarrow + )</td>
<td>( S.\text{neg} \leftarrow \text{false} )</td>
</tr>
<tr>
<td>3: ( S \rightarrow - )</td>
<td>( S.\text{neg} \leftarrow \text{true} )</td>
</tr>
<tr>
<td>4: ( L \rightarrow B )</td>
<td>( B.\text{pos} \leftarrow L.\text{pos}, ) ( L.\text{val} \leftarrow B.\text{val} )</td>
</tr>
<tr>
<td>5: ( L_0 \rightarrow L_1 , B )</td>
<td>( L_1.\text{pos} \leftarrow L_0.\text{pos} + 1, ) ( B.\text{pos} \leftarrow L_0.\text{pos}, ) ( L_0.\text{val} \leftarrow L_1.\text{val} + B.\text{val} )</td>
</tr>
<tr>
<td>6: ( B \rightarrow 0 )</td>
<td>( B.\text{val} \leftarrow 0 )</td>
</tr>
<tr>
<td>7: ( B \rightarrow 1 )</td>
<td>( B.\text{val} \leftarrow 2 , B.\text{pos} )</td>
</tr>
</tbody>
</table>
filled in as prescribed by the semantic rules. The effect of our attribute grammar in this case is to specify that the translation of "-101" is -5.

When an ag is used to specify the translation activity of a compiler, the attributes will be such things as data types of expressions, symbol tables for use in translating identifiers, register usage information, or machine code generated for a statement. The property of locally-defined semantics makes it easy to construct, understand, and modify code generators written using ags.

Attribute grammars.

In this section we make precise the term "attribute grammar". Most of this is taken directly from [12].

Definition: An attribute grammar is an ordinary CF grammar augmented with attributes and semantic functions as described below.

Grammar: A reduced context-free grammar $G = (V_N, V_T, P, S)$. We write $V$ for $V_N \cup V_T$. A production $p \in P$ is written

$$p: X_0 \rightarrow X_1 X_2 \ldots X_{n_p}$$

where $n_p \geq 1$, $X_0 \in V_N$, and $X_k \in V$ for $k=1,2,\ldots,n_p$. We write $p[k]$ to mean $X_k$ for $k=0,1,\ldots,n_p$. We assume the grammar is standardized if necessary with a 0-th production $0:S \rightarrow S'$ so that the start symbol $S$ occurs in no other production. A parse tree of $G$ is a finite ordered tree whose nodes are labelled with symbols from $V$, such that: for each interior node $t$ there is a production $p \in P$ such that $t$ is labelled with the symbol $p[0]$,
t has \( n_p \) sons, and the \( k \)-th son of \( t \) is labelled with the symbol \( p[k] \). We say that \( p \) applies at \( t \), or equivalently that \( t \) is a type-\( p \) node.

If \( t \) is labelled with \( X \), we also say that \( t \) is a type-\( X \) node.

**Attributes:** For each \( x \in V \), there are disjoint finite sets \( I(x) \) and \( S(x) \) of inherited and synthesized attributes respectively. For \( x = S \), the start symbol, and for \( x \in V_T \), we require that \( I(x) = \emptyset \). We write \( A(x) \) for \( I(x) \cup S(x) \). As a technical convenience we require \( A(x) \cap A(y) = \emptyset \) if \( x \neq y \). The attributes of a symbol identify the various components of its meaning. There is a distinguished synthesized attribute of the start symbol which identifies the translation of the terminal string. A production \( p \) has the attribute occurrence \( (a,k) \) if \( a \in A(p[k]) \) for \( k = 0,1, \ldots n_p \). Attribute occurrences are to be understood as variables which are used in writing the semantics for a production.

Inherited attributes transmit information down the parse tree toward the leaves, while synthesized attributes transmit information up the tree toward the root. The start symbol may not have inherited attributes because it has no ancestors; terminal symbols have no inherited attributes because they have no associated semantics. The values of a terminal symbol's synthesized attributes are given initially; in a compiler this is the job of the lexical scanner. The term "attribute" is used ambiguously to mean some \( a \in A(X) \), as in "an attribute of a nonterminal"; to mean some occurrence \( (a,k) \), as in "an attribute of a production"; or to mean a field of the tree, as in "an attribute of a node". It should always be clear from the context which sense is intended.
Semantic functions: For each production $p \in P$ there is a set of semantic functions as follows: for every synthesized occurrence $(a,k)$ with $k=0$, and for every inherited occurrence $(a,k)$ with $k=1,2,\ldots,n_p$, there is a semantic function $f^p_{(a,k)}$ mapping certain other attribute occurrences of $p$ into a value for occurrence $(a,k)$. The dependency set of $f^p_{(a,k)}$, denoted by $D^p_{(a,k)}$, contains those attribute occurrences of $p$ used in the definition of the semantic function. The semantic functions specify the meanings of parse trees locally, in terms of only a node and its immediate descendents. We do not consider the language in which the functions are written; we assume only that we can identify the attribute occurrences referenced in a function and that the function can be translated into machine code to do the evaluation.

Dependency constraints.

Naturally a semantic function cannot be evaluated before the occurrences in its dependency set have been computed. This is the only constraint on the order of evaluation imposed by an ag. We are studying ways to find a specific evaluation order which obeys this constraint. We therefore need some concepts and notation for studying the dependencies of an ag.

The dependency graph of a production $p \in P$, denoted by $DG_p$, is a convenient representation of the constraints imposed locally by the semantics of the production. The nodes of $DG_p$ are the attribute occurrences of $p$; there is a directed arc from $(a',k')$ to $(a,k)$ if $(a',k') \in D^p_{(a,k)}$, i.e. if $(a',k')$ is used in the evaluation of $(a,k)$. The arcs of $DG_p$ may be thought of as data flow paths. Figure 1.3 shows one of the rules of our example grammar and two ways of drawing its dependency graph. The
FIGURE 1.3

Production

5: \( L_0 \rightarrow L_1 B \)

Semantics

\( L_1 \cdot \text{pos} = L_0 \cdot \text{pos} + 1, \)

\( B \cdot \text{pos} = L_0 \cdot \text{pos}, \)

\( L_0 \cdot \text{val} = L_1 \cdot \text{val} + B \cdot \text{val} \)
more elaborate of the pictures is intended to suggest a portion of a parse tree and provide a clearer picture of the data flow up and down the tree.

If we have a parse tree of an ag, we can construct a dependency graph which represents all data flow paths in the tree. This graph is the result of "pasting together" copies of the DG's for productions occurring in the tree, as illustrated in Figure 1.4. A path in this graph from one attribute to another indicates that the second attribute depends, possibly indirectly, on the first -- and hence that the evaluation of the second necessarily follows the evaluation of the first in time. If a subtree is detached from the rest of the parse tree, some data flow paths through the root of the subtree will be interrupted. Data flows into the subtree through inherited attributes of the root, and out of the subtree through synthesized attributes of the root. For this reason these attributes are sometimes referred to as input attributes and output attributes respectively. In a production, the input occurrences are those of the form \((i,0)\) where \(i \in I(p[0])\) and the output occurrences are those of the form \((s,0)\) where \(s \in S(p[0])\).

If the dependency graph of a parse tree contains a directed cycle, some of its attributes cannot be determined. In this case the tree is said to be circular semantically. An ag is circular if it can generate a circular parse tree. Circularity is obviously undesirable, in that we would like an ag to specify the meaning of every string in the language. Knuth [11] has given an algorithm to test an ag for circularity, but the task is not an easy one [16].
Evaluation of attribute grammars.

A semantic tree of an ag is a parse tree in which each node labelled with XeV is a structured variable whose field selectors are the elements of A(X). In order to determine the translation of a string, we parse it and build a semantic tree, and then we fill in the fields of each node in accordance with the semantic functions. Upon proper termination of this process, the translation is given by the value of the root's distinguished attribute. The filling-in process is called evaluating the semantic tree. An evaluator for an ag is a program which will accept any of its semantic trees and evaluate the tree to yield a translation. In this thesis we think of the time during which an evaluator operates as run-time, since we view the attribute grammar formalism as a high-level language for specifying a certain class of programs: namely, the syntax-directed translators.

The significance of the attribute grammar formalism is prescribed by a defining evaluator which provides a conceptual model for evaluating semantic trees. Evaluation begins with all fields of the semantic tree undefined. At each step, some attribute of a node is chosen whose corresponding semantic function is ready to evaluate; that is, all of whose argument occurrences are already defined. The chosen semantic function is executed and the corresponding field of the tree node is defined by setting it to the computed value. The process continues until all attributes in the tree have been defined (alternatively, until the distinguished attribute of the root has been defined). The defining evaluator is non-deterministic, since at each step any attribute can be chosen which is ready. The remainder of this thesis is devoted to finding reasonably implementable mechanisms for making this choice.
Conventions.

In what follows, we assume that some fixed ag is understood to be the subject of discussion unless otherwise noted. We further assume that the goal of evaluation is to yield the value of the start symbol's distinguished attribute, and hence that evaluation may terminate when this has been done. If it is desired to enforce the evaluation of all the attributes in semantic trees, the ag must be modified slightly. A special 'done' attribute, synthesized, is added to each nonterminal's set S(X). A dummy semantic rule is added to each production, defining the (done,0) occurrence in terms of all other occurrences defined within the production as well as all the (done,k) occurrences of sons. The idea is that the sets $D^P_{(\text{done},0)}$ should contain every attribute occurrence whose evaluation is to be enforced. If in addition the start symbol's distinguished attribute is considered to depend on the (done,k) occurrences of production 0, then termination will not occur until every attribute in the tree is defined. Of course, these 'done' attributes may be ignored at run-time (as if the execution of the dummy functions involves no action); they serve only to cause the construction of a certain kind of evaluator.
2. THE COROUTINE MODEL OF ATTRIBUTE GRAMMAR EVALUATION

A "model of evaluation" describes a class of structurally similar programs which implement ags. We have restricted our attention to models of evaluation in which a semantic tree is explicitly built and filled in with attribute values; moreover, the "filling in" is to be done by a single deterministic processor which wanders around in the tree evaluating semantic functions. After discussing the merits of this restriction, we present such a model of evaluation based on distributed control. Since an ag gives semantics on a production-by-production basis, it is natural to look at the activities during an evaluation from the point of view of an individual node in the semantic tree. We obtain for each production a flowchart prescribing the possible activity sequences at a node where the production applies. A semantic tree is evaluated by attaching to each node a coroutine executing a copy of the appropriate flowchart, then initiating the root coroutine. The locus of activity moves around in the tree, being directed at each node by the attached flowchart, until evaluation is complete. At the end of this chapter we compare the coroutine model to the previously known methods of implementing ags.

Models of evaluation.

Any ag specifies a mapping from a set of strings into some domain of "translations" of those strings. We are interested in finding systematic ways to implement ags, that is to produce for any given ag a program which carries out the specified translation. This problem can be approached most generally as a task in automatic programming: given (an ag as) a specification of a desired behavior, to write a program which exhibits
the behavior. From this point of view an ag merely prescribes what output is to be yielded for each input, and says nothing about how these outputs are to be calculated. Naturally there will be a great many programs which implement any particular translation, and some of the best of them may bear little internal resemblance to the ag which specified the translation. Consider for example the ag of chapter 1, which defined a mapping from binary integers into the numerical values they denote. One fairly good implementation of that mapping is the following program:

```
begin char c, int n := 0;
    int u = (read(c); c = "\+" | 1 | -1);
    while read(c); c ≠ blank
        do n* := 2; (c = "1" | n+ := u) od;
    print(n)
end
```

There is no obvious correspondence between elements of this program and the nonterminals, attributes, or semantic functions of chapter 1's ag. Several tricks have been used to eliminate the 'nos' attribute, replace the addition of various powers of two by successive doublings and incrementings, encode the 'neg' attribute in a more convenient way, and even to avoid building a parse tree at all! Such clever-trick implementations are very nice when available; but to find them is a very hard problem, one that is well beyond the current state of our knowledge. In any case, it is a problem in artificial intelligence and not attribute-grammar theory.

In order to get a tractable problem we make some simplifying assumptions. To begin with we insist that any implementation must translate a
string by executing precisely the same semantic function calls as would
the defining evaluator for the ag; thus we rule out any attempts to use
special properties of particular semantic functions to make an optimized
implementation. This amounts to studying ag schemas with uninterpreted
function symbols, and is a standard way to avoid considering what is
called "world knowledge" in A.I. circles. Once this is decided, finding
an implementation of an ag simply involves arranging that, for any input
string, a proper sequence of semantic function calls is performed and that
the result of each call is properly conveyed as an argument to later calls
requiring it.

Our second simplifying assumption is that a semantic tree will be
built and filled in by the implementation. The tree serves both to enum-
erate the semantic function calls necessary to translate the input string
and to communicate the results of such calls to succeeding calls as re-
quired. The problem of implementing an ag is thus reduced to constructing
some sort of sequencing mechanism that accepts a semantic tree and causes
its occurrences of semantic functions to be called in a suitable order —
in short, an evaluator as defined in chapter 1. The main advantage of
this approach is that it is workable: we can actually find ways to produce
such implementations mechanically, whereas if we do not assume the use of
explicit semantic trees it is hard to see where to start. This is not to
say that semantic-tree models are particularly good implementations. In
fact, they suffer from a great deal of storage inefficiency. All attri-
butes of all nodes in a tree are retained throughout the translating pro-
cess — wasteful because almost all attributes are essentially temporary
variables, quantities produced solely for use in later computations. The
effect is that upon termination, one attribute of the root is the desired
translation, while the entire remainder of the tree is just so much
garbage.

We can contemplate adding some sort of storage reclamation device
to our evaluators, and in chapter 6 we hint at how this might be done for
certain ags; but it must be admitted that semantic-tree models have a lot
of "catching up" to do to compare favorably with hand-made implementations
of complex ags. For some purposes this may be inconsequential, as for
instance in a command processor or control-card reader which never receives
long strings to translate; for other purposes, such as production com-
pilers for large languages, it may be fatal. We reiterate that we are
using semantic-tree models because they are something we can deal with,
and because by understanding them we may learn something of value in
producing more efficient implementation methods.

The coroutine model.

Imagine that we have an evaluator which consists of a single deter-
ministic processor with a cursor that always points to some "current"
node of the semantic tree. The processor can move the cursor down to any
son of the current node or up to the father of the current node. It can
inspect the current node to see which production applies there, and it
can call for the evaluation of a semantic function at the current node.
When started with the cursor pointing to the root of a semantic tree, it
runs for a finite time and halts after having completed the evaluation.
How might such an evaluator be controlled?

Our point of departure is the idea of watching the evaluation of
a semantic tree from the point of view of some individual node in it.
During an evaluation the cursor is moving around in the tree computing attributes. If we watch at one node, at first nothing seems to happen; the cursor has not yet reached us. Suddenly the cursor arrives at our node from above. Perhaps some semantic functions of our node are evaluated. At any rate, after a while the cursor moves away from us in some direction. Later, it returns from the direction it went; perhaps some more semantic functions of our node are evaluated. And so it goes, until one day the cursor leaves us going toward the root, never to return.

From a local point of view, evaluation consists of comings and goings of the cursor interspersed with evaluations performed while it is at our node.

We begin to see the idea of distributed control: we make each node somehow responsible for directing the actions of the processor whenever the cursor is there. The possible actions are to evaluate a semantic function of the node, and to move the cursor in some direction to a neighboring node. A type-p node has \( n_p + 1 \) neighboring nodes. Neighbor 0 is the node's father, while for \( 1 \leq k \leq n_p \), neighbor \( k \) is the \( k \)-th son of the node. Thus a type-p node may direct the cursor to move in any of \( n_p + 1 \) possible directions. The node divides the rest of the tree into \( n_p + 1 \) regions: for \( 1 \leq k \leq n_p \), the \( k \)-th region is the \( k \)-th subtree; region 0 is the remainder of the tree after the node and its subtrees have been removed. From the node's point of view, moving the cursor to neighbor \( k \) has the effect of sending the processor off to wander around in region \( k \), doing evaluations as appropriate before returning to the node.

We will associate with each node a flowchart made up of actions of the kinds just discussed. Since the permitted actions depend on the production which applies at the node, it is natural to let that production determine what flowchart is attached to the node. It is pleasing to let
the production-by-production semantics of asgs give rise to production-by-
production evaluation strategies. Whenever the cursor is at a node, the
processor's activity is to be directed by that node's flowchart. It is
necessary, when the cursor returns to a node after being sent away, that
execution of the node's flowchart be resumed where it left off; this may
be accomplished by letting execution at each node be done by a coroutine
associated with that node.

Coroutines [23] are program segments which cooperate as equals,
each coroutine viewing its partners as subroutines to which it may tem-
porarily send control. This is achieved by a generalization of the ordi-
nary subroutine call/return mechanism. When a main program invokes a
subroutine, it first saves its program counter and then transfers to the
subroutine. When control is returned to the main program, its execution
begins where it left off, at the saved program-counter address. In con-
trast, if control is sent to the subroutine again, its execution begins
again at its beginning. In the coroutine linkage mechanism this asym-
mety is removed. Each coroutine has its own program counter pointing
to where its execution was last suspended. Whenever a coroutine receives
control its execution continues from the point of interruption. Since
each coroutine "remembers what it's doing" while inactive, the effect is
similar to parallel processing; however, synchronization of the processes
is accomplished by explicitly yielding control to another coroutine as
determined at compile-time. This is the "quasi-parallel processing" of
SIMULA 67 [24]. The coroutine organization thus provides the conceptual
power of parallel processing while avoiding the overhead of run-time
scheduling.
In the coroutine implementation of our evaluator the cursor is represented by the locus of control — the current node is the one whose coroutine is active at the moment. Since a coroutine can have no direct knowledge of activities that take place while it is inactive and yet these activities may affect the coroutine's own course of action, a means of exchanging information is required. Communication between coroutines is done by sending a parameter along with control when a coroutine allows one of its neighbors to resume execution. From the point of view of the coroutine, the resumption looks like a parametrized subroutine call; since a parameter is also sent by the neighbor when returning control, the "subroutine call" will seem to return a result. The coroutine may use this information about events it could not witness directly in order to change its strategy to conform to the shape of the semantic tree.

