CUCS-165-85

A Comparison of Storage Optimisations in Automatically-Generated Attribute Evaluators

by

Rodney Farrow[1] and Daniel Yellin

Computer Science Department
Columbia University
New York, New York 10027

[1]EECS, Computer Science Division
University of California, Berkeley
Berkeley, CA 94720

revised version: 1 September 1985

This research was partially supported by the National Science Foundation under grant DCR-83-10930, and partially supported by the Defense Advanced Research Projects Agency under contract number N00039-84-C-0165

Table of Contents

List of Figures

ABSTRACT

Attribute grammars are a value-oriented, non-procedural extension to context-free grammars that facilitate the specification of translations whose domain is described by the underlying context-free grammar. Just as parsers for context-free languages can be automatically constructed from a context-free grammar, so can translators, called attribute evaluators, be automatically generated from an attribute grammar. A major obstacle to generating efficient attribute evaluators is that they typically use large amounts of memory to represent the attributed parse tree. In this report we investigate the problem of efficient representation of the attributed parse tree by analyzing and comparing the strategies of two systems that have been used to automatically generate a translator from an attribute grammar: the GAG system developed at the Universitat de Karlsruhe and the LINGUIST-86 system written at Intel Corporation. Our analysis will characterize the two strategies and highlight their respective strengths and weaknesses. Drawing on the insights given by this analysis, we propose a strategy for storage optimization in automatically generated attribute evaluators that not only incorporates the best features of both GAG and LINGUIST-86, but also contains novel features that address aspects of the problem that are handled poorly by both systems.

As the cost of raw computer power has plummeted, it has become a cliche to observe that there is a "crisis" in our ability to produce the software needed to effectively use that power. It is not only that more programmers need to be trained; even with an unlimited supply of programmers, the cost of producing the needed software mounts astronomically. One response to this problem is to change the way we "program". *Programming by specification* calls for solving problems by giving a rigorous specification of their solution, rather than by describing in detail how they are to be solved. Of course this puts a much larger burden on the computer system, especially its software, to calculate this solution, and it is not always either possible or feasible. However, in some problem domains we know enough that this technique can be used. One widely-appreciated but non-trivial example of this is the use of parser-generator programs to automatically construct parsers from a context-free grammar that describes the strings to be recognized. The history of the development of these parser-generator programs shows that even after context-free grammars were recognized as an effective description, much research was still needed before efficient, widely-usable parsers could be automatically built.

More recently, attribute grammars have been proposed as an appropriate means of describing still more of the tasks of translation and compilation. As was the case for parser-generators based on context-free grammars, much research is needed in order to build efficient attribute evaluators that are competitive with hand-coded translators. One of the most serious obstacles to implementing attribute evaluators is that they use enormous amounts of space to represent the attributed parse tree. Several optimizations have been proposed in the literature and variations of some of these have been implemented in experimental translator-writing-systems based on AGs. In this paper we examine in detail the storage optimization strategies of two such systems that have been used to generate substantial compiler front-ends: GAG [10] developed at the University of Karlsruhe, and LINGUIST-86 [2] developed at Intel Corp.

Section 1 contains a brief introduction to attribute grammars, tree-walk evaluators and storage optimization for tree-walk evaluators. Sections 2 and 3 give an overview of GAG's and LINGUIST-86's storage allocation policies, respectively. Section 4 describes the effect of these two sets of storage optimizations on evaluators generated from different attribute grammars for Pascal. Important similarities and differences between the two systems are noted. The differences between the two systems often suggest how the various optimizations can be improved and some of these improvements are briefly described. We also explore how the more effective techniques of one system can be incorporated into the other. Finally, section 5 contains some suggestions for new attribute storage optimizations that

complement those done by GAG and LINGUIST-86 but that are not currently addressed by either system. The appendix to this paper proves an important storage optimization problem to be NP-complete.

1. Introduction to attribute storage optimization

Many attribute grammar (AG) evaluators which have been developed employ a tree-walk evaluation strategy [18]. For any source language input, this strategy builds an explicit semantic tree: a parse tree in which each node is labeled by its corresponding symbol X and each node contains fields (*attribute-instances*) corresponding to the attributes of X. Translation into the target language is performed by *walking* over this explicit tree evaluating the attribute-instances. After all attribute-instances have been evaluated the translation resides in a distinguished attribute of the root. At any moment during the evaluation of the semantic tree, the *locus-of-control* of the evaluator resides at some particular production-instance in the semantic tree. It can choose to execute either an $EVAL_{X.att}$ instruction or a $VISIT_k$ instruction. An $EVAL_{X.att}$ instruction calls for the evaluation of an instance of X.att of the current production-instance. A $VISIT_k$ instruction calls for the evaluator to move its *locus-of-control* to an *adjacent* production-instance; i.e., if $k > 0$ then a $VISIT_k$ instruction causes it to move to the k^{th} son and if $k = 0$ it moves to its parent. Besides the $EVAL_{X.att}$ and $VISIT_k$ instructions the only other instructions used by a tree-walk evaluator are those used to determine the flow of control.

Many tree-walk evaluation strategies have been devised. One of the simplest is an *alternating-pass* strategy that makes depth-first left-to-right and depth-first right-to-left passes over the semantic tree. A more flexible strategy is that of *ordered* evaluation, which tailors the traversal over the semantic tree to the particular attribute grammar at hand. Both of these strategies can be implemented by several different methods, such as: sets of recursive procedures, sets of coroutines, or stack automata [9].

As semantic trees can be very large, it is important for tree-walk evaluators to conserve as much storage as possible. Several techniques have been developed to accomplish this. First of all, the attribute-instances of the semantic tree should contain only pointers to complex objects and not the values themselves [16, 13, 3, 14]. We shall assume that every tree-walk evaluator does so and we shall not discuss this optimization further.

Another technique is to implement the instances of an attribute as some separate data structure, rather than as components of semantic tree nodes. The two possibilities that are used by GAG and LINGUIST-86 are implementing the instances of an attribute as the

values on a *stack*, and as the contents of a single, statically-allocated variable. Such *attribute storage optimizations* are done extensively by GAG and LINGUIST-86, and they are quite effective. Our purpose herein is a detailed analysis of this variety of optimization.

Saarinen first suggested [15] taking as many attributes as possible out of the semantic tree and putting them into a stack. To this end he distinguished between *significant* attributes and *transient* attributes[1] . A *transient* attribute is one whose lifetime consists of a single visit to a production-instance from its parent; all other attributes are *significant*. If X.att is a transient attribute it can be implemented as a stack, as will be illustrated later, so that storage need not be allocated in the semantic tree for <u>any</u> instance of X.att. When another instance of X.att is defined it is PUSHed onto the stack; it is POPped off the stack when it is no longer needed. Although an instance of X.att will take up stack space, it will only do so for the duration of its lifetime.

Another possibility is to implement instances of an attribute as the value currently in a global variable. Not all attributes can be implemented this way, but many can, thus further reducing the size of the semantic tree. This optimization was investigated by Ganzinger in a formal setting [6]. He found that the problems involved were similar to those for register allocation.

An especially common kind of semantic function are those of the form, [Y.att1 = Z.att2]. These are called *copy rules*. If Y.att1 and Z.att2 are merely different occurrences of the same attribute, say X.att, and if X.att is implemented as a global variable, then the copy rule just copies the value of the global variable onto itself. Such a semantic function can be eliminated. If X.att is implemented as a stack then the copy rule calls for duplicating the top of the stack and then later popping off the second copy. Often these copy rules can be eliminated also, thus avoiding PUSH and POP operations on the stack and keeping the stack from growing as deep. When copy rules are eliminated like this we say they have been *subsumed* by the storage optimization policy.

2. GAG

GAG [10] uses the ordered evaluation strategy which it implements using stack automata. GAG first computes the order in which the attributes of a symbol will be evaluated and uses this information together with each production's dependency graph to compute *visit sequences* for each production. The *visit sequence* VS_p of a production p gives the order in

[1] these were refered to in [15] as *temporary* attributes

which attributes of the production are evaluated and right-part nodes visited. It consists of an ordered list of EVAL and VISIT instructions (for a tree-walk evaluator) that are to be executed for this production. Each production has one visit sequence that is used for any instance of that production in any semantic tree.

After fixing an evaluation order GAG decides how to implement the attributes by choosing one of 3 possible storage mechanisms for each attribute; an attribute is either: implemented as a *global variable*, implemented as a *global stack*, or allocated space in the semantic tree. If the latter mechanism is used for the attribute X.a then every instance of X.a in the semantic tree will be allocated its own storage cell. Otherwise all instances of X.a in the semantic tree will share either 1 storage cell (if X.a is implemented as a global variable) or 1 stack (if X.a is implemented as a global stack). An attribute is transient only if all its occurrences are transient. An attribute-occurrence, X_i.att, is transient only if there is no $VISIT_0$ in the visit sequence for its production between any two EVAL instruction that define or reference X_i.att. For purposes of this definition, if an inherited attribute X.a is defined before the j^{th} visit to X, then the visit sequence for every production p having X as its left-part symbol is assumed to have an implicit reference to X_0.a before the $EVAL_{att}$ and $VISIT_k$ instructions pertaining to the j^{th} visit to p. Similarly, if a synthesized attribute X.a is defined during the j^{th} visit to X, then the visit sequence for every production p having a right-part occurrence of X, say X_m, is assumed to have an implicit reference to X_m.a before the $EVAL_{att}$ and $VISIT_k$ instructions following the j^{th} visit to X_m. Only transient attributes are implemented as global variables or global stacks by GAG.

2.1. Global Variables and Global Stacks

The *lifetime* of an attribute-instance in a semantic tree is the period of time between its *computation* (i.e. instantiation) and its last *application* (i.e. reference). For tree-walk evaluators, the lifetime of any attribute-instance N.att can be expressed as a pair of visit numbers $(att_{initial}, att_{final})$, where $att_{initial}$ is the number of the visit to N during which N.att is first computed, and att_{final} is the number of the visit to N during which N.att is last applied. After an attribute-instance's last application we no longer need to save its value as its role in the translation process is completed.

After the evaluation strategy has been fixed GAG analyzes the lifetime of each transient attribute to see whether it can be a global variable or global stack. If it discovers that, in *any* semantic tree, the lifetime of *any* instance of an attribute X.a cannot overlap the lifetime of any other instance of X.a, then it implements X.a as a global variable. This means that X.a is not allocated space in the semantic tree; rather, all X.a values are stored

in a global variable, call it X_a. Any copy rules between different X.a instances then look like x_a = x_a; these are all eliminated.

If X.a cannot be made into a global variable but GAG discovers that, in *any* semantic tree, the lifetimes of any two instances of X.a are either *disjoint* (the requirement for a global variable) or *properly nested* then GAG implements X.a as a global stack. N2.a's lifetime is *properly nested* in N1.a's lifetime if N2.a's entire lifetime is contained between two consecutive applications of N1.a when evaluating the semantic tree. If X.a is implemented as a global stack then, whenever a new instance of X.a is defined its value is pushed onto the top of the stack. Upon last application of that attribute-instance its value is popped off the stack. Any use of the value of this attribute-instance is translated into a reference to the top element of the stack. In between 2 consecutive references to this instance of X.a, another instance of X.a may have its value pushed onto and popped off of the stack.

The way in which attributes are implemented as a stack is different for inherited attributes than it is for synthesized attributes. At any time during attribute evaluation the inherited attributes that are on a stack are attributes of nodes that are ancestors of the current node in the semantic tree. At the same time the synthesized attributes that are on a stack are attributes of nodes that are siblings of ancestors, rather than the ancestors themselves. Figure 2-1 illustrates the difference. This is similar to the different stack-contents of a top-down parser versus a bottom-up parser.

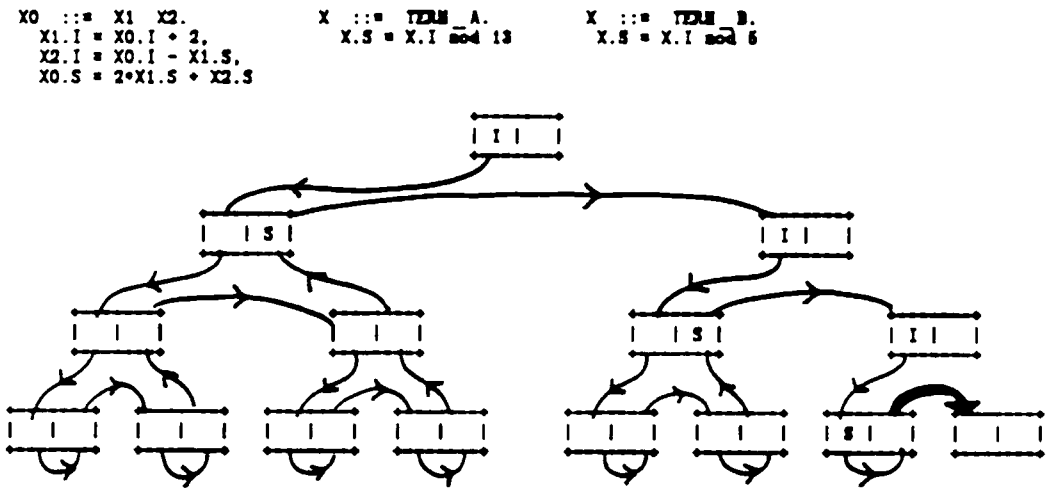


Figure 2-1: What attributes are on the stack? synthesized vs. inherited. The path indicates evaluation order; labelled attribute-instances are on the stack when the evaluators *locus of control* is at the tip of the path.

