LINGUIST-86
Yet Another Translator Writing System Based On Attribute Grammars

Rodney Farrow
Intel Corporation

ABSTRACT

LINGUIST-86 is a commercially-developed translator-writing-system based on attribute grammars [K]. From an input attribute grammar it generates a set of high-level language source modules that form an alternating-pass attribute evaluator [JW]. LINGUIST-86 generates attribute evaluators efficient enough to run on a microcomputer at speeds competitive with other translators on the system. The Attributed Program Tree is kept on secondary storage rather than in randomly-accessed memory, thus allowing non-trivial inputs to be evaluated on a microcomputer system. LINGUIST-86 also applies an optimization called static subsumption that eliminates many copy-rules from the generated evaluators. LINGUIST-86 is itself written as an 1800 line attribute grammar and is self-generating.

Introduction

Translator-writing-systems based on attribute grammars have been maturing in the academic computer science community for the past decade [B] [F] [L] [GRW] [JW] [RST] [KZ]. LINGUIST-86 is a new translator-writing-system based on attribute grammars. It was developed in a commercial software development environment and will be used to build compiler and translator products. LINGUIST-86 generates attribute evaluators written in high-level programming languages, including Pascal. There are three aspects of LINGUIST-86 that make it particularly noteworthy. Two of these, the basic attribute evaluation paradigm and an optimization we call static subsumption, contribute to the efficiency of the automatically generated evaluators. The third is that LINGUIST-86 is itself written as a 1800-line attribute grammar and is self-generating.

One of the weaknesses of automatic translator generation from attribute grammars is that the resulting evaluators require a lot of memory, much of it to store the Attributed Parse Tree (APT). LINGUIST-86 supports an attribute evaluation paradigm that stores most of the APT on sequentially-accessed mass-storage such as disk or tape [S]. About 48K bytes of memory are available to LINGUIST-86 for holding dynamic data that represents the input attribute grammar. In this space LINGUIST-86 puts : part of the APT; name-table entries that store the source text of identifiers; dictionary entries for grammar symbols, attributes, semantic functions; the linked lists that represent sets, sequences, and partial functions; and the run-time stack that supports recursion and procedure call/return. Even though the APT for the LINGUIST-86 attribute grammar is more than 42K bytes long, everything fits because at any one time most of the APT is stored in temporary disk files. An integral part of this evaluation paradigm is that attributes must be evaluated in alternating passes [J] [JW] [PJ1]. Consequently, LINGUIST-86 generates evaluators only for those attribute grammars that can be evaluated in alternating passes.

LINGUIST-86 avoids another potential inefficiency of automatically generated attribute evaluators: the typical abundance of copy-rules. Copy-rules are semantic functions that copy attribute values around the APT without changing them. They are especially important to "locality of reference", one of the attractive properties attribute grammars have for specifying the semantics of languages. However, a straight-forward implementation by explicit copying of values can be quite inefficient when, as in many attribute grammars, between 40 and 60 percent of the semantic functions are copy-rules. LINGUIST-86 implements an optimization, called static subsumption, that eliminates most copy-rules but does not require any modification to the attribute grammar formalism. Not modifying the form of attribute grammars is important. Raiha [Ra] reports that attribute grammars which have been augmented with "global attributes" (partially as an alternative to numerous explicit copy-rules) are not as easy to write and understand, especialy for non-expert users.

The remainder of this paper is organized into five sections. The first section contains a brief introduction to attribute grammars and our terminology and notation. The second section discusses the attribute evaluation paradigm and LINGUIST-86's implementation of it. Optimizations of the attribute evaluator are considered in section three; static subsumption is discussed in this section. The actual form of the input to LINGUIST-86 is presented in section four and the attribute grammar for LINGUIST-86 is briefly described. Section five describes some of the operational characteristics of LINGUIST-86 as a program.

I. An Introduction to Attribute Grammars

Attribute grammars were first proposed by Knuth [K] as a way to specify the semantics of programming languages. The basis of an attribute grammar is a context-free grammar that describes the abstract syntax of the language. This context-free grammar is augmented with ATTRIBUTES and SEMANTIC FUNCTIONS. Attributes are associated with the symbols of the grammar, both terminal and non-terminal. We write "S.A" to denote attribute A of symbol S. Semantic functions are associated with productions.

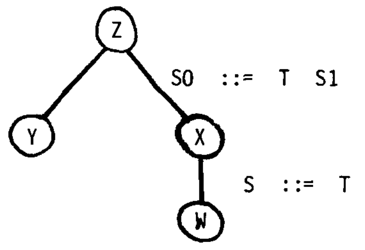
```
1  S0 ::=  V S1
2     S1.A = IncrIfZero(T.B, S0.A)
3     S0.C = S1.C

4  S  ::=  T
5     S.C = IncrIfZero(T.B, S.A)
```

Figure 1

Lines 2,3 and 5 of figure 1 are examples of semantic functions. The notation of this example is widely used. In the production of line 1, S0 and S1 denote separate occurrences of the same symbol, the numeric suffixes distinguish both the symbols and their attributes. A semantic function specifies the value to be associated with an attribute of some symbol of the production, e.g. S1.A. Semantic functions are pure functions, i.e. no side-effects. Their only arguments are attributes of the production.

