EVALUATORS FOR ATTRIBUTE GRAMMARS

by Ralph Charles Merkle

TABLE OF CONTENTS

INSTRUCTIONS TO THE READER

The reader who is unacquainted with attribute grammars is advised to read the paper slowly. (No other method will be feasible...) He might also wish to read Knuth[1], which introduces most of the concepts in more detail. The reader who knows what an attribute grammar is, but is somewhat shaky on the topic, is advised to skim the introduction to refresh his memory. Researchers in the field are advised to read the abstract, ignore most of the introduction, but read the last two pages of that section, (starting on page 12), pay close attention to the definitions, and then skim the rest of the paper, paying particular attention to the algorithms, and reading the text where the algorithms are opaque. The main results of the thesis consist of three algorithms:

1.) a circularity test which is usually linear.

2.) An algorithm to rank the attributes in time N log N, in primary memory bounded by the depth of the parsing stack, and using a tape as secondary storage.

3.) An algorithm to evaluate the attributes, which adds N log N time, adds small primary memory, and uses a disk as a secondary storage device. A bound on the size of an attribute is assumed. (It has to fit in primary memory...)

Both 2 and 3 assume an arbitrary attribute grammar.

EVALUATORS FOR ATTRIBUTE GRAMMARS

Ralph C. Merkle

ABSTRACT

It is possible to evaluate an arbitrary attribute grammar in an amount of primary memory bounded by the depth of the parsing stack, and with a disk as a secondary storage device. This evaluation will require time N log N. (This is exclusive of the memory and time requirements of the semantic functions.) The method appears to offer value as a research tool: it allows the rapid production of compilers for arbitrary attribute grammars that are not inefficient. It does not appear to be efficient enough for practical applications. In this, it is marginal. For some applications, it might prove tolerable.

The method used to evaluate the attributes with a small primary memory appears to have applications outside of its use in attribute grammars. In particular, the algorithm obtained will evaluate an arbitrary address trace with a small primary memory. This result appears to be novel.

In addition, it is shown that the circularity test for attribute grammars, while in general of exponential complexity, can, in almost all cases of interest, be performed in linear time.

INTRODUCTION

Context Free Grammars, (CFG's), have proven extremely useful, and have gained widespread acceptance for the definition of the syntax of programming languages. The presentation of a new language would seem incomplete without an appendix giving its syntax, typically in BNF. A universally accepted, well defined, and easily comprehensible method for describing syntax has been achieved, and much theoretical work has used this solid basis as a starting point for the automatic construction of parsers, used in syntax driven compilers. This happy state of affairs does not exist with respect to the semantics of programming languages. Many methods of defining the semantics of a language have been proposed, but for various reasons, none of them appears to have dominated the others, nor have any of them gone terribly far in replacing English as a semantic meta-language. Naturally, everyone has rushed to fill this gap, in the hopes that their pet semantic meta-language would suffer the happy fate of CFG's, aiding humanity and insuring the author's fame and fortune. Knuth joined this group of would be saviors with a paper in which he proposed the use of attribute grammars, (which he also defined). The author, after reading Knuth's paper, became enamored of attribute grammars and decided that here was the ideal semantic meta-language.

An attribute grammar is a method of attaching a "meaning" to a given phrase in a language. The concept of an attribute grammar is an extension to the concept of a context free grammar. A phrase in a context free language has an associated parse tree. The attribute grammar works directly with the parse tree. It attaches "attributes" to each of the symbols in the parse tree: these attributes are defined to be the meaning of the associated symbol. The following grammar for expressions can serve as an example.

```
E  ::=  E  +  P
E  ::=  P
P  ::=  identifier
P  ::=  (  E  )
```

A sample sentence in this language might be: A+B+(C+D). The attributes that we might wish to associate with E and P might be the type of the expression, and the code needed to evaluate the expression. Every occurrence of E in the parse tree would have two attributes: type and code. The type might have two values, integer or real, while the code might be a sequence of machine language instructions. In this simple example, P would have the same two attributes. Every occurrence of P would have associated attributes defining the type and the code required to evaluate it. (In general, this need not be the case. Different symbols can have different attributes.)

The attributes for a given node in the parse tree are defined in terms of the attributes of the offspring of that node, and also in terms of the attributes of the parent of that node.

Arbitrary semantic functions are allowed to define each attribute. Given the production:

```
E ::= E + P
```

we might have a semantic function, rtype, which determines the resultant type of the left hand E, given the types of the right hand E and the P. We might say:

```
E.type := rtype(E.type,P.type)
```

(The double use of "E" renders this ambiguous, but we assume the reader can understand what is meant. More precise definitions and usage will be considered later.) Notice that this semantic function is tied to a particular production. In general, each production will have an associated set of definitions which will allow the computation of various attributes of the symbols in the production. Also note that, in general, we might define attributes of symbols in the right part in terms of attributes of symbols in the left part, even though this has not been done in this example.

The principal advantage of attribute grammars is the ease with which concepts can be described, and the close link between the semantic definitions, and the actual grammar. Because an attribute grammar includes, as an integral component, a CFG, the semantics of the language are tied quite closely to the syntax, and no room is allowed for ambiguity about the meaning of a particular syntactic construct to creep in. (Semantic definition languages which do not include a grammar as an integral component can result in rigorous semantics, but no clear connection between the rigorous semantics and the actual language.) Attribute grammars are also quite flexible, allowing the use of arbitrary semantic functions. The semantics of any language, as long as the semantics are effectively computable, can be cast into the form of an attribute grammar.

The power of attribute grammars has proven a mixed blessing. On the one hand, it is easy to write an attribute grammar for a language, but on the other hand, there is no guarantee that it will be possible to find an efficient implementation of the attribute grammar. Converting an attribute grammar into a working compiler is a non-trivial feat. While the concept of an attribute grammar has gradually been gaining acceptance, the nasty problems of implementation have kept them from practical use, and forced them into the role of an interesting semantic definition language, i.e., an academic toy. Many people have recognized this problem, and some have been sufficiently brave to leap into the fray. The objective is to produce a practical compiler-compiler, based on attribute grammars. The author has attacked this problem, and it is the major purpose of this thesis to acquaint the reader with the results.

Knuth's original paper[1] recognized two main problems which attribute grammars give rise to. First, given an attribute grammar, how do you evaluate the attributes? This requires that the attributes be evaluated in some particular order, subject to the constraint that all the attributes which serve as arguments to a semantic function must be defined before that semantic function is invoked. This problem is not as trivial as it sounds, because, in general, any attribute in the parse tree can be defined in terms of any other attribute in the parse tree. Untangling this plate of spaghetti can be involved. The other problem that Knuth defined was the circularity problem. The question is just this: Given an attribute grammar, is it impossible to produce a parse tree in which the attributes are defined in terms of each other in a circular fashion? That is, is it possible to evaluate all the attributes for an arbitrary parse tree?

Knuth proposed solutions to both of these problems, but did not concern himself with computational efficiency. His proposed solution to the circularity problem is, in the worst case, of exponential complexity. Worse, the circularity problem is of intrinsically exponential complexity. The work of Jazayeri[4,5] shows that any algorithm to solve the circularity problem must be, in the worst case, of exponential complexity. In this thesis we remove much of the sting from this result by proposing an algorithm which is linear in most cases of interest, showing that the exponential cases are essentially pathological.

Knuth's proposed solution to the problem of evaluation of all of the attributes of a parse tree is quite simple. Put the entire parse tree in memory. The semantic functions impose a partial ordering on the attributes, the order in which it is possible to evaluate the attributes. Using an algorithm proposed by Knuth[2], convert this partial order into a total order, (in linear time, but also linear space, i.e., all the attributes have to be in main memory at once.) Given a total order which satisfies the constraints of the partial order, evaluate each of the attributes, in accordance with the total order. During the process of evaluation, any attribute which has been defined must be kept in main memory, for it could be used, at any time, in defining another attribute. The method, while quite efficient in terms of speed, and also completely general, in that it works for all attribute grammars, uses a great deal of memory. This sharply reduces its widespread applicability. It would be highly desirable to devise algorithms which were efficient, but which used much less memory. Such algorithms need not handle all attribute grammars. If such algorithms handled a significant subset of the attribute grammars efficiently, then this would be sufficient for many purposes. Ideally, an efficient algorithm, which used little memory and executed quickly, and which would handle all attribute grammars, would be desirable.

Previous work has tended to focus on either efficient methods of dealing with a subclass of the attribute grammars, or inefficient methods of dealing with all possible attribute grammars. Fang[8] proposed a method in the latter category. The work of Fang was never intended to be efficient, in terms of either space or time, but was intended to provide a highly flexible research tool. It is, in fact, more flexible than the concept of an attribute grammar given by Knuth. This is because circular definitions can sometimes be resolved, if the functions involved do not use the information in a circular fashion. To give an example:

```
A=AND(FALSE,B)
B=OR(FALSE,A)
```

In this sequence of definitions, A and B are defined in terms of each other. In spite of this, it is clear that A=FALSE, and that the value of B is not, in fact, needed in the evaluation of A. Fang's system would allow this. (provided that the AND function checked its left argument first, and if it proved to be false, ignored its right argument.)

In the methods proposed by Bochmann[3] and Jazayeri[4], on the other hand, the objective has been to define a subclass of the attribute grammars for which it is possible to produce efficient evaluators. In this, they have succeeded. The method of Jazayeri is probably superior, because of its greater power, and comparable efficiency. If the left to right bias is built into the hardware, as it appears to be on some machines, then Bochmann's method is superior. (Jazayeri's method allows right to left passes.) It should be pointed out that the method described by Bochmann in [3] is not, strictly speaking, left to right. He appears to have made a mistake, and included too many attribute grammars in the class which he claims he can handle with a fixed number of left to right passes. For the first pass of his algorithm, all is well. On subsequent passes, however, it might be that an attribute, evaluated on a previous pass, and thus supposedly available, is in the wrong position, and thus not available without additional computation. This additional computation would involve exactly the concepts that Jazayeri used in his alternating semantic evaluator. To put it another way, Bochmann is traversing a tree. Some attributes he evaluates upon first reaching a node, other attributes he evaluates upon leaving the node. These latter attributes will appear in their correct end-order position, not in the pre-order position that is desirable. Because some attributes from the previous pass appear in end-order positions, they cannot be used to evaluate attributes that appear in pre-order in the current pass. Oops.

The original objective of the current work was to design algorithms which would allow the efficient evaluation of an arbitrary attribute grammar in a small, fixed, primary memory with one tape drive as a secondary storage device, and in linear time. It has proven necessary to relax these criteria somewhat. In particular, the problem has been divided into phases, and the following results have been obtained.

PHASE 1.) Ranking of the attributes. It has proven possible to rank the attributes in 4 passes for an arbitrary attribute grammar, using an amount of primary memory proportional to the depth of the parsing stack. These passes are alternating left to right, right to left.

PHASE 2.) Sorting the attributes, once they are ranked. Many well known algorithms are available to deal with this problem, typically running in time N log N. It should be noted that the sorting problem, given a ranking, can be done in fewer passes than a general purpose sort.