GAG assumes that no production contains a reference to a right-part inherited attribute-occurrence for any potentially stackable attribute X.I. Consequently, the last reference to

such an X.I is during the VISIT to that X-node and hence must correspond to a reference to a left-part occurrence of X.I in some other production. Thus, X.I is pushed onto the stack in one production, where it is a right-part occurrence, and popped off the stack in a different production, where it is a left-part occurrence. In order for this work correctly, all productions that have X as their left-part symbol must pop X.I before finishing that VISIT. When a right-part X.I is defined by a copy rule from a left-part X.I (i.e. `x[1].I = x[0].I`) we would like to not duplicate the current top of the stack; i.e. not push a new value onto the stack and then not pop it off later. This would save some code and keep the stack from growing. Unfortunately, this copy rule can be only be eliminated in some cases: when the production that contains the copy rule makes no reference to the left-part occurrence X[0].I after VISITing X[1]. The reason for this restriction is that the VISIT to X[1] will pop the X.I stack and if X[1].I was not pushed onto the stack before the VISIT then the value that gets popped will be the value corresponding to X[0].I. This can be allowed only if X[0].I will not be used again.

2.2. Finding global variables and global stacks

To decide whether a transient attribute X.a can be implemented as a global variable or a global stack GAG checks whether each production that contains an occurrence of X.a satisfies certain sufficient conditions concerning how X.a is used in that production. If the appropriate conditions are satisfied for all the productions then X.a is implemented as a global variable or global stack, accordingly. Otherwise X.a is allocated space in the semantic tree. The sufficient conditions are given below, together with some illustrations.

2.2.1. Sufficient conditions for global variables

We consider inherited and synthesized attributes separately. First, let X.a be an inherited transient attribute, defined before the i^{th} visit to X and never referenced after that visit. Suppose production p has X as its left-part. X.a will not be implemented as a global variable only if, between the first attribute evaluation of the i^{th} visit and the last reference to X_0.a, either:

1. a right-part occurrence of X.a is defined, or
2. some right-part node Y is visited and Y derives X.

In the first case, X.a certainly cannot be implemented as a global variable since the value of the right-part occurrence would overwrite the value of the left-part occurrence and the left-part occurrence still needs to be referenced. In the second case, GAG assumes that the visit to Y will result in a nested visit to X (in Y's subtree) and the evaluation of its attribute X.a. This also would overwrite the previous value of the global variable although it still needs to be referenced. This last assumption made by GAG can be overly pessimistic; often global analysis can determine that even though Y is visited and Y derives X, no new

occurrence of X.a will be evaluated.

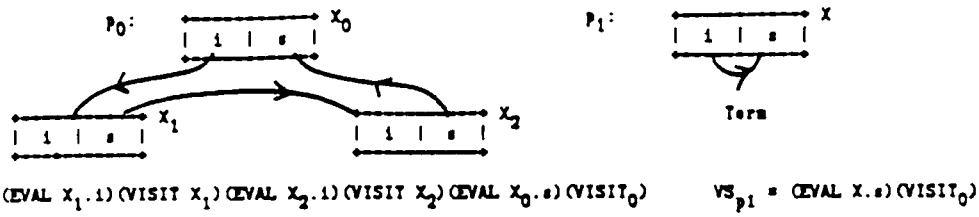


Figure 2-2: GAG implements X.i as a global variable

Synthesized transient attributes are examined in a similar fashion. Let X.a be an synthesized transient attribute, defined during the i^{th} VISIT to X and never referenced after executing ≥ VISIT$_0$ to X's parent. For each production p having a right-part occurrence of X, say X_j, X.a will not be implemented as a global variable only if, between the first attribute evaluation following the i^{th} visit to X_j and the last reference to X_j.a, either:

1. a left-part occurrence of X.a is defined, or
2. some right-part node Y is visited and either Y = X or Y derives X.

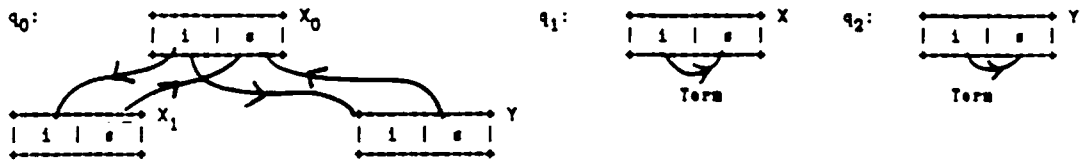
Figure 2-2 gives an example of an attribute, X.i, that GAG implements as a global variable. An examination of the sequence VS_{p0} in conjunction with the dependency graph D_{p0}, reveals that after evaluating X_1.i, X_0.i is never referenced again. After examining all of the productions of the attribute grammar GAG concludes that any two instances of X.i in any semantic tree will have disjoint lifetimes and X.i can be implemented as a global variable. In figure 2-3, however, an examination of VS_{q0} reveals that the lifetime of X_1.i is nested in the lifetime of X_0.i. This makes it impossible to implement X.i as a global variable; X_1.i's value would overwrite the value of X_0.i although the latter value is still needed to compute Y.i.

2.2.2. Sufficient conditions for global stacks

Again we consider inherited attributes separately from synthesized ones. Let X.a be an inherited transient attribute defined before the i^{th} VISIT to X and never referenced after that VISIT, and suppose production p has X as its left-part. X.a can not be made into a global stack only if there is a right-part occurrence of X in p, say X_j ($j>0$), and after defining X_j.a but before visiting X_j, X_0.a is referenced. In such a case the lifetime of X_j.a is not properly included in the lifetime of X_0.a; their lifetimes are *intertwined*. However, if X_j is visited before X_0.a is referenced then X_j.a's lifetime is properly included in X_0.a's lifetime; upon returning from the visit to X_j, the value of X_j.a is no longer needed since X.a is transient.

In figure 2-3, X.i can be implemented as a global stack. Upon visiting the production q_0,

X_1.i would be evaluated by using the value on top of the X.i stack as the value of X_0.i. X_1.i's value would then be pushed onto the top of the X.i stack, where it will be referenced during the visit to X_1. Before returning from that visit the top of the stack is popped, once again revealing the value of X_0.i. Upon returning to q_0 this value is used to compute the value of Y.i. The value of X_0.i can then also be popped off the top of the stack as it is no longer needed to compute any other attribute-instances.

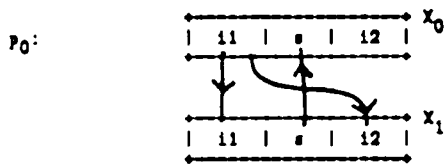


VS_{q0} = ŒVAL X_1.i) (VISIT X_1) ŒVAL Y.i) (VISIT Y) ŒVAL X_0.s) (VISIT$_0$)

VS_{q1} = ŒVAL X.s) (VISIT$_0$) VS_{p2} = ŒVAL X.s) (VISIT$_0$)

Figure 2-3: GAG implements X.i as a global stack

On the other hand, figure 2-4 shows a production with transient attribute X.i1 that GAG does not stack. The problem is that X_1.i1 is defined, and pushed on top of the stack, before the reference to X_0.i1 is used to define X_1.i2, which in turn happens before the VISIT to X_1 that pops the X.i1 stack.



VS_{p0} = ŒVAL X_1.i1) ŒVAL X_1.i2) (VISIT X_1) ŒVAL X_0.s) (VISIT$_0$)

Figure 2-4: A transient attribute GAG does not stack

GAG's sufficient conditions for stacking synthesized attributes are a little more lenient than for inherited ones. Sometimes these attributes are stacked even though their lifetimes are *intertwined*. The rule that a production must satisfy is that a synthesized attribute can not be stacked only if there are two distinct attribute-occurrences of X.s, X_i.s and X_j.s, such that

1. X_i.s is defined before X_j.s, and
2. the last reference to X_i.s occurs after the definition of X_j.s, and
3. the last reference to X_i.s occurs before the last reference to X_j.s

Note that for the purpose of this analysis the VISIT$_0$ that ends this VISIT to the production counts as the last reference to a left-part X_0.s. Notice further that, unlike the case for inherited attributes, there can be a reference to X_i.s between the definition of X_j.s

and the last reference to X_j.s; figure 2-5 illustrates such a case. GAG implements X.s as a global stack even though both X_1.s and X_2.s need to be referenced in order to define Y.i and X_0.s. To make these evaluations, GAG references the top <u>two</u> values on the stack.

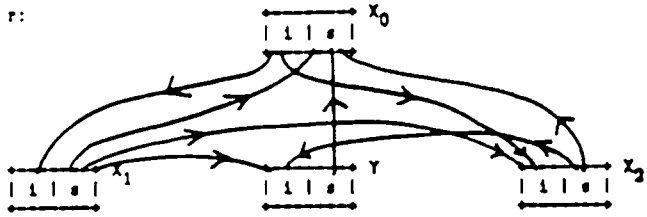


Figure 2-5: GAG allows reference to value within stack

GAG does not implement all synthesized transient attributes as stacks. In figure 2-6, for example, GAG does not implement XA.s1 as a global stack. This is because after the computation of XA_0.s1 and XA_0.s2, XA_1.s1 would no longer be on the top of the stack and could therefore not be popped off the stack. The same is true for the example in figure 2-7.

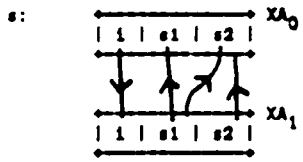


Figure 2-6: An example of GAG not stacking a synthesized attribute

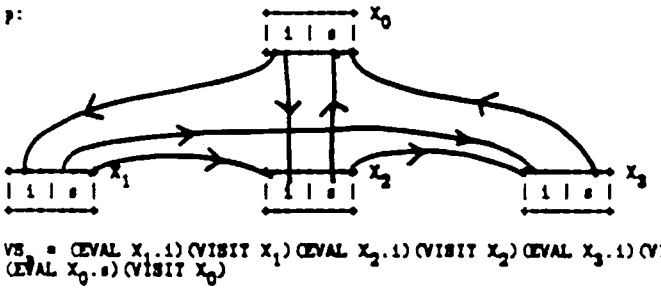


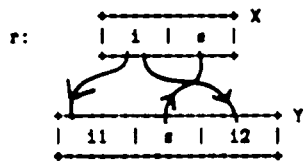
Figure 2-7: A second example of GAG not stacking a synthesized attribute

2.3. Combining Global Variables and Global Stacks

Once GAG has determined which attributes are to be implemented as global variables and global stacks it further attempts to optimize storage by combining several global variables into a single global variable and combining several global stacks into a single global stack. This policy can produce some startling effective optimizations. For example, figure

2-2 shows a grammar in which the two attributes (X.i and X.s) interact to simulate updating a global variable during a left-to-right evaluation pass. After GAG determines that both X.i and X.s can be implemented as global variables, it decides that they can be combined into a single global variable. The evaluator generated by GAG uses a single global variable to store all the attributes of the semantic tree and this variable gets updated during the traversal of the tree. Combining global stacks can also have beneficial effects: if X.i and Y.i are both being implemented as global stacks and there exists a production p: X ::= Y with a semantic function [Y.i = X.i] then we may be able to eliminate this copy rule by combining the X.i and Y.i stacks. The elimination of this copy rule means that for every instance of p in the semantic tree, one less storage cell is needed. Similarly, combining X.i and Y.i global variables eliminates copy rules of this form.

Not any two global stacks can be combined into one. For example, if X.i and Y.i1 of figure 2-8 were being implemented as global stacks they could not be combined into a single global stack as X.i needs to be referenced to evaluate Y.i2 after Y.i1 has already been placed on the top of the stack. If Y.i2 were being implemented as a global stack as well, it could be combined with X.i. Similar comments apply to global variables.



VS_r = (EVAL Y.i1)(EVAL Y.i2)(VISIT Y)(EVAL X.s)($VISIT_0$)

Figure 2-8: X.i and Y.i1 stacks cannot be combined

Furthermore, no 2 inherited or synthesized attributes of the same symbol can be implemented as the same global stack or variable, as demonstrated by figure 2-9. In this case, the X.i1 and X.i2 global stacks cannot be combined as upon visiting this production both X.i1 and X.i2 need to be referenced. In general, if X.i, Y.i1, and Y.i2, are global stacks or variables, we can combine the X.i stack with Y.i1 or with Y.i2 but not with both. This gives quite a bit of choice on how to combine global stacks together. Any combination gives rise to a *partition* of the global stacks, where all the global stacks in the same partition element will be made into one global stack. There are several criteria by which the efficacy of a partition could be judged: the total number of global variables and global stacks needed, the number of copy rules eliminated, or the total amount of space used to implement all of the stacks and global variables. This last criterion would be good to use if storage optimization is our chief concern, but the space used will generally depend on the

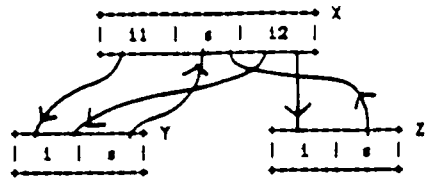


Figure 2-9: No two attributes of the same symbol can share a global stack

structure of the individual semantic trees, which can only be approximated when building an evaluator. The other two criteria are static criteria that can be determined when building the evaluator. A partition that eliminates the greatest number of copy rules will not necessarily result in the least number of global variables and global stacks, as illustrated by figure 2-10.