The underlying context-free grammar of an attribute grammar describes a language. Any string in this language has a parse tree associated with it by the grammar. The nodes of this parse tree can be labelled with symbols of the grammar. Each interior node of this tree has two productions associated with it. The left-hand-side (LHS) production is the production that derives this node. The children of this node are labelled with the symbols in the right-hand-side of the LHS production. The right-hand-side (RHS) production is the production that applies at the parent of this node, which is labelled with the left-hand-side symbol of the RHS production. This node and the siblings of this node are labelled with the symbols in the right-hand side of the RHS production. Leaves of the tree don't have LHS productions; the root doesn't have a RHS production. Figure 2 shows a tree fragment that contains a node X and its parent, siblings and children, as well as the LHS production and RHS production associated with X.



An APT Fragment
Figure 2

An ATTRIBUTED PARSE TREE (APT) is a parse tree in which each node contains fields that correspond to the attributes of its labelling grammar symbol. Each of these fields is an ATTRIBUTE-INSTANCE. Each attribute has a set of possible values analogous to the "type" of a variable in a programming language. However, each attribute-instance takes on precisely one such value; attribute-instances are not variables. The values of attribute-instances are specified by the semantic functions.

The semantic functions of a production represent a template for specifying the values of attribute-instances in the APT.

Consider figure 2 again. X is an APT node that is associated with S1 in production "S0 ::= T S1" and X is associated with S in "S ::= T". The semantic function "S1.A = IncrIfZero(S0.A, T.B)" indicates that the value of attribute-instance X.A will be the result of evaluating IncrIfZero(Z.A, Y.B), where Z and Y are other APT nodes (that correspond to grammar symbols S and T) as shown in the figure. Similarly, the semantic function "S0.C = S1.C" indicates that attribute-instances Z.C and X.C will have the same value.

Since two different productions are associated with each attribute-instance, there could be two semantic functions that independently specify its value, one from the LHS production and one from the RHS production. Attribute grammars avoid require that either all instances of a particular attribute be defined in their LHS production, or that they all be defined in their RHS production. Attributes whose instances are all defined in their LHS production are called SYNTHESIZED attributes; attributes whose instances are all defined in the RHS production are called INHERITED attributes. Every attribute is either inherited or synthesized. The start symbol has no inherited attributes; terminal symbols have no synthesized attributes. From the point of view of an individual production these conditions require that the semantic functions of a production MUST define EXACTLY the right-hand-side occurrences of inherited attributes and all synthesized attributes of the left-hand symbol. Inherited attributes propagate information down the tree, towards the leaves. Synthesized attributes propagate information up the tree, toward the root.

Although the semantic functions of an attribute grammar specify a unique value for each attribute-instance, in order to actually compute this value the values of any other attribute-instances that are arguments of the defining semantic function must first be available. In the example of figure 2, before X.A can be computed Z.A and Y.B must have already been computed. Such dependency relations restrict the order in which attribute-instance can be evaluated. In extreme cases an attribute-instance can depend on itself; such a situation is called a circularity and occurs only in ill-defined attribute grammars. In general, it is an exponentially hard problem [JOR] to determine that an attribute grammar is non-circular; i.e. no APT that can be generated by the attribute grammar contains a circularly defined attribute-instance. Fortunately there are several interesting and widely applicable sufficient conditions that can be checked in polynomial time [B] [JW] [KW] [Ka].

The result of the translation specified by an attribute grammar is realized as the values of one or more (necessarily synthesized) attribute-instances of the root of the APT. In order to compute these values the other attribute-instances must be computed. An ATTRIBUTE EVALUATION PARADIGM is a meta-algorithm for computing attribute-instances such that no attribute-instance is computed before all dependent attribute-instances are available and such that all attribute-instances of the root are computed. An attribute evaluation paradigm may work correctly only on a subset of all well-defined attribute grammars, but it must work correctly on any APT of an acceptable attribute grammar.

Attribute grammars are attractive specification tools. Two principal reasons for this are their LOCALITY OF REFERENCE and their NON-PROCEDURAL NATURE. We say that an attribute grammar has locality of reference in that the values it defines (i.e. the attribute-instances) are specified exclusively in terms of other attribute-instances local to a production. An attribute grammar does not contain any global variables or hidden state which can affect the translation. Each local piece of an attribute grammar, i.e. production, communicates with the outside world only through strictly defined interfaces: the attribute-occurrences of the production.

Like a context-free grammar, an attribute grammar is a description rather than an algorithm. Just as a context-free grammar specifies phrase-structure independently of a parsing algorithm, so does an attribute grammar specify semantics or translation without presuming an evaluation order. Because semantic functions are pure functions, the definition of attribute-instance is determined by the attribute grammar and the APT, not by the algorithms of the semantic functions. Thus, an attribute grammar is a locally specified, non-procedural description of values rather than an alogorithm for computing values.

II. The Attribute Evaluation Paradigm

LINGUIST-86 supports the strategy of "evaluation in alternating passes" proposed by Jazayeri [J] [JW] [JP1]. For alternating pass evaluation there is a simple strategy, described by Schulz [S], that allocates most of the APT to secondary storage and that achieves both prefix and postfix visits to every APT node during each pass. LINGUIST-86 uses alternating pass evaluation is to take advantage of this. The evaluation strategy calls for storing a linearized version of the APT in an intermediate file. When an APT node, N, is encountered during the course of attribute evaluation it is read from the intermediate file onto a stack in memory. N is kept on the stack while the sub-tree descended from N is visited (and those nodes get put on the stack "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. When the evaluation pass over N's subtree is finished node N is written to the intermediate file. Because of the the evaluation order, the nodes of N's subtree will have already been written to the file.