PHASE 3.) Evaluating the attributes, once they have been sorted. It has proven possible to evaluate the attributes in time N log N, in a fixed primary memory, with a disk as a secondary storage device. This is subject to the restriction that it is possible to fit the attributes into primary memory for the actual evaluation of each one. That is to say, if a semantic function operates on 4 attributes, and produces a 5th, then it is necessary that all 5 of these attributes fit into the primary memory. This puts a bound on the size of a given attribute. This result is also subject to the restriction that it does not include the time spent in the semantic functions, nor the space that they use. If the semantic functions take forever to execute, then this method will not take time N log N, but will, instead, add time N log N to the execution time of the semantic functions. If the semantic functions need huge gobs of memory, then the additional memory requirements are fixed, but the total core requirements might be arbitrarily large. In essence, this algorithm uses the disk drive to simulate log N sequential storage devices, (tape drives.) Thus, if an installation is willing to devote 10 or 20 tape drives to the task, it is possible to evaluate gargantuan programs. It is not clear that this will occur in practice. It is possible to improve the efficiency of this algorithm if we give it a primary memory of size log N. (It should be mentioned that the "fixed" primary memory mentioned before neglects the log N factor that appears because it requires log N bits to represent a number of size N. It should also be mentioned that the N for this phase is more closely related to the total size of the evaluated attributes than it is to the size of the input string.)

AN EXAMPLE

In the following section, we give a detailed example of an attribute grammar. The discussion has been adapted from Knuth[1].

Suppose we are given the following grammar for binary numbers:

```
B       ::=     0
B       ::=     1
L       ::=     B
L       ::=     L       B
N       ::=     L
N       ::=     L       .       L
```

(The symbols 0,1, and . are the terminals; the nonterminals are B, L, and N standing respectively for bit, list of bits, and number.) This grammar says in effect that a binary number is a sequence of one or more 0's and 1's, optionally followed by a radix point and another sequence of one or more 0's and 1's.

We can define the following attributes for each symbol:

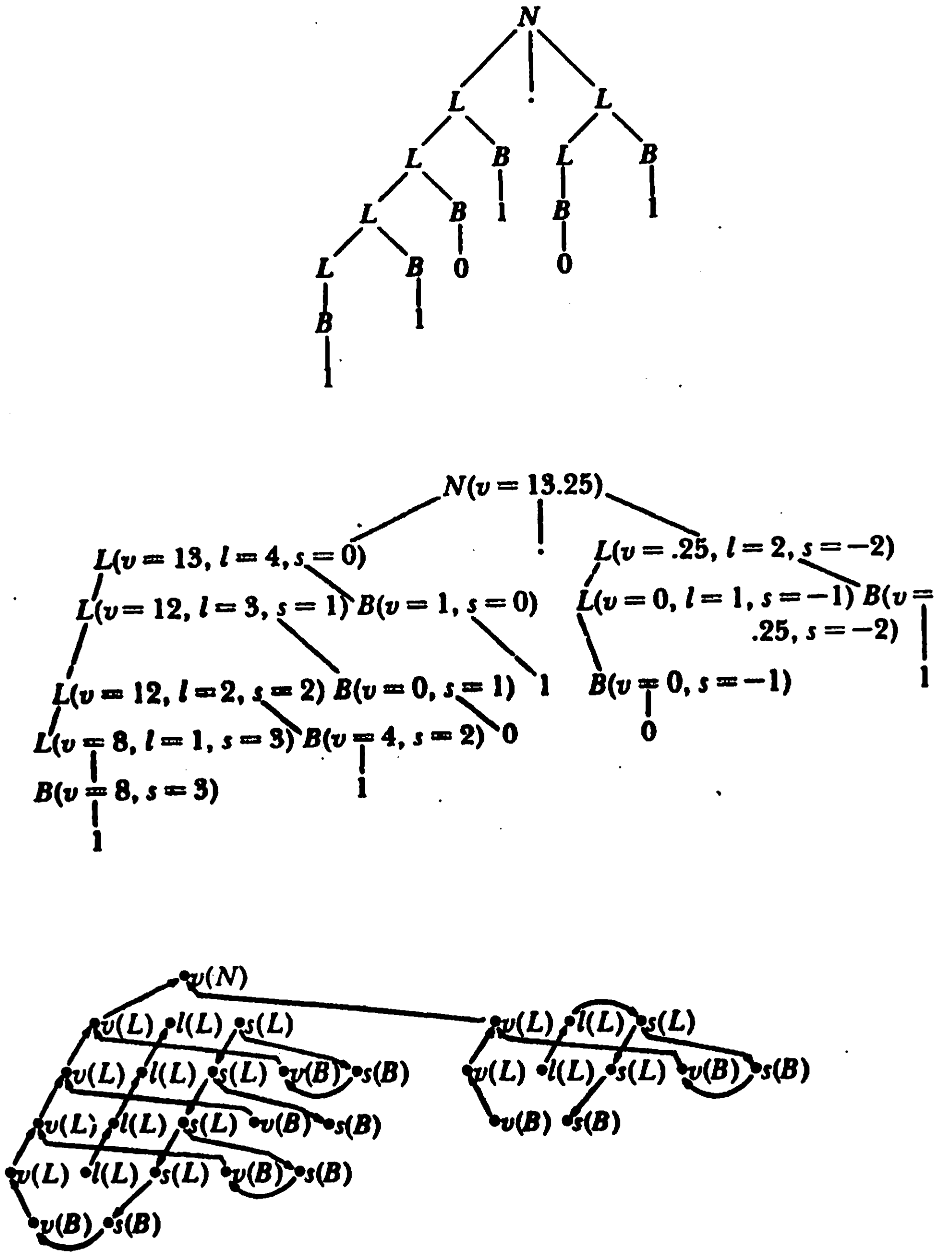
Each B has a "value" B.v which is a rational number.
Each B has a "scale" B.s which is an integer.
Each L has a "value" L.v which is a rational number.
Each L has a "length" L.l which is an integer.
Each L has a "scale" L.s which is an integer.
Each N has a "value" N.v which is a rational number.

These attributes can be defined as follows:

SYNTAX					SEMANTICS
B	::=	0			B.v := 0
B	::=	1			B.v := 2 ^ B.s
L	::=	B			L.v := B.v, B.s := L.s, L.1 := 1
L1	::=	L2	B		L1.v := L2.v + B.v, B.s := L1.s, L2.s := L1.s+1, L1.1 := L2.1 + 1
N	::=	L			N.v := L.v, L.s := 0
N	::=	L1	.	L2	N.v := L1.v + L2.v, L1.s := 0, L2.s := -L2.1

In the semantic rules shown above, the attribute on the left hand side of the assignment statement is defined by the expression on the right hand side of the assignment statement. The reader should note that some attributes of symbols in the right part are defined in terms of other attributes in the production, including attributes of the left part, while some attributes of the left part are defined in terms of attributes in the right part. This provides a two way flow of information in the parse tree. We are using both synthesized attributes, i.e., attributes in which the flow of information is towards the root of the parse tree, and inherited attributes, or attributes in which the flow of information is towards the leaves of the tree. The attributes B.v, L.v, L.1, and N.v are synthesized, i.e., are involved in the upwards flow of information, while B.s and L.s are inherited, and bring information down towards the leaves. (The parse "tree" is thought of as being upside down, with its root in the air, and its leaves on the ground.) The evaluation of all the attributes in this grammar would involve going up and down the parse tree.

How does this grammar work? For a given binary number, 100101.1010001, how would this grammar produce the "meaning"? The "meaning" of this string of 0's and 1's is the value attribute associated with the distinguished symbol, N. That is, N.v is defined to be the meaning of the string. In essence, the grammar associates a scale factor with each binary digit in the string, and multiplies this scale factor by the value of the digit. Each digit then contributes the proper amount to the value of the entire string, which is obtained by summing the value contributed by each digit. The scale associated with a digit cannot be determined by examing the digit alone. The scale associated with a given digit can only be determined by examining the context around the digit, i.e., by counting from the radix point. It this feature of collecting information from all parts of the string, and using it to aid in the definition of local components, which gives attribute grammars their power. The meaning of a sub-tree is context dependent. An illustration of this is given in the diagram, which is from Knuth[1]. The reader who has had difficulty in following the discussion is advised to stare at the diagram, and trace through the application of the various semantic functions, along with the flow of information in the parse tree.



The diagrams are from Knuth[1]

DEFINITIONS

We now give the formal definitions which will be used in the rest of the paper.

To begin with, an attribute grammar utilizes a CFG as an integral component. We therefore start by defining a grammar. A grammar is a 4-tuple, G = {V,N,S,P}. V is a finite vocabulary of terminal and non-terminal symbols. N, a subset of V, is the set of nonterminal symbols. S is the distinguished symbol. P is the set of productions. In the following discussion, P[i] will denote the ith production, P[i,j] will denote the jth symbol in the ith production, and P[i,0] will denote the left part of the ith production. We shall denote the rank of a set by /set/. /G/ is 4, /{a,b,c}/ is 3, /P/ is the number of productions, /P[i]/ is the number of symbols in the ith production. (note that /P[i]/ = 1 if the right part is empty.) This gives the following definitions.

V	the set of terminals and nonterminals.
N	The set of nonterminals.
S	The distinguished symbol.
P	The set of productions.
P[i]	The ith production.
P[i,j]	The jth symbol in the ith production
P[i,0]	The leftpart of the ith production.

/Set/ The rank of the Set.

/P/ The number of productions.

/P[i]/ The number of symbols in the ith production.

To take the 4th production of our sample grammar as an example,

P[4] = L ::= L B
P[4,0] = L
P[4,1] = L
P[4,2] = B
/P[4]/ = 3

We associate with each symbol X in V a finite set of synthesized attributes, S(X), and a finite set of inherited attributes, I(X). We require that I(S)={}, that is, the distinguished symbol has no inherited attributes, and likewise, for each terminal T, S(T)={}, that is, no terminal has any synthesized attributes.

S(X) The synthesized attributes of X.

I(X) The inherited attributes of X.

The semantic functions for each production must allow us to define all the synthesized attributes of the left part, and all the inherited attributes of the right part. We assume that all inherited attributes of the left part were defined in the production "above" this one, i.e., nearer the root, while the synthesized attributes of the right part have been defined in the production below this one. The semantic functions can take any attributes in the production as arguments. Each synthesized attribute of the left part, and each inherited attribute in the right part, must appear on the left hand side of an assignment statement exactly once. If it does not appear, the corresponding attribute is undefined. If it appears more than once, the attribute is overdefined.

We also need to define clearly notation for the set of attributes associated with each nonterminal, each production, and with each possible parse tree.

X.att For X in V, this is the set of attributes associated with X.

T.att This is defined to be the same as Root(T).att. T.att is simply a notational convenience.

P[i].att The set of attributes associated with the ith production. (We might wish to refer to P[i].att as a multiset, or bag. If a symbol occurs twice in the rightpart, the corresponding attributes will occur twice in P[i].att.)