The number of stacks used is not a very important measure of storage efficiency since there will be at most some constant number of stacks anyway, and each one will contribute only a constant amount of overhead. The key measure is the total number of storage cells these stacks will use. Eliminating a copy rule from a production means that for *each* occurrence of that production in a semantic tree one less storage cell will be used. Hence total number of copy rules eliminated is a better measure. Unfortunately it is an NP-complete problem to determine the optimal way to combine the global stacks and vars so to eliminate as many copy rules as possible, as shown in the appendix.

To determine how to combine global variables and global stacks GAG uses a *first fit* strategy that combines together any two stacks or variables that can be combined. It does no analysis of any expected savings gained by doing this combination versus some other, incompatible, combination.

```
p1: X ::= Y term1.                    {(X.a,Y.a), (Z.a,X.b), (Z.b,Y.b)}
          Y.a = X.a;
          Y.b = constant;     A partition eliminating all 3 copy rules and using 3 global variables

p2: X ::= Z term2.
          Z.a = X.b;
          Z.b = constant;

p3: Y ::= Z  term3.                   {(X.a,Y.a,Z.a), (X.b,Y.b,Z.b)}
          Z.a = constant;
          Z.b = Y.b;          A partition eliminating only 2 copy rules, but using 2 global variables

p4: Z ::= term4.
```

Figure 2-10: Two different partitions of global variables X.a, Y.a, Z.a, X.b, Y.b and Z.b

3. LINGUIST-86

LINGUIST-86 is an AG-based translator-writing-system that generates attribute evaluators that use the alternating pass evaluation strategy [7]. These evaluators store a linearized version of the semantic tree in intermediate files on secondary storage, and so they avoid using large amounts of main memory to represent the semantic tree. LINGUIST-86 attempts to further improve the use of storage by its evaluators through an optimization called *static subsumption*. This eliminates copy-rules and decreases both the stack space needed to evaluate an attribute grammar and the size of the intermediate files.

3.1. LINGUIST-86 evaluation paradigm

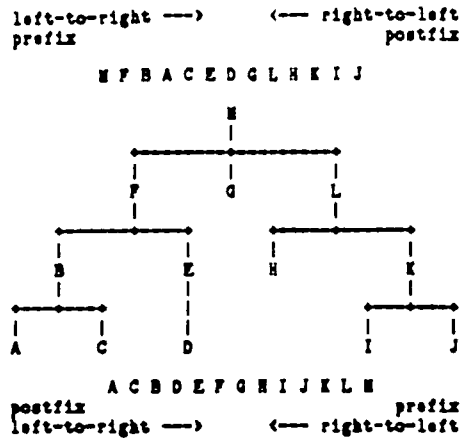
The basic idea of LINGUIST-86's evaluation paradigm is that when a semantic tree node, N, is VISITed during attribute evaluation it is read from the intermediate file onto the top of a stack in memory. N is kept on the stack while the sub-tree descended from N is visited. The nodes of this sub-tree get stacked on top of N and attribute-instances in that subtree are assigned values. The evaluation of the sub-tree may use the values of some attribute-instances of N and may define other attribute-instances of N. When the evaluation pass over N's subtree is finished N is written to the intermediate file and popped off the stack. Because of the the evaluation order, the nodes of N's subtree will have already been written out and removed from the stack. LINGUIST-86's paradigm for semantic tree traversal and attribute evaluation in a left-to-right pass is given in figure 3-1, which describes the process of VISITing a sub-tree whose root is an instance of X0.

```
read all attribs of X1 from input file onto stack
eval inherited attribs of X1 for this pass
visit the sub-tree whose root is X1
write all attribs of X1 to output file

read all attribs of X2 from input file onto stack
eval inherited attribs of X2 for this pass
visit the sub-tree whose root is X2
write all attribs of X2 to output file
                ...

read all attribs of Xn from input file onto stack
eval inherited attribs of Xn for this pass
visit the sub-tree whose root is Xn
write all attribs of Xn to output file

eval synthesized attribs of X0

pop all attribs of X1 ... Xn off top of stack

return from visiting sub-tree rooted at X0
```

Figure 3-1: Attribute Evaluation Paradigm of LINGUIST-86

This evaluation paradigm calls for reading nodes in prefix order and writing them in postfix order. Two intermediate files are used per pass: semantic tree nodes are read from one intermediate file and written to the other intermediate file. The output file of a left-to-right pass is a left-to-right, postfix ordering of the nodes of the semantic tree. The input file for a right-to-left pass is a right-to-left, prefix ordering of the semantic tree nodes.

Thus, if the output file of a left-to-right pass is read backwards it can be used as the input file for a right-to-left pass. The same is true for a right-to-left pass followed by a left-to-right pass. This trait is illustrated by the diagram below.



LINGUIST-86 generates in-line code to read and write semantic tree nodes and to evaluate semantic functions. This code is organized as a set of mutually recursive "production-procedures". For each production and for each pass, there is a separate production-procedure. These are partitioned into: the set of production-procedures that are used for pass 1, the set used for pass 2, the set used for pass 3, etc. Each production-procedure has one value/result parameter, a semantic tree node, that corresponds to the left-part of the production. Space for the production's right-part semantic tree nodes are allocated as local variables of its production-procedures. Thus, the stack of semantic tree nodes is intermixed with the system run-time stack that supports procedure call/return, parameter passing, and recursion. The body of each production-procedure does the following:

- read right-part semantic tree nodes from the input intermediate file,
- compute values by evaluating semantic functions and use these to define attribute-instances,
- call production-procedures to VISIT right-part sub-trees, and
- write right-part semantic tree nodes to the output intermediate file.

This organization is similar to that of a recursive descent compiler. A simple production and the corresponding production-procedure for a left-to-right pass is shown in figure 3-2.

In this scheme transient attributes are dealt with quite easily by just not reading them from or writing them to the intermediate file. The attributes of each symbol are partitioned into significant and transient attributes depending on whether or not the attribute will be used in a later pass than the one during which it is defined. Only the values of significant

```
X0  ::=  Y  X1.
  X1.I    = X0.I,
  Y.I     = X0.I,
  X0.S    = X1.S,
  X1.PRE  = UnionSetof(Y.OBJ,X0.PRE),
  X0.POST = IncrIfTrue(IsIn(Y.I,X1.PRE),X1.POST);


procedure P14_PP1 (VAR X0 : X_type);
   /* this is a right-to-left pass */
 VAR  Y  : Y_type;
      X1 : X_type;
 begin

   GetNode(X);
     X1.I = X0.I;
     X1.PRE = UnionSetof(Y.OBJ,X0.PRE);
   PP1( X );            /* Visit X1 */
   PutNode(X);

   GetNode(Y);
     Y.I = X0.I;
   PP1( Y );            /* Visit Y */
   PutNode(Y);

    X0.S = X1.S;
    X1.POST = IncrIfTrue(IsIn(Y.I,X1.PRE),X1.POST);

 end P14_PP1;
```

Figure 3-2: A production-procedure generated by LINGUIST-86

attributes are transfered between the intermediate files and the stack. In a LINGUIST-86-generated attribute evaluator all transient attributes are implemented on a single stack.

Being able to keep the semantic tree on secondary storage and not in main memory is an important, integral part of LINGUIST-86's evaluation paradigm. However, there are other aspects of LINGUIST-86's storage optimization strategy that we wish to analyze and compare with GAG's strategy, such as static subsumption and the stack-implementation of transient attributes. Thus it is useful to notice that LINGUIST-86's paradigm could easily be modified so that the semantic tree was kept in main memory; simply use a large buffer rather than the intermediate file.

3.2. Static Subsumption

In LINGUIST-86's basic paradigm, the various production-procedures communicate with one another by passing a pointer to the stack-resident structures that hold the values of attribute-instances. The principle behind static subsumption is that this communication can be achieved just as well by copying the values of attribute-instances from the stack to specific global variables. Consider the production-procedure for a production with symbol X as its left-part. Suppose that the value of any instance of inherited attribute X.I is always copied to global variable X_I just before VISITing an X-node. Instead of accessing the value of X.I as a field of its VAR parameter, the production-procedure can now access global variable X_I for the value of this left-part attribute-instance. Similarly, if the production-procedure defines left-part synthesized attribute X.S by assigning a value to global variable X_S, then any production-procedure containing X in its right-part can use this

global variable, either to copy it into the local, stack-resident structure for a node, or to use it in the evaluation of semantic functions.

In most cases, copying attribute-instances back and forth is more expensive than passing a pointer and making indirect references through it. However, if the semantic function that defines an attribute-instance is a copy-rule whose right-part is a different instance of the same attribute then no explicit code is required to implement this semantic function. The proper value is already in the global variable. We say that such a copy-rule is "subsumed" by the static allocation of the attribute. Figure 3-3 shows a simple example of how copy-rules can be subsumed; the subsumed copy-rules are commented out.

```
X0 ::= Y X1.
  X1.I    = X0.I.
  Y.I     = X0.I.
  X0.S    = X1.S.
  X1.PRE  = UnionSetof(Y.OBJ,X0.PRE),
  X0.POST = IncrIfTrue(IsIn(X1.S,X1.PRE),X1.POST);

VAR
  X_I : I_attribType;
  Y_I : I_attribType;
  X_S : S_attribType;

procedure P14_PP1 (VAR X0 : X_type);
   /* this is a right-to-left pass */
  VAR  Y  : Y_type;
       X1 : X_type;
  begin

    GetNode(X);
   /* X1.I := X0.I; */
      X1.PRE := UnionSetof(Y.OBJ,X0.PRE);
    PP1( X );
    PutNode(X);

    GetNode(Y);
      Y_I := X_I;
    PP1( Y );
    PutNode(Y);

  /* X0.S := X1.S; */
     X0.POST = IncrIfTrue(IsIn(X_S,X1.PRE),X1.POST);

  end P14_PP1;
```

Figure 3-3: An example of static subsumption, X.I, Y.I and X.S are statically allocated.

The penalty for eliminating this explicit copying is paid at those points where the static attributes are not defined by subsumable copy-rules. In these cases a new value must be assigned to the global variable for propagation to the sub-tree. However, the previous value of the global variable is not "dead"; it may still be used later by this production-procedure, or by a production-procedure at some ancestor node. Hence the old value must be saved in a temporary variable in the production-procedure's stack-frame. Sometime after VISITing the subtree but before exiting this production-procedure the saved value must be restored to the global variable. Figure 3-4 shows the production-procedure of the earlier example modified as would be required if attributes X.PRE and X.POST were statically allocated to global variables X_PRE and X_POST.

```
VAR
  X_I    : I_attribType;
  Y_I    : I_attribType;
  X_S    : S_attribType;
  X_PRE  : PRE_attribType;
  X_POST : POST_attribType;

procedure P14_PP1 (VAR X0 : X_type);
   /* this is a right-to-left pass */
  VAR
    Y           : Y_type;
    X1          : X_type;
    X_PRE_QZP   : PRE_attribType;  /* save copy of left-part  value of X_PRE  */
    X_PRE2_ZQP  : PRE_attribType;  /* hold copy of right-part value of X_PRE  */
    X_POST2_ZQP : POST_attribType; /* hold copy of right-part value of X_POST */

  begin

    GetNode(X1);
   /* X1.I = X0.I */
      X_PRE2_ZQP = UnionSetof(Y.OBJ,X_PRE);
      X_PRE_QZP = X_PRE; X_PRE = X_PRE2_ZQP;
    PP1( X1 );
      X_PRE = X_PRE_QZP;          /* restore value of X0.PRE */
      X_POST2_ZQP = X_POST;       /* capture value of X1.POST so it isn't lost when VISITing Y */
    PutNode(X1);

    GetNode(Y);
      Y_I = X_I;
    PP1( Y );
    PutNode(Y);

   /* X0.S = X1.S; */
      X_POST = IncrIf(IsIn(X_S,X_PRE2_ZQP),X_POST2_ZQP);

  end P14_PP1;
```

Figure 3-4: An example of the costs of static subsumption.