The coroutines we consider will all adhere to a special convention about the interpretation of these parameters. A coroutine sends control to its k-th neighbor when it wishes some evaluation to take place in region k; usually the intent is to make available to the coroutine some attributes which it needs but which are computed by semantic functions of other (neighboring) productions. These attributes will be either inherited attributes of the node itself (X₀), defined by the production applying at neighbor 0, or else synthesized attributes of some k-th son (Xₖ), defined by the production at neighbor k. (Recall that the semantics of the production applying at the node itself define the synthesized attributes of the node and the inherited attributes of its sons.) We therefore declare that when control is sent to a coroutine, the parameter sent along will be a set of names of attributes which are now available. Note that when control is received from the k-th neighbor of a coroutine, these are attributes of Xₖ (the node itself if k=0, the k-th son otherwise).
Flowcharts.

Let us now speak more precisely. For any \( p : X_0 \rightarrow X_1 X_2 \ldots X_n \) we divide the attributes of symbols occurring in \( p \) into two groups, those defined by the semantics of \( p \) and those defined outside of \( p \).

For \( k=0 \), define \( \text{SENT}_p(X_0) = S(X_0) \) and \( \text{RECEIVED}_p(X_0) = I(X_0) \);
for \( 1 \leq k \leq n \), define \( \text{SENT}_p(X_k) = I(X_k) \) and \( \text{RECEIVED}_p(X_k) = S(X_k) \).

Then a flowchart for \( p \) is a directed graph whose nodes are labelled with "instructions" of two kinds:

(a) symbols of the form \( r_p^{(a,k)} \) where \( a \in \text{SENT}_p(X_k) \)
or (b) symbols of the form \( \text{MOVE}_k(A) \) where \( A \in \text{SENT}_p(X_k) \), \( X_k \in V_N \)

Instructions of form (a) are called evaluate-instructions, and those of form (b) are called move-instructions. We also use the word "instruction" to mean a so-labelled node of the flowchart, but no confusion should arise. If \( c \) is a node in a flowchart, then \( \text{INSTR}(c) \) is the instruction labelling it. If \( \text{INSTR}(c) = i \), we write \( c : i \) to denote this.

An evaluate-instruction has a single out-arc leaving it; if \( \text{INSTR}(c) = r_p^{(a,k)} \) then \( \text{NEXT}(c) \) is \( c \)'s successor node in the flowchart. When \( c \) is executed, the semantic function \( r_p^{(a,k)} \) is executed and the result is stored in the appropriate field of the semantic tree. Control then passes to the instruction \( \text{NEXT}(c) \).

A move-instruction may have several out-arcs leaving it, each bearing a distinct label. If \( \text{INSTR}(c) = \text{MOVE}_k(A) \) then each exit from \( c \) is labelled with a subset of \( \text{RECEIVED}_p(X_k) \). The set of labels on exits of \( c \) is given by \( \text{EXITS}(c) \). If \( A' \in \text{EXITS}(c) \), then \( \text{NEXT}(c,A') \) is \( c \)'s \( A' \)-successor, i.e. the node \( c' \) such that the arc \((c,c')\) is labelled \( A' \).
We also say that $X_k$ is the nonterminal associated with instruction $c$. When $c$ is executed, the current coroutine is temporarily suspended at $c$. Control is sent to the $k$-th neighbor of the current node, and the parameter $A$ is transmitted along with control. Every attribute occurrence $(a,k)$ where $a \in A$ should already have been evaluated. When neighbor $k$ finally returns control to the coroutine, it will also send along some parameter $A'$. This parameter determines which exit is taken from $c$ when execution of the coroutine is resumed; namely, instruction $\text{NEXT}(c,A')$ is the next one to be executed. Thus to the coroutine, a $\text{MOVE}_k(A)$ instruction looks like a subroutine call which sends a parameter, causes some activity in region $k$, returns a result depending on the details of that activity, and transfers control based on the returned result.

The flowcharts we are interested in have certain structural properties in addition. An initial node is a node with no in-arcs, and a terminal node is one with no out-arcs. A normal flowchart is one that satisfies:

1. It is acyclic;
2. There is precisely one initial node, which is labelled with a $\text{MOVE}_0$ instruction;
3. Each terminal node is labelled with a $\text{MOVE}_0$ instruction.

We ask that our flowcharts be acyclic because we have no use for loops and their absence simplifies a number of definitions and proofs. Evaluation of a semantic tree is begun by giving control to the root's coroutine; thus every coroutine receives control first from above. Each coroutine's program counter is initially set to point to its flowchart's
initial node, as if execution had been suspended at the dummy MOVE_0 instruction there. This enables us to avoid treating coroutine start-up as a special case. We want control to finally return to the root coroutine before evaluation terminates, both for neatness and in consideration of certain attractive implementation techniques. Requiring every terminal node to be a MOVE_0 instruction guarantees this. Any evaluation performed by a set of normal flowcharts either aborts (see chapter 3) or terminates through the root after a finite number of steps. This property is so convenient that we don't talk about flowcharts that lack it; hereafter, by a flowchart we will always mean a normal flowchart.

Coroutine evaluation.

Suppose that an ag G is given. A coroutine evaluator for G is a set \( \{ C_p | p \in P \} \) of flowcharts, one for each production, which satisfy certain correctness criteria. A coroutine is an activation of one of these flowcharts, consisting of a copy of the flowchart, a "program counter" which may point to instructions in the flowchart, and an associated semantic tree node through which evaluate-instructions reference attributes of the tree and move-instructions reference neighboring coroutines. To evaluate a semantic tree of G we create for each tree node a coroutine which is an incarnation of the flowchart for the production applying there. Every coroutine's program counter is initialized to point to its flowchart's initial node. Control is then sent to the root's coroutine along with the parameter \( \emptyset \). After a finite number of steps control will have returned to the root coroutine, which then executes a MOVE_0 yielding control to the "outside", and evaluation is complete. Such a computation is called an execution of the set of flowcharts. In
order for this computation to go smoothly, the set of flowcharts must be well-formed in a certain sense. Informally, we say that a set of flowcharts is valid if:

(1) No evaluate-instruction is ever executed until all the attributes referenced by the semantic function have been computed;
and (2) Whenever control is sent to a coroutine along with a parameter $A$, the resumed move-instruction has an exit labelled $A$.

If these conditions are satisfied then any attempt to evaluate a semantic tree will result in a well-defined computation. Notice that condition (2) especially relates to the harmonious cooperation of sets of coroutines; it will turn out that the possible communication sequences used by flowcharts are central to our study. A valid set of flowcharts for $G$ is in fact a coroutine evaluator for $G$ if it also satisfies the completion-condition: whenever an execution of the set of flowcharts terminates, the distinguished attribute of the root must have been evaluated. The next chapter will show how to verify that a set of flowcharts is valid and complete. Not surprisingly, the correctness-proof techniques are the key to mechanically constructing coroutine evaluators. By working backwards with the correctness-proof methodology we can build a set of flowcharts in such a way that, by construction, they satisfy a proof of evaluator correctness. This important matter is discussed in chapter 4.
Relation to previous work.

We will close this chapter by reviewing briefly the ag implementation techniques previously reported in the literature. We state without proof certain assertions relating these techniques to our present effort; the proofs will be found in later chapters. The first implementation of ags was a general system based on parallel processes due to Fang [15]. Bochmann [12] proposed evaluating certain ags sequentially in a fixed number of left-to-right passes over the semantic tree. In its one-pass form this idea was published earlier by Koster [21] for affix grammars and by Lewis, Rosenkrantz, and Stearns [17] for attributed translations; since Bochmann's work includes these, we do not discuss them separately. Jazayeri [16] suggested extending Bochmann's idea to a fixed number of alternately left-right, then right-left passes. Warren [25] described a method of constructing for any ag a recursive procedure that traversed the semantic tree in a fashion determined by analysing the ag's dependency constraints and performed the evaluation. The coroutine model is an outgrowth of that work.

The first actual implementation of ags was done by Isu Fang, a student of Knuth's. Fang's system, called FOLDS, was a more or less direct implementation of the defining evaluator. The semantic functions were written in an ALGOL-like language which allowed explicit parallelism and the manipulation of VDL structured objects. Semantic trees were evaluated by an interpreter which created a separate parallel process for each attribute in the tree. Each process was an incarnation of a semantic function, which attempted to compute the value of its attribute. Any reference by a process to an attribute not yet computed caused "passivation" of the process. Its execution was suspended and it was placed on a
list of processes waiting for the attribute in question. Whenever an attribute was successfully computed, all processes on its waiting list were re-activated. A process which depended on several attributes might be awakened from one waiting list only to be placed on another.

The interpreter used a routine called develop to create new processes. When no existing processes were ready to run, develop would be called on to select an attribute and create an incarnation of its semantic function. Develop simply processed the semantic tree in depth-first left-to-right order.

Fang's system could evaluate any ag, but it was not a practical method of evaluation. The use of a run-time scheduler to determine evaluation order caused large space and time overheads. As Fang himself noted, "a nondeterministic approach such as this is bound to be inherently less efficient than deterministic approaches". The coroutine model allows the implementation of ags with as much generality as Fang's parallelism, but permits the scheduling of evaluations to be done once and for all at evaluator-generation time.

Bochmann was the first to describe a deterministic approach. He began by considering the possibility of evaluating an ag in a single left-to-right pass through the semantic tree. A recursive routine, evaluate-subtree, traversed the tree in depth-first left-to-right order performing evaluation of attributes. To ensure that all the semantic functions of a node could be executed it was necessary to compute all of the node's inherited attributes before visiting it. The body of evaluate-subtree performed the following algorithm: for each subtree in left-to-right order, first evaluate all its inherited attributes, then call evaluate-subtree on it to finish its evaluation; finally, compute the current node's syn-
thesized attributes and return. Not every ag can be evaluated this way. Bochmann stated the condition that the semantic functions' dependencies must satisfy: no inherited attribute of a son can depend on any of its synthesized attributes or on any attributes of sons to its right. This represented a very strong condition on the attribute grammar.

To increase the number of ags that could be evaluated, Bochmann proposed to allow the evaluation to occur in several left-to-right passes. On each pass, all the attributes computed by previous passes could be used. He gave an algorithm to determine whether an ag could be evaluated this way and to find the attributes which could be evaluated on each pass. The iterative algorithm assumed initially that all remaining attributes could be evaluated on the next pass. The one-pass dependency condition was checked; if any attribute failed the check (could not be evaluated) it was eliminated from the current pass and the condition checked again. If the condition was satisfied, all attributes not eliminated were assigned to the current pass and the process was repeated. The algorithm terminated when all attributes had been assigned to a pass or when it was found that the next pass could not evaluate any of the remaining attributes. In the latter case the ag could not be evaluated in a fixed number of left-to-right passes.

Jazayeri suggested a modification of Bochmann's method in which on alternate passes the direction of traversal was reversed. Observing with Fang that "the left-to-right bias is not all-pervasive" in programming language semantics, he described an Alternating Semantic Evaluator that made every other pass from right to left. Jazayeri showed that certain left-recursive situations could be evaluated in a single right-to-left pass even though no fixed number of left-to-right passes was suffi-
cient. However, this proposal did not remedy the difficulty with evaluation in passes. Such evaluators visit each node several times, but in a highly constrained fashion. Each node of the tree is visited once before any is visited a second time; each gets a second visit before any gets a third; and so on in lockstep. Evaluation is done uniformly throughout the tree: all occurrences of X.a must be evaluated "at the same time", on one sweep through the tree. Consequently, evaluation in passes runs into trouble when "nested passes" are required, as in the example of Figure 2.1. The attribute grammar shown there is more easily understood if one knows that, for B nodes, the first visit brings B.a and yields B.x, while the second visit brings B.b and yields B.y. Any semantic tree of this ag can be evaluated by making two visits to each B node. The catch is in the semantics for rule 2: because of the definition

\[ B_1.a + B_0.b + 1 \]

the first visit to the son \( (B_1) \) cannot be made until during the second visit to the father \( (B_0) \), since \( B_0.b \) is not available until then. Neither Bochmann's nor Jazayeri's methods can handle this grammar, because there is no attribute of B all of whose occurrences in a tree can always be evaluated in the first pass. On the other hand, in the coroutine model it is perfectly possible that two or more MOVE\(_1\) instructions should occur along a path between two successive MOVE\(_0\) instructions; that is, a visit to a node may perform several visits to a subnode. This is just what is required to deal with "nested passes", and our construction has no trouble with the example above. Figure 2.2 shows a coroutine evaluator for that ag, and Figure 2.3 shows a trace of its execution on a small tree.
FIGURE 2.1A

Nested passes required.

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: A→B</td>
<td>B.a+1, B.b+B.x+1, A.r+B.y+1</td>
</tr>
<tr>
<td>2: B₀→B₁ b</td>
<td>B₀.x+B₀.a+1, B₁.a+B₀.b+1, B₁.b+B₁.x+1, B₀.y+B₁.y+1</td>
</tr>
<tr>
<td>3: B→b</td>
<td>B.x+B.a+1, B.y+B.b+1</td>
</tr>
</tbody>
</table>

Dependency graphs of productions:

![Graphs](image-url)
Example:

FIGURE 2.1B

parse tree

dependency graph

semantic tree after evaluation
FIGURE 2.2

C_1: \phi 

C_2: f(a, b) \rightarrow \{a, b\} 

C_3: f(a, b) \rightarrow \{a, b\} 

A \rightarrow f(x, y) 

B \rightarrow f(x, y) 

Move_0 (a) 

Move_1 (a) 

f(x, y)
Any ag that can be evaluated in passes has a coroutine evaluator, since any noncircular ag at all has one. In fact, any such ag has a uniform evaluator (see chapter 6). However, there exist uniform ags that cannot be evaluated in passes; we have just seen such a one. So we see that our method is more general that either Bochmann's or Jazayeri's. But even in a stricter sense our method includes theirs: for any evaluator that works in passes, there is a corresponding coroutine evaluator that simulates it -- evaluates every tree in precisely the same order.

In an earlier work, Warren constructed evaluators with the form of recursive routines which traversed the semantic tree in patterns tailored to given ags by analysing their dependency constraints. The body of such a routine was a case statement consisting of code sequences to handle each situation that could be found upon arriving at a node. In operation, the control stack of the evaluator automatically kept track of what was going on at a node while its subtrees were being traversed; however, this was not sufficient and the evaluator had to leave "flags" on the tree nodes to remember what was going on from one visit to the next. Similarly, in order to construct the evaluator it was necessary to use a special data structure, the "history graph", to keep track of possible activity sequences at each type of node.

The coroutine model was suggested by these necessities and especially by the similarity of the history graph to a set of flowcharts. It seems that the earlier work's insistence on planning for a visit -- and the resulting concern with the situation upon arrival from above --, rather than planning for the entire evaluation as seen from a node, was unfortunately asymmetrical. Making the conceptual shift to looking at an entire evaluation from a particular node's viewpoint reveals the role of flags.
and history graphs more clearly and immediately suggests a generalization of the earlier work. In a coroutine evaluator the program counters of inactive coroutines play the role of the flags in "treewalk evaluators". The $\text{MOVE}_k(A)$ instructions, for $k > 0$, correspond to $\text{VISIT}(k,A)$ instructions exactly. However, the coroutine evaluators make use of $\text{MOVE}_0$ instructions to send control upward instead of the asymmetrical subroutine-return mechanism of the treewalk evaluators. The portion of a flowchart between two $\text{MOVE}_0$ instructions corresponds to a "plan"; the particular $\text{MOVE}_0$ that an inactive coroutine is suspended at thus corresponds to the "quiescent-state" flag that a treewalk evaluator would have left behind. The generalization of the earlier work is this: since upward transfers are done by the same machinery used for downward ones, parameters may be transmitted up as well as down. In effect, the treewalk evaluator formalism allows $\text{MOVE}_k$'s to send parameters, and hence $\text{MOVE}_0$'s to be multi-exit, but restricts $\text{MOVE}_0$'s to be parameterless, and hence $\text{MOVE}_k$'s to be single-exit. Elimination of this asymmetry provides a cleaner way to handle the need for "look-down". A son may send his father information about substructure through ordinary channels, without introducing new concepts.