LINGUIST-86's paradigm for APT traversal and attribute evaluation in a left-to-right pass is given in figure 3. Depicted is the process of "visiting" the sub-APT whose root is an instance of XØ.

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

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

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

eval synthesized attribs of XØ
return from visiting sub-APT rooted at XØ
```

Attribute Evaluation Paradigm
Figure 3

Although Schulz [S] describes an interpretive approach that uses a single intermediate file, LINGUIST-86 generates in-line code to read and write APT nodes and to evaluate semantic functions. Two intermediate files are used per pass; APT nodes are read from one intermediate file and written to the other intermediate file.

This model of attribute evaluation requires reading nodes in prefix order and writing them in postfix order. The output file of a left-to-right pass is a left-to-right, postfix ordering of the nodes of te APT. The input file for a right-to-left pass is a right-to-left, prefix ordering of the APT nodes. Thus, if the output file of a left-to-right pass is read backwards it can be the input file for a right-to-left pass. Of course, 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.

```
left-to-right --->       <--- right-to-left
prefix                              postfix
        M F B A C E D G L H K I J

                     M
                     |
           +---------+---------+
           |         |         |
           F         G         L
           |                   |
    +------+----+        +-----+------+
    |           |        |            |
    B           E        H            K
    |           |                     |
+---+---+       |                 +---+---+
|       |       |                 |       |
A       C       D                 I       J

         A C B D E F G H I J K L M
postfix                              prefix
left-to-right --->       <--- right-to-left
```

The attribute evaluation strategy does not tell how to build the first linearized APT file. There are two approaches that fit immediately in the evaluation paradigm. One is for the parser to emit tree nodes in bottom-up order. This creates an intermediate APT file that is identical to what would have been created by a left-to-right attribute evaluator. No attribute evaluation is done because there is no prefix encounter of these nodes; the first attribute evaluation pass is right-to-left.

The other approach is for the parser to emit nodes in prefix order, like a recursive descent parser. Accepting the next node from the parser takes the place of reading the next APT node. In this case, evaluation of semantics can be done during the same pass as parsing, and the first semantic pass is a left-to-right pass.

LINGUIST-86 supports both of these strategies. Part of its input is an indication of which strategy is to be used. The only difference in the attribute evaluators is whether the first attribute evaluation pass is right-to-left (the first approach) or left-to-right (the second approach). LINGUIST-86 itself uses the first method.

The code that reads/writes APT nodes and evaluates semantic functions is organized as a set of mutually recursive procedures called "production-procedures". There are distinct sets of production-procedures for each pass. Each production-procedure has one value/result parameter, an APT node, that corresponds to the left-hand-side of the production. Space for those APT nodes that correspond to the production's right-hand-side are allocated as local variables of a production-procedure. Thus, the stack of APT nodes is intermixed with the system run-time stack that supports procedure call/return, parameter passing, and recursion. The body of each production-procedure reads right-hand-side APT nodes from the input intermediate file, computes values and defines attribute-instances, calls production-procedures that correspond to the productions that apply at right-hand-side nodes, and writes right-hand-side APT nodes to the output intermediate file. This organization is similar to that of a recursive descent compiler. A production from the LINGUIST-86 attribute grammar is reproduced on the next page along with the corresponding production-procedure LINGUIST-86 generated for one pass.

IV. Optimizations of the Evaluators

LINGUIST-86 performs several useful optimizations of this paradigm. An obvious one is to reduce the amount of data transfered between the intermediate files and memory by not writing any instances of attributes that are defined during this pass but never referenced after this pass. Our experience with the attribute grammar for LINGUIST-86 and with an attribute grammar for Pascal indicates that the majority of attributes are referenced only during the same pass in which they are defined.

The effect of not writing "dead" attribute-instances to the intermediate file turns out to be very similar to an evaluation strategy suggested by Saarinen [Sa]. For any "tree-walk" evaluator [KW] he proposes that attributes be divided into "significant" attributes and "temporary" attributes. An attribute is "significant" if it is referenced in a later "visit" than the one in which it was defined; otherwise it is "temporary". "Significant" attributes are kept in the data structure for a node (roughly corresponding to LINGUIST-86's file-resident APT) and "temporary" values are kept on a stack (LINGUIST-86's stack-resident local variables of production-procedures). In [JP2] such an approach is described in detail for pass-structured attribute evaluators, especially alternating pass evaluators. The authors' experience is also that most attributes are "temporary" attributes.

A second optimization used in LINGUIST-86 loosens restrictions on the order of attribute evaluation within the production-procedures of a pass; it is tied in with the tests for evaluability. The paradigm calls for evaluating inherited attributes of right-hand-side nodes just before calling the production-procedures for those nodes, and for defining synthesized attributes of the left-hand-side node only after visiting all of the sub-APTs of the right-hand-side nodes. However, there is nothing to prevent us from evaluating a synthesized attribute-instance of the left-hand-side, X, before visiting some right-hand-side sub-APT so long as all the attribute-instances that X depends on have already been evaluated. Similarly, some inherited attribute-instances of right-hand-side nodes can be evaluated earlier than called for by the paradigm. Such evaluation criteria means that LINGUIST-86 will evaluate some attributes earlier than the "ordered ASE" of [JP1].

The really important optimization is static subsumption. Its effect is to eliminate copy-rules and to decrease both the stack space needed to evaluate an attribute grammar and the size of the intermediate APT files. Static subsumption can be applied to the entire class of "tree-walk" evaluators [KW], but we will discuss it only for alternating pass evaluators, a strict sub-class.

Static subsumption was correctly formulated only after we reexamined information passing in the evaluation paradigm. The paradigm calls for allocating on the stack the memory to store attribute-instances by making them local variables of a recursive procedure. Right-hand-side attribute-instances are directly addressable as local variables of the corresponding production-procedures. The left-hand-side attribute-instances are referenced through a pointer that is passed as an argument to the production-procedure. This pointer represents the only communication between the various production-procedures.

An alternative to passing a pointer is to copy the attribute-instances of a node into a global memory area just prior to calling the production-procedure and then, after returning from that call, copying these attribute-instances back to the local, stack-resident, variables. If the attribute S.A is always copied to a specific global variable, S_A, before entering a production-procedure for S then code in the production-procedure can always reference this global variable for the value of the left-hand-side attribute-instance. Similarly, if a production-procedure defines left-hand-side attribute S.B and assigns the value to a global variable, S_B, then any production-procedure for a production with S in the right-hand-side can reference this global variable, either to copy it into the local variable 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