The need to save/restore the global variable of a statically allocated attribute is especially burdensome in the case of synthesized attributes, and LINGUIST-86 is not very good about subsuming copy rules between synthesized attributes. Even if synthesized attribute X.S is statically allocated, a copy rule [X0.S = X1.S] will be subsumed only if:

- it is evaluated during a left-to-right pass and X1 is the right-most symbol of the production, or
- it is evaluated during a right-to-left pass and X1 is the left-most symbol;

i.e. only if no other sub-tree is VISITed after VISITing X1. This is because LINGUIST-86 does no global analysis to determine whether a particular VISIT will assign to the global variable, hence any useful value in that variable must be saved before the VISIT and restored after it. Only when no VISIT intervenes between the definition of the global variable (during a VISIT) and the use of that variable (to define a left-part attribute allocated to the same global variable) will the variable not be saved/restored and hence be eligible for subsumption.

Static subsumption can be even more widely applied by allocating several different attributes to the same global variable. The major restriction is that two different attributes of the same symbol can not be allocated to the same global variable. Many more copy-rules are subsumable by such a strategy and hence can be eliminated. In the example above, X.I and Y.I could be allocated to the same variable, thereby enabling us to eliminate the copy-rule

[Y.I = X.I]. On the other hand, global variables may have to be saved/restored more frequently. In general, the extra code necessary to save/restore a global variable is as much as the code saved by subsuming several copy-rules.

Static subsumption can also reduce the amount of space needed to store attribute-instances. When an attribute X.A is statically allocated no field is needed for it in X nodes. This can result in significant decrease both in the stack space needed for semantic tree nodes and in the size of the semantic tree file. However, stack space will be needed whenever a right-part occurrence of X.A is not defined by a subsumable copy-rule because then the global value must be saved in a local variable on the stack.

3.2.1. Determining static allocation of attributes

LINGUIST-86's static allocation paradigm calls for us to decide, for each attribute, whether the attribute should be statically allocated, and if so then with which other statically allocated attributes should it be combined. In making these decisions LINGUIST-86 tries to save as much code space as it can by eliminating copy-rules; it tries to maximize the net code space savings of subsumed copy rules minus extra code necessary to save and restore global variables. This is a combinatorial problem that is infeasible to solve exactly for any realistic number of attributes. Instead, LINGUIST-86 uses a heuristic in order to narrow the search space and then uses a polynomial-time approximation on the resulting, smaller, problem. The heuristic is that if two attributes with the same name, say X.FOO and Y.FOO, are each statically allocated then they will be statically allocated to the same global variable. This substantially reduces the complexity of the problem: for each attribute X.FOO LINGUIST-86 only needs to decide whether it should be static. If it is static then it will share storage with all the other attributes Y.FOO that are also static.

This reduced problem is still NP-complete and the way LINGUIST-86 solves it depends on how many attributes there are with a given name. The attributes are partitioned into classes with the same name and the members of each class are analyzed independently from the members of any other class. If there are 13 or fewer attributes in a class then LINGUIST-86 examines each of the 2^{13} possible combinations looking for the "best" one. Otherwise, the polynomial-time approximation described below is used. The "best" solution is the one that saves the most code space.

If there are more than 13 attributes in a given class then LINGUIST-86 starts by assuming that all attributes in the class are statically allocated. Each attribute is then checked to see if it costs more in code size for it to be static than it would if it were normally allocated. This check is based on how many copy rules would no longer be subsumed if the attribute

were not static, versus how many times the attribute would no longer have to be saved and restored. LINGUIST-86 assumes that it takes three subsumed copy rules to offset a single save and restore, even though this assumption is overly pessimistic in many cases.

If so indicated, the attribute is removed from the statically allocated set. When an attribute is changed from being statically allocated to being allocated in the semantic tree node it can become more expensive for other attributes to be statically allocated. Hence, all remaining static attributes must be reexamined until the process stabilizes. This is an $O(n^3)$ algorithm and it does not always find an optimal set of attributes to statically allocate.

3.2.2. Static allocation of significant attributes

The static subsumption paradigm does not require that the static attributes be transient; LINGUIST-86 can statically allocate significant attributes. A significant, static attribute does not take up space in the semantic tree except at those places where the value of an instance of the attribute is changed, i.e. where an instance is defined by other than a copy rule. There the previous value of the static variable is saved in the semantic tree as a temporary value that is associated with the production, rather than associated with the symbol (i.e. node).

The implementation of a significant, static attribute is the same as that of a transient static attribute for the pass during which the attribute is defined. On later passes the treatment is similar in that, upon entry to a production-procedure, the value in the global variable is saved in a stack-resident temporary whenever the static attribute is redefined by a non-copy rule. However, the treatment is different in later passes in that the global variable is redefined with the value that was computed earlier and was saved in the semantic tree. Figure 3-5 shows the situation of figure 3-4 if X.PRE were a significant attribute. The production-procedure PP14_PPj shows the implementation of X.PRE on passes after X.PRE is computed.

This strategy will only save code space, i.e., eliminate enough copy rules, if the value of the attribute is not changed very often. For a significant attribute the code needed to save and restore the global variables must be generated in several passes, whereas the copy rules that can be eliminated still occur in only one pass - the one during which the attribute is defined. Thus, a significant attribute will be statically allocated only when the subsumable copy rules greatly outnumber the non-subsumable definitions. For instance, in the Pascal AG written for LINGUIST-86, only three attributes (out of 892) are both statically allocated and significant.

```
VAR
  X_I    : I_attribType;
  Y_I    : I_attribType;
  X_S    : S_attribType;
  X_PRE  : PRE_attribType;
  X_POST : POST_attribType;


procedure P14_PP1 (VAR X0 : X_type);
    /* this is a right-to-left pass */
  VAR
    Y           : Y_type;
    X1          : X_type;
    X_PRE_QZP   : PRE_attribType;  /* save copy of left-part  value of X_PRE  */
    X_PRE2_ZQP  : PRE_attribType;  /* hold copy of right-part value of X_PRE  */

  begin

    GetNode(X1);
      /* X1.I = X0.I */
      X_PRE2_ZQP = UnionSetof(Y.OBJ,X_PRE);
      X_PRE_QZP = X_PRE; X_PRE = X_PRE2_ZQP;
    PP1( X1 );
      X_PRE = X_PRE_QZP;
    PutNode(X1);

    GetNode(Y);
      Y_I = X_I;
    PP1( Y );
    PutNode(Y);

    PutSignifImps(X_PRE2_ZQP);
  end P14_PP1;


procedure P14_PP2 (VAR X0 : X_type);
    /* this is a right-to-left pass after pass 1 */
  VAR
    Y           : Y_type;
    X1          : X_type;
    X_PRE_QZP   : PRE_attribType;  /* save copy of left-part  value of X_PRE  */
    X_PRE2_ZQP  : PRE_attribType;  /* hold copy of right-part value of X_PRE  */
    X_POST2_ZQP : POST_attribType; /* hold copy of right-part value of X_POST */

  begin
    GetSignifImps(X_PRE2_ZQP);

    GetNode(X1);
      X_PRE_QZP = X_PRE; X_PRE = X_PRE2_ZQP;
    PP1( X1 );
      X_PRE = X_PRE_QZP;          /* restore value of X0.PRE */
      X_POST2_ZQP = X_POST;       /* capture value of X1.POST so it isn't lost when VISITing Y */
    PutNode(X1);

    GetNode(Y);
    PP1( Y );
    PutNode(Y);

      /* X0.S = X1.S; */
      X_POST = IncrIf(IsIn(X_S,X_PRE2_ZQP),X_POST2_ZQP);

  end P14_PP2;
```

Figure 3-5: Static allocation of significant attributes.

4. Comparing the efficacy of GAG and LINGUIST-86

In order to see how well the various storage optimizations work, and to compare their effectiveness, we would ideally like to take several attribute grammars and generate attribute evaluators from them with both systems and then compare the results. Unfortunately, the two systems accept quite different input forms with respect to such features as: type structure, built-in functions, special default and *short-hand* notation, etc; so much so that the existing attribute grammars for each system would have to be substantially rewritten in order to be accepted by the other system.

The next best form of comparison would be to take a pair of attribute grammars that describe the same (or very similar) translations and look at how much each system can improve the time-efficiency and space-efficiency of the attribute evaluators it generates over the efficiency of unoptimized versions of those evaluators. However, it can be misleading to compare these figures for GAG and LINGUIST-86 because the two systems are targeted to such widely different computer systems. GAG generates evaluators for large main-frame computers with 32-bit words, multi-megabyte address spaces, and virtual memory. LINGUIST-86 generates evaluators for micro-computers with 16-bit words, address space in the range of 100-500 *kilo*bytes, and no virtual memory. For example, when we find that LINGUIST-86's storage optimizations have no perceivable effect on running time, is it because too few instructions were saved? or because LINGUIST-86-generated attribute evaluators run in an environment where they are I/O bound anyway and have no mechanism for trading space for time (i.e. a virtual memory system)?

The comparison we do think meaningful is to relate the effects of storage optimization in terms of the input attribute grammar, i.e. the attributes and semantic functions. Shown below are such statistics for two attribute grammars for Pascal, one designed to be input to GAG and the other designed for LINGUIST-86. Keep in mind that these are different attribute grammars. Although other attribute grammars have been written for both systems, these two are the ones that describe most nearly the same translation and for which reasonably compatible figures are available. The figures for GAG's grammar are from [10, 1]; those for LINGUIST-86's grammar are from [5].[2]

	GAG	LINGUIST-86
total # attribs	386	898
# attribs in nodes (i.e. not optimized)	52 (13%)	166 (18%)
# attribs as global var	291 (75%)	—
# attribs as stacks	43 (11%)	—
# transient attribs	—	732 (82%)
# static attribs	—	367 (41%)
total # semantic rules	998	2030
# copy-rules	727 (74%)	1147 (57%)
# copy-rules eliminated	372	748
% of all semantic rules	38%	37%
% of copy-rules	51%	65%

This data immediately suggests two observations:

- the strategies of each system are reasonably effective,

[2]The GAG figures reflect the options of expanding INCLUDINGs, and uniting a stack together with a global variable. These are the choices most nearly compatible with what LINGUIST-86 does.

- the degree of optimization performed is quite similar between the two systems.

This second point should not be too surprising for both systems have much the same underlying philosophy. The essentials of this philosophy, as contrasted to other approaches suggested in the literature, are:

- Both GAG and LINGUIST-86 systems are satisfied to minimize storage requirements even if they cannot find the best solution to the problem. Contrast this to the exhaustive analysis of global storage allocation presented in [6].
- Neither system will evaluate any attribute-instance twice, hence there is no space/time tradeoff. On the contrary, some of the optimizing techniques (static subsumption of copy rules) also eliminate the need to copy duplicate values around the tree. Compare this to the algorithms given by Reps [14] which bound the number of attributes ever needed to be stored and accessed simultaneously but which may require multiple evaluation of attribute-instances.
- Both GAG and LINGUIST-86 build "static" tree-walk evaluators, with the evaluation order at each production completely determined. Contrast this with the attribute evaluation paradigm described by Katayama [11], or with the similar paradigm implemented by Jourdan [8]. This paradigm also calls for transient attributes to be allocated on a stack in much the same way as for LINGUIST-86. However, this paradigm deals with significant attributes by re-evaluating them on each VISIT during which they are used. Furthermore, this is a more "dynamic" paradigm that does much less analysis of ultimate evaluation order, and hence does not do as much storage optimization; nothing similar to copy rule subsumption is done, for example.
- The optimizations used in GAG and LINGUIST-86 are precompiled into the code of the evaluator and are tree-independent; no extra run-time analysis of a semantic tree is needed to apply the optimizations.
- Both approaches are flexible: they allow attributes to share storage cells even if not *all* of their occurrences will have the same value or disjoint lifetimes.
- Both, although GAG moreso than LINGUIST-86, take their evaluation strategy into consideration when deciding upon which attributes should share storage.

Despite these similarities in underlying strategy, GAG and LINGUIST-86 are different in many respects. Most of these differences are incidental and contribute little to the total effectiveness of the systems. However, there are a few substantive differences that suggest how one or the other, or both, systems can be improved. These will be discussed in section 4.2. But first, let us examine the important features of storage optimization as performed by both GAG and LINGUIST-86.

4.1. Analysis of storage optimizations

For both GAG and LINGUIST-86 the issues for storage optimization can be characterized as:

1. Whether instances of an attribute should be implemented as:
   a. components of semantic tree nodes, or
   b. elements of a stack, or
   c. values assigned to a global, static variable.
2. For attributes implemented as stacks, how deep will the stacks grow, i.e. how much memory will they require.
3. Which of the attributes that are implemented as stacks or variables can be combined together in order to:
   - eliminate copy rules, or
   - reduce the number of stacks and variables.

These optimizations affect how much memory is needed for the evaluator's data, as well as the code size of the evaluator (eliminated copy rules, pushing and popping stacks, etc.). It is our opinion that the effects on data storage are far more important than the effects on the evaluator's code, either the memory needed to store this code, or the time needed for its execution.

The evaluator's code is independent of the size of the semantic tree; it does not grow with the size of the input string being processed. Experience with both GAG and LINGUIST-86 is that memory needed for the evaluator's code is much less a problem than the memory needed for the semantic tree.