It should be clear, then, that any treewalk evaluator may be simulated by a coroutine evaluator. Moreover, if VISITs are allowed to return results -- easily done -- we can implement any coroutine evaluator by an ordinary recursive routine. Thus it is not necessary that our coroutine evaluators actually be implemented via a coroutine linkage; if the target language for the evaluators offers no or only a very expensive coroutine linkage, recursive techniques may be substituted with acceptable efficiency.
3. VERIFICATION OF COROUTINE EVALUATORS

In this chapter we show how to prove that a set of flowcharts is in fact a correct coroutine evaluator for an ag. From the verification technique we can derive a program synthesis technique that allows us to construct evaluators systematically; however, for the moment we assume that a purported evaluator is already given. We base our approach on the method of "inductive assertions" [26]. Predicates are attached to points on each flowchart and the "verification conditions" are checked as usual. In order to allow the verification of each flowchart independently of the others, we use "interface predicates" to specify the manner of interaction between coroutines [27, 28]; each flowchart is then shown to obey the interface specifications under the assumption that all the others do so. As O.J.Dahl remarks, "A simple correctness proof is not necessarily easy to find, unless certain key assertions about the program variables are given". We therefore assume that along with the set of flowcharts we are given the attached predicates and the interface specifications. Because we have to deal with only a stereotyped class of programs, the coroutine evaluators, these predicates and specifications may be expressed in a special form allowing the existence of a correctness proof to be checked by some simple computations on finite sets.

Correctness of coroutine evaluators.

The execution of a flowchart instruction may be undefined under certain circumstances. An evaluate-instruction calls for the execution of a semantic function, and is meaningful only if the attributes referenced by the function have already been computed and stored in the tree.
A move-instruction sends control to a suspended neighboring coroutine, causing it to resume execution. The exit taken from the resumed instruction depends on the parameter sent along with control; thus the resumption is meaningful only if an appropriately labelled exit is present in the neighbor's flowchart. An attempt to perform an instruction whose execution is undefined is called an evaluate-fault or a move-fault respectively.

To correctly evaluate a semantic tree, the corresponding execution of the set of flowcharts must proceed without causing a fault, terminate after a finite number of steps, and evaluate the distinguished attribute of the root before quitting. To prove that a set of flowcharts is a correct coroutine evaluator for an ag, we must show that any execution of them is successful in this sense. Actually, any set of normal flowcharts will give rise to only finite computations, as we shall prove in a moment. A set of flowcharts is valid if no execution can cause a fault, and a valid set of flowcharts is complete if every execution results in computing the distinguished attribute. The bulk of this chapter is devoted to presenting a methodology for proving the validity and completeness of a set of flowcharts. Before going on, we dispose of the question of termination once and for all.

**Theorem:** Any execution of a set of normal flowcharts terminates after a finite number of steps, providing that all semantic functions do so and that a fault is considered to halt execution.

**Proof:** We ignore the possibility of faults, since if one occurs we are done. The key to the proof is that in a normal flowchart any path encounters a MOVE instruction in a finite number of steps. Normal flowcharts
are acyclic, so any path encounters a terminal node in a finite number of steps, and by normality this is a \( \text{MOVE}_0 \). We now proceed by induction on the maximum depth of the semantic tree. The hypothesis is that whenever control is sent down to the root of a subtree of maximum depth \( N \), after a finite number of steps it is sent back up. If \( N=1 \), the root has no nonterminal sons and hence its flowchart has no \( \text{MOVE}_k \)'s for \( k>0 \); thus any path from the resumed instruction can execute only a finite number of evaluate-instructions, each terminating, before reaching a \( \text{MOVE}_0 \) and returning control. Now suppose the hypothesis holds for depths up to \( N \), and consider sending control to the root of a subtree of depth \( N+1 \). Again, any path from the resumed instruction can execute only a finite number of instructions before hitting a \( \text{MOVE}_0 \). Each of these is either an evaluate-instruction, or a \( \text{MOVE}_k \) sending control down to the root of a subtree no deeper than \( N \); either way, the instruction terminates. We conclude that the hypothesis holds for all \( N \). The theorem is proved by noting that any execution of a set of flowcharts consists of sending control down to the root of a (finite) semantic tree and waiting for control to be returned.

| The inductive assertion method. |

We now briefly review the basic method of proving a program correct, originally due to Floyd [29] and later elaborated by Hoare [26]. The idea is to characterize a piece of program text by a pair \((P,Q)\) of assertions about the program variables, with the following interpretation: if \( P \) is true just before the text is executed, and if the execution terminates properly, after termination the assertion \( Q \) will be true. This
claim is expressed by the notation \( P<text>Q \). If \( P \) represents what the program can assume about its input data, and \( Q \) represents the intended effect of running the program, then the pair can be taken as a specification of the program's behavior. We can verify that the program meets its specifications by demonstrating that \( P<text>Q \) is a theorem deducible from the semantics of the programming language. For instance,

\[
(*) \quad b=3 \; <a:=b/f> \; a\neq 0
\]

is deducible from the semantics of ALGOL 60. The notion of correctness captured by this formalism is called partial correctness, because it is only established that the behavior will be correct if the program terminates, and not that the program actually will terminate properly. In the example just given, the truth of the assertion \((*)\) does not rule out that execution might abort, say because \( f=0 \) or because \( f \) is a typed procedure which does not halt when called.

To facilitate proving that a program meets its specifications, we attach suitable assertions to points on its flowchart. \( P \) is placed at the flowchart entry, \( Q \) is placed at the exit, and intermediate assertions are placed as convenient subject to the requirement that every loop must contain at least one assertion. The intention is that each assertion be true whenever control reaches it during an execution. Once we have an annotated flowchart, we construct the "verification conditions" as follows. A "verification path" is a path through the flowchart which starts and ends at an assertion but contains no assertions within. For any verification path, let \( Q_1 \) be the assertion at the start, \( Q_j \) the assertion at the end, and \( S_1S_2\ldots S_m \) the sequence of instructions lying along the path; then the corresponding verification condition is \( Q_1 <s_1s_2\ldots s_m> Q_j \).
Since every loop is cut by at least one assertion, there are only a finite number of these verification conditions. If every verification condition can be proved true, then we have shown \( P \langle \text{program} \rangle Q \). In the process we have also proved that each intermediate assertion \( Q_i \) will hold whenever control is at that point, provided that \( P \) is true when the program is started. This can be shown by induction on the number of times control reaches an assertion during execution; hence the name "inductive assertion method". Of course termination of the program must be shown separately.

Much depends on the choice of intermediate assertions. If the annotations are too few or too weak, it may not be possible to prove the verification conditions even though the program is correct. In general the assertions must come from a knowledge of "how the program works", and verification is usually undertaken on the assumption that adequate assertions are given along with the program.

In our case --- verifying coroutine evaluators --- most of the power of this approach is unused. Our flowcharts have no loops, and the required assertions are quite simple and stereotyped. We are more interested in establishing the truth of the intermediate assertions than of any input-output relationships for the flowcharts. We will show the impossibility of faults by establishing at each instruction an assertion sufficient to guarantee the instruction's proper execution.

The inductive-assertion method so far described deals with only a single program. We need some way of verifying each flowchart locally, independently of the rest, if we are to demonstrate the absence of faults in any execution of the set and not just in the execution of a particular connection of coroutines. Clint [27] and Dahl [28] have given a technique for verifying coroutines based on the use of interface assertions to...
specify their behavior. For our purposes we need only a special case of the technique, which may be stated as follows in terms of coroutine evalu-
ators. Choose for each nonterminal X an assertion $\phi(X)$ which is to hold just before any move-instruction with associated nonterminal X is exe-
cuted; that is, $\phi(X)$ is to hold whenever control is passed across a type-X node of the tree. Technically these $\phi$'s are inductive hypotheses about the interfaces. Each flowchart $C_p$ is then proved to make $\phi(p[k])$ true prior to executing a $\text{MOVE}_k$ instruction; in these proofs we are allowed to assume that $\phi(p[k])$ holds immediately after each $\text{MOVE}_k$, since by hypo-
thesis the k-th neighbor made it true before sending back control. If all of these proofs can be carried out, we may conclude that in any exe-
cution of the set of flowcharts, all the interface assertions will be obeyed; we will have justified our use of the induction hypotheses that $\phi(p[k])$ holds after each $\text{MOVE}_k$.

Because sending control to a coroutine allows it to resume exe-
cution where it left off, the apparent effect to $C_p$ of a $\text{MOVE}_k$ instruction may be time-dependent -- interface behavior may not be specifiable simply in terms of the situation at the moment of the MOVE, but may depend on the entire history of communication between the coroutines. This can be dealt with by imagining a "mythical variable" which records the dialog across an interface, and allowing the interface predicate to be a function of this "variable".

The general plan of a verification proof can now be seen. We are given a set of annotated flowcharts and some interface assertions. Using axioms about instruction semantics and the interface assertions, we verify that each annotation will hold when control reaches it. Finally we show that the annotation in front of each move-instruction is enough to ensure
the interface specifications are met, and that the annotations guarantee faultless execution. The plan is rather involved, so for clarity we repeat it in outline form:

1. Verify that every annotation will hold when reached.
   1.1. Assume interface specs as inductive hypotheses.
   1.2. Verify that every annotation will hold when reached, given assumptions (1.1).
      1.2.1. Prove every verification condition, using (1.1) and axioms for instruction semantics.
      1.2.2. Prove that the initial assertion on each flow-chart will hold when execution begins.
      1.2.3. CONCLUDE that every verification condition will hold when reached, given (1.1).
   1.3. Prove that the annotation in front of each move-instruction implies the corresponding interface assertion holds, given (1.1).
   1.4. CONCLUDE that use of hypotheses (1.1) was justified and that every verification condition will hold when reached.
2. Prove that each annotation implies the faultless execution of its corresponding instruction.
3. CONCLUDE that no execution can ever give rise to a fault.

Reading off of the above (1.2.1, 1.2.2, 1.3, and 2) we see that for each instruction we must prove the verification conditions originating there, that the annotation there implies the truth of the corresponding inter-
face assertion, and that the annotation implies the faultless execution of the instruction; in addition, for each flowchart's initial instruction we must show that its annotation will hold when execution begins. In all of these proofs we may use the assumption that interface assertions will hold after each move-instruction, and we may use axioms characterizing the semantics of the individual instructions.

Naturally we cannot carry out an actual verification without being given a particular ag and annotated set of flowcharts. Instead we work through a typical verification formally. We find that the necessary assertions can be expressed as finite sets, and we derive some simple set-theoretic tests on these sets, the passing of which guarantees the existence of a correctness proof based on the assertions the sets represent. For convenience and clarity, we do the typical verification in two parts, one establishing the absence of evaluate-faults, the other the absence of move-faults.

The evaluation-condition.

In this section we develop the machinery for proving that a set of flowcharts can never give rise to an evaluate-fault. The proof is quite straightforward, perhaps so much so that the inductive-assertion technique will seem like overkill; but it will at least be a good introduction to the method of proving the absence of move-faults, which is less obvious.

We are primarily concerned with which attribute occurrences are "available" when control is at a given point in some flowchart. By "available" we mean "set to the value prescribed by the ag"; we use the predicate available(a,k) to assert that occurrence (a,k) has been
computed. We will commonly use assertions that are conjunctions of this predicate, such as

\[ \text{available}(a_1, k_1) \land \text{available}(a_2, k_2) \land \ldots \land \text{available}(a_m, k_m) \]

We will use the shorthand notation \( P(A) \) for the above, where \( A = \{(a_1, k_1), \ldots, (a_m, k_m)\} \). It is an easy lemma that \( P(A) \Rightarrow P(B) \) iff \( B \subseteq A \). Let us now investigate the semantics of flowchart instructions in terms of available. First of all, no instruction ever renders an attribute unavailable, so the following are deducible from the semantics of our instructions:

\[ \text{available}(a, k) < f_{(a', k')}^P > \text{available}(a, k) \]

\[ \text{available}(a, k) < \text{MOVE}_k(A) > \text{available}(a, k) \]

Free variables are to be understood as universally quantified. Next, we observe that no matter what the situation before, if execution of an evaluate-instruction terminates properly then its associated attribute is available afterward. This is expressed by:

\[ \text{true} < f_{(a, k)}^P > \text{available}(a, k) \]

From the foregoing we can conclude the following most general facts about the available-semantics of our instructions:

(1) \( P(\text{AVAIL}) < f_{(a, k)}^P > P(\text{AVAIL} \cup \{(a, k)\}) \)

(2) \( P(\text{AVAIL}) < \text{MOVE}_k(A) > P(\text{AVAIL}) \)
These properties hold for any set \( AVAIL \) of attribute occurrences of \( p \), and capture the behavior of our instructions rather completely with regard to available. We take these as the axioms of instruction semantics to be used in our verifications.

The other thing we can use in our proofs is the interface specification. Recall that a coroutine passes control to a neighbor in order to make available some attributes that it cannot itself compute, and that the neighbor sends back along with control a parameter revealing which attributes it succeeded in computing. Without formalizing this returned information we will not know enough to prove our verification conditions. First we define an operator for notational convenience:

\[
\text{if } A \in A(X_k) \text{ then } A \ast k = \{(a,k) | a \in A\}
\]

The requirement that every attribute mentioned in a move-instruction's parameter be available is then expressed as the demand that \( P(A \ast k) \) hold immediately before a \( \text{MOVE}_k(A) \) instruction. As an induction hypothesis we may assume

\[
(3) \quad \text{true } <\text{MOVE}_k(A) \rightarrow A'> P(A' \ast k)
\]

We have used the notation \( \rightarrow A' \) to cope with the fact that move-instructions are potentially multi-exit. The truth of \( P \) on the arc leading into a move-instruction \( M \) together with the assertion \( P <M \rightarrow A'> Q \) allows us to conclude that \( Q \) holds on that out-arc of \( M \) labelled with \( A' \). Naturally different predicates may hold on different arcs out of one instruction.

The formulas (1), (2), and (3) are our tools for proving things in the current verification scheme.
To show that an evaluate-instruction \( f^P_{(a,k)} \) will execute without causing an evaluate-fault, we must establish that the assertion \( P(D^P_{(a,k)}) \) holds just before it; recall that \( D^P_{(a,k)} \) is the set of attribute occurrences referenced by the semantic function \( f^P_{(a,k)} \).

We assume that the set of flowcharts to be verified is given complete with attached assertions. Specifically, we assume the existence of a set \( AVAIL(c) \) of attribute occurrences of \( p \) associated with each node \( c \) in flowchart \( C_p \). The set \( AVAIL(c) \) is interpreted as the predicate \( P(AVAIL(c)) \) attached just before instruction \( c \). We write \( P_c \) to denote this predicate.

We now capitalize on our use of sets to represent predicates. By carrying out a couple of typical proofs we will arrive at some simple set-theoretic conditions, easily checked, on the sets \( AVAIL(c) \). The truth of these conditions will be enough to complete a correctness proof of the annotated flowcharts. Thus the stereotyped nature of our flowcharts and the assertions we want to prove about them will allow us to reduce the entire inductive-assertion method to a little set arithmetic.

Because every flowchart node has an attached predicate, the verification paths are each only one instruction long. Each verification condition simply has the form \( P_c \langle \text{INSTR}(c) \rangle P_c' \), where \( c' \) is a successor of \( c \). For each evaluate-instruction \( c : f^P_{(a,k)} \) we get one verification condition \( P_c \langle f^P_{(a,k)} \rangle P_{\text{NEXT}(c)} \), and for each move-instruction \( c : \text{MOVE}_k(A) \) we may get several verification conditions \( P_c \langle \text{MOVE}_k(A) \rangle A' \rangle P_{\text{NEXT}(c,A')} \), one for each \( A' \in \text{EXIT}(c) \). Let us consider the two kinds of instructions one at a time.
Suppose that node \( c \) is in \( C_p \) and that \( \text{INSTR}(c) = f^p(a,k) \). We must show the truth of the verification condition for \( c \) and that \( P_c \) implies the faultless execution of \( c \). The latter condition is \( P_c \Rightarrow P(D^p_p(a,k)) \).

By our lemma (easy) this is true iff

\[
(E1) \quad D^p_p(a,k) \subseteq \text{AVAIL}(c)
\]

We must also show the verification condition, i.e. that \( P_c f^p_p(a,k) \Rightarrow P \text{NEXT}(c) \).
Axiom (1) is the only relevant fact we've got. If we knew that the right side of (1) implied \( P \text{NEXT}(c) \) we could use the inference rule

\[
(P Q \Rightarrow R) \Rightarrow P R
\]

(see, for instance, Hoare [26]) to conclude that the verification condition holds; this will be true iff

\[
(E2) \quad \text{AVAIL(NEXT(c))} \subseteq \text{AVAIL}(c) \cup \{(a,k)\}
\]

Now suppose that node \( c \) is in \( C_p \) and \( \text{INSTR}(c) = \text{MOVE}_k(A) \). We first consider showing that \( P_c \) implies that \( c \) obeys the interface specification, i.e. that \( P_c \Rightarrow P(A^k) \). As before, this will be true iff

\[
(E3) \quad A^k \subseteq \text{AVAIL}(c)
\]

To show the verification condition originating at \( c \) corresponding to the exit \( A'c \text{EXITS}(c) \), we use the inference rule

\[
(P1 Q1 \& P2 Q2) \Rightarrow (P1 P2) (Q1 Q2)
\]

to obtain from (2) and (3) the assertion
\[ P(AVAIL(c)) <\text{MOVE}_k(A)\rightarrow A'> P(AVAIL(c) \cup A'^k) \]

We have also substituted the set \( AVAIL(c) \) for the free variable \( AVAIL \).
This will prove the verification condition \( P_c <\text{MOVE}_k(A)\rightarrow A'> P_{\text{NEXT}}(c,A') \)
provided that

\[ (E4) \quad AVAIL(\text{NEXT}(c,A')) \subseteq AVAIL(c) \cup A'^k \]

Assuming that all the conditions E1-E4 are satisfied by the \( AVAILs \)
at every node, we have verified almost all the assertions required to show
the absence of evaluate-faults. It only remains for us to prove that the
initial assertion on each flowchart will be true when execution begins.
Let \( c_{p0} \) be the initial node in flowchart \( C_p \). We must somehow show that
\( P(AVAIL(c_{p0})) \) is true whenever a copy of \( C_p \) is initiated. We recall
that the synthesized attributes of terminal symbols are given in advance;
hence initially the assertion \( P(INIT_p) \) holds, where

\[ INIT_p = \{(a,k) | 1 \leq k \leq n_p \text{ & } p[k] \in V_T \text{ & } a \in S(p[k])\} \]

As usual, this can be proved iff

\[ (E5) \quad AVAIL(c_{p0}) \subseteq INIT_p \]

Our methodology for showing that no execution of a set of flowcharts
can cause an evaluate-fault may be summarized as follows. We are
given a set of flowcharts together with a set \( AVAIL(c) \) for every node \( c \).
At each evaluate-instruction we check conditions E1 and E2. At each
move-instruction we check condition E3 and, for every exit, E4. At each
initial node of a flowchart we check condition E5. This combined group of checks we call the evaluation-condition. If the annotated flowcharts satisfy the evaluation-condition, we know there is a proof that no execution of them can ever give rise to an evaluate-fault.