```
function$list0 = function COMMA  function$list1 ->     FunctionListLimb.
  ERR = if IsIn(function.OBJ, function$list1.CYCLIC)
          then local$circularity$msg
          else no$msg
        endif,                                                        # pass 2
  function$list1.FUNCTS   = union$setof(function.OBJ, function$list0.FUNCTS), # pass 2
  function$list1.USED$AOS = union(function.DEF$AOS, function$list0.USED$AOS), # pass 2
  function$list0.MSGS     =  cons$msg (commaNT.LINE,ERR,null$name,
                            merge$msgs(function.MSGS, function$list1.MSGS )), # pass 3
  function$list0.TEXT$LIST =
    if not function.EVAL then  function$list1.TEXT$LIST
                        else
      cons3(function.SYM$NAME,function.OBJ,function.TEXT,function$list1.TEXT$LIST)
    endif                                                             # pass 4


   { Production-procedure for pass2.   This is a left-to-right pass }

  procedure FUNCTIONLISTLIMBPP2 (VAR FUNCTIONLIST0 : FUNCTIONLIST_PQZ_type);
    VAR
       FUNCTIONLISTLIMB : FUNCTIONLISTLIMB_PQZ_type;
       FUNCTION         : FUNCTION_PQZ_type;
       COMMA            : COMMA_PQZ_type;
       FUNCTIONLIST1    : FUNCTIONLIST_PQZ_type;

    begin
      GetNodeFunctionListLimb(FUNCTIONLISTLIMB);

      GetNodeFunction(FUNCTION);
        {  eval inherited attribs here  }
        FUNCTION.LHSID   := FUNCTIONLIST0.LHSID;
        FUNCTION.LIMBAOS := FUNCTIONLIST0.LIMBAOS;
        FUNCTION.AOS     := FUNCTIONLIST0.AOS;
        FUNCTION.USEDAOS := FUNCTIONLIST0.USEDAOS;
        FUNCTION.SYNSYM  := FUNCTIONLIST0.SYNSYM;
      FunctionPP2(FUNCTION);
      PutNodeFunction(FUNCTION);

      GetNodeComma(COMMA);
      PutNodeComma(COMMA);

      GetNodeFunctionList(FUNCTIONLIST1);
        {  eval inherited attribs here  }
        FUNCTIONLIST1.LHSID   := FUNCTIONLIST0.LHSID;
        FUNCTIONLIST1.LIMBAOS := FUNCTIONLIST0.LIMBAOS;
        FUNCTIONLIST1.AOS     := FUNCTIONLIST0.AOS;
        FUNCTIONLIST1.SYNSYM  := FUNCTIONLIST0.SYNSYM;
        FUNCTIONLIST1.FUNCTS  := UNIONSETOF(FUNCTION.OBJ, FUNCTIONLIST0.FUNCTS);
        FUNCTIONLIST1.USEDAOS := UNION(FUNCTION.DEFAOS, FUNCTIONLIST0.USEDAOS);
      FunctionListPP2(FUNCTIONLIST1);
      PutNodeFunctionList(FUNCTIONLIST1);

        { eval synthesized attribs here }
      FUNCTIONLIST0.SEMFUNCTS := FUNCTIONLIST1.SEMFUNCTS;
      FUNCTIONLIST0.CYCLIC    := FUNCTIONLIST1.CYCLIC;
      if ISIN(FUNCTION.OBJ, FUNCTIONLIST1.CYCLIC)
         then FUNCTIONLISTLIMB.ERR := LOCALCIRCULARITYMSG
         else FUNCTIONLISTLIMB.ERR := NOMSG;

      PutNodeFunctionListLimb(FUNCTIONLISTLIMB);
    end;
  end FUNCTIONLISTLIMBPP2;
```

A production of the LINGUIST-86 attribute grammar and its automatically generated production-procedure attribute-instance is a copy-rule whose right-hand-side 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". Shown below is a simple example of how copy-rules can be subsumed; the subsumed copy-rules are commented out.