The time savings of eliminated copy rules is also not significant; in [2] the effect on running time of not eliminating any copy-rules is reported as being too small to notice. The GAG researchers also report [1], [10], p.67 that the effect on run-time of GAG's attribute storage optimizations was minimal.

The most important goal for these optimizations is to *keep attributes from being components of semantic tree nodes*. The big savings in space comes from not having to keep all instances of an attribute simultaneously allocated in the semantic tree.

Next in importance is to keep the stacks from growing too large. This is especially important if the generated evaluator processes lists of elements (e.g., lists of statements) by recursively VISITing the elements on the list rather than iteratively VISITing them. The best way to keep stacks small is to implement an attribute as a global variable rather than as a stack. Eliminating copy rules also helps to keep stacks small since a copy rule that

can be eliminated is pushing a [redundant] value onto some stack. By this reasoning, eliminating copy rules whose source and target are both the same global variable is not very useful.

Combining global variables and combining stacks is not very effective except when it eliminates copy rules between stacks. The storage overhead for using one more global variable or one more stack is quite small -- a couple of words at most. The number of different attributes is quite small, relative to the number of attribute-instances in a semantic tree. The decrease in storage possible by combining stacks and global variables is probably no more than 400-500 words, and is likely much less. Combining stacks is only useful when it allows us to eliminate copy rules; combining global variables does not save much storage.

Thus, we feel that the following should be the major goals of an attribute storage optimization strategy, listed in order of importance.

1. allocate as few attributes as possible in the semantic tree nodes,
2. implement as many attributes as possible as global variables,
3. keep attribute stacks shallow,
4. combine attributes implemented as stacks so as to eliminate redundant PUSH operations, and so help keep stacks shallow.

Let us now consider how GAG and LINGUIST-86 differ from one another in achieving these goals. We will also consider how one system may be able to borrow more effective ideas and techniques from the other.

4.2. Substantive differences between GAG and LINGUIST-86

4.2.1. Transient attributes.

The most important goal is to keep attributes from being implemented as components of semantic tree nodes. The attributes GAG will implement as either stacks or global variables must be transient, but not all transient attributes can be so implemented. On the other hand, LINGUIST-86 implements all transient attributes either on the stack or as static variables, and can also do so for a few significant attributes. Thus one might think that LINGUIST-86 would allocate fewer attributes in the semantic tree. Nonetheless, GAG optimizes 87% of its attributes, versus 82% for LINGUIST-86. We believe this is because GAG's strategy of *ordered* attribute evaluation is much more flexible than LINGUIST-86's alternating-pass evaluation. An attribute that is *significant* under alternating-pass evaluation can be *transient* under ordered evaluation. For example, in figure 4-1, since Y.i references X.i but must be defined on a later pass than X.i, X.i is not a transient attribute under alternating pass evaluation. An ordered evaluator could visit first X_1, then X_2 and then Y,

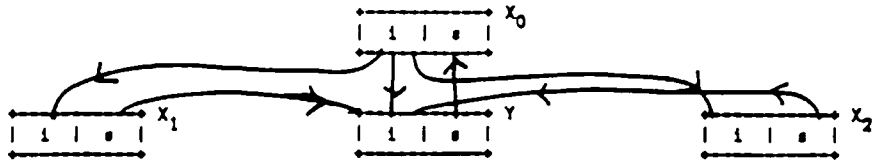


Figure 4-1: X.i is transient only using an ordered evaluation strategy

and so make X.i into a transient attribute.

4.2.2. Global variables.

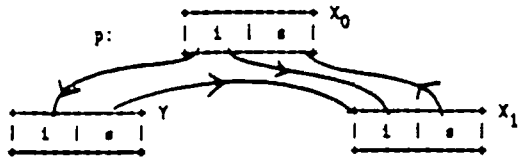
GAG can implement a transient attribute as a global variable rather than as a stack. LINGUIST-86 only uses stacks; it does not implement attributes as global variables. Even when an attribute could be implemented as a global variable the best LINGUIST-86 does is to statically allocate it. This results in saving and restoring attribute-instances needlessly.

LINGUIST-86 should implement attributes as global variables. This would substantially improve LINGUIST-86's stack requirements, especially for list constructs that are VISITed recursively. One way to do this within LINGUIST-86's framework is to determine when a statically-allocated variable need not be saved and restored. However this would require that information describing how attributes are used be propagated from one production to another and LINGUIST-86 does not do this. An alternative is to incorporate GAG's algorithm for finding global variables into LINGUIST-86. This would also be a non-trivial change to LINGUIST-86 since the evaluation order at a production would need to be computed before storage optimization is done. Either way, LINGUIST-86 needs to do more global analysis.

4.2.3. *Popping from above* versus *popping from below.*

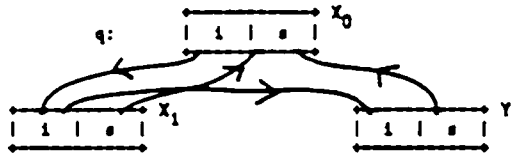
LINGUIST-86 allocates and deallocates space for attribute-instances in the same procedure, and saves and restores static variables in the same procedure; GAG pushes values onto an attribute stack in one production and pops values off the stack in a different production. Let us refer to this difference as *popping from above* versus *popping from below*. Each strategy has advantages and disadvantages. *Popping from below*, implemented by GAG, can save a lot of stack space as illustrated by figure 4-2. In this example, GAG implements X.i as a stack. Since X can derive itself, the stack could grow as the height of the tree. By popping X_0.i off the stack from "below" before visiting X_1, the stack will have height 1 whenever X_1 is VISITed. However, "popping from below" can cause many attributes not to be implemented as a stack that could be so implemented if GAG would *pop from above* instead. Figure 4-3 illustrates such a case. In this example, GAG would not make X.i into a stack as X_1.i's value is needed after the visit to X1. Furthermore, as was discussed in

section 2.1, this strategy also inhibits the elimination of many copy rules.



VS_p = (EVAL Y.i)(VISIT Y)(EVAL X_1.i)(VISIT X_1)(EVAL X_0.s)(VISIT$_0$)

Figure 4-2: An example where *popping from below* is advantageous



VS_q = (EVAL X_1.i)(VISIT X_1)(EVAL Y.i)(VISIT Y)(EVAL X_0.s)(VISIT$_0$)

Figure 4-3: An example where *popping from above* is advantageous

On the other hand, the reason that LINGUIST-86 is able to stack all transient attributes is because it keeps those attribute-instances on the stack for a relatively long time. For instance, LINGUIST-86 allocates stack space for a synthesized attribute-instance before VISITing the sub-tree that will define that attribute-instance, even though that value will not be defined until just before the end of that VISIT. In the interim, which can be quite a long time, that place on the stack is not being used. GAG would avoid pushing that value onto the stack until just prior to finishing the VISIT, if it was able to stack the attribute in the first place.

The diagram of figure 2-1 showed which attribute-instances would be on a stack when GAG was visiting a node. It is reproduced below in figure 4-4, changed to show which attribute-instances LINGUIST-86 would have on its stack when visiting that node. There are many more stacked nodes for LINGUIST-86.

For GAG there is a simple addition that gets the best of both techniques: use a fourth way of implementing attributes, global stacks that are *popped from above*. If an attribute can not be implemented as a stack if it is *popped from below* then check to see if it can be stacked if the *pop from above* convention is used. This does not increase the stack space needed by any attribute that is stacked by GAG's current policy, but it does allow more attributes to be stacked.

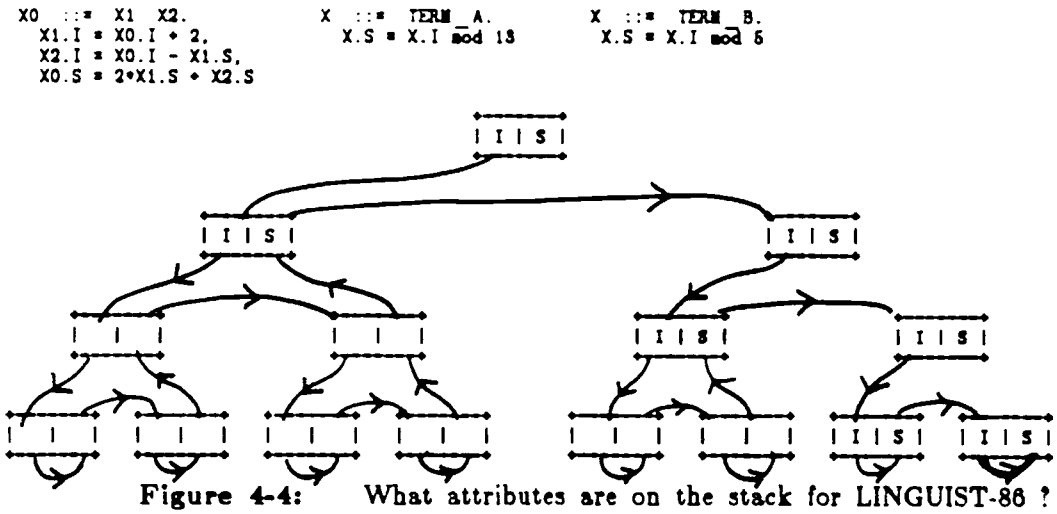


Figure 4-4: What attributes are on the stack for LINGUIST-86 ?

The path indicates evaluation order; labelled attribute-instances are on the stack when the evaluators *locus of control* is at the tip of the path.

4.2.4. Combining attributes to eliminate copy rules.

GAG combines attribute stacks and global variables (section 2.3); LINGUIST-86 statically allocates different attributes to the same variable (section 3.2.1). Both of these optimizations allow copy rules between different attributes to be subsumed. GAG's policy for such combinations is a simple *first fit*, whereas LINGUIST-86 analyzes how many copy rules can be subsumed by various combinations, and combines only attributes with the same name. Because it does more analysis LINGUIST-86 subsumes more of these copy rules. Of course, since LINGUIST-86 only subsumes copy rules between attributes with the same name, it would never subsume one such as [X.A = Y.B] and GAG could. Looking at the figures for the two Pascal attribute grammars, though, GAG subsumed 51% of its copy rules and LINGUIST-86 subsumed 65% of its copy rules; thus indicating that this latter situation is relatively rare.

GAG decides how to group stacks together without considering how many copy rules will be eliminated as a result of this grouping. LINGUIST-86 analyzes many different possibilities looking for one that saves the most code in the evaluator. By ignoring copy rules in its strategy for grouping attributes together, GAG misses many opportunities to subsume copy rules and hence to conserve stack space. However LINGUIST-86's strategy of optimizing the code size of the evaluator can also cause it not to subsume copy rules and hence to use more stack space.

Ideally, both systems should combine attributes so as to minimize the space needed for stacks, however this is an intractable problem. The difficulty is that the number of times a copy rule is executed depends on the structure of the semantic tree (i.e. input program) and

so the "best" copy rule to eliminate may vary from one input to another. Nonetheless, both systems can be improved. GAG should combine attributes only if it will eliminate some copy rules. LINGUIST-86 should combine attributes based on how much stack space is saved, rather than the amount of code saved.

4.2.5. More thorough global analysis using evaluation order.

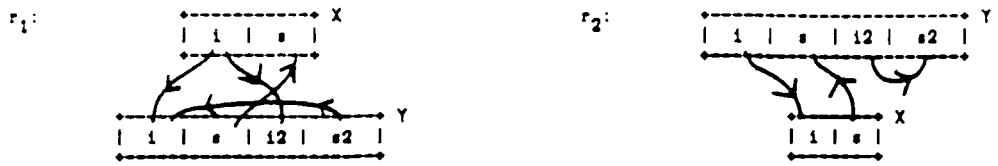
GAG does storage optimization analysis after the evaluation order has been determined and it uses this information in its analysis. LINGUIST-86 does its analysis (i.e. which attributes to statically allocate [together]) before the complete evaluation order has been fixed. As a result, LINGUIST-86 is too pessimistic about the cost of statically allocating some attributes and so misses out on potential optimizations. LINGUIST-86 should decide how to statically allocate attributes after the evaluation order is known. This would also make it easy to incorporate GAG's strategy for finding attributes that can be global variables.

Still more global information could be effectively used by both systems for storage optimization. In particular, it would be useful to collect summary information about the effect on attributes of VISITing non-terminals. We suggest computing the following information:

```
for each non-terminal, X
   for each visit to X, VISIT_i(X)
      for each attribute Y.A
         USE(X,i,Y.A) = true iff
           a VISIT_i to an X-node can ever reference
           or define some instance of Y.A
```

Such USE information could be used by GAG to implement more attributes as global variables. In section 2.1, we saw that GAG would not implement an inherited transient attribute X.a as a global variable if, in a production where X is the left-part, there is a visit to a right-part node Y before the last reference to X.a, and Y derives X. This is out of recognition that the visit to Y could cause a nested visit to X which could overwrite the value of the left-part occurrence of X.a. With USE information GAG could determine more precisely whether or not this particular visit to Y could actually overwrite X.a if it were implemented as a global variable. Figure 4-5 illustrates such a case. In production r_1, X.i is referenced after the first visit to Y and therefore GAG would not implement it as a global variable. Global analysis leads to the realization, however, that any occurrence of X.i in Y's subtree will not be evaluated until the second visit to Y, when the former value of X.i is no longer needed.