Let T be any derivation tree obtainable in the grammar, having only terminal symbols as labels of its terminal nodes, but allowed to have any symbol of V, (not only the start symbol, S) as the label of the root.

We also will wish to refer to sub-trees of a parse tree, and we will do so by simple indexing: T[1] is the first (leftmost) subtree, T[2] is the second subtree, and so forth and so on. If we wish to refer to the symbol which is at the root of a parse tree, we will just say Root(T).

T[i] The ith subtree of T.

Root(T) The symbol which is at the root of T.

Rootproduction(T) The production which is at the root of T.

In the course of the paper we shall also be needing various dependency graphs. A dependency graph is a graph whose nodes are attributes, and whose arcs indicate the semantic dependencies among those attributes. Several different kinds of dependency graphs will be defined and used. In general, we shall use D(arg) to indicate some form of dependency graph. The type of "arg" will indicate the type of dependency graph.

D(production) A dependency graph whose nodes are taken from production.att, and whose arcs represent the direct semantic dependencies in the production.

D(T) A dependency graph whose nodes are taken from the derivation tree, T, and whose arcs represent the direct semantic dependencies among the attributes in T.

DTOP(T) A dependency graph whose nodes are taken from Root(T).att, but whose arcs represent the indirect semantic dependencies in T. (a,b) is in DTOP(T) iff a,b are in Root(T).att and (a,b) is in D(T)*, (where * represents the transitive closure operation.) That is, there must be a directed path from a to b in D(T). If T is a terminal, then DTOP(T)={}.

CIRCULARITY

We shall first briefly review Knuth's circularity test, and then give a modified version that is linear for the class of grammars under consideration.

Knuth makes the observation that the circularity problem for attribute grammars is equivalent to the problem of determining if there exist oriented cycles in any DTOP(T). That is, an attribute grammar is circular iff there exists an attribute, a, and a derivation tree, T, such that (a,a) is in DTOP(T). This is equivalent to saying a grammar has a circularity iff there is a directed cycle in D(T), for some T, or to saying that there is some derivation tree, T, and some attribute, a, such that (a,a) is in D(T)*.

To determine the dependency graph, DTOP(T), associated with a parse tree, T, would seem to require that we compute the transitive closure of D(T). (Recall that DTOP(T) is defined in terms of D(T)*). Because of the particular nature of attribute grammars, however, it is possible to simplify this computation. We observe that we can compute D(T) knowing only D(Rootproduction(T)) and the dependency graphs derived from the sub-trees, DTOP(T[1]), DTOP(T[2]), etc. etc. That is, all we need know is the dependencies of the production at the root of T, and the transitive closure of the dependencies of the sub-trees. Just as we can factor the problem at the root into a problem involving the separate subtrees, and limited interaction among the

T[i], so we can apply the same concept recursively to each subtree in turn. That is, we can factor the algorithm for computing DTOP(T) into an algorithm which computes DTOP(T[i]) for each of the sub-trees, and then uses that information to compute DTOP(T) for the whole tree. To be a trifle more precise, we can write a recursive routine, CLOSE(T), which will compute DTOP(T) . We can write the code for Close(T) as follows:

```
Function Close(T);
Begin
  Temp := {};
  For I := 1 to /Rootproduction(T)/ - 1 Do
     Temp := Temp union Close(T[I]);
  Temp  := Temp union D(Rootproduction(T));
  Temp := Normalclosure(Temp);
  Return({(a,b): a,b are in Root(T).att and (a,b) is in Temp});
End.
```

(The function, Normalclosure, computes the transitive closure of a relation (which can also be viewed as a directed graph) using normal methods, e.g., Warshall's algorithm.) The algorithm uses the "union" operator to refer to the union of two directed graphs. The union of two graphs is just the graph consisting of the union of the set of nodes, and the union of the set of arcs. It should be noted that all graphs in the algorithm are subgraphs of D(T). This implies in particular that the statement

```
    Temp := Temp union D(Rootproduction(T));
```

"pastes together", in Knuth's terms, the different sub-graphs in Temp.

This algorithm will determine only if a particular derivation tree has a circularity. It will not determine if a derivation tree with a circularity exists. An analysis of this algorithm, however, leads one to the test for circularity for an attribute grammar. The key observation is that the number of possible values that Close(T) can take on is finite. Because of this finiteness, the algorithm can be analyzed by exhaustive search. We can determine what the algorithm will do in all possible cases, because the number of cases, though large, is bounded.

We observe that Close(T) = DTOP(T), that is, (a,b) is in Close(T) iff (a,b) is in D(T)* and a,b are in Root(T).att. Now, there are a bounded number of graphs whose nodes are in Root(T).att. This is because /Root(T).att/ is bounded. If a directed graph can have no more than a bounded number of nodes, then there are a bounded number of possible directed graphs.

Beyond this, there are a bounded number of possible symbols. This means that, even though there are an infinite number of possible trees, Close(T) will produce a bounded number of graphs. Each such possible graph will have, as nodes, the attributes of some symbol. This means that we can represent the possible values for Close(T) by the following data structure:

S:Array[V] of sets of graphs

That is to say, for each symbol, X, there is a set of graphs, S[X]. Each graph in the set has nodes which are taken from X.att. Each call to Close(T) must produce a value which is in S[Root(T)]. If we can determine the set of all possible values which Close(T) can return, then we can determine if there is a circularity in the grammar.

We can now write the algorithm which determines if there is a circularity in the grammar.

```
   S:Array[V] of sets of graphs;

Begin
   For X in V Do S[X] := { the null graph };

Repeat
   Pick P[I] from P;
   Temp := {};
   For J:= 1 to /P[I]/ - 1 Do
     Temp := Temp union Pick any from S[P[I,J]];
   Temp := Temp union D(P[I]);
   Temp := Normalclosure(Temp);
   Temp := { (a,b):a,b are in P[I,0].att and (a,b) is in Temp};
   Add Temp to S[P[I,0]];
   Until  no more graphs can be added to any S[X];
End.
```

The "Pick" primitive indicates the non-deterministic selection of some item from a set. "Pick x from integers" might be used to select an integer, x, nondeterministically. Its use in the algorithm should be self-explanatory.

This is the algorithm as given by Knuth[1]. Because of the use of the non-deterministic "Pick" primitive, it would appear to require exponential time. This appearance is completely justified. Jazayeri has shown[5] that the circularity test for attribute grammars is of exponential complexity, i.e., that the general problem of determining whether or not a given attribute grammar has a circular definition, for some parse trees, is intractable. With hopes for a general and practical solution to this problem rudely shattered, the search turns to less general methods. The objective is to define a restricted, but natural, subclass of the attribute grammars for which a reasonable solution to the circularity problem exists. It is the purpose of this section to point out a subclass of the attribute grammars for which the complexity of the circularity test is linear in the size of the grammar, using a modification of the the test for circularity proposed by Knuth[2].

The restriction that is proposed is simply to place a bound on the complexity of an individual production. This restriction seems a plausible one. Most grammars in use today have only a few symbols per production.

We can give the modified circularity test as follows:

```
   S:Array[V] of sets of graphs;
   Active: Set of V;

Begin
```

```
   For X in nonterminals Do S[X] := { the null graph };
   Active := V;
   Comment   V is terminals union nonterminals;

Repeat
   Pick X from Active;
   Delete X from Active;
   For All I such that X is in the right part of P[I] Do
    Repeat
    Temp := {};
    For J:= 1 to /P[I]/ - 1 Do
      Temp := Temp union (Pick any from S[P[I,J]]);
    Temp := Temp union D(P[I]);
    Temp := Normalclosure(Temp);
    Temp := { (a,b):a,b are in P[I,0].att and (a,b) is in Temp};
    If Temp is not in S[P[I,0]] Then
     Begin
      Add Temp to S[P[I,0]];
      Add P[I,0] to Active;
     End;
    Until no new values can be added to S[P[I,0]];
   Until Active = {};
End.
```

This algorithm is the same as Knuth's, except that we now impose some constraints on the non-deterministic selection methods which were used previously. These constraints, along with our assumption that each production is of bounded complexity, can be used to show that the algorithm given is of linear complexity in the size of the grammar.

To begin with, we observe that steps 4 through 14, i.e., the inner Repeat clause, take a bounded amount of time to execute. In this repeat clause, we are considering a single production. This one production, in conjunction with the sets S[P[I,1]], S[P[I,2]], ... S[P[I,/P[I]/-1]], will be used to add new graphs, if possible, to S[P[I,0]]. Statement 14 forces us to keep adding new graphs, until we can add no more using only this production. That is to say, until no matter what choice is made by the non-deterministic Pick primitive on line 7, the result is always already in S[P[I,0]]. It might not seem obvious that this repetition will require a bounded amount of time, but this can be seen more clearly in light of the following observations. First, the complexity of a production is bounded. This means that the complexity of each member of S[X] must be bounded, also. This implies that the size of each set, S[P[I,J]], must be bounded. Now, the complexity of each production is bounded, so /P[I]/ is bounded. Therefore, in steps 4 through 14, we are examining how a bounded number of S[P[I,J]], each of which has bounded complexity, can interact. While the number of interactions might be large, it is bounded. Therefore, steps 4 through 14 require bounded time.

The next question is: How many times are steps 4 through 14 executed? We observe that statement 3, the For clause, selects a single production, then executes statements 4 through 14 for this production. That is, each time we select a production, we execute statements 4 through 14 once. Asking how many times statements 4 through 14 are executed is therefore the same as asking how many productions are selected by the For statement, statement 3. We now come to the crux of the matter. A single production can be selected by statement 3 at most a bounded number of times. To demonstrate this, we observe that a production can only be selected if some symbol in its right part is active. A symbol, X, can become active at most a bounded number of times, because X can only become active if we add a new graph to S[X], and the number of possible graphs in S[X] is bounded. This can be clearly seen be examining statements 11 through 13. /S[P[I,0]]/ is bounded. Each time we add a new graph to S[P[I,0]], we mark P[I,0] as active. This is the only way, outside of the initialization, that a symbol can be marked as active. Thus, we can mark each symbol as active at most a bounded number of times.

Now, each production can be selected by statement 3 at most a bounded number of times, because to select a production, P[I], we must have marked some symbol in its right part as active. But we can mark each such symbol as active only a bounded number of times, and there are a bounded number of symbols in the right part. Therefore, each production can be selected by statement 3 only a bounded number of times.

Finally, we note that the number of productions is linear in the size of the grammar. Therefore, the total execution time of this algorithm is linear in the size of the grammar. This is because statements 4 through 14 execute in a bounded period of time, these statements are executed a bounded number of times per production, and the number of productions is linear in the size of the grammar. Q.E.D.

RANKING THE ATTRIBUTES

In the next section, we shall consider the problem of ranking the attributes. This problem is distinct from the problem of sorting the attributes, and from the problem of evaluating the attributes, once sorted. In the methods of both Jazayeri[4], and of Bochmann[3], the problems of ranking, sorting, and evaluation have been merged into one. This produces more efficient evaluators, but results in less flexibility. The division into three distinct phases increases flexibility, to the point where arbitrary attribute grammars can be handled, but only at the price of decreased efficiency. While the methods of Jazayeri and of Bochmann require a fixed number of passes to deal with the attribute grammars which meet the restrictions that they impose, the method to be described requires log N passes, to deal with any arbitrary attribute grammar.