```
 S  ::= X S1 -> ListProd.
   S1.A = S.A,
   X.A  = S.A,
   S.DEFS  = S1.DEFS,
   S1.PRE = UnionSetof(S.PRE,X.OBJ)
   S.POST =
     IncrIfTrue(IsIn(X.A,S1.PRE),S1.POST)

VAR
  X_A, S_A : A_attribType;
  S_DEFS   : DEFS_attribType;

procedure ListProdPPi (VAR S : S_type);
    { this is a right-to-left pass }
  VAR  X  : X_type;  S1 : S_type;
  begin

    GetNodeS(S);
    { S1.A := S.A }
      S1.PRE := UnionSetof(X.OBJ,S.PRE);
    PPi( S );
    PutNodeS(S);

    GetNodeX(X);
      X_A := S_A;
    PPi( X );
    PutNodeX(X);

   { S.DEFS := S1.DEFS; }
     S1.POST =
       IncrIfTrue(IsIn(X.A,S1.PRE),S1.POST)

  End ListProdPPi;
```

Static subsumption also reduces the amount of stack space needed to store attribute-instances. If a collection of attribute-instances is being used only to transmit information around the APT via copy-rules then explicit fields in the record that represents the APT node need not be allocated. This can result in significant decrease both in the stack space needed for APT nodes and in the size of the APT file.

The penalty for eliminating this explicit copying is paid at those points where the affected attributes are not defined by subsumable copy-rules. In these cases a new value will be assigned to the global variable for propagation to the sub-APT. However, the previous value of the global variable is not "dead"; it may still be referenced in some other part of the tree. Hence the old value must be "saved" in a temporary variable in the production-procedure's stack-frame. After processing the sub-APT (i.e. upon return from the call to the production-procedure) the saved value must be "restored" to the global variable. A further complication is that the newly-computed right-hand-side value may be used elsewhere in this production-procedure concurrently with references to the "old" value of the global variable, and after the old value has been restored to the global variable.

Below is the production-procedure of the earlier example modified as would be required if attributes S.PRE and S.POST were statically allocated to global variables PRE and POST.

```
procedure ListProdPPi (VAR S : S_type);
    { this is a right-to-left pass }
  VAR
    X  : X_type;
    S1 : S_type;
    PRE_QZP   : PRE_type;
    POST_QZP  : POST_type;
    PRE2_ZQP  : PRE_type;
    POST2_ZQP : POST_type;

  begin

    GetNodeS(S);
    { S1.A := S.A }
      PRE2_ZQP := UnionSetof(X.OBJ,S.PRE);
      PRE_QZP := PRE; PRE := PRE2_ZQP;
    PPi( S );
      PRE := PRE_QZP;
      POST2_ZQP := POST;
    PutNodeS(S);

    GetNodeX(X);
      X_A := S_A;
    PPi( X );
    PutNodeX(X);

   { S.DEFS := S1.DEFS; }
     POST =
       IncrIf(IsIn(X.A,PRE2_ZQP),POST2_ZQP)