LINGUIST-86 could exploit USE information to statically allocate more attributes. Recall that LINGUIST-86 will subsume a copy rule between between different occurrences of a



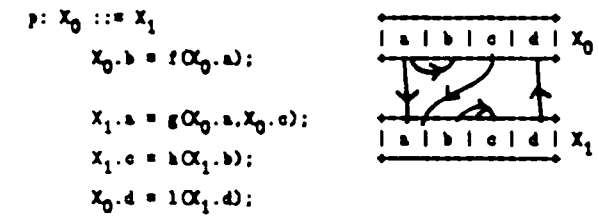
VS_{r1} = (EVAL Y.i2)(VISIT Y)(EVAL Y.i)(VISIT Y)(EVAL X.s)($VISIT_0$)

VS_{r2} = (EVAL Y.s2)($VISIT_0$)(EVAL X.i)(VISIT X)(EVAL Y.s)($VISIT_0$)

Figure 4-5: X.i could be implemented as a global variable

statically-allocated synthesized attribute only if the source occurrence belongs to the right-part node that is visited last during this pass. This is because a visit to a right-part node after the source of the copy rule is defined could cause some other attribute that is statically-allocated to the same variable to be defined. This would overwrite the contents of that variable and so destroy the source of the copy rule. USE information would enable LINGUIST-86 to tell that this could not happen, which would allow this copy rule to be subsumed, which would make it less expensive to statically allocate this attribute, which would cause more attributes to be statically-allocated.

Finally, if USE information were available GAG could implement some significant attributes as stacks or global variables. Recall that for an attribute to be a stack it is necessary that all instances of that attribute have lifetimes that are either disjoint or properly nested. The sufficient condition that GAG uses includes the restriction that no lifetime can contain a $VISIT_0$. This is because GAG doesn't know enough about what happens "above" the current locus of control in the semantic tree; the worst-case is assumed to happen and so no such attributes are stacked. However, it can happen that all attribute-instance lifetimes are either disjoint or properly nested even though the attribute is not transient. For example, attribute X.a of figure 4-6 is not stacked by GAG as it is not transient; X.a is defined before visiting X for the first time, but is referenced during the second visit to X. Nonetheless, the lifetimes of instances of X_0.a and X_1.a are disjoint and X.a can be implemented as a stack or global variable.



VS_p = (EVAL X_0.b)($VISIT_0$)(EVAL X_1.a)(VISIT X_1)(EVAL X_1.c)(VISIT X_1)(EVAL X_0.d)($VISIT_0$)

Figure 4-6: X.a is a significant attribute but can be implemented as a stack

Situations like the one shown in figure 4-6 would be correctly detected if GAG's sufficient

conditions for stacking attributes were modified to allow a $VISIT_0$ in the lifetime of a left-part attribute-occurrence, say X.A, but to treat visits to any right-part node Y_j (in the same or other productions) as a reference to a right-part occurrence of X.A if during that visit any instance of X.A could be referenced. This latter information would be supplied by the USE computation.

5. Suggestions for Improvement

Several suggestions for improving GAG and LINGUIST-86 were presented in section 4. Briefly, these were:

- in LINGUIST-86, implement attributes as global variables,
- in GAG, implement attributes as stacks that are *popped from above*,
- in both GAG and LINGUIST-86, combine attributes so as to minimize the space requirements for stacks,
- in both GAG and LINGUIST-86 (but especially LINGUIST-86), use more thorough global information to determine the applicability of the optimizations.

In this section we show how both GAG and LINGUIST-86, by considering storage optimizations at an earlier time in the generation cycle, can create an evaluation paradigm explicitly designed to optimize storage.

5.1. Computing the evaluation order for storage-efficiency.

We have shown several examples where the order of evaluation of semantic functions was crucial in being able to implement a storage optimization. However, in neither LINGUIST-86 nor in GAG do potential storage optimizations influence the choice of evaluation order. This is particularly unfortunate because in both ordered evaluation and in alternating pass evaluation there are many arbitrary choices that go into computing the evaluation order, choices that could be made so as to facilitate storage optimizations. Because a major source of storage savings is achieved by implementing transient attributes as global variables or global stacks, we suggest that appropriate heuristics be used when the evaluation order is computed in order to increase the number of transient attributes.

For most evaluation strategies, one can view the process of fixing visit sequences (VS_p) for the productions of the grammar as a two stage process: First, for each nonterminal X of the grammar, each attribute X.a is assigned a visit number i, indicating that any occurrence of X.a in any semantic tree will be evaluated on the i^{th} visit to X. Secondly, each production p is examined and a final evaluation order VS_p is decided upon. This evaluation order must be consistent with both the dependencies given by the semantic functions of p

and with the visit numbers assigned to the attributes of p. Both of these steps usually make some arbitrary choices.

First we examine the second stage of the process. After visit numbers are assigned to all the attributes, each production p must be assigned a visit sequence VS_p. This entails completing into a total order the partial order given by the semantic functions of p and the visit numbers assigned to the attributes of p. The partial order is represented by an *augmented dependency graph* [12], having the attributes of p as vertices and an edge (X_j.a, X_k.b) if X_j.a is an argument to the semantic function defining X_k.b or if j = k and X_j.a has a lower visit number than X_j.b. This partial order is completed into a total order by adding edges to this graph. We suggest replacing some of the arbitrary choices of this process by heuristics that increase the number of transient attributes. In particular, one element of choice involves the order in which the inherited attributes of a right-part node and the synthesized attributes of a left-part node are evaluated before visiting the node. GAG often arbitrarily decides to evaluate one attribute before another, preventing the attribute from being implemented as a stack. This was illustrated by figure 2-4, where GAG chose to evaluate X_1.i1 before X_1.i2, and by figure 2-6, where GAG chose to evaluate XA_0.s1 before XA_0.s2. We suggest substituting this arbitrary choice by the *read before next write* heuristic, given by Sethi in [17].

The *read before next write* heuristic is illustrated in figure 5-1. If A is needed to define both B and C, and we wish for A to share storage with C, and B and C are unrelated (i.e. neither depends on the other), then this heuristic calls for adding an edge from B to C. This makes it appear as though B is needed to define C and will call for the evaluation of B before C. Since A will not be referenced after the computation of C, A and C can share storage.

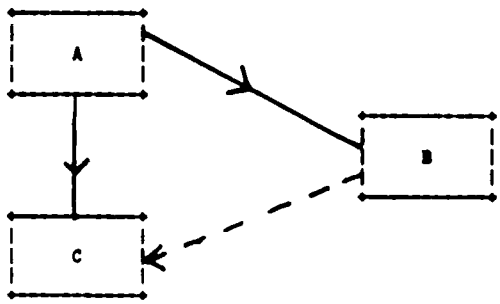


Figure 5-1: The *read before write* heuristic

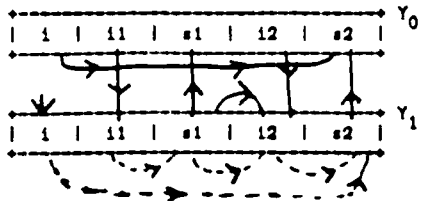
This heuristic can be applied to our case as follows: Say that we have a production of the

form [p: X_0 ::= ... X_j ...] where $X_0 = X_j$ and the inherited attribute X_0.i1 is used to define the inherited attributes X_j.i1 and X_j.i2. In this case X_0.i1 plays the role of A, X_j.i2 plays the role of B, and X_j.i1 plays the role of C. As X_j.i2 would be evaluated before X_j.i1, X.i1 can be made into a stack. Similarly, if the synthesized attribute X_j.s1 is used to define the synthesized attributes X_0.s1 and X_0.s2, then X_j.s1 plays the role of A, X_0.s2 plays the role of B, and X_0.s1 plays the role of C. This heuristic will allow GAG to implement many more attributes as global stacks. Note, however, that this heuristic will not allow all transient attributes to be implemented as global stacks; if both X_0.i1 and X_0.i2 are used to define both X_j.i1 and X_j.i2 then application of this heuristic could add either the edge (X_j.i1, X_j.i2) or (X_j.i2, X_j.i1) but not both.

Now we return to the first stage of determining an evaluation order, assigning visit numbers to attributes, and we show how heuristics can operate even at this early stage to increase the number of transient attributes. In order to determine an assignment of visit numbers to the attributes of X, a graph G_X is formed. This graph has the attributes of X as vertices and an edge (X.a, X.b) if X.b is directly or indirectly dependent upon X.a in some semantic tree. This graph will result in a partial ordering on the attributes of X. In order to make an assignment of visit numbers to the attributes of X, this partial order must be extended so that for every inherited attribute X.i and synthesized attribute X.s, either (X.i, X.s) or (X.s, X.i). Different evaluation strategies use different strategies to extend this partial order. A "greedy" strategy calls for evaluating attributes on the earliest visit possible. This strategy is used by LINGUIST-86 in assigning pass numbers to attributes. A "lazy" strategy calls for evaluating attributes on the last visit possible. This method is inherent in GAG's ordered evaluation strategy. Each of these strategies will sometimes make an attribute transient where the other fails to do so. In figure 5-2, the dotted lines indicate the graph G_Y. This graph gives a partial order <Y.i1, Y.s1, Y.i2, Y.s2> and <Y.i, Y.s2>. This partial order must be extended so that either (Y.i, Y.s1) or (Y.s1, Y.i). The greedy strategy evaluates Y.i as early as possible, extending the order to include (Y.i, Y.s1) and resulting in the visit sequence VS_p. The lazy strategy evaluates Y.i as late as possible, extending the order to include (Y.s1, Y.i) and resulting in the visit sequence VS_p'. Whereas Y.i is a transient attribute using VS_p', it is a significant attribute using VS_p.

Figure 5-3 gives another attribute grammar fragment. Here also the two strategies result in different visit numbers being assigned to X.i3. Whereas the greedy strategy results in the visit sequence VS_p and X.i1 being a transient attribute, the lazy strategy results in the visit sequence VS_p' and X.i1 being a significant attribute. Hence we see that each strategy can make some attributes transient that the other makes significant.

An augmented dependency graph for a production [p: Y_0 ::= Y_1] (where Y_0 and Y_1 are 2 instances of the same nonterminal)

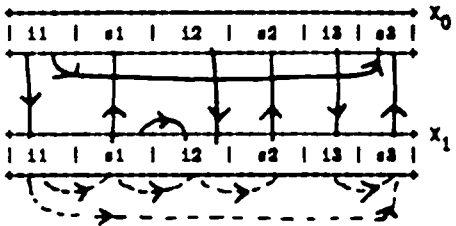


VS_p = (EVAL Y_1.i) (EVAL Y_1.i1) (VISIT Y_1) (EVAL Y_0.s1) (VISIT$_0$) (VISIT Y_1.i2) (VISIT Y_1) (EVAL Y_0.s2) (VISIT$_0$) *GREEDY*

VS_p' = (EVAL Y_1.i1) (VISIT Y_1) (EVAL Y_0.s1) (VISIT$_0$) (EVAL Y_1.i) (EVAL Y_1.i2) (VISIT Y_1) (EVAL Y_0.s2) (VISIT$_0$) *LAZY*

Figure 5-2: The advantage of using a lazy strategy

An augmented dependency graph for a production [p: X_0 ::= X_1] (where X_0 and X_1 are 2 instances of the same nonterminal)



VS_p = (EVAL X_1.i1) (EVAL X_1.i3) (VISIT X_1) (EVAL X_0.s1) (EVAL X_0.s3) (VISIT$_0$) (EVAL X_1.i2) (VISIT X_1) (EVAL X_0.s2) (VISIT$_0$) *GREEDY*

VS_p' = (EVAL X_1.i1) (VISIT X_1) (EVAL X_0.s1) (VISIT$_0$) (EVAL X_1.i2) (EVAL X_1.i3) (VISIT X_1) (EVAL X_0.s2) (EVAL X_0.s3) (VISIT$_0$) *LAZY*

Figure 5-3: The advantage of using a greedy strategy

Instead of adopting a purely greedy strategy or a purely lazy strategy, we suggest that heuristics be designed to increase the number of transient attributes. These heuristics would describe how to extend the partial order of the G_X graphs, thereby assigning visit numbers to the attributes of X, and would use the information contained in the G_X graphs and the dependency graphs of the productions. In [4] Farrow describes how such heuristics could be integrated into an algorithm for computing the evaluation order.

Figures 5-4 and 5-5 graphically illustrate one possible heuristic for adding edges to the G_X graphs, designed to make attributes transient. To use this heuristic we need to distinguish between two different kinds of edges that may occur in a G_X graph, *transitive closure* edges and *defining* edges. A defining edge (X.a, X.b) in a G_X graph indicates that in some production X.a is an argument to the semantic function defining X.b. A transitive closure edge (X.a, X.b) indicates that X.a can indirectly define X.b. A defining edge in a G_X graph

is distinguished from a transitive closure edge by the word "def" which appears over the arrow. An arrow without a "def" marker may be *either* a transitive closure edge or a defining edge.

