The Automatic Inversion

of

Attribute Grammars

by

Daniel Yellin[1]   and   Eva-Maria M. Mueckstein[2]

CUCS-135-84

[1]Computer Science Department
Columbia University
New York, New York, 10027

[2]IBM T. J. Watson Research Center
Yorktown Heights, New York, 10598

Table of Contents

List of Figures

ABSTRACT

Over the last decade there has developed an acute awareness of the need to introduce abstraction and mathematical rigor into the programming process. This increased formality allows for the automatic manipulation of software, increasing productivity and, even more importantly, the manageability of complex systems. Along these lines, attribute grammars constitute a formal mechanism for specifying translations between languages; from a formal description of the translation a translator can be automatically constructed. In this paper we consider taking this process one step further: given an attribute grammar specifying the translation from language L_1 to the language L_2, we address the question of whether the inverse attribute grammar specifying the inverse translation from L_2 to L_1 can be automatically generated. We show how to solve this problem for a restricted subset of attribute grammars. This inversion process allows for compatible two-way translators to be generated from a single description. To show the practical feasibility of attribute grammar inversion, we relate our experience in inverting an attribute grammar used as an interface for a formal database accessing language, SQL. The attribute grammar is used to paraphrase SQL database queries in English.

1. Introduction

This paper discusses a method to invert attribute grammars. Given an attribute grammar (AG) defining a translation $T: L_1 \to L_2$, we show how to automatically synthesize the inverse attribute grammar specifying the inverse translation $T^{-1}: L_2 \to L_1$. To do so we impose restrictions on the the attribute grammars we consider.

Our research has been motivated by both theoretical interests and practical applications. Theoretically, this paper adds to a theory of inversion. It demonstrates, for a particular framework based on attribute grammars, how inversion of subprocesses (context-free productions and semantic functions) leads to the inversion of the entire process (the AG). It also shows that a strong duality between syntax and semantics exists in attribute grammars and that this duality can be exploited for purposes of inversion. Along practical lines, attribute grammar inversion promises to be a powerful tool for software development. Because it can be accomplished automatically, it increases production efficiency and insures the consistency of complex software.

Efficiency can be enhanced in systems where two-way translations are needed. In particular, if there is a need for an attribute grammar $T: L_1 \to L_2$ and its inverse $T^{-1}: L_2 \to L_1$, then by writing the attribute grammar T and automatically generating the inverse attribute grammar T^{-1} only half of the labor need be performed. More importantly, T^{-1} is guaranteed to be the actual inverse of T; $T^{-1}(T(s)) = s$ for all s in the domain of T. If T^{-1} were to be written manually and independently of T, it would be difficult to prove that this property is preserved. Furthermore, if at some later date T is changed or updated, T^{-1} can be automatically generated from the updated attribute grammar T. Hence *consistency* between the two translators can be maintained.

Attribute grammar inversion can also be used to translate between high level programming languages. For example, suppose that L_A and L_B are programming languages and $T_A: L_A \to I$ and $T_B: L_B \to I$ are attribute grammars describing the translations from A and B into an intermediate language I. If we can generate the inverse attribute grammar T^{-1}_B then we can create the translation $T_{AB}: L_A \to L_B$ by forming the composition $T_{AB} = T^{-1}_B \circ T_A$. (A method of composing AGs without using an intermediate representation is discussed by Ganzinger in [6]). These ideas can be extended to a distributed system with k processors linked together, each using its own command language. If programs need to be shared between processors, we can define a canonical form and write invertible translators from this canonical form into each command language. By automatically generating the inverse translators we would be able to translate a program written for one processor into the command language of some other processor. Furthermore, using this method one can create n^2 translators (translating from any one of n languages into any other one) from only n specifications, instead of n^2. This is illustrated schematically in figure 1-1. Other applications of inverting translation specifications are discussed in [23].

The organization of this paper is as follows: Section 2 contains a brief introduction to attribute grammars and presents an example grammar which will be used throughout the
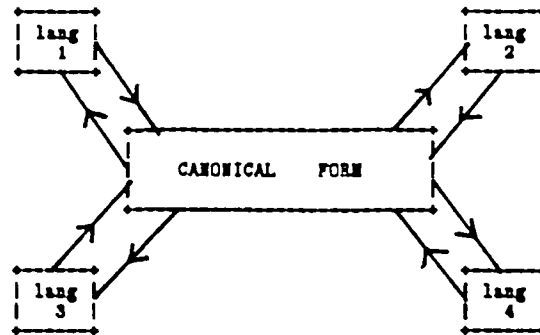
Figure 1-1: Inverse attribute grammars used for two-way translations

paper. In section 3 we introduce a restricted form for attribute grammars and discuss the inversion algorithm. In section 4 we relate our experience in inverting an actual attribute grammar. Section 5 summarizes our results and suggests areas for future research.