  End ListProdPPi;
```

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, S.A and X.A could be allocated to the same variable, thereby enabling us to eliminate the copy-rule "X.A := S.A". On the other hand, global variables may have to be saved/restored more frequently.

In general, the extra code neccessary to save/restore a global variable is as much as the code saved by "subsuming" several copy-rules. For static subsumption to be effective only those attributes that participate in several subsumable copy-rules should be statically allocated.

LINGUIST-86 statically allocates attributes to decrease the code size of the evaluator. There are other criteria that might be important, e.g. time efficiency of the evaluator due to not executing the subsumed copy-rules, and the reduced stack height caused by not having to allocate fields in the APT nodes for the statically allocated attributes. But these considerations vary from one APT to another, average values for them are hard to get, and we have not had any serious problems with either stack-height or the time-efficiency of semantic functions.

LINGUIST-86 allocates all static attributes with the same name to the same global variable. The algorithm to determine which attributes will be statically allocated starts by assuming that all attributes 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 what percentage of the semantic functions that define this attribute are subsumable copy-rules. The percentage is based on the relative costs of save/restoring a static variable and copying the attribute-instances. If so indicated the attribute is removed from the statically allocated set. When an attribute changes from statically allocated to being allocated in the APT node it can make it more expensive for other attributes to be statically allocated. Hence, all remaining static attributes must be reexamined until the process stabilizes. This is an n-cubed algorithm and it does not always find an optimal set of attributes to statically allocate. However, the algorithm is correct and easy to implement and it does eliminate many copy-rules.

Our experience shows that it is very effective to allocate to the same global variable all inherited attributes that have the same name. An enormous amount of context information is copied down the tree via inherited attributes. Static subsumption can eliminate such copy-rules with very little cost because this context information is not often updated.

We have used static subsumption to generate evaluators for both the LINGUIST-86 attribute grammar and our attribute grammar for Pascal. Static subsumption eliminated nearly 20% of the semantic function evaluation code in LINGUIST-86. It eliminated about 13% of the code that evaluates semantic functions in the Pascal attribute evaluator. At first blush this is disappointing in an optimization that can eliminate up to half of the semantic functions. However, each copy-rule generates very little code, whereas semantic functions that aren't copy-rules can be quite large. Furthermore, if an optimizing compiler eliminated 10% of the generated code with a single optimization it would be an enormously successful optimization. We also timed versions of LINGUIST-86 that were generated with and without having static subsumption applied. Because the evaluators are I/O bound there was no noticable difference in the two times.

Static subsumption has some elements in common with a method investigated by Ganzinger [G], who suggests trying to allocate attribute-instances to statically defined variables. His main purpose is to conserve storage by allocating many attribute-instances to the same memory location. However, he tries to do such allocation independent of an attribute evaluation strategy and his results are pessimistic. On the other hand, LINGUIST-86's static subsumption optimization is tailored to a particular evaluation strategy, and static subsumption permits saving and restoring the values in global variables.

The effect of static subsumption is similar to some optimizations done by the translator-writing-system GAG [KZ]. GAG uses attribute-stacks and static variables to hold some attribute-instances during attribute evaluation. The important difference is that GAG's optimizations are applied based on user-supplied input in the attribute grammar: TRANSFER, INCLUDING, and CONSTITUENT specifications. LINGUIST-86's static subsumption is based solely on the attribute grammar, not on instructions provided outside the attribute grammar formalism.

IV. The Input To LINGUIST-86

The input to LINGUIST-86 is an attribute grammar. This includes: a list of grammar symbols, a list of attributes for each symbol, a list of productions, and a list of semantic functions associated with each production.

The input to LINGUIST-86 is also the input to our LALR parse-table builder. This latter program ignores all the attributes and semantic functions and just picks out the productions of the underlying context-free grammar. To assure that the phrase-structure expected by the LINGUIST-86-generated attribute evaluator is the same as that generated by the parser we submit exactly the same input file to both LINGUIST-86 and the parse-table builder.

LINGUIST-86 does not contain a programming language embedded within it. LINGUIST-86 does not support the definition of constants, types, or variables; the types of attributes are uninterpreted identifiers; any identifier that is not a grammar symbol, attribute, or attribute type is treated as an uninterpreted constant or function. All type-checking, storage allocation, and interpretation of types, constants, and functions is done by the compiler for the target programming language.

The form of the attribute grammars accepted by LINGUIST-86 goes beyond the bare-bones specification in several areas: intrinsic attributes, limb symbols, implicit copy-rules, and the form of semantic functions. These extensions reflect the pragmatic nature of LINGUIST-86 as a tool that generates evaluators. Each of them is discussed below.

Intrinsic attributes are attributes that are already defined before attribute evaluation starts, much as the APT is defined before evaluation starts. In our system, any intrinsic attributes are set by the parser, just as the APT is built by the parser. The attribute grammars we have written use intrinsic attributes to denote the name-table-index of terminal symbols and the location in the source of the text that corresponds to a leaf of the APT. As far as LINGUIST-86 is concerned an intrinsic attribute is like any other attribute except that it is evaluated before any pass. No semantic function can define an intrinsic attribute. Our use of intrinsic attributes, including the term itself, was proposed by Schulz [S].

LINGUIST-86 expects every production that has non-trivial semantics to have a limb symbol. The limb must be declared as a grammar symbol; it forms the third type of grammar symbol, after non-terminals and terminals. The limb is used to construct the name of its production-procedure, and to synchonize the identification of productions with the parser. A limb symbol can have attributes just as any other symbol can. However, these limb-attributes do not provide communication between levels of the APT; rather, they serve as names for common sub-expressions used in the semantic functions. Being able to name common sub-expressions is useful. It makes writing the semantic functions easier (i.e. shorter). It makes the code that gets generated shorter. It can reduce the number of attributes that must be written to and read from intermediate files. Like any other attributes, limb attributes must be declared in the symbol section of the input.

Normally, the semantic functions of a production must define all synthesized attributes of the left-hand-side symbol of the production and all inherited attributes of any right-hand-symbol of the production; if not this is an error. In many cases where such definitions are ommitted LINGUIST-86 will supply "implicit" copy-rules.

Our formula for inserting these implicit copy-rules has two flavors: one for synthesized attributes of the left-hand-side and one for inherited attributes of the right-hand-side. If R.A is an inherited attribute of right-hand-side symbol R which is not defined by any semantic function of this production, and if there is an attribute L.A of the left-hand-side symbol L with the same attribute name, A, then an implicit copy-rule of the form R.A = L.A will be inserted as a semantic function of this production. If L.B is a synthesized attribute of the left-hand-side symbol L which is not defined by any semantic function of the production and if there is exactly one symbol, R, in the right-hand-side of the production such that R has a synthesized attribute named B, and if there is only one occurrence of R in the right-hand-side of the production, then an implicit copy-rule of the form L.B = R.B will be inserted. When the listing file for the input attribute grammar is generated each implicit copy-rule is listed immediately after all of the explicit semantic functions of the production.

These implicit copy-rules are analogous in effect to the TRANSFER construct of the GAG system [KZ]. The major difference is that the TRANSFER construct is explicit, whereas our insertion of copy-rules is implicit. LINGUIST-86 does not have any mechanism analogous to GAG's INCLUDING or CONSTITUENT constructs; we feel they detract from an attribute grammar's locality of reference.

The attribute grammar for LINGUIST-86 has 584 semantic functions. 3Ø2 of these are copy-rules and 276 of the copy-rules are implicit copy-rules.

The final "extension" supported by LINGUIST-86 concerns the form of semantic functions. The right-hand-side of semantic functions can contain expressions involving some standard infix operators (+, -, AND, OR, =, <>, >, <), constants (e.g. Ø, 14, true), as well as a value-producing control flow construct, e.g.