As a final note, we remark that it is not strictly necessary to be supplied the AVAIL sets along with the flowcharts. Conditions E5, E2, and E4 may easily be used to find the largest AVAIL sets consistent with them: since our flowcharts are acyclic, we can topologically sort them, use E5 to choose the largest permitted AVAILS for initial nodes, then use E2 and E4 to propagate the sets down the flowcharts. If the AVAIL sets so obtained do not satisfy conditions E1 and E3, it is clear that no smaller sets would do so, and so the set of flowcharts cannot be proven free of evaluate-faults by this method. The flowcharts could nevertheless be free of evaluate-faults, perhaps because they obey a (nonstandard) communication convention of their own.

The closure-condition.

We now turn our attention to the more difficult problem of showing a set of flowcharts to be free of move-faults. In a sense we are trying to show that the flowcharts are "closed" under MOVEing, i.e. that no parameter transmission can take execution "out of the flowchart" due to lack of the required exit. Our plan is the same as before, but the situation is more complicated. This is primarily due to the time-dependence of the behavior we are investigating, which requires the use of mythical variables, and to the complexity of the move-instruction semantics.

When control is passed to an inactive coroutine, the acceptability of the passed parameter depends on the particular move-instruction at
which its execution was suspended. This in turn depends on the path so far taken through its flowchart, which was of course guided by the parameters previously received from its neighbors. Since these neighbors' activities were controlled in part by messages from this coroutine and in part by others beyond its reach, we can see that the progress of the various coroutines in an execution is a highly interrelated affair. As shown by Clint[27] and Dahl [28], the key to proving harmonious cooperation of such coroutines is to study the communication history of a computation. We look at an interface between two coroutines and watch the sequence of parameter transmissions across that interface. To allow independent verification of the two coroutines, we invent a predicate on transmission sequences and insist that each time control is passed across the interface the predicate must hold on the communication history to date. To prove a flowchart correct in isolation, we show that each time it sends control away the predicate holds on the history to date, under the inductive hypothesis that the predicate holds on the history each time it receives control (i.e. on the assumption that the other coroutine is correct).

**Definition:** A protocol for a nonterminal X is a sequence \( A_0 A_1 A_2 \ldots A_m \)

where \( A_i \in I(X) \) if \( i \) is even and \( A_i \in S(X) \) if \( i \) is odd.

A protocol is a possible sequence of parameter transmissions across a type-X tree node. We denote protocols by symbols such as \( \pi \) and \( \pi' \). Concatenation of protocols is indicated as usual for sequences, by juxtaposition; if \( H \) is a set of protocols, then \( H\pi \) denotes the set \( \{ \pi'\pi | \pi' \in H \} \). Since the \( A(X)'s \) are disjoint, any \( \pi \) is "for" only one nonterminal.
The basis for our treatment of cooperation is the choice of a finite set of **legal** protocols for each nonterminal. Our interface specifications enumerate all the ways in which pairs of coroutines are permitted to interact; whenever control is passed across a type-X node of the tree, the transmission sequence to date must be a legal protocol for X. This simple form of interface assertion happens to be adequate for dealing with any noncircular ag, and is relatively easy to work with. It is natural to require that if \( \pi \) is a legal protocol for \( X \), so is every prefix of \( \pi \). Tracing along any path in a flowchart, we can write down the protocol engaged in by noting, in sequence, the parameters of and exits taken from the encountered \( \text{MOVE}_k \) instructions. More interestingly, by comparing the protocols leading to a particular \( \text{MOVE}_k \) instruction with the set of legal protocols for \( X_k \), we can determine what exits it must have to be sure of avoiding move-faults. The only thing guaranteed by the interface specifications is that the received parameter will extend the current protocol into another legal protocol; thus if \( \pi \) is the protocol engaged in along some path up to \( c: \text{MOVE}_k(A) \), then when \( c \) sends control away the current protocol will be \( \pi A \), and so if \( \pi AA' \) is a legal protocol for \( X_k \) then \( c \) had better have an exit labelled \( A' \). According to the interface specification, \( A' \) would be a permissible reply.

We now get somewhat more formal. To verify the flowchart \( C_p \), we invent a number of "mythical variables" [27,28] to keep track of the communication history with each neighbor. For \( 0 \leq k \leq n_p \), the variable \( h_k \) contains at each moment the protocol (for \( p[k] \)) which is the transmission sequence to date across neighbor \( k \). Each \( \text{MOVE}_k \) instruction has the additional "effect" of concatenating new elements onto the end of \( h_k \). These variables and their updating are mythical in the sense that they are not
a part of the actual execution, but play a role only in the verification
proofs. Their use will become clearer shortly. We assume that along
with the set of flowcharts we are given a number of sets \( H_k(c) \) for each
flowchart node \( c \). If \( c \) is in \( C_p \), then for \( 0 \leq k \leq n_p \) the set \( H_k(c) \) con-
tains some legal protocols for \( p[k] \). Each set \( H_k(c) \) is interpreted as
the assertion \( Q_k(H_k(c)) \) attached just in front of instruction \( c \), where

\[
Q_k(H) \text{ means } (h_k = \pi_1 \text{ or } h_k = \pi_2 \text{ or } \ldots \ h_k = \pi_m) \text{ where } H = \{ \pi_1, \pi_2, \ldots \pi_m \}
\]

Intuitively \( H_k(c) \) is the set of protocols which may have been engaged in
with neighbor \( k \) by the time control reaches \( c \). These assertions will of
course need verifying. We will write \( Q_{k+c} \) for \( Q_k(H_k(c)) \), and \( Q_c \) to
mean \( (Q_0_c \& Q_1_c \& \ldots \ Q_{n_p} c) \). We observe that \( Q_k(H) =\Rightarrow Q_k(H') \text{ iff } H\subseteq H' \).

In addition to the flowchart annotation sets, we must be given the
interface specifications in the form of a set \( \text{LEGAL}(X) \) of protocols for \( X \),
for each nonterminal \( X \) in the grammar. We make the requirement that if
\( \pi \in \text{LEGAL}(X) \), then \( \pi' \in \text{LEGAL}(X) \) for all prefixes \( \pi' \) of \( \pi \). In particular,
the empty prefix \( \Lambda \) is a member of every \( \text{LEGAL}(X) \). We define for each \( X \)
the predicate \( \text{legal}_X(\pi) \) so that \( \text{legal}_X(\pi) \text{ iff } \pi \in \text{LEGAL}(X) \). Having said
this, we immediately drop the subscript \( X \) since no confusion can result.

Now that we have established the kinds of assertions we will be
using, let us investigate the semantics of flowchart instructions in these
terms. The evaluate-instructions of course have no effect on the current
protocols at all, and so we assert:

\[
(4) \text{ for any } k, \ Q_k(H) <_{\text{fp}} (a',k') > Q_k(H)
\]
To characterize the move-instructions satisfactorily, we must analyse their effects in some detail. We shall need to prove, for instance, that when a suspended MOVE is resumed the received parameter corresponds to some exit label. Attaching assertions to the out-arcs of such an instruction cannot help us, for naturally if control proceeds along one of those arcs there has been no move-fault -- it is precisely the situations where no arc can be taken that we are concerned about! We must therefore decompose each move-instruction in order to talk about their inner workings. We think of an instruction $\text{MOVE}_k(A)$ as being composed of a "GO\(_k\)(A)" instruction with a single exit followed by a "BRANCH" instruction whose exits are those of the original MOVE. The coroutine linkage is carried out as usual by GO instructions; on exit from a GO, a parameter has been received but not yet used to guide a multi-way branch. The GO instructions cannot cause faults. It is the BRANCH instruction which may cause a move-fault if the received parameter does not match one of its exits. We invent the mythical variable $r$ to hold the received parameter until a BRANCH can act on it. The apparent effect of a GO\(_k\) instruction is to extend $h_k$ by two elements, the transmitted parameter and then the received parameter. Thus we can assert:

$$h_k = \pi <\text{GO}_k(A) > h_k = \pi Ar$$

and $$h_j = \pi <\text{GO}_k(A) > h_j = \pi$$ for $j \neq k$

We use the inference rule $(P1 \text{ or } P2) \Rightarrow (Q1 \text{ or } Q2)$ to generalize these, and combine them with a conditional expression to get
(5) \[ Q_j(H) \langle \text{GO}_k(A) \rangle Q_j(\text{if } j=k \text{ then } H\text{Ar} \text{ else } H \text{ fi}) \]

as our axiomatization of the \text{GO} instruction with respect to protocols.

The \text{BRANCH} instruction has no effect on the \text{h}_k's, but it takes exit \text{A} only when \( r = A \). We have, then, that

\[ Q_k(H) \langle \text{BRANCH}+A \rangle Q_k(H) & r=A \]

Since the text \( \langle \text{MOVE}_k(A)\rightarrow A' \rangle \) is equivalent to \( \langle \text{GO}_k(A) \rangle \langle \text{BRANCH}+A \rangle \)
we can use the inference rule \( (P<s_1>Q \text{ & } Q<s_2>R) \Rightarrow P<s_1;s_2>R \) to get the
following axiom for the whole move-instruction:

\[ Q_j(H) \langle \text{MOVE}_k(A)\rightarrow A' \rangle Q_j(\text{if } j=k \text{ then } HAA' \text{ else } H \text{ fi}) \]

We now have a sufficient characterization of the semantics of flow-chart instructions with respect to protocols, as given by (4)-(7). We
now need to formalise the inductive hypothesis that after a move-instruction the interface assertion holds. We will use this assumption at the
earliest point possible: just after resumption, and before the \text{BRANCH}.

We need to know this ahead of the \text{BRANCH} in order to show that its execution does not cause a fault. We may write

\[ \text{true} \langle \text{GO}_k(A) \rangle \text{legal}(\text{h}_k) \]

We now begin the verification process, using (4)-(8). Suppose
that \( c \) is in \( C_p \) and \( \text{INSTR}(c) = f^p_{(a,k)} \). No interface-assertion considerations have to be made, and we are not now concerned with showing the faultless execution of evaluate-instructions, so all we have to prove is
the single verification condition originating at c. We must show that
\[ Q_c ^{f_p (a, k)} \circ Q_{\text{next}} (c). \]
It is clear from (4) that this will be true iff
\[
(C1) \quad \text{for } 0 \leq k \leq n_p, \quad H_k (c) \subseteq H_k (\text{next}(c))
\]

Now suppose that \( \text{instr}(c) = \text{move}_k (A) \). As well as proving all the
verification conditions originating at c, we must show that execution of
c obeys the interface specification and does not cause a fault. We begin
with the verification conditions. For each \( A' \in \text{exits}(c) \) we must show
\[ Q_c ^{\text{move}_k (A) \rightarrow A'} \circ Q_{\text{next}} (c, A') \]

We can prove this by proving instead \( Q_{jc} ^{\text{move}_k (A) \rightarrow A'} \circ Q_{jc} \circ \text{next}(c, A') \) for
all \( 0 \leq j \leq n_p \). When \( j \neq k \), we see from (7) that this holds only when
\( Q_{jc} \Rightarrow Q_{jc} \circ \text{next}(c, A') \); that is, iff
\[
(*) \quad H_j (c) \subseteq H_j (\text{next}(c, A')) \quad \text{for all } j \neq k
\]
On the other hand, for \( j = k \) we see from (7) that we need \( Q_k (\text{next}(c, A')) \Rightarrow
Q_k (\text{next}(c, A')) \), which will be the case iff
\[
(**) \quad H_k (c) A'A' \subseteq H_k (\text{next}(c, A'))
\]
Combining (*) and (**), and remembering that we were to prove one of
these verification conditions for each exit of c, we get the following
condition which is equivalent to the validity of all of c's verification
conditions:
for all $A' \in \text{EXITS}(c)$, for all $0 \leq j \leq n_p$,
\[
\text{if } j=k \text{ then } H_k(c) A' \text{ else } H_j(c) \text{ fi } \subseteq H_j(\text{NEXT}(c,A'))
\]

Next we attempt to show that execution of $c$ obeys the interface specifications: specifically, we must prove that when control is sent out of the flowchart by $c$, the current protocol with neighbor $k$ is legal. Since $h_k$ will hold the protocol engaged in up to $c$, $h_k A$ will be the protocol to date once $c$ transmits its parameter. We must show that $Q_c$ implies $\text{legal}(h_k A)$. Then the following computation applies:

\[
Q_{kc} \Rightarrow \text{legal}(h_k A)
\]

iff

\[
(h_k = \pi_1 \text{ or } \ldots \text{ or } h_k = \pi_m) \Rightarrow \text{legal}(h_k A) \text{ where } H_k(c) = \{\pi_1, \ldots, \pi_m\}
\]

iff

\[
(h_k = \pi_1 \Rightarrow \text{legal}(h_k A)) \text{ and } \ldots \text{ and } (h_k = \pi_m \Rightarrow \text{legal}(h_k A))
\]

iff

\[
\text{legal}(\pi_1 A) \text{ and } \ldots \text{ and } \text{legal}(\pi_m A)
\]

iff

\[
\pi_1 A \in \text{LEGAL}(X_k) \text{ and } \ldots \text{ and } \pi_m A \in \text{LEGAL}(X_k)
\]

Of course if $Q_{kc} \Rightarrow \text{legal}(h_k A)$ then $Q_c \Rightarrow \text{legal}(h_k A)$. So $c$ obeys the interface specification provided that

\[
\text{(C3) } \text{ for all } \pi \in H_k(c), \pi A \in \text{LEGAL}(X_k)
\]

Finally, we must demonstrate that $Q_c$ implies that execution of $c$ cannot give rise to a move-fault; i.e. that $Q_c$ before the transfer of control guarantees a corresponding exit for any received parameter. We must prove, then, that $Q_c < \text{GO}_k(A) > rc\text{EXITS}(c)$. Using (5) with $j=k$ together with (8) yields the theorem

\[
Q_{kc} < \text{GO}_k(A) > Q_k(H_k(c) A) \Rightarrow \text{legal}(h_k)
\]
We can massage this formula as follows:

\[ Q_{kC} <G_0, (A)> (h_k = \pi_1. Ar \ or \ ... \ h_m = \pi_m. Ar) \ & \ legal(h_k) \]

iff 

\[ Q_{kC} <G_0, (A)> \ legal(\pi_1. Ar) \ or \ ... \ legal(\pi_m. Ar) \]

To establish the desired result, we need the right side of this to imply \( r \in EXITS(c) \). We see that

\[ (legal(\pi_1. Ar) \ or \ ... \ legal(\pi_m. Ar)) \Rightarrow r \in EXITS(c) \]

iff 

\[ (legal(\pi_1. Ar) \Rightarrow r \in EXITS(c)) \ and \ ... \ (legal(\pi_m. Ar) \Rightarrow r \in EXITS(c)) \]

and this will be true provided that

\[ (C4) \ for \ all \ \pi \in H_k(c), \ \pi Ar \in LEGAL(X_k) \Rightarrow r \in EXITS(c) \]

If so, we can conclude that \( Q_{kC} <G_0, (A)> r \in EXITS(c) \), and so no move-fault is possible.

After some lengthy calculation, we have arrived at a methodology for establishing that a set of flowcharts always cooperate properly, so that no execution of them can give rise to a move-fault. The only thing we have not verified is that the initial assertion on each flowchart will hold when execution begins, but this is easy:

\[ (C5) \ for \ each \ initial \ node \ c_{p0} \ in \ C_p, \ for \ all \ 0 \leq k \leq n_p, \]
\[ H_k(c_{p0}) = \{ \Lambda \} \]

is the last condition we need. It simply reflects the fact that before execution is underway, no transmissions have been exchanged with any other coroutine. The conditions C1 to C5 are collectively the closure-condition.
Any set of annotated flowcharts and their legal-protocol sets which satisfy the closure-condition and the evaluation-condition as well are provably valid -- any execution of them will run without faults and terminate in a finite number of steps.

The completion-condition.