Rather than attempt to introduce the ranking algorithm slowly, a piece at a time, we shall give the entire algorithm at once, and then explain, afterwards, how it works. It is given in a Pascal-like language. The notation is similar to that of the previous section, with one or two minor exceptions.

Data structures:

```
Tree  = Record
          att: set of attributes;
          d:   a relation of attributes, i.e., a set of pairs  of
```

```
         attributes.  It can also be viewed as a directed graph;
         offspring: an array of trees;
        End;

Attribute = Record
              position,pred:integer;
             End;
```

It should be noted that we are considering attributes with respect to a parse tree: that is to say, two attributes are distinct if they belong to different nodes in the parse tree, even though those two nodes are labeled with the same nonterminal, and even though the two nodes are expanded by the same production.

Initialization of the data structure:

We shall assume that the input string has already been parsed, and that we have available an explicit parse tree. Each node of the parse tree is represented by an object of type Tree, as given above. We initialize each field of a given node, T, as follows:

T.att This field remains constant throughout the algorithm. It is the set of attributes associated with this node. When T is a terminal, each of the attributes must also be initialized. For each attribute, a, of such a terminal, a.pred:=1. The value of a.position is computed by the algorithm, and is initially undefined.

T.d This field is initialized to show the dependencies among the attributes associated with this node and its direct descendants. Initially, for each subtree T, T.d = D(Rootproduction(T)), from the notation of the previous section. T.d will next contain DTOP(T), along with some other stuff. (The other stuff is DTOP(T[I]), or DTOP of the offspring of T.) DTOP(T) will be computed during the first pass. T.d will finally be used to hold a somewhat different relation, a total ordering on T.att, (instead of just the partial ordering, defined by DTOP(T)). (It should be noted at this point, that the construct (T.offspring[1].att union ... union T.offspring[j].att) will appear quite frequently. The variable j is a dummy variable, and is used simply to indicate the last offspring of the node, T.)

T.offspring The direct descendants of T, appearing in the same order as the corresponding symbols in the right part of P. Note that T.offspring[I], in the notation of this section, is identical with T[I], in the notation of the last section. The change was made to bring the syntax into accord with Pascal.

THE ALGORITHM

Procedure Pass1(T:Tree);

Comment: This procedure computes DTOP(T) for every node in the derivation tree, and adds the resulting relation to T.d. (To be exact, it computes DTOP(T) union DTOP(T[1]) union DTOP(T[2]) union etc. etc.)

```
Begin
  For x in T.offspring and x a nonterminal Do Pass1(x);
  Temp:=T.offspring[1].d union ... union T.offspring[j].d;
  T.d:= T.d union {(x,y) in Temp such that x,y in {T.att union
  T.offspring[1].att union ... union T.offspring[j].att}};
  Normalclose(T.d);
End;
```

Procedure Normalclose(R:relation);

Comment: This procedure computes the transitive closure of R using normal methods.

```
Begin
  While there exists x,y,z such that (x,y),(y,z) in R and (x,z)
  not in R Do Add(x,z) to R;
End;
```

Procedure Pass2(T:Tree);

Comment: Pass2 determines the order in which the attributes may appear. At the end of pass 2, for all nodes, T, if a,b in T.att, then (a,b) in T.d implies a must be evaluated prior to b, i.e., b depends on a. Pass 2 also converts the partial ordering imposed by the semantic constraints into a total ordering. It does so by adding artificial constraints.

```
Begin
  Normalclose (T.d);
  While there exists x,y in {T.att union T.offspring[1].att union ... union T.offspring[j].att} such that (x,y) not in T.d and (y,x) not in T.d Do
  Begin
    Add((x,y)) to T.d;
    Normalclose(T.d);
  End;
  For x in T.offspring and x a nonterminal Do
  Begin
    While there exists y,z in x.att such that (y,z) in T.d and (y,z) not in x.d Do Add((y,z)) to x.d;
    Pass2(x);
  End;
End;

Procedure Pass3(T:Tree);
Comment: Pass3 computes the lengths of various sequences of attributes that must be adjacent in the final ordering.
Begin
  For x in T.offspring and x a nonterminal Do Pass3(x);
  For a in T.att Do
  Begin
    Temp:= {x in {T.offspring[1].att union ... union T.offspring[j].att} such that (x,a) in T.d and for all b in T.att, (b,a) in T.d implies (b,x) in T.d};
    Comment: To put it another way, Temp is the set of attri-
    butes of the offspring of T which precede a in the total ord-
    ering defined by T.d, but follow all other attributes in
    T.att which precede a;
    a.pred:= (Sum of Temp.pred) + 1;
    Comment: Temp.pred denotes the bag of numbers produced by
    applying the field selector, pred, to each of the elements of
    the set, Temp, in turn;
  End;
End;

Procedure Pass4(T:Tree);
Comment: Pass4 determines an integer, giving the order in the
ranking, for each attribute.
  Function Predecessor(x:attribute);
  Begin
    Comment: This function selects the predecessor of its argu-
    ment in the ordering defined by T.d, and the predecessor is
    in T.att, or T.offspring[j].att, for some j;
    If there exists a y such that (y,x) in T.d and for all z,
    (z,x) in T.d implies (z=y) or (z,y) in T.d Then
    predecessor:= y
    Else
    Predecessor := "QUIT";
  End;
  Function Last;
  Begin
```

```
    Comment:  This function selects the attribute in T.att  which
    follows all other attributes in T.att in the ordering defined
    by T.d;
    select x such that for all y, y<>x implies (y,x) in T.d;
    Last:=x;
  End;
Begin
  Comment:  The body of Procedure Pass4;
  If T=Root Then For x in T.att Do x.position :=  Sum  of  {y  in
  T.att such that y=x or (y,x) in T.d}.pred;
  Temp:=Last;
  Repeat
    If Predecessor(Temp) not in T.att Then If Temp in T.att Then
    Predecessor(Temp).position:=Temp.position-1
    Else
    Predecessor(Temp).position:=Temp.position-Temp.pred;
    Temp:=Predecessor(Temp);
  Until Temp="QUIT";
  For x in T.offspring and x a nonterminal Do Pass4(x);
End;
```

```
Begin
  Comment:  this is the main procedure;
  Pass1(Root);
  Pass2(Root);
  Pass3(Root);
  Pass4(Root);
End;
```

Before going into a detailed discussion of the algorithm, some global comments are in order. The algorithm is clearly divided into 4 separate passes, executed consecutively. Passes 1 and 3 carry information upwards in the tree,towards the root. Passes 2 and 4 carry information downwards in the tree, towards the leaves. The first pass carries information about the dependency relations of the attributes upwards. The second pass does two things: it carries information about the dependency relations of the attributes downwards in the tree, and it also imposes additional constraints on the dependencies, i.e., it adds artificial "semantics", whose sole purpose is to reduce the number of acceptable orderings and thus simplify the problem. The constraints added by the second pass are not, in general, sufficient to produce a unique ordering of the attributes, and so, in the third pass, this ordering process is completed. The third pass begins the job of determining what the final numbering should be. It does this by counting the number of attributes in various subsequences of the final ordering. By the time pass 4 is executed, enough information about the ordering has been gathered to allow the actual assignment of numbers to each attribute, which define their final position in the ranking.

Pass 1 computes DTOP(T) union DTOP(T[1]) union DTOP(T[2]), etc. etc. for each node in the derivation tree. It generates the transitive closure of the relation defined by the semantics of the production used, and by the relations computed for the offspring.

Pass 2 finishes the job of determining how two attributes associated with the same node are related. That is, DTOP(T) does not give the complete set of constraints on the order of the attributes in X.att. DTOP(T) provides only that information about the ordering which can be deduced from the subtree, T. We must also add the additional information which can be gained by examining the rest of the tree. The logic of pass 2 which insures that if x is needed by y, then (x,y) will appear in T.d is very similar to that used in pass 1, and so will be skipped.

In addition, Pass 2 imposes further constraints which insure that all the attributes associated with a particular production, i.e., the attributes of T and the attributes of all the offspring of T, are totally ordered. Because we are adding additional constraints, we might accidentally add one constraint too many, and produce an artificially circular definition. Examining the code for Pass2 indicates that we add new items to the relation in two places. In the first While statement, we cannot possibly get into trouble, for we have tested the relation to insure that the two elements were previously unordered with respect to each other. Assume, therefore, that when we add the pair (y,z) in the second while statement, that we have introduced a circularity in x.d. But if we have a circularity in x.d at that point, then we must surely have had a circularity in T.d. The circularity in x.d can be made to involve only elements of x.att, i.e., the attributes associated with the node x. This is because, in any possible circular path, the attributes from x.offspring can be removed. If we have the sequence of attributes "a b c" in the circularity, and a and c are in x.att, while b is not, then we can remove b, giving "a c", and the pair (a,c) will be in x.d. This is because x.d was computed by taking the transitive closure over the attributes associated with the production which expands x. Therefore, the circularity will exist among the attributes in x.att, ignoring attributes from lower nodes in the tree. But if the circularity involves only elements of x.att, then it must have already appeared in T.d, for in pass 1, all elements in x.d that involved only attributes in x.att were added to T.d. This means that, contrary to our assumption, there was a circularity in T.d. Thus, we will not introduce a circularity that is not already there.

The other thing to note about pass 2, is that it defines a total ordering on the attributes associated with a particular production. That this is so can be seen by examining the first While loop. If any two attributes have no relation to each other, they are forced to have a relation to each other. When this process terminates, every pair of attributes is related. This defines a total ordering.

By the time pass 2 has finished, we have imposed a total ordering on the attributes associated with each production, but we have not defined a total ordering on all attributes in the derivation tree. We must add an additional constraint to the constraints already added in order to obtain a total ordering on the attributes. This total ordering will be needed by pass 3.

Before proceeding further, we shall define some terminology.

Pass 2 order: a,b are in pass 2 order iff there exists a node, T, such that a,b are in T.att, and (a,b) is in T.d. It is understood that the value of T.d refered to is the value after the completion of pass 2 but prior to the start of pass 3.

Given pass 2 order, we wish to define a total order, which specifies completely the order in which all the attributes in the derivation tree will appear. We observe that pass 2 order fails to be a total order because two attributes in separate subtrees need not be related. We note that distinct subtrees must necessarily have a common ancestor, and it seems intuitively reasonable to define our additional constraints by forcing a relationship between the attributes of a node and the attributes of all the ancestors of that node. We can then derive an indirect relationship between any two attributes by tracing back to a common ancestor, and using the relation among the ancestor's attributes to define the relation between the two attributes in question. (In the following, all variables have the values they would have after the completion of pass 2.) Accordingly, we define our total order as follows:

The pair, (a,b) is in total order iff one or more of the following three conditions is met:

1.) (a,b) is in pass 2 order.