CASE i: a and c are synthesized attributes, b and d are inherited attributes, a and d are currently unrelated: add (a, d)

CASE ii: c and e are synthesized attributes, b and d are inherited attributes, b and e are currently unrelated: add (b, e)

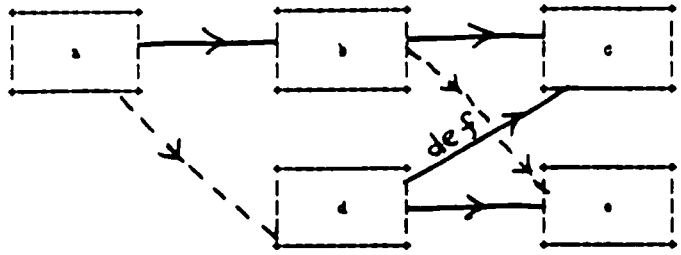


Figure 5-4: A heuristic for adding edges, cases i and ii

CASE iii: a and c are inherited attributes, b and d are synthesized attributes, c and d are currently unrelated: add (d, c)

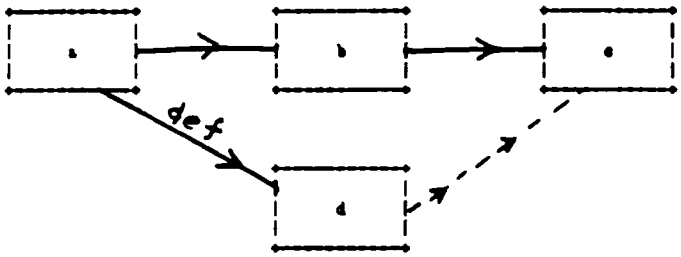


Figure 5-5: A heuristic for adding edges, case iii

Case i of the heuristic states that if **a** and **c** are synthesized attributes, **b** and **d** are inherited attributes, (**a**, **b**) and (**b**, **c**) are edges in G_X[3], (**d**, **c**) is a defining edge in G_X, and there is no relationship in G_X between **a** and **d**, then add the edge (**a**, **d**) to G_X. In order to understand the logic behind this heuristic, consider the consequence of adding the "opposite" edge (**d**, **a**) to G_X. Any assignment of visit numbers to the attributes of X based on this graph will necessarily assign an earlier visit number to **d** then to **b**, since there exists a path from **d** to **b**. Hence **d**'s lifetime must start on a visit prior to the one in which **b** is defined. Yet since there exists an edge from **b** to **c**, **c** must be defined after **b**, and since there is a defining edge from **d** to **c**, **d**'s lifetime must extend into the visit defining **c**. Hence **d** would have to be a significant attribute. To prevent this from happening, the edge (**a**, **d**) is added. A similar logic applies to cases ii and iii of the

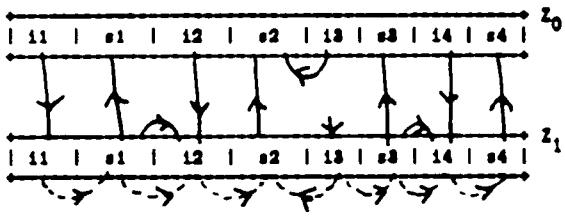
[3] when we say that (x,y) is an edge without specifying its type, then it can be either a defining or transitive closure edge

heuristic.

Let us attempt to add edges to the graph G_Y of figure 5-2 using this heuristic. We find that case i can be applied to this graph, with Y.s1 playing the role of **a**, Y.i2 the role of **b**, Y.s2 the role of **c**, and Y.i the role of **d**. This causes the addition of the edge (Y.s1, Y.i), resulting in the same visit sequence as produced by the lazy strategy and making all attributes transient. If we apply the heuristic to the example of figure 5-3, once again all the attributes are made transient. This time, however, it is the greedy visit sequence which is produced. In this example cases ii and iii of the heuristic are found to be applicable. In case ii, X.i1 plays the role of **d**, X.s1 plays the role of **e**, X.i3 plays the role of **b**, and X.s3 plays the role of **c**, causing the addition of the edge (X.i3, X.s1). In case iii, X.i1 plays the role of **a**, X.s1 plays the role of **b**, X.i2 plays the role of **c**, and X.s3 plays the role of **d**, causing the addition of the edge (X.s3, X.i2) and completing the G_X graph.

Given any G_X graph which can be completed so as to make all inherited attributes transient, this heuristic, unlike the greedy and lazy strategies, will not add any edges forcing an inherited attribute to be significant. However, it does not guarantee to necessarily complete the G_X graph at all. Therefore, it may still be necessary to apply one of the other strategies or another heuristic after applying this heuristic. Nonetheless, a final example, given in figure 5-6 illustrates the power of method. In this example, both the greedy and lazy strategies result in the visit sequence VS_p, making Z.i3 significant. The heuristic results in the visit sequence VS_p', making all attributes transient.

An augmented dependency graph for a production [p: Z_0 ::= Z_1] (where Z_0 and Z_1 are 2 instances of the same nonterminal)



VS_p = (EVAL Z_1.i1) (EVAL Z_1.i3) (VISIT Z_1) (EVAL Z_0.s1) (EVAL Z_0.s3) (VISIT$_0$) (EVAL Z_1.i2) (EVAL Z_1.i4) (VISIT Z_1) (EVAL Z_0.s2) (EVAL Z_0.s4) (VISIT$_0$) *GREEDY / LAZY*

VS_p' = (EVAL Z_1.i1) (VISIT Z_1) (EVAL Z_0.s1) (VISIT$_0$) (EVAL Z_1.i2) (EVAL Z_1.i3) (VISIT Z_1) (EVAL Z_0.s2) (EVAL Z_0.s3) (VISIT$_0$) (EVAL Z_1.i4) (VISIT Z_1) (EVAL Z_0.s4) (VISIT$_0$) *HEURISTIC*

Figure 5-6: The advantage of using the heuristic

Finally, we contemplate the following basic organization of the evaluator-generator to make better use of heuristics for increasing storage optimizations:

1. partition the attributes into equivalence classes based on whether X.A is copied to Y.B in any production. The members of an equivalence class will be candidates to share storage for the heuristics of this section,
2. assign visit numbers to attributes using the heuristics of this section to make as many attributes as possible transient,
3. determine the final visit sequence for each production using the heuristics of this section to increase the number of attributes that can be implemented as global variables and stacks, and
4. finally, carry out the current storage optimizations, using the improvements suggested in section 4.

6. Conclusions

In this paper we have examined in depth the storage optimizations performed by two significant AG-based translator-writing systems: GAG and LINGUIST-86. This examination has been illustrated by many small, but concrete, examples showing how each system performs. We have seen that, although there are significant differences between the two, there are also very basic similarities that unite their approaches. We have argued that both strategies would benefit by:

1. taking storage optimization into consideration when determining evaluation order,
2. using global analysis to increase the number of attributes that are implemented as global variables, implemented as global stacks, and statically subsumed, and
3. combining attributes implemented by global variables or stacks based on whether there are any copy rules between them and on how much stack space would be saved by combining them.

AKNOWLEDGEMENT

We would like to thank Phillip Garrison of the University of California, Berkely for helping us run the examples through GAG and Ron Farrer of Intel Corporation for helping us run the examples through LINGUIST-86.

I. APPENDIX

In this appendix we show that combining global variables and global stacks so as to optimize the number of copy rules eliminated is an NP-complete problem. Since the problem of combining global variables and global stacks are the same, we will focus our attention on combining global stacks, bearing in mind that the analysis is equally valid for global variables. Recall from section 2.3 that the main restriction on combining global stacks is that no two inherited attributes nor two synthesized attributes of the same context-free symbol are allowed to share the same stack. In addition, if X is the left-part of some production p, Y is on the right-hand side, and the inherited attribute X.i is referenced after defining the inherited attribute Y.i but before visiting Y, then the X.i and Y.i global stacks cannot be combined.

Let us phrase the problem of combining global stacks as one of partitioning a set: Given a set S of attributes, each which can be made into a global stack, we wish to find a partition of the set such that all the attributes in any subset of the partition can be made into one global stack. We call such a partition *valid*. We would like to find an *optimal valid partition*- one which eliminates as many copy rules as possible. We shall now prove the following theorem:

Theorem 1: Given an attribute grammar G and a set S of attributes, each which can be made into a global stack, finding whether there exists a valid partition of the attributes into global stacks such that at least K copy rules are eliminated is NP-complete.

Proof: Certainly the problem is in NP (Guess a partition. Verification can be done in P-time).

To show that it is NP-complete, we shall reduce 3-satisfiability to it. This shall be done as follows: given any instance of 3-satisfiability, we shall create an attribute grammar for that instance in p-time and show that for a certain K, we can eliminate K copy rules from the attribute grammar *iff* the given clause is satisfiable. To this end let $U = \{u_1,...,u_n\}$ and $C = \{c_1,...,c_m\}$ be any instance of 3-satisfiability. (U is the set of literals, C of clauses). Let the context-free symbols of our grammar be $\{X, u_1, u_2, ..., u_n, c_1, c_2, ..., c_m\} \cup \{ S \} \cup \{TermA_1, ..., TermA_{100m}\} \cup \{TermB_1, ..., TermB_{100m}\} \cup \{AttA_1,...,AttA_{50m}\} \cup \{AttB_1,...,AttB_{50m}\} \cup \{(i,j) \mid 1 \leq i \leq m, 1 \leq j \leq 10\}$

The $TermA_j$, $TermB_j$, $AttA_j$, $AttB_j$, and (i,j) context-free symbols do not have any attributes and are used only to distinguish between different productions. The symbol S is the distinguished start symbol. The associated attributes of each symbol and the

productions of the attribute grammar are given in figure I-1. The context-free symbols which do not have any attributes are not listed. Let ATTRIBUTES be the set of all of these attributes. (There are 3n + m + 1 attributes). Note that individually, each attribute in this set can be made into a global stack and that any two attributes of different context-free symbols can share the same stack[4]. This attribute grammar doesn't have any synthesized attributes. This is not very realistic but they not needed for the proof. It would be a simple matter to augment this attribute grammar with synthesized attributes.

context-free symbol	inherited transient attributes
X	⟨X.1, X.2, ..., X.n, X.att⟩
u_i $(1 \leq i \leq n)$	$\langle u_i.pos, u_i.neg \rangle$
c_j $(1 \leq j \leq m)$	$\langle c_j.att \rangle$

There will be 4 types of productions in addition to a single start production of type 0:

Type 0: S ::= X.

X.1 = constant$_1$;
X.2 = constant$_2$;
.
.
.
X.n = constant$_n$;
X.att = constant$_{n+1}$;

Type 1A: X ::= u_i TermA$_j$. $1 \leq i \leq n$; $1 \leq j \leq 100m$.

u_i.pos = X.i;
u_i.neg = constant;

Type 1B: X ::= u_i TermB$_j$. $1 \leq i \leq n$; $1 \leq j \leq 100m$.

u_i.pos = constant;
u_i.neg = X.i;

Type 2A: X ::= u_i AttA$_j$. $1 \leq i \leq n$; $1 \leq j \leq 50m$.

u_i.pos = X.att;
u_i.neg = constant;

Type 2B: X ::= u_i AttB$_j$. $1 \leq i \leq n$; $1 \leq j \leq 50m$.

u_i.pos = constant;
u_i.neg = X.att;

Type 3: X ::= c_i (i,j). if u_i or u_i' is in c_i, $1 \leq j \leq 10$.

c_i.att = X.i;

Type 4: u_i ::= c_i. if u_i or u_i' is in c_i.

c_i.att = u_i.pos if u_i is in c_i, c_i.att = u_i.neg if u_i' in c_i.

Figure I-1: The attribute grammar constructed from an instance $U = \{u_1,....,u_n\}$, $C = \{c_1,....,c_m\}$ of satisfiability

The basic idea behind the proof will be as follows: we will attempt to partition the attributes into n + 1 subsets corresponding to n + 1 global stacks. For i = 1,....,n if u_i is

---

[4]Actually, in the attribute grammar as given, each attribute could be made into a global variable. But by adding one production for each of these attributes the grammar can be easily modified so that these attributes must be global stacks, not global variables. The grammar is left as is to show the applicability of the theorem to global variables as well as global stacks.

true, u_i.pos will be in the i^{th} subset with X.i along with the clause attributes c_l.att corresponding to the clauses which u_i satisfies. If u_i is false, u_i.neg will be in the i^{th} subset with X.i along with the clause attributes c_l.att corresponding to the clauses which u_i' satisfies. All the remaining attributes will be in the $(n + 1)^{st}$ subset along with the attribute X.att. There will exist such a partition eliminating the proper number of copy rules iff C is satisfied.

Formally, we claim that C is satisfiable *iff* there exists a *valid partition* of ATTRIBUTES into subsets $S_1, S_2, \ldots, S_t$ corresponding to t global stacks such that at least 150mn + 11m copy rules are eliminated.