```
  S.A = if S.B <> Ø then T.A + 3
                    else WidthOf(T.C)
        endif.
```

Control flow constructs can be nested within one another but they can not occur within the operands of infix operators, or arguments to external functions. i

Furthermore, the left-hand-side of a semantic function can be a list of attribute-occurrences, allowing a single semantic function to define more than one attribute-occurrence. If the right-hand-side of such a semantic function is an

```
attrib$list.PASS$MSGS, attrib$list.MSGS = null$msg$list,


attrib$list0.STATICS$LIST, attrib$list0.PUBLICS, attrib$list0.PRIVATE =
  if    AttribIsStatic(attrib.OBJ)      then
    if EvalPF(attrib$list0.STATICS,attrib.NAME) <> bottom then
                                             attrib$list1.STATICS$LIST
                                                             else
      consPF(attrib.NAME,attrib.TYPE$NAME,attrib$list1.STATICS$LIST)
    endif,
    attrib$list1.PUBLICS,
    attrib$list1.PRIVATE
                                           else
    attrib$list1.STATICS$LIST,
    if    EarliestPass < LatestPass  then
      # it can not happen that EarliestPass > LatestPass
      cons2(attrib.NAME,attrib.TYPE$NAME, attrib$list1.PUBLICS),
      attrib$list1.PRIVATE
    elsif EarliestPass = ThisPass    then
      # and hence LatestPass = ThisPass as well
      attrib$list1.PUBLICS,
      cons2(attrib.NAME,attrib.TYPE$NAME, attrib$list1.PRIVATE)
    else
      attrib$list1.PUBLICS,
      attrib$list1.PRIVATE
    endif
  endif,
```

Two examples of semantic functions that define more than one attribute-occurrence
Figure 5

expression not containing a control-flow construct then the expression is interpreted as the common value of all attribute-occurrences on the left-hand-side; if the right-hand-side expression is an "if" expression the "then" and "else" clauses must also contain a list of expressions. In the latter case the left-hand-side list of attribute-occurrences must have the same length as the right-hand-side lists of expressions and expression values are assigned to attribute-occurrences pair by pair. Figure 5 gives two examples of semantic functions that define more than one attribute-occurrence.

LINGUIST-86's attribute grammar is 1800 lines long. It contains 159 symbols, 318 attributes, 72 productions, 1202 attribute-occurrences, and 584 semantic functions. 302 of the semantic functions are copy-rules, a little more than 50%. The percentage of copy-rules is in line with what other researchers have reported [PJ2]. The attribute grammar is evaluable in 4 alternating passes.

V. Operating Characteristics

Although we have refered to LINGUIST-86 as a translator-writing-system it is actually the most important piece of such a system. Other pieces of the system include an LALR parse-table builder and the parser that interprets those tables, a program that generates a lexical scanner for a set of regular expressions, a package that implements a name-table for identifiers, and a package that supports list-processing.

LINGUIST-86 was itself designed with these tools and generated by them. Its internal organization is indicative of how other generated programs would be organized. LINGUIST-86 is an overlayed, pass-structured program consisting of seven overlays and six passes. Each overlay except the fourth makes a pass over the input or some intermediate representation of the input.

The first overlay scans and parses the input, builds the table of all identifiers encountered, writes a right-parse of the input to an intermediate file, and writes a list of all syntactic errors to another intermediate file. The next two overlays do semantic analysis and in the process builds the dictionary of symbols, attributes, and semantic functions. The fourth overlay analyzes the attribute dependencies that are in the dictionary to determine the alternating pass evaluability of the attribute grammar. The fifth overlay collects a sequence of semantic messages that will be used to generate the listing. The sixth overlay creates the listing output file. The seventh overlay generates one pass of the output attribute evaluator. This overlay is rerun once for each pass of the output evaluator; if the input attribute grammar is evaluable in 4 alternating passes then then the seventh overlay is run 4 times.

Overlay 1 contains the automatically generated scanner tables and parser tables and their interpreters, the scanner and parser. Overlays 2, 3, 5, and 7 are each separate passes of the automatically generated attribute evaluator. The intermediate files that these overlays manipulate contain the APT. Overlays 4 and 6 are independent, hand-written procedures.

There are about 55K bytes of code in LINGUIST-86, of which more than 23K bytes are automatically generated attribute evaluators. LINGUIST-86 took 6 man-months to develop, starting from the initial design and going through the point that an automatically generated version of LINGUIST-86 generated itself. Maintainence to and enhancement of LINGUIST-86 has continued since that time, including the addition of the static subsumption optimization.

The size of the 4 automatically generated attribute evaluation passes is:

pass 1	4292 bytes
pass 2	6538 bytes
pass 3	5414 bytes
pass 4	7215 bytes
husk	4065 bytes.

The husk of an attribute evaluator module is everything except the semantic functions; included in the husk are the production-procedure declarations, calls to GetNode and PutNode, and recursive calls to production-procedures. For a given grammar the size of the husk is the same for every pass. In some sense the husk represents the "overhead" in the attribute evaluator. The amount of code generated by the semantic functions can be found by subtracting the code size of the husk from the code size of the attribute evaluator modules. The above figures indicate that the "overhead" in the attribute evaluators is significant. They also show that passes 1 and 3 do less work than passes 2 and 4.

LINGUIST-86 runs on an 8086-based microcomputer system. The system has between 96K and 128K bytes of memory available, depending on configuaration; it does not support virtual memory. Input files, output files, and intermediate files are on either floppy disk or rigid disk.