For a valid set of flowcharts to constitute a coroutine evaluator for an ag, it must also be established that any execution of them will result in computing the distinguished attribute of the root -- i.e. will result in yielding the translation of the input string. In terms of the evaluation-condition formulation, this is easily stated:

for all terminal nodes $c$ of $C_0$, the root flowchart, we must have $AVAIL(c)$ containing the distinguished attribute.
4. CONSTRUCTION OF COROUTINE EVALUATORS

Given an attribute grammar, we wish to mechanically implement it. In this chapter we show how a set of flowcharts can be built up instruction by instruction to finally obtain a coroutine evaluator for the ag. In keeping with modern practice, the evaluator and its proof of correctness are constructed at the same time. We begin with a trivial flowchart for each production consisting of only the required initial node, a dummy $\text{MOVE}_0$ instruction. We give rules which specify what additions can or must be made to a partially constructed set of flowcharts. These rules ensure that at any stage of the construction, the set of (incomplete) flowcharts so far built up will satisfy the conditions for well-defined execution: no semantic function will be evaluated before its argument attributes are available, and whenever control is passed to a coroutine its flowchart will have the necessary exit. These rules are rather permissive, in that at a given stage of construction there are generally many allowed choices for a next addition to the flowcharts.

This can be viewed as a nondeterministic algorithm [30] for building evaluators. Any permitted sequence of additions to the initial flowcharts is a computation of the algorithm. Some computations do not halt; that is, it is possible to go on extending an incomplete evaluator indefinitely without "getting anywhere". However, for any noncircular ag the algorithm does have halting computations, and any halting computation yields a correct evaluator for the given ag. Some computations yield "better" evaluators than others, to be sure.

The algorithm given here is the bare skeleton of a construction technique. It contains only the rules necessary to guarantee the correct-
ness of a generated evaluator. All strategy, or means of working toward a desired goal, is omitted; nondeterministic guesses take its place. Although this algorithm could be executed, for example by breadth-first simulation of the nondeterminism, we do not suggest that it ought to be. It is not intended for practical use, but as a framework for developing and comparing various realistic constructions. A whole family of related algorithms may be obtained by adding some strategy or additional choice rules to the general construction [31]. In the next chapter we discuss some possible strategies: heuristic criteria are used to narrow the space of possible computations so as to allow only "good" evaluators to result, sometimes at the expense of failing entirely on certain args. In addition, a later chapter describes a special case of particular practical interest, the "uniform evaluators".

The construction method we present is closely related to the verification concepts of chapter 3. There we assumed we were given an annotated set of flowcharts and some interface specifications describing how the coroutines were supposed to cooperate. Both the attached assertions and the interface specifications could be represented as finite sets: the sets AVAIL(c) of attribute occurrences available at c; the sets $H_k(c)$ of possible histories at c of communication with the k-th neighbor; and the sets LEGAL(X) of legal communication histories across nonterminal X. We derived some simple set-theoretic conditions on these sets the satisfaction of which was sufficient to guarantee the correctness of the coroutine evaluator.

Our construction is obtained by reversing these conditions into "permissions" for adding to the set of flowcharts. The key assertions in the verification methodology become bookkeeping variables in the gen-
eral construction algorithm. In addition to the set of partially constructed flowcharts, we maintain sets $AVAIL$ and $H_0 \ldots H_n$ for each node in them and a $LEGAL$ set for each nonterminal. At any moment the set of flowcharts can be proved valid (though not in general complete) by using the current sets as annotations and interface specifications. Each step in the construction is made so as to leave this invariant. An evaluate-instruction may be added if doing so will leave the evaluation-condition satisfied; a move-instruction may be added at any time, but doing so may require the addition of new exits from existing move-instructions in order to preserve the closure-condition. Each addition is accompanied by the appropriate adjustment to the bookkeeping sets. If after some number of steps the completion-condition is also found to be satisfied, the construction halts with a correct evaluator.

Before presenting the general algorithm, we repeat without comment the set-theoretic conditions derived in the last chapter. Our current numbering differs from that used before for our convenience of exposition.

For an evaluate-instruction $c: f_{(a,k)}^p$ in $C_p$ --

$F1. \quad D_{(a,k)}^p \subseteq AVAIL(c)$

$F2. \quad AVAIL(\text{NEXT}(c)) \subseteq AVAIL(c) \cup \{(a,k)\}$

$F3. \quad \text{For } 0 \leq k \leq n, \quad H_k(c) \subseteq H_k(\text{NEXT}(c))$

For a move-instruction $c: \text{MOVE}_k(A)$ in $C_p$ --

$M1. \quad A^*k \subseteq AVAIL(c)$

$M2. \quad \text{For all } A' \in \text{EXITS}(c), \quad AVAIL(\text{NEXT}(c,A')) \subseteq AVAIL(c) \cup A'^*k$

$M3. \quad H_k(c)A \subseteq \text{LEGAL}(p[k])$
M4. For all $p \in H_k(c)$, $\forall AA' \in \text{LEGAL}(p[k]) \Rightarrow A' \in \text{EXITS}(c)$

M5. For all $A' \in \text{EXITS}(c)$, for all $0 \leq j \leq n_p$,

$\begin{cases} 
\text{if } j = k \text{ then } H_j(c)AA' & \text{else } H_j(c) \text{ fill } \in H_j(\text{NEXT}(c, A'))
\end{cases}$

For an initial node $c_0$ in $C_p$ --

1. $\text{AVAIL}(c_0) \subseteq \text{INIT}_p$

2. For all $0 \leq k \leq n_p$, $H_k(c_0) = \{A\}$

We demonstrate the correctness of the construction algorithm by showing that the initial set of flowcharts satisfies these conditions and that every step preserves them.

Construction conventions.

The evaluator under construction is at any moment a collection \{C_p | p \in P\} of directed graphs some of whose nodes are labelled with instructions. Certain terminal nodes of the graphs may be temporarily unlabelled; these are termed pending nodes. Pending nodes are created when it is necessary to add an exit out of some node: a new arc leading to a new node is added, but the instruction to label the new node has not yet been chosen. Along with the graphs we maintain for each node $c$ in $C_p$ a set $\text{AVAIL}(c)_p$ of attribute occurrences of $p$, and sets $H_k(c)_p$ for $0 \leq k \leq n_p$, of protocols for nonterminal $p[k]$; and for each nonterminal $X$, a set $\text{LEGAL}(X)$ of protocols for $X$. Under the interpretation of the last chapter, these sets constitute a proof of the validity of the set of flowcharts at the end of any step.

After initialization, three kinds of steps may be taken. An eval-
uate-instruction may be added at a pending node, producing a single exit to a new pending node. A move-instruction may be added at a pending node, after which new exits are added from this or other MOVEs as required. By applying these two steps only, the initial graphs may be grown into tree-shaped graphs, since paths may diverge at move-instructions but never converge. The third type of step is not strictly necessary, but allows several pending nodes to be merged into a single one; arbitrary acyclic flowcharts may be produced this way. We describe each of these steps below, giving the condition under which the step may be taken and the actions involved in taking it. A computation of the algorithm consists of initializing, followed by taking any sequence of permitted steps; if at the end of some step the set of flowcharts satisfies the completion-condition the computation may halt. After any halting computation, the set of flowcharts is a correct coroutine evaluator for the ag, as may be established using the AVAIL, $H_k$, and LEGAL sets with the methods developed in chapter 3.

**Initialization.**

Initially, each graph $C_p$ consists of a single node $c_{p0}$ labelled $\text{MOVE}_0(\Lambda)$ and no arcs. For each $p \in P$ let $\text{AVAIL}(c_{p0}) = \text{INIT}_p$ and let $H_k(c_{p0}) = \{\Lambda\}$ for $0 \leq k \leq n_p$. The initial move-instructions are dummies, never actually executed but only resumed; since no parameter is ever transmitted by one of them, they should not have the effect of tacking an element onto $H_0$; this is the meaning of the rather odd parameter-symbol $\Lambda$. With this understanding, we note that validity conditions I1 and I2 are thereby satisfied. Since they involve only initial nodes, these conditions must remain satisfied regardless of additions to the flowcharts. Let $\text{LEGAL}(X) = \{\Lambda\}$ for
all nonterminals X. Add an arc labelled $\emptyset$ from $c_{00}$, the initial node in the flowchart for production 0, to a new pending node. This corresponds to the root coroutine receiving control from "outside" at the start of execution. Note that if the graphs are viewed as a set of flowcharts by treating pending nodes as "halt" instructions, any execution at this point would terminate immediately upon initiation; the root coroutine would quit without ever doing anything. Although valid, this is about as far from complete as they come.

It is easy to see that the M-conditions are satisfied by this initial set of flowcharts; the F-conditions are vacuously satisfied since there are no evaluate-instructions.

Adding evaluate-instructions.

A semantic function is ready to evaluate if all its argument attributes have been computed, but it has not yet been evaluated. An evaluate-instruction may be added at any point where a semantic function is ready to evaluate. Thus we may state --

STEP: Add instruction $f^p_{(a,k)}$ at pending node c in $C_p$.

CONDITION: $D^p_{(a,k)} \subseteq AVAIL(c)$ and $(a,k) \notin AVAIL(c)$.

ACTION:

1. Label c with the instruction $f^p_{(a,k)}$.
2. Add an arc from c to a new pending node c'.
3. Let $AVAIL(c') = AVAIL(c) \cup \{(a,k)\}$.
4. For $0 \leq k \leq n_p$, let $H_k(c) = H_k(c')$. 

66
Clearly taking this step does not affect the M-conditions or the I-conditions. Since the CONDITION was true if the step was taken, F1 holds for the new instruction c. Action (3) ensures that F2 holds, and (4) ensures the truth of F3.

Adding move-instructions.

The addition of move-instructions to a set of flowcharts is considerably more complicated than adding evaluate-instructions, because the global problem of communication and cooperation between coroutines is involved. A MOVE_k for any k can always be added at any pending node, unlike evaluate-instructions; but after doing so, it is necessary to make sure that the consequences of executing it are taken into account. Essentially we must see that any instruction which might be resumed as a result of executing the new MOVE does in fact have the exit label that that resumption will require; that is, we must preserve the closure-condition. Our basic device for doing this is the LEGAL set of the nonterminal associated with the just-added instruction. If the parameter transmission performed by the new instruction can be the last in a sequence π of transmissions, then the protocol π must be included in the LEGAL set. As we know, the LEGAL communication histories determine which exits a move-instruction must have; adding to the LEGAL set may then require adding corresponding exits out of some other MOVES. Moreover, we must use the LEGAL set to see what exits the new move itself needs.

Definition: An instruction c: MOVE_k(4) is suspended with protocol π = π'A
iff π' ∈ H_k(c).

67
Briefly, if an instruction is "suspended with \( \pi \)" then whenever the transmission sequence across its nonterminal has been \( \pi \), it is possible that execution of its coroutine was last interrupted when it sent control away; consequently whenever the transmission sequence has been \( \pi \), it is possible that the next transmission will result in resuming it.

**Definition:** To add an \( A' \)-exit to instruction \( c \): \( \text{MOVE}_k(A) \) in \( C_p \) ---

1. Add an arc labelled \( A' \) from \( c \) to a new pending node \( c' \).
2. Let \( \text{AVAIL}(c') = \text{AVAIL}(c) \cup A'^*k \).
3. For \( 0 \leq j \leq n_p \), let \( H_j(c') = \begin{cases} j=k & \text{if } j=k \\ \text{else} & H_j(c) \end{cases} \).

Note that the truth of conditions M2 and M5 is explicitly preserved by the act of adding an exit to a move-instruction and there is no effect on the other conditions.

We now give the details of the step for adding a \( \text{MOVE} \).

**STEP:** Add instruction \( \text{MOVE}_k(A) \) at pending node \( c \) in \( C_p \).

**CONDITION:** \( A'^*k \subset \text{AVAIL}(c) \).

**ACTION:**

1. Label \( c \) with the instruction \( \text{MOVE}_k(A) \).
2. Let \( \text{LEGAL}(p[k]) = \text{LEGAL}(p[k]) \cup H_k(c)A \).
3. For all \( \pi \in H_k(c) \), for all instructions \( c' \) suspended with \( \pi \), add an \( A \)-exit to \( c' \) if there is not one already.
4. For all \( \pi \in H_k(c) \), for all \( A' \) such that \( \pi AA' \in \text{LEGAL}(p[k]) \), add an \( A' \)-exit to \( c \).
Let us check to see that the M-conditions are in fact preserved when a step of this kind is taken. M1 is preserved because the step is taken only if the CONDITION is true. Action (2) ensures that M3 is true by simply making it so. Condition M4 is slightly more subtle; it is the one which insists that each move-instruction have enough exits. If we show that all exits required by M4 get added if not already present, then we are done -- recall that M2 and M5 are preserved by the addition of an exit.

The preservation of M4 is accomplished by actions (3) and (4) which add any missing exits. We know that M4 held before this step was taken; afterwards, it could only fail to hold for the new node c, or for some old node c' which now needs a new exit because LEGAL(p[k]) has increased in size. It is clear that (4) adds all the exits M4 requires for c. As for any c' needing a new exit, it would have to be due to an implication of the form \([\pi BA \in \text{LEGAL}(p'[k'])] \Rightarrow \text{AEXITS}(c')\) where c': MOVE\(_k\)'(B) is in flowchart C\(_p\)' and p'[k'] = p[k] and \(\pi \in \text{H}_k\)'(c'). For this to be a new exit, the left side of the implication must have just become true; i.e. the protocol \(\pi BA\) was just added to \(\text{LEGAL}(p[k])\). This last fact implies that \(\pi \in \text{H}_k\)(c). Now \(\pi \in \text{H}_k\)'(c') means that c' is suspended with \(\pi B\), and so c' was one of the instructions processed by the iteration of (3). Therefore c' received an A-exit just as required by M4. We conclude that every old node which needed a new exit was actually given one, and so M4 remains satisfied after this step.

**Merging a set of pending nodes.**

The construction steps given so far will always produce flowcharts that are trees; control may branch out from move-instructions, but two paths never come together. Any ag which has a coroutine evaluator has one
of this form. However, for reasons such as concern over evaluator size, or to cater to specific versions of the general algorithm, we may wish to build evaluators in which paths occasionally come together. For instance, to tailor this construction into a version that behaves like the "treewalk evaluator" construction of [25], we want to identify nodes in flowcharts with the "evaluation states" used there. This requires that instruction sequences leading to the same evaluation-state come together in a single node.

We therefore give an additional rule which permits a step to merge several pending nodes into one. By allowing the merger of only pending nodes we ensure that no loops can be formed.

STEP: Merge pending nodes $c_1, c_2, \ldots, c_m$ in $C_p$.

CONDITION: For all $i, j$, $AVAIL(c_i) = AVAIL(c_j)$.

ACTION:

1. Create a new pending node $c'$.

2. Let $AVAIL(c') = AVAIL(c_1)$.

3. For $0 \leq k \leq n_p$, let $H_k(c') = H_k(c_1) \cup H_k(c_2) \cup \ldots \cup H_k(c_m)$.

4. Make all arcs going to a $c_i$ go to $c'$ instead, and delete the nodes $c_1, c_2, \ldots, c_m$.

We insist that all merged nodes have the same $AVAIL$ set for the following reason: if they did not, the new node $c'$ would have to have the intersection of all $AVAIL(c_i)$ as its $AVAIL$ set in order to satisfy F2 and M2, and this set would be strictly smaller than some of the $AVAIL(c_i)$'s. This would amount to "forgetting" that some attributes had been evaluated, and there would be no way to prevent their being computed again on the
path out of c'. We wish to avoid this unpleasant prospect. However, if a more general merge is desired we have only to replace the CONDITION by \texttt{true} and make (2) set \texttt{AVAIL(c')} to the intersection as mentioned.

The merge clearly is relevant only to conditions F2, F3, M2, and M5, since the nodes involved are themselves neither evaluate-instructions nor move-instructions. F2 and M2 are preserved because of the equality of all of the \texttt{AVAIL}s, guaranteed because the step is taken only if the CONDITION yields \texttt{true}. F3 and M5 are preserved because action (3) takes the union of all \texttt{H}_k's to get \texttt{H}_k(c'); thus the containments are not disturbed.

**Terminating the general construction.**

We will now give a slight modification of the general construction just presented which is guaranteed to terminate in a finite number of steps regardless of what nondeterministic choices are made. Restrictions are placed on what steps may be taken in order to guarantee that each step represents progress toward completing an evaluator. There are two reasons for presenting this modification: to prove the theorem "Every noncircular attribute grammar has a coroutine evaluator", and to serve as a foundation for developing later constructions whose termination will be ensured by the termination of this one.

The terminating general construction (TGC) is the same as the general construction just given, with the additional restriction that every step taken satisfy all of the following:
(1) No merge steps are taken.

(2) A move-instruction is added at c only if no evaluate-instruction can be added at c.

(3) A $\text{MOVE}_k(A)$ instruction is added at c only if $A^*k = \text{AVAIL}(c)$.

(4) No $\text{MOVE}_k$ instruction is added at c if the $k$-instruction immediately preceding c was also a move-instruction, for $k > 0$.

(5) A $\text{MOVE}_0$ instruction is added at c only if no other instruction can be added at c.

If no next addition to the flowcharts is permitted, but the completion-condition is not satisfied, we say the computation fails. If the completion-condition is satisfied, the computation succeeds.

Restriction (4) enforces that a move to a son is made only if it will be the first time control is given to the son, or if a new input has been made available to the subtree since the last time the son got control.