2.) If there exist nodes, T and T', with T' an ancestor of T, such that a is in T'.att, and b is in

T.att, and neither (a,b) nor (b,a) is in (pass 2 order)*, then (a,b) is in the total order.

3.) (a,b) is in the total order if it is in the transitive closure of the union of the relations defined in 1 and 2.

This definition means, essentially, that we first see if two attributes are in (pass 2 order)*, but if they aren't in (pass 2 order)*, then we seek out the common ancestor, associate each attribute with an attribute of its common ancestor, and use the order defined by the order of the ancestors.

We can view this in another fashion, as follows. Each node, T, in the derivation tree, has a subtree descending from it. Now, there are many attributes in this subtree. We group these attributes into categories. We associate an attribute in the subtree with the first attribute of the root which must follow it in the final order. Put another way, with each attribute of the root of the subtree, we associate those attributes in the subtree which must directly proceed it in the final order. Now all we do is order the categories in the order defined by the order of the attributes of the root. In this way, the ordering of the attributes at the root is used to provide an ordering on all the attributes in the subtree. If a and b are two attributes in the subtree, associated, respectively, with two attributes, c and d, of the root of the subtree, then (a,b) is in the final order iff (c,d) is in (pass 2 order)*, and (b,a) is in the final order iff (d,c) is in (pass 2 order)*.

While the total order involves some elaborate definitions, we never actually compute the total order, explicitly, in pass 3. We must know that such a total order, with the properties described, exists, but that is all. The actual code of pass 3 performs simple operations on integers. In particular, pass 3 assigns an integer value to a.pred, a value associated with each attribute.

In the third pass, we are counting the number of attributes in a sub tree which have been associated with each attribute in the root of the subtree. In our imaginations, we may view these counts, which are computed in T.pred, (a name which is derived from the fact that T.pred counts the immediate predecessors of an attribute), as counting the number of attributes which have been placed in sequence next to the root attribute. Conceptually, the process can be imagined as the concatenation of sequences of attributes. It can be viewed as a generalized syntax directed translation schemata[6]. (In passing, it should be noted that the following algorithm can be extended to give a method of determining the resulting translation for an arbitrary syntax directed translation schemata.)

If there were no constraint on memory, we could sort, (not just rank), the attributes in accordance with the final order during the third pass by using the following syntax directed translation schemata, (read reference [6] before proceeding.)

The desired translation is a string of attributes, in an order which can be evaluated. To define our translation, we associate a sequence of the syntax directed translation schemata with each attribute. Thus, for a given symbol, X, which has m attributes, we would have m sequences for the translation schemata. The rules for our translation schemata are simple. All the attributes associated with a single production can be divided into two groups, the attributes associated with the left part, and the attributes associated with symbols in the right part. Denote members of the former group by L, and of the latter group by R. We have imposed a total ordering on all the attributes associated with a single production. Therefore, we could, if we so desired, list the attributes in this order. The resulting list might appear as follows:

L L L   R R R L   R R L   R L   R R R R L   R L

Now, as can be clearly seen, adjacent to each attribute from the left part is a group of attributes from the right part. We associate with each L, the concatenation of the sequences associated with the adjacent R's. The attribute, L, is concatenated on the right of the resulting sequence. That defines, for each production in the derivation tree, the syntax directed translation schemata. (As the reader might note, we have taken a liberty with the strict definition of a SDTS. We have given a rule for each production in the derivation tree, and so might have different rules for two different applications of the same production. It is assumed that this will not cause confusion.) This syntax directed translation schemata will translate an input string into a sequence of attributes, and the sequence of attributes will be in an order which can be evaluated.

Those satisfied with this explanation can skip to page 49. Those desiring a more detailed treatment can continue.

We will let each attribute associated with each node, be the header of a sequence of attributes. Thus, if a node has 5 attributes, then it will conceptually have 5 sequences of attributes associated with it. Each node of the derivation tree will have associated with it several sequences of attributes. Each of these sequences will be composed by concatenating sequences associated with the offspring of the node, and also by concatenating the attributes associated with this node in appropriate places. The attributes associated with this node will be imagined as being at the extreme right of these sequences, that is, all the attributes to the left of a given attribute precede it in the final order, and all following attributes succeed it. To determine the sequences which should be associated with the attributes of node T, given the sequences associated with the attributes of the offspring of T, we can use the following algorithm. Order the subsequences in the ordering defined by T.d on the corresponding attributes. (It should be noted that T.d defines an ordering on all attributes associated with Rootproduction(T)). The subsequence to be associated with an element of T.att, call it a, is found by concatenating a and the subsequences associated with attributes of the offspring of T, which are found to the immediate left of a. (Note that there will be no subsequences which are to the right of all of the attributes of T in the ordering. If there were, then these attributes would be isolated, and could not affect any attributes above themselves. That this is so will be seen more easily after the demonstration that this algorithm will never place two attributes in reverse order in the final ranking.)

The following property, which will be inductively passed up the nodes of the tree, is used to show that two attributes can never be out of order. If, at a given node in the tree, T, and with a given attribute, a, we were to evaluate those attributes in the subtree to the left of a in the ranking, but not including those attributes in the subsequence associated with a, then we would be able to evaluate the attributes in the subsequence associated with a, and we would be able to do so in order from left to right. This is trivially true for the attributes associated with a terminal, for all such attributes are synthetic, and already defined. Assume it is true for all the offspring of T. We now desire to show that it is true for T. This is easily shown. In forming the subsequences associated with the attributes of T, we preserved the ordering of the subsequences of the attributes of the offspring of T from which they were formed. Thus, for each subsequence which is part of the sequence of attributes which is associated with a, we can evaluate the attributes in the subsequences, in order, from left to right. Putting these left to right evaluations together, in order, we can evaluate the newly formed sequence, in order, from left to right. Thus, this property will hold for all the nodes in the tree, and in particular, for the root. The root, however, contains all the attributes in the entire tree, all nicely ordered. If these can be evaluated from left to right, then we are surely safe.

In the third pass, we do not actually perform the concatenation operations outlined above. We simply keep track of the length of the subsequences that would have been formed, had we actually done so. In this way, the fourth and final pass knows how many attributes there are in the tree, and how long each subsequence is. It is then a simple matter to assign the numbers to each attribute which define the ranking. If the total number of attributes in the tree is N, then we assign the last attribute, available at the root, the number N. The attributes of the root which precede T can be numbered simply by subtracting the length of the subsequence to their immediate right, from the position of the attribute of the root to their immediate right. The positions of the offspring of the new root can be assigned in a similar fashion, and so on, throughout the entire tree. The attributes have been ranked. An example of these operations is given in appendix A.

While the algorithm has been given in a form which assumes the entire parse tree is available, it can easily be reformulated into 4 separate passes, where the first pass is done in conjunction with the original parse. Each pass requires only a stack as deep as the parsing stack, and can leave the rest of the data on secondary storage.

SPECIALIZING THE RANKING ALGORITHM

The ranking algorithm, as given, can be made more efficient, at the cost of some generality. If we restrict ourselves to the class of attribute grammars which Kennedy[7] calls "absolutely noncircular", then passes 1 and 2 can be eliminated. This class of attribute grammars is the class which Knuth's original, (but incorrect) circularity test would decide were noncircular. The absolutely non-circular attribute grammars are a proper superset of the attribute grammars which can be dealt with by Jazayeri or Bochmann. The methods of Jazayeri and Bochmann will, in general, prove desirable, (more efficient,) in those cases where they are applicable.

The only function of the first two passes is to decide upon a total order which can be associated with the attributes of each production. If we could decide upon a total order for each production in advance, then we would not need to compute the total order for each particular input string. We note that the existence of a total order on the attributes of a production, for all productions, is equivalent to the existence of a total order for the attributes associated with a symbol, for all symbols. The test for determining if an attribute grammar is absolutely noncircular does not impose a total ordering on the attributes of each symbol, but we can add additional artificial "semantics" which will accomplish this goal. The statements: "This attribute grammar is absolutely noncircular" and "We can impose a fixed total ordering on the attributes associated with each production, which is consistent with their semantic evaluation" are equivalent. Actual computation of a fixed total order for each production is not difficult.

EVALUATION OF THE ATTRIBUTES

The result of the ranking algorithm is a ranking of the attributes. Once the attributes have been ranked, it is necessary to sort them. It is not the purpose of this paper to consider methods of sorting a set of ranked items, such techniques are well covered elsewhere. We assume, therefore, that the attributes have been sorted, by some technique, and are now available to us in sorted form. It is now necessary to evaluate them.

The simplest algorithm of which we can conceive is to evaluate them in order, from left to right, holding the evaluated attributes in memory until they are used. Needless to say, this might prove expensive in terms of primary memory requirements, because we might have to hold all the evaluated attributes in memory at once. The next approach we might take is to somehow organize the attributes so that we can use secondary storage in an efficient manner, i.e., to adopt a multi-pass algorithm, as is done by Jazayeri and Bochmann. If we insist that the algorithm handle an arbitrary attribute grammar, then our options become narrower. Rather than give the algorithm directly in terms of attributes, we shall give it in more general terms, and then show its application to attribute grammars.

EVALUATION OF GENERALIZED ADDRESS TRACES

The first concept that we must develop is that of the generalized address trace. In a normal address trace, a sequence of (binary) operations is specified on a set of operands. The operands are simply the words of a computer, i.e., for the 6400, the operands of an address trace would be 60 bit words. In a generalized address trace, we will adopt the convention that arguments can be arbitrarily complex objects, but have bounded size. We also assume that these named objects can be operated on by a fixed set of arbitrary functions, which accept an arbitrary number of arguments, instead of the usual binary operators normally found in an address trace. The crucial concepts are that we are dealing with an address space, and that within that address space are named (or addressable) objects. The named objects are operated upon by a fixed sequence of operations, whose arguments are specified by names (addresses) within the address space. An example of such a generalized address trace would be the following:

```
A=I
B=PLUS(A,I)
C=TIMES(A,B)
D=SQUAREROOT(C)
E=TIMES(D,C)
B=TIMES(E,A)
```

C=PLUS(B,E)

The operations in such an address trace are fixed, as are the names of the operands, and the sequence of operation applications. The actual arguments, however, need not be known.

Given an arbitrary generalized address trace, it is possible to evaluate that address trace, i.e., to assign values to the variables in accordance with the specified operations. The algorithm that will follow will do this in time N log N, with primary memory requirements that are small, and with a disk as secondary storage device. The time bounds are those involved in shuffling the data around, and do not include the time that might be spent by the operations themselves. The N is more closely related to the total memory requirements of the operands in question. This assumes that it is possible to fit the few operands necessary to evaluate a single result into primary memory at one time, i.e., the individual operands are small, compared with the primary memory size.