2. A Brief Description Of Attribute Grammars

In this section we provide a brief introduction to attribute grammars, present an example attribute grammar used in the rest of the paper, and define a small extension to attribute grammars, namely, *context conditions*.

2.1. Attribute Grammars

Attribute grammars were first proposed by Knuth [15] as a way to specify the semantics of context-free languages. The basis of an attribute grammar is a context-free grammar. This describes the context-free language that is the domain of the translation, that is, those strings on which the translation is defined. This context-free grammar is augmented with *attributes* and *semantic functions*. Attributes are associated with the nonterminal symbols of the grammar. We write "X.A" to denote attribute A of symbol X, and $A(X)$ to denote the set of attributes associated with X. Semantic functions are associated with productions; they describe how the values of some attributes of the production are defined in terms of the values of other 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, N, has two productions associated with it. The left-part production (LP) of N is the production that applies at N deriving N's children. The right-part production (RP) of the node N is the production that applies at the parent of N deriving N and its siblings. Leaves of the tree don't have LP productions; the root doesn't have an RP production.

A *semantic tree* 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*. The

values of attribute-instances are specified by the semantic functions. For example, if a production [p: $X_0 ::= X_1 \cdots X_{np}$] has a semantic function $X_0.A = f(X_2.B, X_4.C)$, then for any instance of p in any semantic tree, the attribute-instance corresponding to $X_0.A$ will be defined by applying the function f to the attribute-instances corresponding to $X_2.B$ and $X_4.C$.

Since two different productions are associated with each attribute-instance, there could be two semantic functions that independently specify its value, one from the LP production and one from the RP production. If we assume that each attribute-instance is defined by only one semantic function, either from the LP production or from the RP production, then we must guard against an attribute-instance not being defined at all because the LP production assumed that the RP production would define it and vice versa. These difficulties are avoided in attribute grammars by adopting the convention that for every attribute, X.A, either: (1) every instance of X.A is defined by a semantic function associated with its LP production, or (2) every instance of X.A is defined by a semantic function associated with its RP production. Attributes whose instances are all defined in their LP production are called *synthesized* attributes; attributes whose instances are all defined in their RP production are called *inherited* attributes. Every attribute is either inherited or synthesized. Inherited attributes propagate information down the tree, towards the leaves. Synthesized attributes propagate information up the tree, toward the root. The inherited attributes of a non-terminal X are denoted by $I(X)$, the synthesized attributes by $S(X)$; $A(X) = I(X) \cup S(X)$. The start symbol has no inherited attributes. From the point of view of an individual production the above conditions require that the semantic functions of a production MUST define EXACTLY all the inherited attributes of the right-part symbols and all synthesized attributes of the left-part symbol. For a given a production [p: $X_0 ::= X_1 \cdots X_{np}$], we often refer to the *attributes of p*, $A(p) = A(X_0) \cup ... \cup A(X_{np})$.

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 semantic tree. In order to compute these values the other attribute-instances must be computed. In extreme cases an attribute-instance can depend on itself; such a situation is called a circularity and by definition such situations are forbidden from occuring in well-defined attribute grammars. In general, it is an exponentially hard problem [9] to determine that an attribute grammar is *non-circular*; i.e. that no semantic tree 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 [3, 10, 12, 14]; e.g., absolute noncircularity [14].

Many translator writing systems have been built using the attribute grammar formalism [16, 19, 13, 4, 7]. Such a system accepts an attribute grammar as input and generates a compiler for the attribute grammar. Part of this task calls for generating an evaluator of semantic trees; such an evaluator must evaluate each attribute-instance of the tree after all attribute-instances that it depends on have already been evaluated. Many strategies for efficient evaluation have been discussed in the literature [22] and include multi-pass [10] and

3

ordered [12] evaluation strategies.

2.2. An Attribute Grammar Example

Figure 2-1 gives an attribute grammar which translates simple English descriptions of mathematical expressions into post-fix Polish notation. This grammar distinguishes between expressions involving only integer values (in which case operators of the form $+_i$ and $*_i$ are required) and those involving a decimal point value (in which case operators of the form $+_r$ and $*_r$ are required). So, for example, it will translate the English phrase 'multiply 5.7 by 8' into the post-fix Polish expression '(5.7,8,$*_r$)' and the phrase 'add 5 to 9' into '(5,9,$+_i$)'.