**Lemma:** For any input grammar, every computation of the TGC succeeds or fails after a finite number of steps.

**Proof:** Restriction (4) forbids two $\text{MOVE}_k$ instructions to occur on a path without an intervening evaluate-instruction for k. So at most $n + 1$ instructions may be added in sequence before adding an evaluate-instruction (namely a $\text{MOVE}_k$ for each $0 \leq k \leq n$), and so every path of length $n + 2$ through $C_p$ must contain an evaluate-instruction. Since there are only a finite number of attribute occurrences for $p$ and no $f_p^{(a,k)}$ is permitted to occur twice on the same path, we can conclude that every path through $C_p$ is shorter than some a priori bound: specifically, that no path through $C_p$ contains more than $N_p^*(n + 2)$ instructions, where $N_p$ is the number of
attribute occurrences of \( p \). Of course the out-degree of each node is bounded a priori, e.g. by the number of distinct labels available for exits, so the total number of instructions in the set of flowcharts is bounded a priori by a number depending only on the input \( \alpha \). Every step in the computation adds an instruction to the flowcharts (merge steps are forbidden), hence the construction must halt after a bounded number of steps.

The informal intention of these restrictions is to ensure that any attribute that can be evaluated within a subtree is in fact evaluated before control is passed up out of the subtree. This is not difficult to show.

**Lemma:** Whenever a \( \text{MOVE}_0 \) instruction is executed by a coroutine attached to node \( n \) of a semantic tree, all attributes of the subtree with root \( n \) which can be evaluated have been evaluated (i.e. there are no attributes "ready to evaluate" within the subtree).

**Proof:** By the method of inductive assertions. Attach in front of each \( \text{MOVE}_0 \) instruction the assertion "there are no attributes ready to evaluate within this subtree". Then for each flowchart we show that the assertion in front of each \( \text{MOVE}_0 \) is true when reached, on the inductive hypothesis that the lemma is true for the other flowcharts. Consider any instruction \( c: \text{MOVE}_0(4) \) in the current flowchart. Restriction (2) and the fact that a move-instruction plainly was added at \( c \) imply that no evaluate-instruction could be added at \( c \), i.e. that at \( c \) no attribute occurrence is ready to evaluate. Restriction (5) implies that no \( \text{MOVE}_k \) for \( k > 0 \) could be added.
at c; since in the general construction a MOVE_k can be added at any point, the only reason a MOVE_k could not be added is restriction (4), and so it follows that for each k, the last k-instruction executed before reaching c was a MOVE_k. Control was sent to neighbor k, who returned it by executing a MOVE_0 of its own; by hypothesis, at that moment the k-th subtree had no attributes ready to evaluate. Since no new inputs to it have been made available, the k-th subtree still has no attributes ready to evaluate by the time c is reached. Thus there are no ready occurrences either at the current tree node or within any of its subtrees when c is executed, and the inductive assertion holds at c. This is true for any MOVE_0, hence we conclude it is true for every one of them. 

\[ \]

**Corollary:** Any execution of a set of flowcharts built by the TGC results in evaluating every attribute of the semantic tree which can be evaluated.

**Theorem:** Every noncircular attribute grammar has a coroutine evaluator.

**Proof:** Given an ag, run the TGC with it as input. By the first lemma, this process halts in a finite time yielding a set of flowcharts. If the computation succeeds, the correctness of the general algorithm implies that they are a coroutine evaluator for the grammar (note that every computation of the TGC is also a computation of the general algorithm).

However, suppose the computation fails -- i.e. the set of flowcharts does not satisfy the completion-condition. Then there must be some semantic tree that the set of flowcharts does not completely evaluate. But the corollary states that every attribute of the tree which can be evaluated
is actually evaluated; hence any unevaluated attributes must depend on other unevaluated attributes. From any unevaluated attribute, then, we can trace a dependency arc back to another unevaluated attribute; but as there are only a finite number of attributes in the semantic tree, this backward trace must finally return to a previously encountered attribute. That is, there must be a circular dependency chain in this tree, and so the given ag is circular.

Corollary: The TGC is a circularity test for ags; that is, for any input ag it determines in a finite number of steps whether the ag is circular.

Corollary: The worst-case running time of the TGC is exponential in the size of the input ag.

Proof: Jazayeri [16] has shown the intrinsic exponentiality of circularity testing for ags.

Implementing the general construction.

Although we do not advocate the direct use of either the general construction or the TGC to build evaluators, we shall propose a number of practical constructions based on them, and it is of interest whether the bookkeeping mechanism used to administrate adding instructions to flow-charts can be implemented reasonably. In this section we sketch a natural realization of the bookkeeping in concrete terms.
There seems to be no problem in implementing the AVAIL sets and operations on them, perhaps by bit strings associated with each flowchart node with one bit position assigned to each attribute occurrence of the relevant production. The main issue we take to be how to make efficient the searches implicit in actions (3) and (4) of adding a move-instruction:

(a) "for all instructions c' suspended with π",
and  
(b) "for all A' such that πAA'εLEGAL(p[k])"

The key to speeding up (a) would seem to be maintaining some sort of cross-reference between protocols and instructions suspended by them, while (b) seems to call for a way of storing the LEGAL sets so as to make access to the extensions of a protocol fast. Since if π is in a LEGAL set so is every prefix of π, and since new protocols added to a LEGAL set are always one-element extensions of protocols already there, it is natural to think of storing a LEGAL set as a tree with nodes labelled by A's. Each path from the root to a node represents one legal protocol, and the node's sons are labelled with all legal one-element extensions of that protocol. This makes (b) easy to do, and causes no problems when new protocols must be added: because the new protocol is just an extension of some old one, we need only add a son to some node of the LEGAL tree. Now we note that any protocol we ever need to manipulate is a legal one, and so will occur in a LEGAL set. Thus we can represent any protocol uniquely by a reference to its corresponding node in some LEGAL tree. This makes the LEGAL trees a natural place to hold the "cross-reference" information needed to make (a) fast. We simply associate with each node in a LEGAL tree a list of pointers to the flowchart nodes which are "suspended with"
the corresponding protocol. The definition of "suspended with" shows that the lists an instruction belongs on are determined only by the sets $H_k$ at the instruction. Since these are known when the move-instruction is added, we can easily insert the new MOVE on the proper lists at the same time.

In such a concrete implementation, the actions involved in adding a move-instruction would be translated as something like:

1. Label $c$ with the instruction $\text{MOVE}_k(A)$.

2. For each $\pi$ in $H_k(c)$, add a son labelled $A$ to the LEGAL-tree node for $\pi$ if it doesn't have one. Add a pointer to $c$ to the son's instruction list.

3. For each $\pi$ in $H_k(c)$, for all instructions on the list belonging to $\pi$'s LEGAL-tree node, add an $A$-exit to the instruction.

4. For each $\pi$ in $H_k(c)$, find the son labelled $A$ of $\pi$'s LEGAL-tree node; for each of the son's sons, if any, add to $c$ an exit with label corresponding to the label on the grandson.

Since (2), (3), and (4) iterate over the same set one could no doubt merge them into a single iteration. This approach seems certain to yield about as cheap an implementation of the general algorithm's bookkeeping as one could expect.
5. PRACTICAL CONSTRUCTIONS

In this chapter we explore the problem of finding practical implementation methods for attribute grammars. Although we present a number of specific constructions, we are not so much presenting answers as we are a research tool. The general nondeterministic algorithm given in the preceding chapter provides a least common denominator for comparing methods of evaluator construction. Practical constructions may be viewed as extensions of this algorithm to include criteria for directing the computation. By adding strategy it is hoped to make the construction "more deterministic", that is, to reduce the space of possible computations to such an extent that only "good" evaluators are produced; we want to make the outcome less dependent on fortuitous nondeterministic guesses and more the result of effective decision-making. Viewing different construction techniques as modifications of the general construction makes their relationships clearer: the essential strategic elements of each are factored out and made explicit within a common framework. We are indebted to Hayes [31] for suggesting the importance of separating control information from information describing permitted changes of state.

The main problem in constructing coroutine evaluators is deciding on how the semantic trees will be traversed, which is determined by the pattern of move-instructions in the flowcharts. When nothing more can be done at a node, a good evaluator will pass control along directly to the next place where evaluation should be done, while a poor one will wander aimlessly around in the tree before reaching a useful place. How is an evaluator construction to know which move-instructions to add and where? The general algorithm avoids the issue by simply guessing, with some
guesses leading to better evaluators than others. In this chapter we develop some more insightful ways of making these choices. We consider classes of local strategies which guide the decision of whether to add a move-instruction via purely tactical considerations, the situation at the site of the proposed addition. The limitations of such short-sighted criteria lead us to contemplate global strategies in which some overall planning of the evaluator's structure is done before construction begins. Unfortunately at present we know very little about such approaches.

**Local strategies for evaluator construction.**

To produce a compact, efficient evaluator for an ag it is desirable to avoid adding move-instructions which cannot take control to a point where some evaluations can be done. In this section we look at strategies based on predicting, at a given point in a flowchart, in which regions of the tree it might be possible to proceed with evaluation. By requiring that control be passed in such a direction we may avoid the addition of many fruitless move-instructions. We derive two classes of local strategies, one of universal applicability and another which may produce somewhat better evaluators at the expense of failing entirely on some ags. A particular member of these classes is obtained by specifying a source of predictive information. We will present a heirarchy of four such sources, ranging from very simple to the best possible such information; the corresponding evaluator-construction heirarchies probably offer a wide enough selection for practical purposes. However, since these local strategies only accept or reject individual proposed move-instructions, there is no overall direction to the construction of an evaluator.
The Terminating General Construction of chapter 4 builds evaluators in which passing control to a subtree always results in evaluating as much of that subtree as possible. This suggests we think of the subtrees, i.e. neighbors $k$ for $1 \leq k \leq n_p$, as subroutines to be called to make more attributes of the sons available. It would be helpful if we had some way of predicting in advance how many new attributes a proposed "call" would actually yield. Suppose we have a semantic tree and we are at a node where production $p$ applies; consider neighbor $k$ for $k \neq 0$. There are chains of dependencies running through region $k$ of the tree which begin at inherited $(\text{SENT}_p)$ attributes of the neighbor and end at synthesized $(\text{RECEIVED}_p)$ attributes of the neighbor. These chains impose certain relationships between the set of currently available attributes and the set that might be made available by "calling" neighbor $k$. No output can be produced by a visit to the neighbor unless all the inputs it depends on, however indirectly, have been made available before the visit. Unfortunately, the chains between inputs of $X_k$ and outputs of $X_k$ may depend on the details of the tree structure in region $k$. When constructing an evaluator, we know only the structure that will exist in the immediate vicinity of the coroutine: namely the production $p$ which applies at its node, and hence the symbols $X_1 \ldots X_n$ labelling the roots of the subtrees. We need some way of predicting what dependency chains may run through region $k$ from only the symbol $X_k$.

Imagine that we have for each nonterminal $X$ a directed graph $G_X$ whose nodes are the elements of $A(X)$. We can interpret these graphs as characterizing the dependency chains running through any semantic tree regions below a type-$X$ node. That is, an arc from $a$ to $a'$ in $G_X$ would signify the prediction that a region below a type-$X$ node might have
a dependency chain running from X.a to X.a'. We could use such a set of
graphs as heuristic information to guide TGC's otherwise nondeterministic
choice of a move-instruction to be added at a given point.

**Definition:** A set of **heuristic graphs** (for a given ag) is a set \( \{ G_x | x \in V_N \} \)
of directed graphs such that the graph \( G_x \) has nodes \( A(X) \) and
arcs only from \( I(X) \) to \( S(X) \).

**Definition:** Given a set \( \{ G_x \} \) of heuristic graphs, the **yield** of a \( MOVE_k \)
instruction \((k \neq 0)\) at a node \( c \) in \( G_p \) is the set
\[
\{ \text{ac}S(p[k]) \mid (a,k) \notin AVAIL(c) \text{ but for all } i \text{ with an arc from } i \text{ to } a \text{ in } G_p[k], (i,k) \in AVAIL(c) \}.
\]

In words, an attribute of \( X_k \) is in the yield of a \( MOVE_k \) if the heuristic
graph for \( X_k \) indicates no dependency chains from uncomputed inputs of \( X_k \)
to the attribute in question. If the yield of a \( MOVE_k \) instruction is
empty, the heuristic graph suggests that a visit to neighbor \( k \) will not
result in any new attributes being evaluated. This is the kind of move-
instruction we should avoid.

We can now define a family of evaluator constructions based on
this idea. Let \( \Gamma \) be any rule for obtaining from an ag a set of heuristic
graphs for it. Then we make the

**Definition:** The **\( \Gamma \)-heuristic general construction** ( \( \Gamma \)-HGC ) is the same
as the TGC of chapter 4, but with the restriction that every step
taken satisfy the following (where "yield" is with respect to the
set of heuristic graphs obtained by \( \Gamma \)):  

81
(*) No move-instruction with an empty yield is added at $c$ unless every move-instruction has an empty yield at $c$.

The additional restriction (*) affects the operation of TGC only by constraining an otherwise free choice of which move-instruction to add when TGC must add one. If the heuristic graphs are good predictors of dependency chains, then this constraint will improve the resulting evaluators by avoiding some fruitless moves. On the other hand poor predictions cannot do much damage compared to nondeterministic guesses. Note that the extra constraint only applies when some $G_{y_k}^X$ indicates a possibly fruitful move-instruction; when all yields are empty, the computation proceeds as in TGC -- before doing a MOVE, each son with new inputs is tried once to see if some new outputs turn up despite the pessimistic predictions. It's easy to see that (*) cannot interfere with the termination or correctness of the underlying TGC.

**Theorem:** For any rule $\Gamma$, the algorithm $\Gamma$-HGC terminates after a finite number of steps for any given $ag$, and succeeds whenever the $ag$ is noncircular.

**Proof:** Every computation of $\Gamma$-HGC is also a computation of TGC. Thus $\Gamma$-HGC halts for all input; computations of $\Gamma$-HGC cannot diverge since computations of TGC do not. Restriction (*) cannot cause failure, for either there is a nonempty yield and (*) allows the corresponding MOVE to be added or else all yields are empty and (*) does not apply. Thus $\Gamma$-HGC fails only if TGC does, and so $\Gamma$-HGC succeeds for any noncircular $ag$ as input.

[]
In the construction just given, the heuristic graphs were used only to bias the construction in favor of "good" evaluators and not in any really essential way. Therefore any set of graphs will work at least to the extent of producing correct evaluators for all noncircular args; we can judge the quality of the heuristic rule \( r \) by the improvement it makes in the resulting evaluators. The ability to use any set of heuristic graphs at all is due to the fact that HGC doesn't take them as the absolute truth: even if every yield is empty, visits are still made to sons with new inputs.

This has its unfortunate aspect. If the heuristic graphs are usually right about the futility of move-instructions with empty yields, then in the resulting evaluators every \( \text{MOVE}_0 \) instruction will tend to be preceded by a sequence of fruitless \( \text{MOVE}_k \)'s. We can build somewhat better evaluators by relying on the heuristic graphs and refusing to add a move-instruction with an empty yield at all. However, the price we pay for the improvement is that success of the construction now depends in a way on the accuracy of the \( G_x \)'s: if they are too pessimistic -- contain too many false indications of dependency chains -- the construction may refuse to add a necessary but apparently futile move-instruction and so fail. If the heuristic rule is good enough for practical purposes this trade-off may be profitable.

**Definition:** The \( \Gamma \)-heuristic **limited construction** ( \( \Gamma \)-HLC) is the same as the TGC of chapter 4, but with the restriction that every step taken satisfy:

\[
(**) \text{ A move-instruction is added only if its yield is nonempty.}
\]
The construction family HLC are "limited" as opposed to general in the sense that they do not always succeed for a noncircular ag. The restriction (**) can sometimes cause failure when TGC would have succeeded. Each rule defines a class of ags which can be implemented by -HLC. We can formalise this as follows.

**Definition:** Suppose that $DG_p$ is the dependency graph for production $p$ and that for $1 \leq k \leq n_p$ we have a graph $G_k$ whose nodes are the elements of $A(p[k])$. Then the graph

$$DG_p[G_1, \ldots, G_n]$$

is the graph obtained from $DG_p$ by adding an arc from $(a,k)$ to $(a',k)$ whenever the graph $G_k$ has an arc from $a$ to $a'$.

**Definition:** An ag is $\Gamma$-adequate if no graph $DG_p[G_1^x, G_2^x, \ldots, G_n^x]$ has a cycle for any $p \in P$, where $\{G_x\}$ are the set of heuristic graphs obtained by $\Gamma$ for the ag.

**Theorem:** For any $\Gamma$, the algorithm $\Gamma$-HLC terminates after a finite number of steps for any given ag, and succeeds iff the ag is $\Gamma$-adequate.

**Proof:** Since any computation of $\Gamma$-HLC is also a computation of TGC, the termination of $\Gamma$-HLC is obvious. To prove the second part we show instead that $\Gamma$-HLC fails iff the ag is not $\Gamma$-adequate. First suppose that a computation of $\Gamma$-HLC fails. Then there is a pending node in some flowchart at which not all attributes are available but at which nevertheless no instruction can be added. Trace a path backwards in $DG_p[G_1^x, \ldots, G_n^x]$, where $G_p$ is the incomplete flowchart, as follows. Start with any unavail-
able attribute occurrence. If the current attribute is in RECEIVED\(_p\)(p[k]) there must be an arc in \(G_{p[k]}\) to it from another unavailable attribute, otherwise it would have been in the yield of a MOVE\(_k\), which therefore could have been added at the pending node; follow the corresponding arc in \(DG_p[...]\). On the other hand, if the current attribute is SENT by \(p\), then there must be an unavailable attribute in its dependency set or else its evaluate-instruction could have been added; follow the corresponding arc backwards in \(DG_p[...]\). This trace backward must extend indefinitely, but there are only a finite number of attribute occurrences of \(p\). Hence some attribute must be encountered twice on the path backward, and so \(DG_p[...]\) contains a cycle; the input ag could not have been \(\Gamma\)-adequate.