We first make the following observations. An operation is performed on a (small) set of operands, and produces a result. The operands are referenced, and the result is produced. The result, and some of the operands, will subsequently be referenced again. (We discount the case in which a result is produced, but never referenced.) If we number the applications of the operations, in order, from the first, then we can assign a "time" to the next reference. Thusly:

```
1      A=PLUS(2,5)
2      B=TIMES(A,8)
3      C=SQRT(B)
4      B=PLUS(C,A)
```

Operand A is produced at time 1. It is next referenced at time 2. It is then referenced again at time 4. For every reference, it is possible to produce the time of next reference, if any. In the worst case, if we simply keep every named object in primary memory from the time it is produced until the time it is last referenced, we might occupy an amount of memory proportional to the length of the input string. Our desire is clear: we wish to arrange matters so that the named objects that we desire to use for the next operation are in memory, while the objects that we are not interested in are out somewhere on disk. How can we arrange this?

The first observation that we make is this: objects are referenced by different operations at different times. Ideally, we should like to take the objects, sort them by time of reference, and then apply the operations to them. (If an object is referenced several times, we can simply make several copies of it, and sort each copy in accordance with the time at which it is referenced.) Unfortunately, we can't sort the objects until they've been generated, and we can't generate them until we've got them sorted. What we need is an incremental sort that will allow us to do a little sorting, which will bring the operands we're interested in to the correct place, then we do a little evaluation, which will let us generate some new objects, then we do a little sorting, which will position the newly generated objects, and so forth. It just so happens that there is a sorting method which can be made to have exactly this property.

Consider the following method for sorting the numbers between 1 and 1024. First, we create two buckets, B(1,512) and B(513,1024). Leaving B(513,1024) alone, we divide B(1,512) into two new buckets, B(1,256) and B(257,512). Again, we divide the first bucket into two, while leaving the second bucket alone. B(1,256) becomes B(1,128) and B(129,256). This process continues, until we have the following buckets:

B(1,2)
B(3,4)
B(5,8)
B(9,16)
B(17,32)
B(33,64)
B(65,128)
B(129,256)
B(257,512)
B(513,1024)

To sort, all we need do is place incoming items into one of the log N buckets. If we should chance to desire the next item in the ordering, and if that item has been passed through the sorting sequence, then it is available in the first bucket, B(1,2). As we remove items from the sorting sequence, the structure of the buckets will have to change. Thus, if we have removed items 1 through 16, then the buckets will look like this:

B(17,32)

B(33,64)

B(65,128)

B(129,256)

B(257,512)

B(513,1024)

At this point, it is clear that the bucket structure has broken down, and must be reorganized before we can proceed further. We do this by taking B(17,32) and breaking it into two buckets. B(17,24) and B(25,32). B(17,24) is, in its turn, broken into two buckets, B(17,20) and B(21,24). B(17,20) is then broken down into B(17,18) and B(19,20). At this point, the process stops. Our original bucket structure has been recreated, except that we now have no bucket of size 16. That bucket, B(17,32), was broken down into a sequence of smaller buckets.

B(17,18)

B(19,20)

B(21,24)

B(25,32)

B(33,64)

B(65,128)

B(129,256)

B(257,512)

B(513,1024)

As the process of simultaneous sorting and removal of sorted items continues, the bucket structure will change again and again. The next items required will always be available at the front of the first bucket. Needless to say, in an actual implementation, the first several buckets will be grouped into one bucket, held in primary memory. The rest of the buckets will be on disk, with only a buffer for each bucket in primary memory.

What is the running time of this process? We can estimate the running time by observing the passage of attributes from bucket to bucket. Initially, an attribute will be tossed into some (more or less random) bucket. (Actually, there will probably be locality effects. When an attribute is produced, it will tend to be placed in one of the next few buckets, because it will be referenced quickly. We ignore this for now, and consider only the worst case.) At irregular intervals, the bucket an attribute is in will be split in two. Eventually, the attribute will be used as an argument, and the process will be terminated. It is obvious from this description that a single attribute can pass through no more than log N buckets. Therefore, the running time is order N log N. (N can be either the total number of attributes, the total number of nodes in the parse tree, or the total length of the input string, because all of these quantities are within a constant factor of each other.)

If there are log(N) buckets, and each bucket has a fixed size buffer in main memory, then the main memory requirements are log(N), while it was promised to do the operation in a fixed primary memory. This can be done by grouping the last several buckets together. For example, the first 7 buckets might have representative buffers in primary memory. The 8th and succeeding buckets would be grouped together, and would all share a single buffer in primary memory. When the 1st through 7th buckets had been exhausted, and we desire the first item from the 8th bucket, we simply take all the items we've placed into the overflow bucket, which holds the items that would have gone into the 8th and succeeding buckets, sort them, and then distribute them among the 8th and succeeding buckets according to the rules previously defined. A given item would go into this overflow bucket only once, and would be sorted, along with other items in the overflow bucket, only once. Because the total number of items in the overflow bucket is bounded, (because only a bounded number of new attributes can be produced by evaluating the bounded number of items in buckets 1 through 7,) the sort will require a bounded amount of effort. This would add a fixed additional overhead per item. The resulting method, while slower, will run in a fixed primary memory, as promised.

The method described has applications beyond attribute grammars. In essence, it describes a set of computations which can be done in a very small main memory. Any computation which can be described by a fixed sequence of operations upon a set of operands of bounded size, can be done with a small main memory, and in time N log N. (Again, both the time and memory requirements ignore the time and memory requirements of the operations themselves. These time and space bounds represent the additional overhead for operand shuffling.) Two observations come to mind:

1.) It looks good. Someone must have thought of it already.

2.) Just how far is it possible to push this result, i.e., what computations actually fall into the class described?

Both of these points deserve further research.

Some comments can be made at once, however. First, computations involving pointer structures would not fall into this category. This is because the names of the operands are not known in advance, but are, instead, computed as they are needed. Secondly, this would appear to be a new result concerning the minimal memory requirements for a given address trace, given complete foreknowledge of the addresses involved. This question is of some interest in operating systems theory.

A more detailed description of the algorithm can be found in appendix B.

APPLYING THE GENERALIZED ADDRESS TRACE

The application of the method of evaluating a generalized address trace to the evaluation of a set of ranked attributes should be obvious. What might not be obvious is that, now that we can evaluate a generalized address trace, we can make some changes to the concept of an attribute grammar which will result in simplified evaluation. We give a simple example of such a change below.

Our change consists of defining two semantic functions, Hash(table,name,value), and Lookup(table,name). These two semantic functions perform the obvious operations on a hash table. If we assume that "table" is an attribute, and "name" is a constant, (a fixed sequence of characters, for example), then we can integrate these semantic functions into our evaluation method quite easily. The 2-tuple, (table,name), constitutes an "address", and can be considered as such in the evaluation of the generalized address trace. If this is done, then the actual "table" need never exist. Instead, the items in the table are tied together by the mechanism which deals with the generalized address trace. The advantage of this is simple: We have placed a bound on the size of a given attribute. This bound will be most felt when dealing with hash tables, which we wish to be of unbounded size. If a hash table is implemented as suggested above, then there is no bound on its size.

TIME INDEPENDENT VARIABLES

The author would like to take this opportunity to discuss why attribute grammars are powerful, and a different method of obtaining the same power. The basic reason that attribute grammars are as flexible and convenient as they are is just this: they free the implementor from the need to collect information scattered all over the derivation tree into one place. This collection function is taken over by the attribute grammar. Now, a top down recursive descent compiler provides a great deal of flexibility in language design, but it has one glaring flaw: you can't find out what's going to be read in, until you've read it in. To call a subroutine which has not yet been defined (a forward reference) can cause the compiler to choke. (This fact is recognized by Wirth in the design of Pascal. He dealt with this problem by the simple expedient of making it illegal.) The usual method of getting around this problem is to define a two-pass compiler. The first pass collects the forward reference information that the second pass will need to know about. In an attribute grammar, such "forward references" can be dealt with trivially. How else could we deal with this problem?

If we adopt the viewpoint that we wish to add some extension to the normal semantics of a top down recursive descent compiler, then our problem can be described thusly: we wish to know, now, information that we cannot possibly learn until some time in the future. Our solution, given this problem statement, is equally simple: we define a new type of variable in which information can be carried from the future to the past. This can be done by adding the following semantics:

1.) There is a special value, "undefined".

2.) Any variable can be assigned the value, "undefined".

3.) If a variable is referenced, and the current value of the variable is undefined, then the value used will be the next value assigned to that variable.

The problem of forward references can now be dealt with easily. If we wish to know, now, about a value which we will discover in the future, i.e., after scanning more of the input, we proceed in the following fashion:

```
      wishtoknow := undefined;
         .
         .
       code which uses wishtoknow
         .
         .
 Comment:  we finally scan the information we wish to know;
       wishtoknow := scan(input);
```

It might be objected that implementation of this concept will prove inefficient. If implemented in full generality, this is true. If we make a few restrictions, then implementation becomes easy. We divide all variables into two types, normal variables, and time independent variables. The normal variables can be used in any way we desire. The time independent variables, however, can not be used to define normal variables, nor can time independent variables be used to change the flow of control of the program, nor can time independent variables participate in pointer structures. Time independent variables can depend, in arbitrarily complex ways, on each other, and on normal variables. With these conventions, implementation becomes simple. We execute the program once, with normal variables behaving in the normal fashion. We do not evaluate any of the time independent variables. Instead, we build a data structure which shows which time independent variables depend on which other time independent variables, and what functions have to be applied to evaluate them. In this structure, the time independent variables will resemble attributes, while the functional dependencies among them will resemble the semantic dependencies among attributes. Once we have completed the construction of the data structure which shows the functional dependencies among the time independent variables, we proceed to the evaluation of them. This process is akin to the related process for attributes. In fact, the same algorithms can be adapted. Attribute grammars and time independent variables are similiar in terms of their power and complexity. The reader is left to contemplate programs which ignore time.

CONCLUSION

We have devised an algorithm for the efficient evaluation of an arbitrary attribute grammar. The general method for evaluation can be specialized, with a resulting increase in efficiency, but a corresponding decrease in generality. A significant sub-result was the creation of an algorithm which can efficiently evaluate a generalized address trace. The broader applicability of this algorithm deserves further research. A usually linear test for circularity was given.

APPENDIX A

One picture is worth a thousand words. These pictures (following page 66) are for the example taken from Knuth[1]. The example was also used earlier (page 14).

The action of the four passes will be exemplified by the changes in the dependency graph. Pass1 adds additional arcs to the dependency graph. Pass2 adds still more arcs. Pass3 assigns a number to attributes in the graph, while Pass4 assigns a rank to each attribute in the graph.

The dependency graph, with the direct semantic dependencies illustrated, is shown as "Before Pass1".