In this AG there are 8 productions and each production has associated semantic functions. In production p_1, <Num1> and <Num2> denote separate occurrences of the same symbol, <Num>; the numeric suffixes distinguish these different occurrences. S.trans is the distinguished attribute of the root; at the end of attribute evaluation the translation resides in this attribute.

Figure 2-2 shows a semantic tree corresponding to the input string 'multiply 80 by 5.8'. Each node in this tree is labelled with its associated grammar symbol and has attribute-instances corresponding to the attributes of that grammar symbol.

2.3. Attribute Grammars and Context Conditions

In this paper we shall consider a small extension to attribute grammars. This extension allows for the attachment of semantic conditions to productions as illustrated in productions p_1 and p_2 of figure 2-1. In general we allow a production p to have a *context condition* of the form:

<CONDITION: expr$_1$ AND expr$_2$ AND ... AND expr$_k$>

where each expr$_i$ is a boolean expression involving constants and attributes of p. A condition of the above form attached to a production is to be interpreted as saying that the production-instance is valid if and only if the condition evaluates to true. If the condition evaluates to false then the production-instance is not valid and the input violates context sensitivities of the attribute grammar. An attribute grammar system allowing conditions on the productions would first parse the input, build a semantic tree, and evaluate the attribute-instances of the tree as in a regular attribute grammar system. It would then evaluate all conditions associated with production-instances of the tree. If all evaluate to true it would return the translation given in the distinguished attribute of the root. If any evaluate to false, however, the translation is defined to be 'error' as the input violates context sensitivities of the attribute grammar.[1] So, for instance, the sentence 'multiply 80 to 5.8' of our example attribute grammar would be parsed and a semantic tree built for it. After evaluation of the attribute-instances in the tree it would be determined that a context-

---

[1] If the underlying context-free grammar of the AG is ambiguous, then the translation of a string is 'error' only if every parse for this string contains violated context conditions.

Context free symbols of the attribute grammar and their attributes:

Context-free symbols	synthesized attributes	inherited attributes
S:	{ trans }	$\emptyset$
Op:	{ trans }	{ type }
Num:	{ trans, type }	$\emptyset$
Integer:	{ trans }	$\emptyset$
Decimal_num:	{ trans }	$\emptyset$
digit:	{ trans }	$\emptyset$

Productions of the attribute grammar and their semantic functions:

p_1: S ::= Op Num1 by Num2.   <Condition: (Op.trans = '$*_r$')
                                      or (Op.trans = '$*_i$') >

    S.trans = Concatenate('(', Num1.trans, ',', Num2.trans, ',', Op.trans, ')');

    Op.type = If (Num1.type = real) or (Num2.type = real) then real else int;

p_2: S ::= Op Num1 to Num2.   <Condition: (Op.trans = '$+_r$')
                                      or (Op.trans = '$+_i$') >

    S.trans = Concatenate('(', Num1.trans, ',', Num2.trans, ',', Op.trans, ')');

    Op.type = If (Num1.type = real) or (Num2.type = real) then real else int;

p_3: Num ::= Integer.

    Num.trans = Integer.trans;

    Num.type = int;

p_4: Num ::= Decimal_num.

    Num.trans = Decimal_num.trans;

    Num.type = real;

p_5: Op ::= add.

    Op.trans = If (Op.type = real) then '$+_r$' else '$+_i$';

p_6: Op ::= multiply.

    Op.trans = If (Op.type = real) then '$*_r$' else '$*_i$';

p_7: Integer ::= digits.

    Integer.trans = digits.trans;

p_8: Decimal_num ::= digits1 '.' digits2.