LINGUIST-86 processes input attribute grammars at between 350 and 500 lines per minute, excluding the time taken to generate the production-procedures. We exclude this time for comparison purposes because it will depend directly on the number of passes needed to evaluate the attribute grammar, not on the number of input lines This is reasonably competitive with the host systems translator products. These compilers process high-level language programs at speeds that range between 400 and 900 lines per minute (depending on the language). The time used by each overlay when processing LINGUIST-86's attribute grammar is shown in the table below.

parser overlay -	80 sec.
first attrib eval overlay -	25 sec.
second attrib eval overlay -	42 sec.
evaluability test overlay -	9 sec.
third attrib eval overlay -	24 sec.
listing generation overlay -	63 sec.
TOTAL	243 sec.

It is quite apparent from these figures that LINGUIST-86 is I/O bound. The two longest overlays are the first, which reads the input, and the last, which generates the listing file. Both are I/O intensive. The first attribute evaluation pass and the third attribute evaluation pass each take about 25 seconds. As is indicated by the code size figures, these two overlays spend a large percentage of their effort just reading and writing the APT file. The second attribute evaluation overlay has significant code other than the production-procedure husks. The timing figures indicate that this pass spends about 20 seconds doing something other than reading and writing APT files. Apparently semantic function evaluation is a minor component of the effort expended by the attribute evaluators.

We have also timed LINGUIST-86 processing our attribute grammar for Pascal. LINGUIST-86 processes this input at a little more than 400 lines per minute.

Conclusions

LINGUIST-86 is an interesting contribution to the study of translator-writing-systems based on attribute grammars because it produces efficient evaluators from an input that is an attribute grammar. Such features as intrinsic attributes, implicit copy-rules, and sophisticated, list-valued expressions as the right-hand-side of semantic functions facilitate writing attribute grammars and generating evaluators for them but they do not impair an attribute grammar's locality of reference or non-procedural nature. The approach embodied by LINGUIST-86 has been shown effective; LINGUIST-86 is itself a non-trivial attribute grammar and is self-generating.

This research suggests at least two general questions that we would like to pursue further. Since attribute evaluation is I/O bound, can the evaluation paradigm and its implementation be modified or streamlined to be faster? Especially, would some form of virtual memory system significantly speed up the evaluators? The second question is whether a more complete and global analysis of the attribute grammar can yeild markedly better static subsumption results. Our initial hand simulations of static subsumption were more effective than the automatically generated versions, but the hand simulations made use of global information.

Acknowledgements

Many of the ideas about attribute evaluation that are contained in LINGUIST-86 were suggested by Dean Schulz in his thesis [S]. Sue Ojeda and Tuyen Quoc each wrote significant pieces of LINGUIST-86. Important and timely support for LINGUIST-86 was provided over the course of several years by Al Hartmann and Tom Wilcox. Rick Schell read an early draft of this paper and suggested many improvements. I am indebted to all of these people. Without their help LINGUIST-86 could not have been written.

References

[B] Bochman, G.V. Semantic evaluation form left to right. Communications of the ACM, Vol 19, Num 2, February 1976.

[F] Fang,I. FOLDS, a declarative formal language definition system. STAN-CS-72-329, Computer Science Department, Stanford University, Stanford, California, December 1972.

[G] Ganzinger,H. On storage optimization for automatically generated compilers. Theoretical Computer Science, 4th GI Conference, K.Weihrauch (ed.), Springer-Verlag, Berlin-Heidelberg-NewYork, March 1979, 132-141.

[GRW] Ganzinger, H., K. Ripken, and R. Wilhelm. Automatic Generation of Optimizing Multi-Pass Compilers. Proc IFIP 1977, Toronto, Ontario, 1977.

[JOR] Jazayeri, M., W. F. Ogden and W. C. Rounds. The intrinsically exponential complexity of the circularity problem for attribute grammars. CACM vol. 8, no. 12, December 1975.

[JW] Jazayeri, M. and K.G. Walter. Alternating semantic evaluator. In ACM '75, Annual Conference. Minneapolis, Minn., Oct. 1975.

[Ka] Kastens, U. Ordered attribute grammars. Acta Informatica 13, Springer-Verlag, 1980, pp.229-256.

[KZ] Kastens, U. and E. Zimmerman. GAG - A Generator Based on Attribute Grammars. Institut fur Informatik II, Universitat Karlsruhe, Bericht NR. 14/80.

[KW] Kennedy,K. and S.K.Warren. Automatic generation of efficient evaluators for attribute grammars. Conference Record of the Third ACM Symposium on Principles of Programming Languages, Jan. 1976.

[K] Knuth,D.E. Semantics of context-free languages. Math. Systems Theory 2 (1968), 127-145. Knuth,D.E. Semantics of context-free languages: correction. Math. Systems Theory 5, No. 1. 95-96.

[L] Lorho, B. Semantic attribute processing in the system DELTA. In Lecture Notes in Computer Science: Methods of Algorithmic Language Implementation, A. Ershov and C.H.A. Koster (Eds.). Spinger-Verlag, New York, 1977.

[PJ1] Pozefsky,D. and M.Jazayeri. A family of pass-oriented attribute grammar evaluators. Proceedings of the ACM 1978 Annual Conference, December 1978.

[PJ2] Pozefsky,D. and M.Jazayeri. Space efficient storage management in an attribute evaluator. ACM TOPLAS, vol. 3, num. 4, Oct. 1981.

[RST] Raiha, K-J, M. Saarinen, E. Soisalon-Soininen, and M. Tienari. The Compiler Writing System HLP (Helsinki Language Processor). Report A-1978-2, Department of Computer Science, University of Helsinki, Helsinki, Finland. March 1978.

[Ra] Raiha,K-J. Dynamic allocation of space for attribute instances in multi-pass evaluators of attribute grammars. Proceedings of the SIGPLAN Symposium on Compiler Construction, Denver, Colo., 1979.

[Sa] Saarinen, M. On constructing efficient evaluators for attribute grammars. In Automata, Language and Programming: 5th Colloq., C. Ausiello and C. Bohm (Eds.). Springer-Verlag, New York, 1978.

[S] Schulz, W. A. Semantic Analysis and Target Language synthesis in a Translator. Ph.D. Thesis, University of Colorado, Boulder, Colorado, 1976.