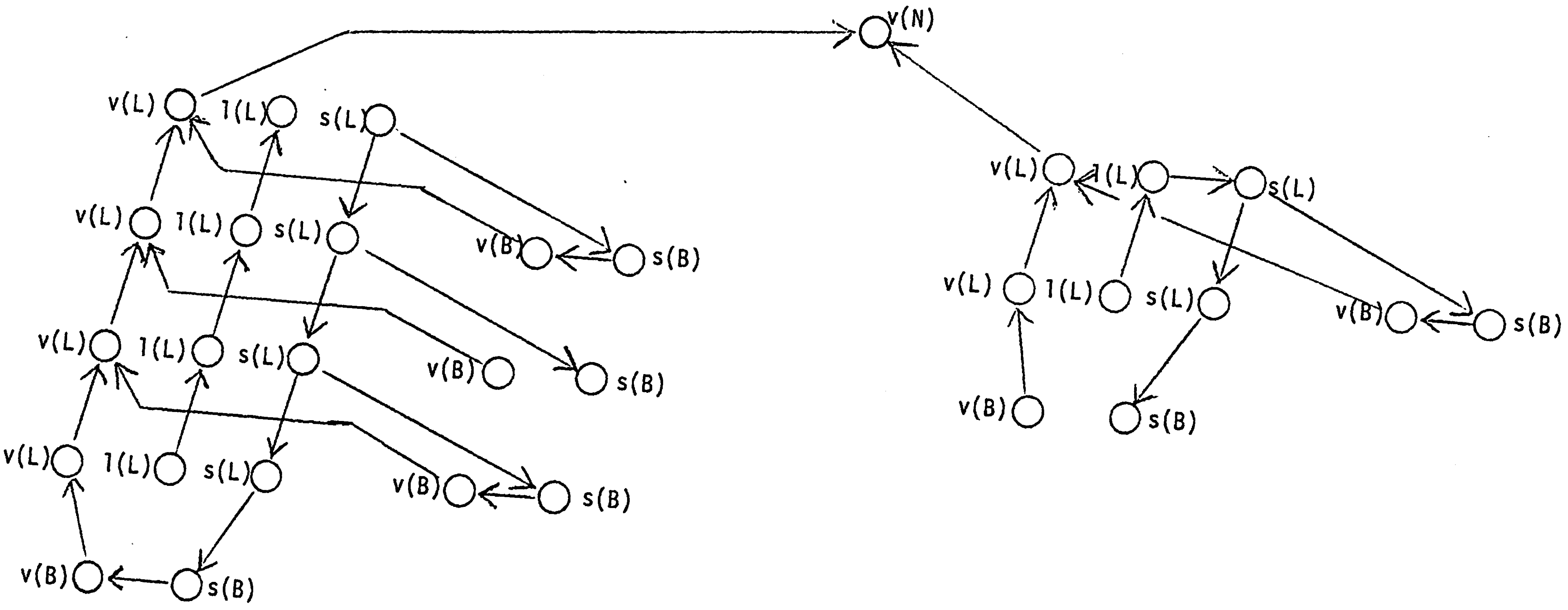
Pass1 makes an upwards sweep of the dependency graph, from the leaves to the root. (Trees are oriented upside down.) The result is shown in "After Pass1". Notice that all dependency information that can be obtained from a subtree has been summarized in the attributes at the root of the subtree.

Pass2 makes a downward sweep of the dependency graph, and imposes a total order on the attributes associated with a production. Because showing all the newly added arcs would leave an illustration bristling with arrows, only the arcs leading from a node to its direct successor in the ordering are shown.

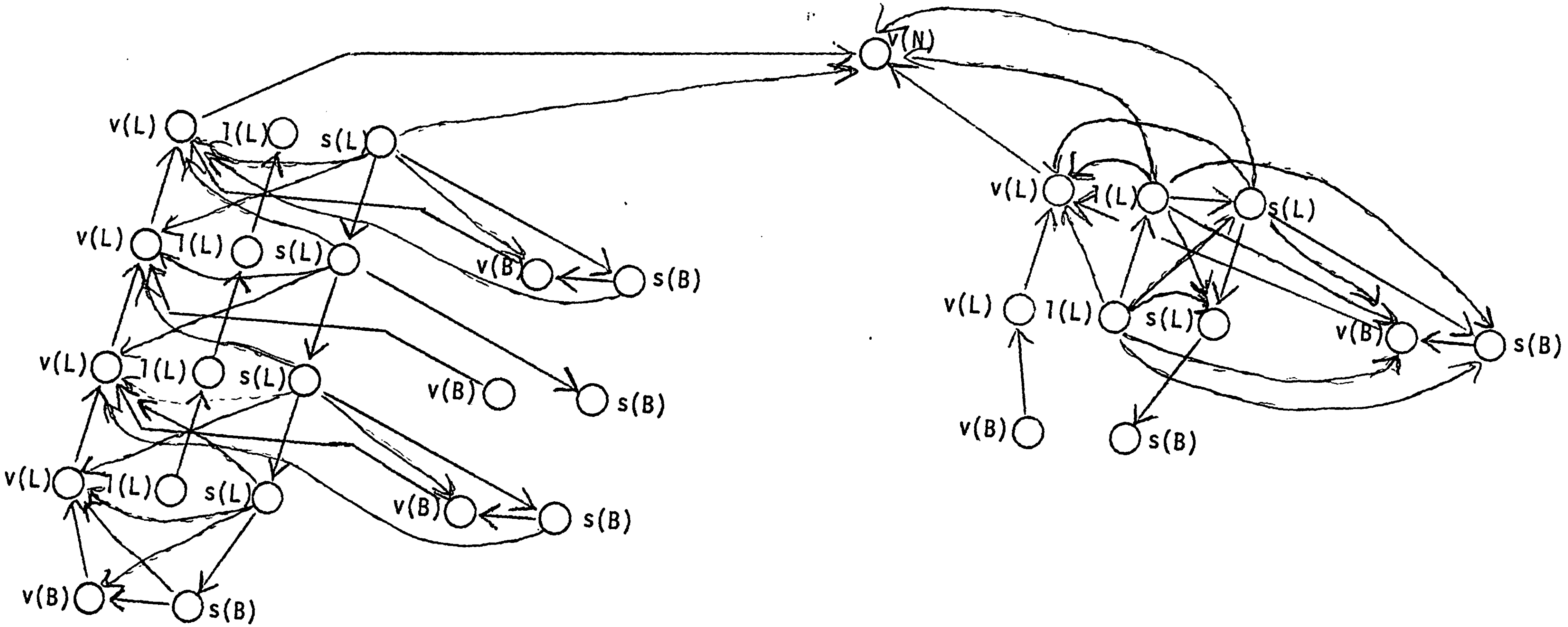
Each attribute in the picture "After Pass Two" is associated with two different productions: the production above and the production below. Therefore, each attribute appears in two different orderings. The first is the ordering of all attributes of the production above, in which the particular attribute is in the right part. The second is the ordering of all attributes of the production below, in which the attribute appears in the left part. The numbers in the circles go with the ordering of the production above, while the numbers below the circles go with the ordering of the attributes in the production below.

Pass3 makes an upward sweep, and counts the number of attributes from each subtree which will directly preceed the attributes of the root of the subtree. The numbers shown in "After Pass3" are the pred field of each attribute.

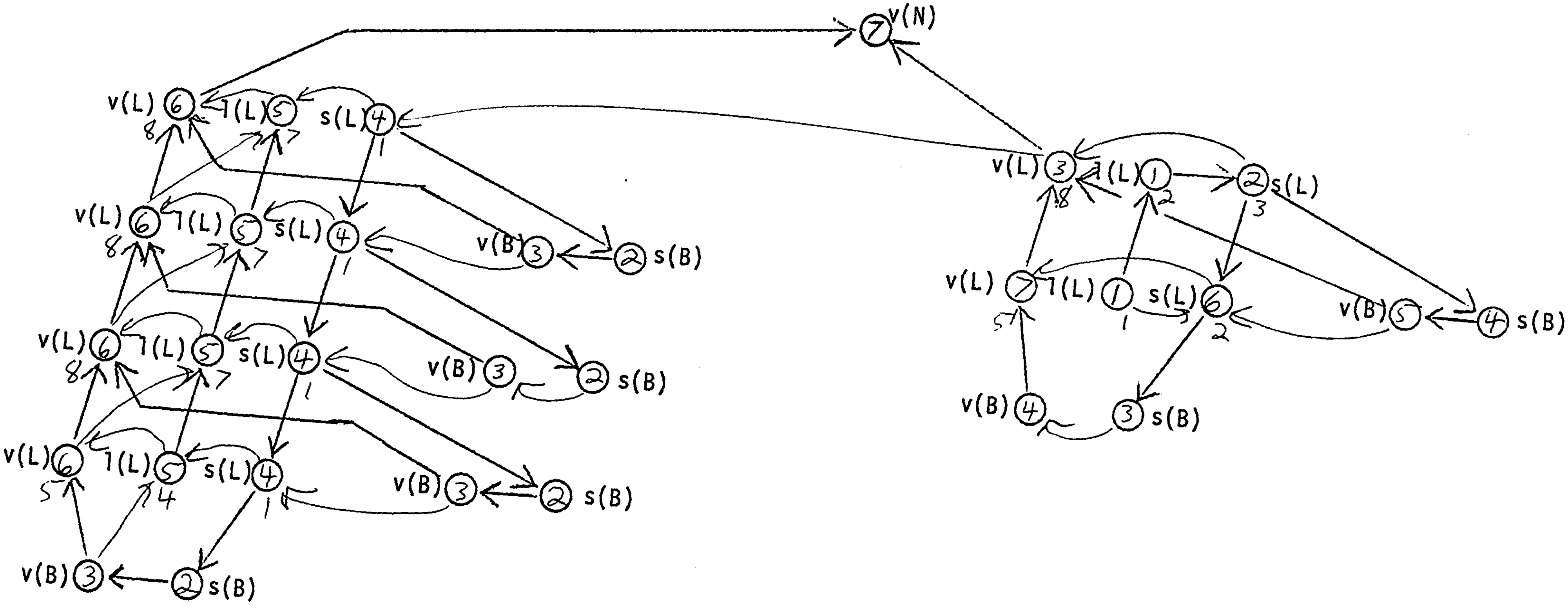
Pass4 makes a downward sweep, and computes the rank of each attribute. The numbers shown in "After Pass4" are the position field of each attribute. The position of the attributes of the right part of a production are computed from the known positions of the attributes in the left part.