    Decimal_num.trans = Concatenate(digits1.trans, '.', digits2.trans);

Figure 2-1: An attribute grammar example

sensitive condition of p_2 is violated; an instance of that production is only valid if Op.trans equals an additive operator (i.e., '$+_r$' or '$+_i$') and in this case Op.trans equals $*_r$. The idea
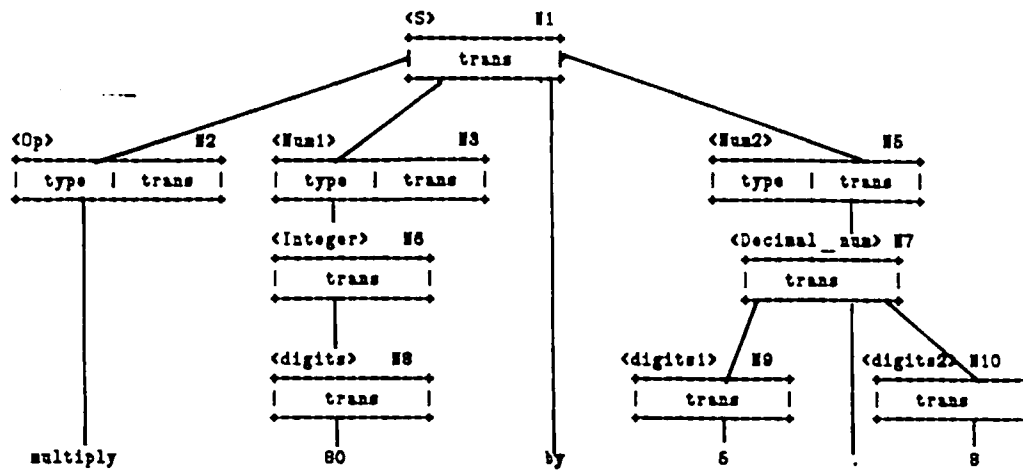
Figure 2-2: A typical semantic tree for the example AG

of context conditions for attribute grammars was first suggested in [20]. By putting further restrictions on the allowable form of conditions, we can make them useful in parsing the input ([11, 21]). In [5] it is shown how context conditions can be incorporated into the regular semantic functions of the productions.

3. Inversion Of Attribute Grammars

In this section we give an algorithm to invert AGs. For example, given the AG above describing the translation from English descriptions of mathematical expressions into post-fix Polish notation, the inversion algorithm will produce a new *inverse* AG describing the translation from post-fix Polish notation into English descriptions of mathematical expressions. In order to perform the inversion, the AG must be in a *restricted inverse form*. A formal definition of this restricted form is given in section 3.2. In essence, it restricts the AG so that each nonterminal of the grammar has a special *trans* attribute, which must be defined by a restricted functional form. For each interior node in a semantic tree, the trans attribute at that node will compute the translation of the subtree beneath it. Although other attributes of the AG influence the translation by passing context sensitive information around the semantic tree, it is the *trans* attribute which ultimately computes the translation. In the next section we introduce the concept of token permuting functions, which will subsequently be used in our definition of restricted inverse form.

3.1. Token Permuting Functions

A function f is a *token permuting function* over an alphabet Δ if and only if it is of the form: $f(Y_1,...,Y_n) = \text{concatenate}(\beta_0, Y_{i1}, \beta_1, Y_{i2},...,Y_{in}, \beta_n)$, where each Y_k ($1 \leq k \leq n$) is a variable taking on values in Δ^*, each β_k ($1 \leq k \leq n$) is a constant in Δ^*, and each Y_k of the left hand side appears once and only once as some Y_{it} ($1 \leq t \leq n$) of the right hand side.

The function f is called a *token permuting function* as it permutes the order of its

arguments and inserts constant tokens of Δ in between them. It is important to emphasize that a token permuting function cannot delete any of its arguments; each Y_k must appear as some Y_{it} and it cannot appear twice. For example, $f(Y1, Y2) = $ concatenate('Hello', Y1, 'and', Y2) is a token permuting function. If Y1 = 'Bob' and Y2 = 'Shirley' then this function would yield the string 'Hello Bob and Shirley'. However, the function $g(Y) = $ concatenate(Y, Y, 'where are you', Y,) is not a token permuting function as it duplicates the value of the string Y several times in the output string.

3.2. Restricted Inverse Form

An attribute grammar, without any restrictions on its semantic functions, is computationally equivalent to a Turing machine. As such, it is almost impossible to formally manipulate, let alone invert. In this section we introduce *restricted inverse form* attribute grammars, in which some semantic functions are required to be token permuting ones. By definition, an attribute grammar $T: \Sigma^* \rightarrow \Delta^*$ is in *restricted inverse form* if it obeys the following constraints:

1. Each nonterminal X has a distinguished synthesized attribute X.trans taking on values in Δ^*. X.trans represents the translation of the substring which X derives.