Now suppose that the algorithm is run on an input ag that is not \(\Gamma\)-adequate. Then there is a \(p \in P\) such that \(DG_p[...]\) contains a cycle. Since any arc from \((a',k')\) to \((a,k)\) in \(DG_p[...]\) implies that in a flowchart the instruction making \((a',k')\) available must precede the instruction making \((a,k)\) available, the existence of this cycle implies that the corresponding flowchart would have to contain an instruction which preceded itself. Our flowcharts are acyclic, so this is obviously impossible. (Even with loops, it would be impossible, but not so obviously.) Hence the algorithm must fail.

\[\square\]

**Proposition:** If \(\Gamma_1\) and \(\Gamma_2\) are two heuristic rules such that for any ag, for any nonterminal \(X\) of the ag, \(G^1_X \subset G^2_X\), then any \(\Gamma_1\)-adequate ag is also \(\Gamma_2\)-adequate.

**Proof:** Obvious.

\[\square\]
Each $\Gamma$ defines a class of ags, the $\Gamma$-adequate ones, and the Proposition tells us that a $\Gamma$ giving smaller heuristic graphs is "better" in the sense of defining a larger class of ags. We can therefore obtain a heirarchy of ag classes with an evaluator construction corresponding to each class. This is a precise formulation of Jazayeri's informal conjecture [16] that there is a heirarchy of circularity tests each corresponding to an evaluation method. As many ag classes may be defined as one can write down $\Gamma$-rules. At one extreme we can take $\Gamma_0$, assigning each ag a set of heuristic graphs with no arcs at all; the class of $\Gamma_0$-adequate ags includes all noncircular ags, since $\Gamma_0$-HLC is really just TGC. At the other extreme, we can take $\Gamma_\infty$, the rule assigning to each ag the set of complete heuristic graphs: each $G_X$ has an arc from every element of $I(X)$ to every element of $S(X)$. The class of $\Gamma_\infty$-adequate ags includes just those ags which can be evaluated by making a single visit to each node of the semantic tree; i.e. the "one-pass" ags where we allow any kind of depth-first pass, not just left-to-right. To see this, note that the yield of a $\text{MOVE}_k$ instruction is empty unless all of $X_k$'s inputs are available; once they are, a $\text{MOVE}_k$ can be added, but after that there are no more inputs to evaluate and so no more $\text{MOVE}_k$'s can follow.

**Obtaining sets of heuristic graphs.**

While it is true that any set of graphs will do as heuristic graphs for HGC, it is also true that the quality of produced evaluators is only improved to the extent that each $G_X$ is an accurate characterization of the possible dependency chains running through a region below a type-X node. If many chains occur which are not represented in the $G_X$'s, then
many futile move-instructions may be generated. On the other hand, if many extraneous arcs appear in the \( G_X \)'s which never represent real chains in trees, then many times all yields will be empty even though one or more move-instructions might be fruitful, and HGC will fall back on trial and error too often (conversely, HLC will fail too often). Therefore the ideal \( \Gamma \) would be one assigning to each grammar the perfect heuristic graphs \( \{ G_X \} \) in which there is an arc from \( i \) to \( s \) in \( G_X \) if and only if there is a semantic tree with a type-\( X \) node such that there is a dependency chain from \( X.i \) to \( X.s \).

We shall see in a moment how to obtain these perfect heuristic graphs. Unfortunately, the computation required is expensive and potentially exponentially time-consuming. It seems worthwhile, then, to capitalize on the flexibility of the HGC and HLC families by finding other cheaper rules for getting somewhat less than ideal heuristic graphs. In this section we present two compromises as well as the ideal solution. We then prove that the three \( \Gamma \)-rules generate a hierarchy of ag classes in accordance with the previous Proposition.

Our development of the ideal heuristic rule is based directly on Knuth's circularity test [11]. We can find for each nonterminal a set containing one graph for each different combination of dependency chains that can actually be observed below the nonterminal in some tree. We maintain for each \( X \) a set \( DG_X \) of graphs, each graph having \( A(X) \) as its node set. Initially, let \( DG_X \) be empty for all \( X \). Then repeat until no graph can be added to any \( DG_X \): for every production \( p \) and graphs \( D_k \) for \( 1 \leq k \leq n_p \), form the graph \( D' \) which has node set \( A(p[0]) \) and an arc from \( i \) to \( s \) whenever the graph \( DG_p[D_1, D_2, \ldots, D_n_p] \) has a path from \( (i,0) \) to \( (s,0) \); if \( D' \) is not in \( D_p[0] \) then insert it and repeat. It can be proved
that this algorithm correctly enumerates in each \( DG_X \) all the possible
graphs \( D_X \) such that some region below a type-X node in some tree has
precisely the dependency chains represented by \( D_X \). See [11] for details.

**Definition:** The heuristic rule \( \Gamma_1 \) assigns to any \( a \) the set \( \{ G_X | X \in V_N \} \)
of heuristic graphs where each \( G_X \) has an arc from \( i \) to \( s \) iff
there is a graph \( D_X \) in \( DG_X \) which has an arc from \( i \) to \( s \).

Our first "compromise" rule is adapted more or less directly from
Knuth's erroneous first attempt at a circularity test [11]. We maintain
for each \( X \) a single directed graph \( IO_X \) whose nodes are the elements of
\( A(X) \). Initially let each \( IO_X \) consist of its nodes and no arcs. Then
repeat until no further arcs can be added to any \( IO_X \): for each production
\( p \), if \( DG_p [ IO_p[1], IO_p[2], \ldots IO_p[n_p] ] \) has a path from \( (i,0) \) to \( (s,0) \) but
\( IO_p[0] \) has no arc from \( i \) to \( s \), add the missing arc and repeat. Upon ter-
mination \( IO_X \) will contain an arc representing each possible dependency
chain under a type-X node, as well as some extraneous arcs in many cases.
These are the graphs used in [25] to guide the construction of "treewalk
evaluators".

**Definition:** The heuristic rule \( \Gamma_2 \) assigns to any \( a \) the set \( \{ G_X | X \in V_N \} \)
of heuristic graphs where each \( G_X = IO_X \).

As a second example of compromise rules we present a simple neces-
sary-but-not-sufficient circularity test adapted from Jazayeri [16].
Define the relation \( R \) by \( (X,a) R (Y,a') \) iff there is a \( p \in P \) and numbers
k and k' with 0 ≤ k, k' ≤ n_p such that p[k] = X, p[k'] = Y, and
(a, k) ∈ D_p^p(a', k'). The relation R simply includes all arcs present in any
DG_p without remembering which arc goes in which place. Let R^* denote
the transitive closure of R. Then we can get a set of heuristic graphs
from R^* by restricting it to a single nonterminal's attributes, once for
each nonterminal.

**Definition:** The heuristic rule \( \Gamma_3 \) assigns to any ag the set \( \{G_X | X \in V_N \} \)
of heuristic graphs where each \( G_X \) has an arc from i to s iff
\((X, i) \in R^* (X, s)\).

**Theorem:** Given any ag, let \( \{G^1_X \}, \{G^2_X \}, \) and \( \{G^3_X \} \) be the sets of heuristic
graphs obtained by rules \( \Gamma_1, \Gamma_2, \) and \( \Gamma_3 \) respectively. Then for
any \( X, \ G^1_X \subseteq G^2_X \subseteq G^3_X \). Moreover, there exist ags for which this
containment is proper.

**Proof:** To start, we show that the \( IO_X \) graphs contain an arc corresponding
to each dependency chain through a subtree below a type-X node, by induction
on the height of the subtree. If the height is 1, any chain consists
of only a single arc reflected in a \( DG_p \), and it is clear that such an arc
will be added to \( IO_X \). If the hypothesis is true for heights up to \( N \) and
we consider a chain in a subtree of height \( N+1 \), we note that the path
consists of segments alternately made up of arcs in a \( DG_p \) and paths
through subtrees of height no more than \( N \). By hypothesis every such sub-
path has a corresponding arc in the \( IO_{p[k]} \) graph. Hence at some point
in the computation of the \( IO \)'s, \( DG_p[IO_{p[1]} \ldots IO_{p[n_p]}] \) will contain a
path corresponding to the original path we were considering and so the
arc in question will be added to \( IO_X \); thus the hypothesis holds for all \( N \).
Now pick an X and consider any arc in $G_1^X$. By definition of $\Gamma_1$, there is an actual subtree with root X exhibiting a dependency chain corresponding to the considered arc. By the induction just given, then, the considered arc also appears in $G_2^X$. Hence $G_1^X \subseteq G_2^X$.

It is easy to see that any $IO^X$ contains only arcs $(i,s)$ such that $(X,i) \rightarrow (X,s)$. The proof is by induction on the number of steps in the computation of the $IO^X$'s. Initially the hypothesis holds vacuously since all the $IO$'s are empty. An arc is added to $IO^X$ when there is a path

$$(i,0) \rightarrow (a_1, k_1) \rightarrow (a_2, k_2) \rightarrow \ldots \rightarrow (s,0)$$

through a $DG_p[\ldots]$. Each arc $(a_j, k_j) \rightarrow (a_{j+1}, k_{j+1})$ is either an arc of $DG_p$ and so $(X_{k_j}, a_j) \rightarrow (X_{k_{j+1}}, a_{j+1})$ holds, or else is an arc already in an $IO_{X_{k_j}}$, and thus $k_j = k_{j+1} = k'$ and $(X_{k'}, a_j) \rightarrow (X_{k'}, a_{j+1})$ holds by hypothesis. By the definition of transitive closure then $(X,i) \rightarrow (X,s)$, and we conclude that the hypothesis continues to hold after each addition of an arc to an $IO$. Therefore any arc in $G_2^X = IO^X$ is also in $G_3^X$, and $G_2^X \subseteq G_3^X$.

We have thus shown the first part of the theorem. To show that the containments may be proper, we present a small example as.

0: \begin{align*}
S & \rightarrow A \\
A & . a + A . x
\end{align*}

1: \begin{align*}
A_0 & \rightarrow A_1 \ a \\
A_1 & . a + A_0 . a, \ A_1 . b + A_1 . y, \ A_0 . x + A_1 . x, \ A_0 . y + 2
\end{align*}

2: \begin{align*}
A & \rightarrow b \\
A & . y + A . a, \ A . x + 2
\end{align*}

3: \begin{align*}
A & \rightarrow bb \\
A & . x + A . b, \ A . y + 2
\end{align*}

Simple computations will reveal that the arc sets of the graphs are
\[ C^1_A = \{(a,y), (b,x)\} \]
\[ C^2_A = \{(a,x), (a,y), (b,x)\} \]
\[ C^3_A = \{(a,x), (a,y), (b,x), (b,y)\} \]

We remark in addition that the phenomena causing the proper containments in this tiny example are quite typical of larger ags.

To sum up this section, we have studied the idea of improving the quality of constructed evaluators by employing knowledge of possible dependencies in semantic trees to guide the otherwise free choice of what direction to move in when no further evaluation at a tree node is possible. We have defined two families of constructions, one completely general and one giving rise to various classes of ags which can be handled by specific strategies. These families are described independently of any particular sources of knowledge about dependencies; any suitable source may be immediately employed. Such a unified view would not have been found without the coroutine model and the general nondeterministic construction to serve as a coordinate system. To clarify these points we presented three practically useful sources of dependency knowledge ranging from perfect knowledge at potentially exponential cost to a rather crude approximation obtainable for the cost of computing the transitive closure of a boolean matrix. Other sources of knowledge are possible, including the author of the attribute grammar.

We have called these local strategies because they affected the operation of the evaluator construction only tactically, influencing each
new decision independently. The "meaning" of an evaluator in the Floyd-
assertion sense [29] is captured by the sets AVAIL(c), $H_k(c)$, and LEGAL(X)
for all appropriate $c$, $k$, and $X$. Of these, only the AVAIL sets are purely
local, reflecting only history-less information about a single point in
a flowchart. The $H_k$ sets record the history of interaction with neigh-
bors, while the LEGAL sets describe overall patterns of interaction
between coroutine pairs; both of these kinds of information may fairly
be regarded as global. The strategies examined so far have made use of
only the AVAIL sets.

**Global strategies for evaluator construction.**

The main fault of the constructions so far considered is the
absence of any overall planning during construction of an evaluator. The
flowcharts grow up little by little in a decentralized way, and as a re-
sult there may be an undesirable amount of diversity in the end product.
The number of protocols employed for a given nonterminal is a measure of
how many different ways that type of tree node may be processed; it is
also a measure of the time needed to build the evaluator and of the space
needed to hold it after it is built. Sometimes the dependencies of an
tag demand that nodes of the same type but in different contexts be pro-
cessed differently, but in many cases it is possible for the treatment
of a node type to be independent of context. If it is possible to con-
struct an evaluator with only one protocol for some nonterminal then most
likely any evaluator that uses several is unnecessarily large. Unfortu-
nately this is just what tends to happen when only local strategies are
employed. Each growing bit of flowchart attacks its neighbors' processing
in whatever convenient or arbitrary way it happens to try first, and as
all these protocols accumulate in the LEGAL sets it becomes necessary to add more and more exits in order to be prepared to do the same thing many different ways.

The idea of global strategy is to make some decisions in advance about the shape of the evaluator, the ways in which coroutines will work and interact. We then attempt to build flowcharts in conformity with the plan. The potential payoffs are reduced redundancy in the flowcharts and less work during the construction. For instance, if we pick a few protocols in advance for each nonterminal (see chapter 6) we have substantially reduced the problem "build an evaluator" into a number of smaller problems like "build flowchart $C_p$ to given specifications". The expediting effect of global planning is well-known to artificial intelligence workers.

There may be other reasons for doing global strategy than just to reduce redundancy. A certain pattern in the flowcharts may be desirable in itself, in which case we wish to know how to induce it in the evaluators we build. For example, if the move-instructions are arranged in just a certain way, the motion through semantic trees will take the form of pre-order traversals. It would then be possible to implement the motion by means of threaded links, allowing the elimination of a stack. Again, it might be possible in some cases to arrange the evaluate-instructions in flowcharts so that a stack-oriented storage management scheme could replace the attribute fields of semantic tree nodes.

The task of devising suitable plans seems combinatoric in nature and amenable to little besides trial and error. When we introduced local strategy it was not too hard to analyse the effect of such a strategy on the progress of the construction, probably because of the independence
of each application of the strategy from all other applications. When trying to choose a global constraint such as a protocol to be used for a nonterminal, it is necessary to consider the interactions of widely scattered choices; it is not clear how to tell if a given protocol can be obeyed everywhere without actually attempting to construct an evaluator using it. We candidly admit to having made little progress in the use of global strategies. However, in the next chapter we present a useful step in that direction which should also be of practical interest.
6. UNIFORM EVALUATION OF ATTRIBUTE GRAMMARS

For those attribute grammars which are "nice" in a sense to be described, we can use a simplified construction to build evaluators with correspondingly simple forms. This class of ags is of practical interest because its membership test is easily understood and natural for the author of a translation to meet; indeed, we claim it imposes on him a valuable discipline analogous to that imposed by "GOTO-less" programming. The size of evaluators produced this way cannot grow more than linearly in the size of the ag, and their simple structure makes more tractable the study of their transformation and optimization. These advantages together with the ease of implementing the construction itself lead us to believe that these "uniform" ags are the largest class of ags with practical significance.

Intuitively, branching arises in a flowchart because of the necessity to process a node in different ways depending on its substructure and its context in the semantic tree. The possibility of branching requires the construction algorithm to iteratively discover all the ways that control might be passed back and forth between coroutines. The construction of each flowchart is inextricably meshed with the construction of the rest by the fact that adding a move-instruction to one of them may demand the addition of exits from move-instructions in others. On the other hand, if an ag is such that any node of a given type may be processed in the same way regardless of context, then branching will be unnecessary in its flowcharts. Such an ag is called uniform. Since we can know the evaluation strategy for each type of node in a uniform grammar before the construction is begun, it is possible to build each flowchart indep-
endently of the others using a simplified algorithm.

By a "way of processing" a type-X node we mean a protocol for X. A protocol describes the order in which groups of attributes are made available at a node. In this chapter we insist that in any protocol \( \pi = A_0A_1 \ldots A_{2m-1} \), the sets \( A_i \) and \( A_j \) must be disjoint when \( i \neq j \). If a node is evaluated according to \( \pi \), there will be \( m \) visits to the node from above; \( A_{2j} \) is the set of input attributes newly available for use on the \( j \)-th visit, and \( A_{2j+1} \) is the set of output attributes produced by the activity during the \( j \)-th visit. An ag is uniform if there is a coroutine evaluator which makes use of only one protocol for each nonterminal X.