150mn copy rule eliminations will come from productions of type 1 and 2 and will insure a partition into n + 1 subsets with, for each i ($1 \leq i \leq n$), either u_i.pos in S_i and u_i.neg in S^* ($= S_{n+1}$) or u_i.neg in S_i and u_i.pos in S^*. 10m copy rules will come from productions of type 3 and will insure that each clause attribute c_l.att ($1 \leq l \leq m$) is in some S_i, $1 \leq i \leq n$. m copy rules will come from productions of type 4 and will insure that if c_l.att is in S_i then either u_i.pos is in S_i and u_i is in c_l or u_i.neg is in S_i and u_i' is in c_l. We can define a truth assignment τ to be such that $\tau(u_i) = T$ if u_i.pos is in S_i and $\tau(u_i) = F$ if u_i.neg is in S_i. In this way we will develope a 1-1 correspondance between valid partitions eliminating 150mn + 11m copy rules and truth assignments which satisfy U.

Hence if the copy rule elimination problem were solvable in p-time, so would be the satisfiability problem. Given U and C, create the AG as above, containing 3n + m + 1 attributes and 300nm + 33m + 1 productions with 300mn + 33m copy rules. This reduction can be done in p-time. The attributes of the attribute grammar make up a set of inherited transient attributes, each which can be made into a global stack. Then find whether a valid partition of the 3n + m + 1 attributes exists such that at least 150mn + 11m copy rules are eliminated. If one can be found then there exists a truth assignment satisfying U, otherwise not.

Proof of claim:

For the proof, it helps to keep the following in mind:

1. If u_i.pos and X.i are in the same subset (stack), then we can eliminate 100m copy rules from productions of type 1A.

2. If u_i.neg and X.i are in the same subset, then we can eliminate 100m copy rules from

productions of type 1B.

3. If u_i.pos and X.att are in the same subset, then we can eliminate 50m copy rules from productions of type 2A.

4. If u_i.neg and X.att are in the same subset, then we can eliminate 50m copy rules from productions of type 2B.

5. If X.i and c_l.att are in the same subset, where u_i or u_i' is in c_l, then we can eliminate 10 copy rules from productions of type 3.

6. If u_i.pos and c_l.att are in the same subset, where u_i is in c_l then we can eliminate 1 copy rule from a prod of type 4.

7. If u_i.neg and c_l.att are in the same subset, where u_i' is in c_l, then we can eliminate 1 copy rule from a production of type 4.

8. X.i and X.j cannot be in the same subset if $i \neq j$ and similiarly X.i cannot be in the same subset with X.att. Also, u_i.pos and u_i.neg cannot be in the same subset since they are attributes of the same context-free symbol.

$\Rightarrow$) Say there exists a truth assignment τ to U satisfying C. The following partition of ATTRIBUTES eliminates 150mn + 11m copy rules:

To each clause c_l associate an integer $int(c_l) = i$ such that either u_i or u_i' satisfies c_l under τ, and if u_j or u_j' satisfies c_l under τ, then $j > i$. (If $int(c_l) = i$ then u_i or u_i' is the smallest numbered literal satisfying c_l under τ).

Let our partition be S_1, ..., S_n,S^* where $S_i = \{X.i\} \cup \{c_l.att \mid int(c_l) = i\} \cup \{u_i.pos \mid \text{if } \tau(u_i) = T\} \cup \{u_i.neg \mid \text{if } \tau(u_i) = F\}$. $S^* = \{X.att\} \cup \{u_i.pos \mid \text{if } \tau(u_i) = F\} \cup \{u_i.neg \mid \text{if } \tau(u_i) = T\}$.

Certainly this partition is valid as it doesn't violate having 2 attributes of the same context-free symbol in the same subset. To see that the required number of copy rules are eliminated, note that since X.i and u_i.pos or u_i.neg are in the same subgroup, we can eliminate 100m copy rules from productions of type 1 for each i, $1 \leq i \leq n$. This totals 100mn copy rule eliminations. Furthermore, since each c_l.att is in the partition S_i, where $i = int(c_l)$, c_l.att is in the same partition with X.i and u_i.pos (if u_i is in c_l) or in the same partition with X.i and and u_i.neg (if u_i' is in c_l). Hence each c_l eliminates 10 copy rules of

type 3 and 1 of type 4. All m clauses therefore cause an elimination of 11m copy rules. Finally, as S^* contains X.att and either u_i.pos or u_i.neg for each i, $1 \leq i \leq n$, we can eliminate 50mn copy rules from productions of type 2, for an elimination of an additional 50mn copy rules. So a total of 150mn + 11m copy rule eliminations is achieved. Figure I-2 gives a set of clauses C, a truth assignment τ, and the attribute partition induced by the above method. For the attribute grammar derived from C and U, this partition would result in the elimination of 150*4*3 + 11*3 copy rules.

The set of clauses $C = (\langle u_1, u_2\rangle, \langle u_3, u_4\rangle, \langle u_2', u_3'\rangle)$.

The truth assignment $\tau(u_1, u_2, u_3, u_4) = (T,T,F,T)$.

$\underline{S_1}$	$\underline{S_2}$	$\underline{S_3}$	$\underline{S_4}$	$\underline{S^*}$
$\langle X.1, u_1.pos, c_1\rangle$	$\langle X.2, u_2.pos\rangle$	$\langle X.3, u_3.neg, c_3\rangle$	$\langle X.4, u_4.pos, c_2\rangle$	$\langle X.att, u_1.neg, u_2.neg, u_3.pos, u_4.neg\rangle$

Figure I-2: A partition of attributes induced by τ

$\Leftarrow$) If there exists a partition of ATTRIBUTES into subsets $S_1, S_2, \ldots, S_t$ such that at least 150mn + 11m copy rules are eliminated, then there exists a truth assignment τ to U satisfying C.

By means of the fact that 150mn + 11m copy rules were eliminated we can deduce what form the partition has. Out of the 200mn copy rules in productions of type 1, we can eliminate at most 100mn of them, as both u_i.pos and u_i.neg cannot be in the same partition as X.i. Similiarly, at most 50mn of the 100mn copy rules in productions of type 2 can be eliminated as we cannot have u_i.pos and u_i.neg in the same partition as X.att. So at most 150mn copy rules were eliminated from productions of types 1-2. If the partition did not eliminate *all* of these 150mn copy rules it would not be able to achieve 150mn + 11m copy rule eliminations. To see why this is true, note that for $1 \leq i \leq n$, each u_i.pos and u_i.neg has associated with it either 100m, 50m, or 0 copy rule eliminations from productions of types 1-2, depending whether it is in the partition with X.i, X.att, or neither of these, and that the total number of eliminations can be found by summing the number of eliminations associated with each individual u_i.pos and u_i.neg. Since the partition achieving the maximum number of copy rule eliminations from these productions achieves 150mn eliminations, any partition which causes less eliminations achieves *at most* 150mn - 50m eliminations; i.e., any u_i.pos or u_i.neg which eliminates fewer copy rules than it does in the maximum partition eliminates at least 50m fewer copy rules. But then the non-maximum partition achieves at most 150mn - 50m + 33m = 150mn - 17m < 150mn + 11m copy rule eliminations, as the remaining productions of types 3 - 4 contain only 33m copy rules. Hence, we see that to achieve 150mn + 11m copy rule eliminations, the partition must

achieve 150mn copy rule eliminations from productions of types 1 - 2.

Achieving 100mn copy rule eliminations from productions of type 1 implies that for each i, $1 \leq i \leq n$, X.i is in the same subset with either u_i.pos or u_i.neg. Achieving 50mn copy rule eliminations from productions of type 2 implies that X.att is in a subset with l1, l2, ..., ln where $li = u_i$.neg if u_i.pos is in the subset with X.i and $li = u_i$.pos if u_i.neg is in the subset with X.i. So we know that the partition consists of at least n + 1 subsets, $S_1, S_2, \ldots, S_n, S^*$ where for each i, $1 \leq i \leq n$, S_i contains X.i and either u_i.neg or u_i.pos. S^* contains X.att and l1, ..., ln. The only remaining question is what subset each of the clause attributes, c_l.att, $1 \leq l \leq m$, fall into. As we must still find 11m copy rule eliminations, this choice is also already made for us. Note that each clause attribute c_l.att can appear in a subset with at most one X.i (it cannot appear in a subset with X.i and X.j if $i \neq j$), hence at most 10m copy rules can be eliminated from productions of type 3 (10 for each clause c_l). To get 11m copy rule eliminations, each clause attribute must also contribute one copy rule elimination from productions of type 4. For this to occur we must have each clause attribute c_l.att ($1 \leq l \leq m$) meet the following condition: c_l.att is in the subset S_i and either *i)* u_i.pos is in S_i and u_i is in c_l or *ii)* u_i.neg is in S_i and u_i' is in c_l. To summarize, we have found that if a valid partition achieves 150mn + 11m copy rule eliminations it must be of the form $S_1, \ldots, S_n, S^*$ with either u_i.pos in S_i and u_i.neg in S^* or u_i.neg in S_i and u_i.pos in S^*. Each clause attribute c_l.att is in some S_i, $1 \leq i \leq n$ and the following property holds: if c_l.att is in S_i then either u_i.pos is in S_i and u_i is in c_l or u_i.neg is in S_i and u_i' is in c_l. We can now define π to be:

$$\pi(u_i) = \begin{cases} T \text{ if } S_i \text{ contains } u_i.pos. \\ F \text{ if } S_i \text{ contains } u_i.neg. \end{cases}$$

This satisfies C, as for any clause c_l, *i)* c_l.att is in S_i for some i, $1 \leq i \leq n$, and *ii)* if u_i.pos is in S_i then u_i is in c_l and $\pi(u_i) = T$ and if u_i.neg is in S_i then u_i' is in c_l and $\pi(u_i) = F$.

End of proof

References

[1] B. Asbrock, U. Kastens, and E. Zimmermann.
Generating an Efficient Compiler Front-End.
Technical Report 17/81, Universitat Karlsruhe, Fakultat Fur Informatik, 1981.

[2] Rodney Farrow.
LINGUIST-86 Yet another translator writing system based on attribute grammars.
In *Proceedings of the SIGPLAN 82 Symposium on Compiler Construction.* ACM, June, 1982.

[3] Rodney Farrow.
Experience with an attribute grammar based compiler.
In *Conference Record of the Ninth ACM Symposium on Principles of Programming Languages.* ACM, January, 1982.

[4] Rodney Farrow.
Covers of Attribute Grammars and Sub-Protocol Attribute Evaluators.
Technical Report, Department of Computer Science, Columbia University, New York, New York 10027, September, 1983.

[5] Rodney Farrow.
Generating a Production Compiler from an Attribute Grammar.
IEEE Software 1(4), October, 1984.

[6] H. Ganzinger.
On storage optimization for automatically generated compilers.
In K. Weirauch (editor), *Theoretical Computer Science - Fourth GI Conference,* . Springer-Verlag, Berlin-Heidelberg-New York, 1979.

[7] M. Jazayeri and K.G. Walter.
Alternating semantic evaluator.
In *Proceedings of ACM 1975 Annual Conference.* ACM, 1975.

[8] Martin Jourdan.
Strongly Non-Circular Attribute Grammars and their Recursive Evaluation.
In *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction.* ACM-SIGPLAN, June, 1984.
Published as Volume 19, Number 6, of *SIGPLAN Notices.*

[9] U. Kastens.
Ordered attribute grammars.
Acta Informatica 13:229-256, 1980.

[10] Uwe Kastens, Brigitte Hutt, and Erich Zimmermann.
GAG:A Practical Compiler Generator.
In *Lecture Notes in Computer Science 141,* . Spring-Verlag, Berlin-Heidelberg-New York, 1982.

[11] T. Katayama.
Translation of Attribute Grammars into Procedures.
ACM TOPLAS 6(3), July, 1984.

[12] K. Kennedy and S. K. Warren.
Automatic generation of efficient evaluators for attribute grammars.
In *Conference Record of the Third ACM symposium on Principles of Programming Languages.* ACM, 1976.

[13] Kari-Jouko Raiha.
Dynamic allocation of space for attribute-instances in multi-pass evaluators of attribute grammars.
In *Proceedings of the SIGPLAN 79 Symposium on Compiler Construction.* ACM, 1979.

[14] Thomas W. Reps.
Generating Language-Based Environments.
PhD thesis, Cornell University, Ithaca, New York, December, 1983.

[15] M. Saarinen.
On constructing efficient evaluators for attribute grammars.
In C. Ausiello and C. Bohm (editor), *Automata, Languages, and Programming: 5th Colloquium.* Springer-Verlag, Springer-Verlag, New York, 1978.

[16] W.A. Schulz.
Semantic analysis and target language synthesis in a translator.
PhD thesis, University of Colorado, Boulder, Colorado, July, 1976.

[17] Ravi Sethi.
Pebble Games For Studying Storage Sharing.
TheoreticalComputerScience 19, 1982.
pp. 69-84.

[18] Daniel M. Yellin.
A Survey of Tree-Walk Evaluation Strategies for Attribute Grammars.
Technical Report, Department of Computer Science, Columbia University, New York, New York 10027, September, 1984.