2. For each production $[p: X_0 ::= \alpha_0 \ X_1 \ \alpha_1 \ X_2 \ ... \ X_{np} \ \alpha_{np}]$ the semantic function defining X_0.trans is of the form

   $X_0.trans = $ if $g_1(atts_1)$ then $f_1(X_1.trans, X_2.trans, ..., X_{np}.trans)$

   elsif $g_2(atts_2)$ then $f_2(X_1.trans, X_2.trans, ..., X_{np}.trans)$

   .
   .
   .

   elsif $g_{s-1}(atts_{s-1})$ then $f_{s-1}(X_1.trans, X_2.trans, ..., X_{np}.trans)$

   else $f_s(X_1.trans, X_2.trans, ..., X_{np}.trans)$

   where each $atts_j \subseteq A(p)$ $(1 \leq j \leq s-1)$, each g_j $(1 \leq j \leq s-1)$ is a boolean function, and each f_j $(1 \leq j \leq s)$ is a *token permuting function* as described above. Note that the arguments to each f_j token permuting function are exactly the *trans* attributes of the production's right-part nonterminals.

3. The value of the translation is specified to be the value of the *trans* attribute of the root (S.trans).

In this definition there is no restrictions on the number of inherited or synthesized attributes a nonterminal can have nor is there placed any restrictions on how they are computed other than the trans attribute. Constraint 2, however, requires that each f_j $(1 \leq j \leq s)$ used to compute the trans attribute is a token permuting function.

Restricted inverse form attribute grammars (RIF grammars) can be viewed as restricted AGs or as a generalized version of syntax-directed translation schema [8]. Like syntax-directed translation schema, RIF grammars associate a special synthesized attribute (the trans attribute), to each nonterminal. This attribute stores the translation of its subtree

7

and is defined by a token permuting function. However, a RIF grammar surpasses a syntax-directed translation scheme in expressive power not only in that it associates context conditions to productions, but in that it allows other attributes to be associated with nonterminals. These "other" attributes influence the translation by determining which token permuting function is chosen to evaluate the trans attribute (they serve as arguments to the g_j boolean expressions). This allows RIF grammars to express context sensitive translations, something syntax-directed translation schema cannot do. For example, it is easy to construct a RIF grammar which accepts strings of the form '$a^i\ b^j\ c^k$' and translates them to 'OK $a^i\ b^j\ c^k$' if $i = j = k$, and to 'NOT OK $a^i\ b^j\ c^k$' otherwise. This language cannot be expressed by any syntax-directed translation schema, since the target language is not context-free. In general, the translations describable by syntax-directed translation schema are fairly restricted (see [1, 2]), whereas RIF grammars can, at least theoretically, describe any translation describable by an attribute grammar. The theoretical power of RIF grammars is discussed in [5].

3.3. The Inversion Algorithm

An attribute grammar in restricted inverse form displays a duality between syntax and semantics, as can be seen by considering a semantic tree of such an AG. On one hand, each node of the tree has an associated context-free *label*. On the other hand, each node can be considered *labeled* by its trans attribute. Inversion of the attribute grammar consists of switching these labels. To make sure that this is possible, we had to restrict the nature of the trans *label*; in restricted inverse form the trans attribute can only be defined by a token permuting function. The inversion process then consists of switching the labels and undoing the permutation specified by this function. This section formally defines the inversion algorithm.

Let T: $\Sigma^* \to \Delta^*$ be an attribute grammar in restricted inverse form. The inverse AG is created modularly from T, production by production. Each production in T will give rise to one or more productions of the inverse attribute grammar. As T translates strings of Σ^* into strings of Δ^*, the inverse AG will translate strings of Δ^* into strings of Σ^*. However, it will only translate those strings of Δ^* that are in the range of T.

Formally, let Δ^*_T be the range of T; i.e., $\Delta^*_T = \{\beta \in \Delta^*$ and there exists a semantic tree translating $\alpha \in \Sigma^*$ to $\beta\}$. Then the attribute grammar T^{-1}: $\Delta^*_T \to \Sigma^*$ is generated from T as follows:

1. For each token δ of Δ, create a terminal δ in T^{-1}.

2. For each nonterminal X in T, create a nonterminal XI in T^{-1} (we call it XI and not X to avoid confusion. We will not be very strict about this usage, however, when our meaning is clear. For example, when we refer to a semantic function f of T as also being a semantic function of T^{-1}, we mean the semantic function f' which is obtained from f by substituting every occurrence of X.A in f by XI.A).

3. Let each nonterminal XI in T^{-1} have the same set of attributes as X in T with

one *additional* attribute: $XI.transinv$. The attribute *transinv* will play same the role in T^{-1} as the attribute *trans* did in T; i.e., the *transinv* attribute will take on values in Σ^* and represents the translation of the substring that XI derives.