The number of visits to a node of type X, the inputs available to and the outputs produced by each visit are fixed by the protocol and will be the same for any occurrence of a type-X node in any semantic tree. This is what prompts the term "uniform". Note, though, that the protocols for different nonterminals may be of different lengths and that different numbers of visits to a node's sons may be nested within a single visit to the node, so that the concept of uniform evaluation is much more general than the "evaluation in passes" of [12] and [16].

We feel that uniformity is a natural condition for the ag-writer to meet, in fact one that is worth encouraging in its own right and not just as an implementation restriction. The general problem of understanding the interplay of unrestricted dependencies in an ag with many attributes and many productions is probably too hard to be manageable; as an indication of the subtlety of these matters we mention the numerous attempts by Knuth to formulate a circularity test, of which one erroneous one and subsequently a correct one were published [11]. Central to the modern theme of "structured" this or that is the freedom to treat a com-
pound entity as atomic at a higher level of abstraction, without the need to consider details of its structure. This requires a simple external characterization of the compound. The locally-defined semantics of ags provide such freedom to a large extent, but the writer must still take into account the dependency structures of lower and higher productions when writing semantic rules. Characterizing these dependencies in a simple way by dividing a symbol's attributes into groups of successive inputs and outputs — into a protocol — is a natural and helpful tool in coping with this. For example, in the definition of a block-structured language one would naturally conceive of a <statement> as first sending its declarations upward to the enclosing <block>, then receiving its symbol table from above, and finally sending its machine-code translation back up.

We suggest that the writer be required to specify the protocol for each nonterminal, since he has to enumerate the attributes anyway and ought to be conscious of the order in which they will be computed. Note that this requirement in no way imposes upon the writer's freedom to ignore the order in which parsing is performed. Obtaining this extra information from the author makes the construction process very easy; as we shall see, we can quickly check the protocols for satisfiability and build the evaluator as we go. Moreover, the fact that the semantics as written do not agree with the author's declared intentions — the case when given protocols can not be used to build an evaluator — is probably an indication of a mistake, and the ability to issue a diagnostic will no doubt be appreciated.
Uniform evaluation with given protocols.

For the moment we assume that we are given an ag complete with a protocol specified for each nonterminal and wish to construct an evaluator.

**Definition:** A protocol for $X$, $X \in V_N$, is a sequence $A_0 A_1 \ldots A_{2m-1}$ where

(a) $A_i \in I(X)$ if $i$ is even, and $A_i \in S(X)$ if $i$ is odd;

(b) if $i \neq j$ then $A_i \cap A_j = \emptyset$;

(c) $A_0 \cup A_1 \cup \ldots A_{2m-1} = A(X)$.

**Definition:** A coroutine evaluator is **uniform** iff no move-instruction has more than one exit (equivalently, iff each $\text{LEGAL}(X)$ for the evaluator consists only of a single protocol and its prefixes).

**Definition:** A set $\{\pi_X | X \in V_N\}$ of protocols is **satisfiable** iff there is a coroutine evaluator (for the given ag) such that for all $X$, $\text{LEGAL}(X) = \{\pi | \pi \text{ is a prefix of } \pi_X\}$.

**Definition:** An ag is **uniform** iff there is a satisfiable set of protocols for it.

Suppose that along with an ag we are given a set $\{\pi_X\}$ of protocols. We can test the set for satisfiability and build the corresponding evaluator, if it exists, at the same time by using a modified version of the general nondeterministic algorithm. The modified version admits the possibility of failure. If it halts successfully it yields a uniform evaluator with the given set of protocols; if it halts unsuccessfully, the
given set of protocols is unsatisfiable; furthermore it halts for any input ag. Although the modified algorithm is also nondeterministic, this nondeterminism is of no consequence: for given inputs either all computations succeed or all computations fail, and every successful computation yields the same evaluator except possibly for different orderings of actions which could be done collateral. Thus the particular computation carried out is irrelevant, and the algorithm may be made deterministic without penalty by introducing an arbitrary rule for choosing among the permitted next steps.

Our method is to modify the general algorithm so that it operates in a goal-oriented manner using the set of protocols as a guide. At any point in a developing flowchart (remember that flowcharts are linear now) a certain prefix of the protocol for each \( X_k \) has already been obeyed. Just after exiting the initial node of a flowchart, for example, the first element of \( X_0 \)'s protocol has been received and none of the other neighbors' protocols have begun yet. We can associate with each point in the flowchart a "goal" for each neighbor \( k \), namely a set \( A \subset A(X_k) \) which is to be the next parameter transmitted to the \( k \)-th neighbor. This set is the element of \( X_k \)'s protocol which immediately follows the portion already obeyed. For any instruction \( c \) and any \( k \), define

\[
\text{GOAL}_k(c) = A_j, \quad \text{where} \quad \pi_{X_k} = A_0 A_1 \ldots A_{2m-1} \quad \text{and} \quad H_k(c) = A_0 A_1 \ldots A_{j-1}
\]

We should note that \( \text{GOAL}_k(c) \) is undefined when \( H_k(c) = \pi_{X_k} \), but we won't refer to such \( \text{GOALs} \). We can now state our modification of the general construction:
**Definition:** The uniform evaluator construction accepts an ag and a set \( \{ \pi_X \mid X \in \mathcal{V}_N \} \) of protocols. Its computations are the same as those of the general algorithm of chapter 4, but with the additional restriction that every step taken satisfy

\[ (*) \text{ A } \text{MOVE}_k(A) \text{ instruction is added at } c \text{ only if } A = \text{GOAL}_k(c). \]

The uniform evaluator construction **succeeds** if after some number of steps the set of flowcharts satisfy the completion-condition, and **fails** if the completion-condition is not satisfied but no further step is permitted.

**Theorem:** The uniform evaluator construction terminates after a finite number of steps for any input ag and set of protocols. If it succeeds it has constructed a uniform evaluator with the given protocols; if it fails the given set of protocols is unsatisfiable.

**Proof:** Termination is clear since the length of any path through a \( C_p \) is bounded a priori: only a finite number of evaluate-instructions may appear on the path since there are only a finite number of attribute occurrences and no \( f^p(a,k) \) can occur twice on a path; and only a finite number of MOVE\(_k\) instructions, for each \( k \), may occur on a path because each successive one must transmit the next element of the finite \( \pi_X \).

We now show that success implies that the desired uniform evaluator has been constructed. Initially each LEGAL(X) contains only prefixes of \( \pi_X \), since each is initialized to \( \{ A \} \). Adding evaluate-instructions has no effect on this assertion. When a move-instruction is added, the update \( \text{LEGAL}(X) := \text{LEGAL}(X) \cup H_k(c)A \) is performed, but since \( A = \text{GOAL}_k(c) \) this does not falsify the assertion. We conclude by induction on the
number of updates that at termination each LEGAL(X) contains only prefixes of \( \pi_X \), and so the resulting evaluator is uniform and has the given set of protocols.

Finally we must show that if the construction fails the given set of protocols is unsatisfiable. Failure occurs when there is a pending node \( c \) at which not all attributes are available but at which nevertheless no instruction may be added. Since no evaluate-instruction may be added, there must be no attribute which is ready to evaluate. Since no move-instruction may be added, there must be for each \( k \) an attribute in \( \text{GOAL}_k \) which is not available. We may trace backwards from any unavailable attribute as follows: the current attribute is not available because the instruction that would make it available needs some other attribute which is also not available; the instruction which would make that attribute available cannot be added because it needs a third attribute which is not available; and so on. The chain must extend indefinitely, for if it ended, the last-reached instruction could be added. On the other hand there are only a finite number of attributes and so one must be encountered twice; hence there is a cycle of dependencies. Any instruction needing an attribute must be preceded by the instruction which makes that attribute available; the presence of the cycle implies that in a uniform evaluator with the given protocols some instruction must precede itself! This is impossible in our acyclic flowcharts, and so the desired uniform evaluator does not exist.

\[ \square \]

The uniform evaluator construction may be simplified considerably since certain complications of the general nondeterministic algorithm are
made unnecessary by the restriction (*). The preservation of the closure-condition can be made much more straightforward. In the general case, when a move-instruction is added it is necessary to find all the other move-instructions to which it might pass control and make sure that they have the necessary exit. This is required because we don't know in advance what protocols will be used. The restriction (*) ensures that the given protocols are the only ones used, and so we do not have to carry out the search-and-update process. Instead whenever we add a move-instruction we can go on to give it the single exit it will need. We can state this easier rule for adding move-instructions and prove that it gives the same result as the general rule, under the restriction (*).

With the new rule, each flowchart can be built independently of the others.

Advantages of straight-line flowcharts.

Aside from the human-engineering benefit from the discipline of uniform ags, there are several reasons to prefer uniform evaluators. First is the fact that the size of uniform evaluators can grow only linearly in the size of the ag, because there is only one path per flowchart and the number of instructions on a path is bounded a priori by a number depending only on the ag. Then there is the fact that the absence of branches makes the evaluators easier to analyse. We might, for instance, propose to save storage during evaluation by retaining only the attributes which will be needed later, rather than keeping them all in a semantic tree. It would be nice if storage could be administrated by some simple scheme such as a stack. Since the flowcharts are linear, we might be
able to bracket the segments corresponding to lifetimes of attributes with GET and FREE operations as opening and closing "parentheses" in such a way that the operations occur in well-nested fashion during evaluations. Such an analysis would seem to be much more difficult in general acyclic flowcharts.

Perhaps the most interesting advantage of uniform evaluator structure is that the entire coroutine linkage mechanism can be eliminated! Since each flowchart is linear, any given move-instruction can transfer control to only one other move-instruction per flowchart -- i.e. per production. We could (almost) replace the MOVE operation by a simple jump to the unique resumed instruction in the other flowchart. "Almost" because it would actually be necessary to examine the new current tree node to see which production applies, then jump to the unique instruction in that production's flowchart. Parameter transmission and even the local program counters can be eliminated, leaving only a finite control moving a cursor around in the tree.
7. SUMMARY AND CONCLUDING REMARKS

We are interested in the automatic implementation of attribute grammars (ags) because of their suitability for specifying the semantic actions of compilers and other translators. The mapping from source language to target language may be described by an ag in a natural, modular, and nonprocedural way, just as syntactic analysis may be described by a context-free grammar. To apply ags in practical situations it is necessary to solve the problem of converting nonprocedural semantic descriptions into efficient procedures for performing the translations. We argue that at present it is more reasonable to look for flexible approaches to the problem than for pat solutions, since the body of experience required to assess particular solutions has not yet been acquired.

We have presented both a framework for studying ag implementation and a number of significant results obtained through it. We introduced a model of ag implementation based on distributed control. In this model an evaluator consists of an acyclic flowchart for each production in the grammar, instances of which are executed as coroutines to perform a translation. We developed a methodology for verifying the correctness of these coroutine evaluators. From it we derived a very general nondeterministic algorithm for building such evaluators; the key assertions in a correctness proof became bookkeeping variables in the algorithm. We presented several practical evaluator constructions obtained from the general algorithm by adding various choice criteria; these constructions yield efficient sequential implementations, and some of them will handle any ag at all. The deficiencies of these constructions motivate a natural restriction defining the class of uniform ags. As well as admitting especially simple imple-
mentations, the uniform ags are argued to be the "structured" semantic definitions, enforcing a healthy discipline on the author of a translation. It seems unlikely that this class could have been discovered without the point of view provided by the coroutine model.

The coroutine model of evaluation is based on the idea of distributed control. Since an ag gives semantics on a production-by-production basis, it is natural to look at the activities during evaluation from the point of view of individual nodes in the semantic tree. From such a local viewpoint evaluation consists of the comings and goings of the processor interspersed with evaluations of attributes performed while it is at the node. We obtain for each production a flowchart prescribing the possible activity sequences at a node where the production applies. A semantic tree is evaluated by attaching to each node a coroutine executing a copy of the appropriate flowchart, then initiating the root coroutine. Coroutines are program segments which cooperate as equals, each coroutine viewing its partners as subroutines to which it can temporarily send control. Whenever a coroutine receives control it continues its execution from the point of last interruption, giving an effect similar to parallel processing. However, synchronization of the processes is accomplished by explicitly yielding control to another coroutine as determined at compile-time; the coroutine organization thus provides the conceptual power of parallel processing without the overhead of run-time scheduling.

To verify the correctness of an evaluator, it is necessary to prove that during any execution, every instruction's execution is well-defined; that is, that there are no run-time errors. This entails showing that no semantic function is executed before its argument attributes have been computed, and that no coroutine sends a neighbor a parameter which the
neighbor's flowchart is not prepared to receive. We use the standard method of inductive assertions, in which predicates are attached to points on each flowchart and shown to be mutually consistent given the semantics of the intervening instructions. It turns out that the key to proving proper cooperation between coroutines is to examine the "protocols", or sequences of parameter transmissions that can occur between neighboring coroutines. The concept of the protocol is the single most valuable insight provided by the coroutine model. The intended manner of cooperation between coroutines is specified by sets $\text{LEGAL}(X)$ of protocols which are to be permitted to occur between the two coroutines communicating across a type-$X$ node of the tree. These sets, together with a set $\text{AVAIL}(c)$ of attribute occurrences and some sets $H_k(c)$ of protocols associated with each point $c$ in a flowchart, constitute the assertions necessary for a correctness proof of a coroutine evaluator. The sets are interpreted as predicates, and the reverse interpretation of proof steps as set operations results in some simple set-theoretic conditions which must hold on the sets. Thus from the initial idea of an inductive-assertion correctness proof, we are able to move to the idea of some finite sets and conditions on them. If satisfactory sets exist for a given coroutine evaluator, then the evaluator is correct.

The general nondeterministic evaluator construction arises from the notion of building up a set of flowcharts and their proof of correctness simultaneously, as is recommended by writers on program verification. Since the assertions required for a verification take the form of sets, we are able to make the assertions into bookkeeping variables of the construction algorithm. From the conditions the sets must satisfy, we deduce how the sets must be updated when additions are made to the flow-
charts. We thus obtain some "steps" or transformations which be applied while leaving invariant the fact that can be interpreted as a correctness proof of the flowcharts. This be viewed as a nondeterministic algorithm for constructing evaluators. The acceptable steps are prescribed, but no rules step to take next are given; this is left as a nondeterministic choice. Any terminating computation of this algorithm yields a correct evaluator but certain sequences of choices will result in "better" evaluators or others.

We obtain a number of practical algorithms for building evaluators by simply adding some choice criteria to the nondeterministic algorithm. These criteria use some knowledge of possible dependency chains in tic trees produced by the given ag in order to guide the choice of instruction to add to a flowchart next; the intention is to thereby out all computations of the general algorithm except those leading "good" evaluators. As many construction algorithms can be written as sources of knowledge of dependency chains; we gave three such ranging from very precise but expensive to fairly crude but easy to compute. The correctness of all these algorithms is immediately clear the fact that they were special cases of the (known) correct algorithm. One class of constructions was able to handle any ag at while another family produced slightly better evaluators at the ex of failing entirely on certain ags. All yielded efficient sequent evaluators which could be generated in machine code if desired.

All these constructions exhibited the weakness that their strategies were purely local in nature; no overall planning of the evaluator's structure was done. As a result, the same job might get d
different ways in different parts of the flowcharts, causing unnecessary duplication of structure within the evaluator and increasing its size. This was reflected in the use of several protocols for a nonterminal when one would have sufficed. This motivated the definition of the uniform ags as those which can be evaluated using only one protocol per nonterminal. It was suggested that the author of a translation be required to specify for each nonterminal the protocol to be used in its evaluation. This class of ags is interesting in that its membership test is easily understood and natural to obey; indeed, we claimed that it imposes a valuable discipline on the writer analogous to that imposed by "GOTO-less" programming. The size of evaluators produced in this way cannot grow more than linearly in the size of the ag, and their simple structure makes more tractible the study of their transformation and optimization. These advantages together with the ease of implementing the construction itself lead us to believe that the uniform ags are the largest class of ags of practical interest.

Much work remains to be done on the automatic implementation of ags. Within the framework of the coroutine model, the big problem is how to effectively generate small evaluators for ags. We have seen that the constructions of chapter 5 do not necessarily use the smallest possible number of protocols for a nonterminal. At present we do not know of a way to find a minimal set of protocols other than trial and error. A related matter concerns the use of loops in flowcharts. The work presented here assumes acyclic flowcharts throughout; but while it is true that we have no use for loops in the sense of repetitive execution of the same instructions, nevertheless the judicious introduction of loops can greatly reduce the size of some evaluators. This situation arises when there
are several independent groups of attributes for a nonterminal such that order within a group is constrained by dependencies but the relative ordering of the groups may vary. By letting each group's instruction sequence be put on a loop through the same control point, all possible permutations of the groups could be represented without combinatorially duplicating the code. Although much of our development does not depend critically on the acyclic-ness, we do not know how to cope fully with this possibility or arrange to take advantage of it.

More significant research lies ahead in the area outside the coroutine model's domain. The explicit building and filling in of trees, while a tractable point of departure, must finally be given up. The amount of space wasted by retaining an entire semantic tree full of intermediate results for the entire duration of the evaluation is unacceptable for large translators. Some way must be found to order evaluations and coordinate temporary storage management to hold down the amount of space needed at any one time. In chapter 6 we made some hints about how this could be approached for certain aigs; the general problem is, as far as we know, completely open.

Perhaps the most difficult and potentially significant optimization problem is how to actually rearrange the semantic functions -- rewrite the aig -- to permit cheaper evaluation. Until this problem is solved, automatic implementations of aigs will probably remain somewhat less efficient than careful manual implementations.
REFERENCES