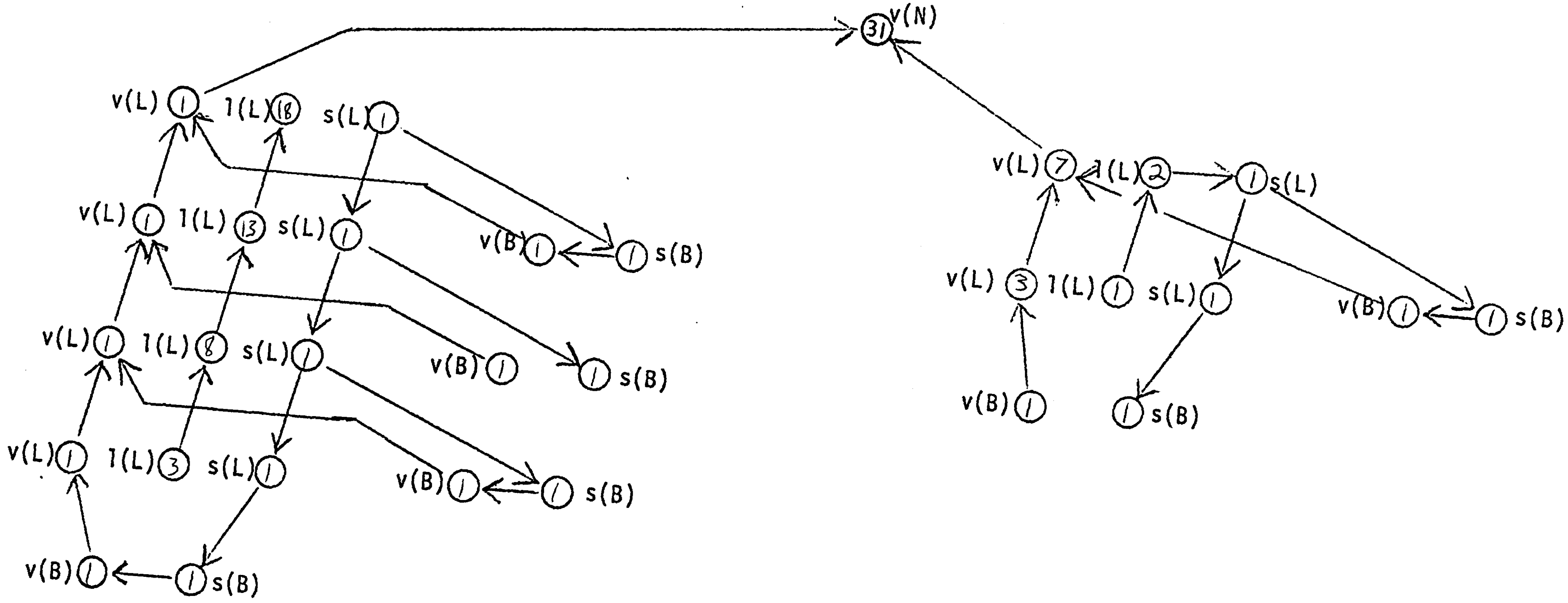
BEFORE PASS ONE



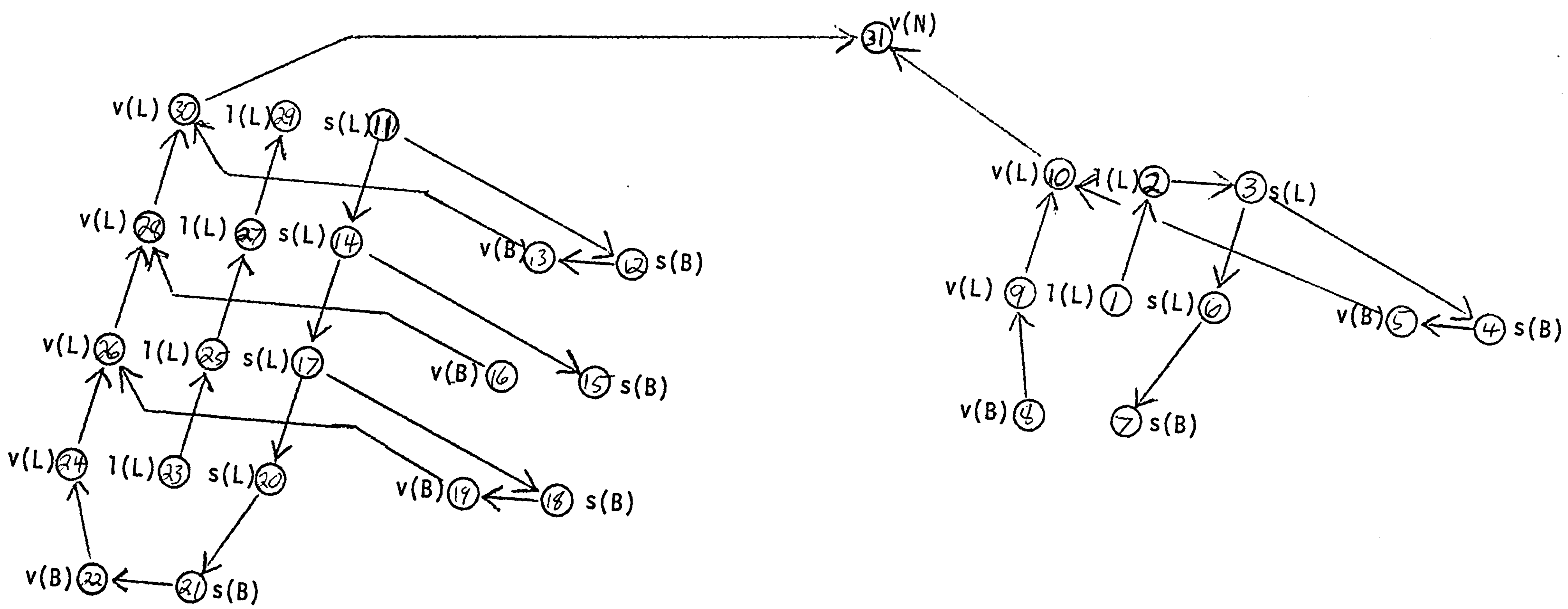
AFTER PASS ONE



AFTER PASS TWO



AFTER PASS THREE



AFTER PASS FOUR

APPENDIX B

In the following section, we give a detailed description of the sorting procedure briefly described on page 55 et sequitur.

The important points of this algorithm are:

1) items can be inserted randomly.
2) items are fetched sequentially, in order.
3) an item can be fetched any time after it has been inserted. Both fetching and inserting can proceed at the same time.
4) this sorting algorithm runs in time N log N.
5) This sorting algorithm can use a small main memory, and a disk drive (or log N tape drives) as a secondary storage device.

The algorithm is given in pseudo-Pascal.

The "Initialize" procedure is responsible for initializing data structures. In particular, it empties all buckets, and indicates the range of numbers that fall into each bucket. The ith bucket contains numbers in the range Lower[i] to Upper[i].

The range of the buckets are initialized to:

1 to 2
3 to 4
5 to 8
9 to 16

17 to 32

etc. etc. etc.

The procedure, "Fetch", takes the next sequential item from the data items that have been put into the bucket structure via the "Put" procedure. If the next data item is not in the first (smallest) bucket, then an error condition has occurred. If the smallest bucket has more than two items in it, then it is split in two. The smallest bucket continues to be split in two, until it has only one or two items in it. The next sequential data item is then taken from the smallest bucket, and returned.

The procedure, "Put", is called with the data items which are to be put into the bucket structure.

It is assumed that a data item is always put into the bucket structure before it is fetched.

```
Constant LogN: the log to the base 2 of N, where
               N is the number of data items.
Type     dataitem = record rank:integer ; value:any end
Var  current:integer;
     next:integer;
     Lower: array[1..LogN] of integer;
     Upper: array[1..LogN] of integer;
     Bucket: array[1..LogN] of bag of dataitem

Procedure Initialize;
Begin
```

```
  current:=1;
  next:=1;
  For I:= 1 to LogN Do
   Begin
    Lower[I]:=2^(I-1)+1;
    Upper[I]:=2^I;
    Bucket[I]:= {};
   End;
End;


Procedure Fetch;
  Comment given the global integer variable,
        next, which specifies the rank
        of the item which it is desired to fetch, this
        procedure will fetch it.  The variable "next"
        is incremented by 1 which means that
        data items will be fetched in sequence, i.e.,
        "next" will take on the values 1,2,3,4,... in that
        order;
Begin
  If next > N Then
      Print "error not enough data items."
   Else Back:Begin
    If next < Upper[current] Then
    Begin
     Comment next is in the current bucket.  Just
```

```
            return it and have done;
  next:= next+1;
  Return(Bucket[current][next-1])
 End;
Comment next is not in the current bucket, so start
        splitting big buckets into little buckets;
current:= current+1;
While Upper[current]-Lower[current] > 2 Do
 Begin
   Comment split the current bucket into two new
           buckets, both of half the size.  The
           new buckets will be current, and current-1;
   Lower[current-1]:=Lower[current];
   bucketsize:=Upper[current]-Lower[current]+1;
   Upper[current-1]:=Lower[current-1]+bucketsize/2-1;
   Bucket[current-1]:={};
   Lower[current]:=Upper[current-1]+1;
   For y in Bucket[current] Do
   If Lower[current-1] <= y.rank <= Upper[current-1] Do
     Begin
      remove y from Bucket[current];
      Add y to Bucket[current-1];
     End;
  End;
  current := current-1;
 Comment now that we have a little bucket, go
```

```
            back and fetch the requested data item;
    Goto Back;
End;



Procedure  Put(x:dataitem);
Comment  this procedure takes a data item and puts
       it into the proper bucket;
Begin
 For I:= current to LogN Do
  If Lower[I] <= x.rank <= Upper[I]
     Then
      Add x to Bucket[I];
 End;
```

BIBLIOGRAPHY

1. SEMANTICS OF CONTEXT-FREE LANGUAGES, by Donald E. Knuth, Mathematical Systems Theory, Vol. 2, No. 2(1968) pp. 127-145.

2. THE ART OF COMPUTER PROGRAMMING, Vol. 1, FUNDAMENTAL ALGORITHMS, by Donald E. Knuth, pp. 258-268.

3. SEMANTICS EVALUATED FROM LEFT TO RIGHT, by Gregor V. Bochmann. Comm. of the ACM, Vol. 19, No. 2, Feb. 1976.

4. ON ATTRIBUTE GRAMMARS AND THE SEMANTIC SPECIFICATION OF PROGRAMMING LANGUAGES, by Mehdi Jazayeri, October 1974, Report No. 1159 of the Jennings Computing Center at Case Western Reserve University.

5. THE INTRINSICALLY EXPONENTIAL COMPLEXITY OF THE CIRCULARITY PROBLEM FOR ATTRIBUTE GRAMMARS, by Mehdi Jazayeri, William F. Ogden, and W.C. Rounds. Comm. ACM 18,12 (Dec. 1975) 697-706.

6. THE THEORY OF PARSING, TRANSLATION, AND COMPILING, Vol. 2:COMPILING, by A. V. Aho and J. D. Ullman, pp. 758 et sequitur.

7. AUTOMATIC GENERATION OF EFFICIENT EVALUATORS FOR ATTRIBUTE GRAMMARS, by Ken Kennedy and Scott Warren, Dept. of Math. Sciences, Rice University, Houston, Texas, 77001. Revised version to appear in POPL proceedings.

8. FOLDS, A DECLARATIVE FORMAL LANGUAGE DEFINITION SYSTEM, by I. Fang, STAN-CS-72-329, Computer Science Department, Stanford University, December, 1972.

EVALUATORS FOR ATTRIBUTE GRAMMARS

by Ralph C. Merkle

RESEARCH PROJECT

Submitted to the Department of Electrical Engineering
and Computer Sciences, University of California, Berkeley,
to partial satisfaction of the requirements for the degree
of Master of Sciences, Plan II.

Approval for the Report and Comprehensive Examination:

COMMITTEE: ______________________________, Research Adviser

______________________ Date

____

______________________ Date