4. For each production $[p: X_0 ::= \alpha_0 \ X_1 \ \alpha_1 \ X_2 \ ... \ X_{np} \ \alpha_{np}]$ in T with the distinguished semantic function

$$X_0.trans = \text{if } g_1(atts_1) \text{ then } f_1(X_1.trans, X_2.trans, ..., X_{np}.trans)$$
$$\text{elsif } g_2(atts_2) \text{ then } f_2(X_1.trans, X_2.trans, ..., X_{np}.trans)$$
$$\bullet$$
$$\bullet$$
$$\text{elsif } g_{s-1}(atts_{s-1}) \text{ then } f_{s-1}(X_1.trans, X_2.trans, ..., X_{np}.trans)$$
$$\text{else } f_s(X_1.trans, X_2.trans, ..., X_{np}.trans)$$

create s productions in T^{-1}, one corresponding to each of the token permuting functions f_j. In particular, for each f_j, $1 \leq j \leq s$, where $f_j(X_1.trans, ..., X_{np}.trans) = \text{concatenate}(\beta_0, X_{i1}.trans, \beta_1, X_{i2}.trans, ..., X_{inp}.trans, \beta_{np})$ create an inverse production $[pI_j: XI_0 ::= \beta_0 \ XI_{i1} \ \beta_1 \ XI_{i2} \ ... \ XI_{inp} \ \beta_{np}]$ with an attached context condition $<\text{COND: (NOT } g_1(atts_1)) \text{ AND (NOT } g_2(atts_2))$ AND...AND (NOT $g_{j-1}(atts_{j-1})$) AND $g_j(atts_j)>$.[2] Let this production have all the semantic functions that p has except that in place of the semantic function defining $X_0.trans$ as given above, it has the semantic function $XI_0.trans = f_j(XI_1.trans, ..., XI_{np}.trans)$. It also has one additional semantic function defining $XI_0.transinv$ given by $XI_0.transinv = \text{concatenate}(\alpha_0, XI_1.transinv, \alpha_1, XI_2.transinv, ..., XI_{np}.transinv, \alpha_{np})$.

5. The value of the translation is specified to be the value of the *transinv* attribute of the root (SI.transinv).

The essence of the inversion algorithm lies in point 4. To make this point more concrete, figure 3-1 shows the inversion of production p_6 of our example attribute grammar of figure 2-1. This production is *split* into two productions in the inverse attribute grammar. Whereas the production p_6 of T specified that Op derived 'multiply' and had a translation of either '$*_r$' or '$*_i$' the inverse productions pI_{6a} and pI_{6b} specify that OpI derives either '$*_r$' or '$*_i$' and in either case has a translation of 'multiply'. OpI's derivation of '$*_r$' or '$*_i$' is specified to be valid only if certain context conditions are satisfied.

Figure 3-2 presents the inverse of the remaining productions of the attribute grammar of figure 2-1. This specification would be produced automatically by the inversion algorithm. Due to space considerations, the inverse of productions p_7 and p_8 are not presented. Note that productions pI_1 and pI_2, while having different semantics, have the same context-free portion; the underlying context-free grammar of the inverse AG is ambiguous. In section 4.2 we show how to remove this ambiguity from the inverse specification.

---

[2]In addition, if p had an attached condition: $<\text{Condition: E}>$, then the condition E is also attached to pI_j.

pI_{6a}: $\overline{Op}I ::= *_r.$ <Condition: OpI.type = real>

   OpI.trans = '$*_r$';

   OpI.transinv = 'multiply';

pI_{6b}: OpI ::= $*_i.$ <Condition: NOT(OpI.type = real)>

   OpI.trans = '$*_i$';

   OpI.transinv = 'multiply';

Figure 3-1:   The inversion of p_6 splits into two productions

If an attribute grammar is in restricted inverse form, then there exists a duality between the context-free portion of the production (the syntax of the production) and the semantic function defining the X_0.trans attribute (the semantics of the production). While the context-free portion defines the strings X_0 can legally derive, the semantic function computing X_0.trans defines the translation of such strings. The inversion process exploits this duality by switching the role of syntax and semantics.

All the attributes of a nonterminal in the original attribute grammar remain in the corresponding nonterminal of the inverse AG. They will be defined properly as all the semantic functions of a production remain in the inverse production as well. Even the trans attribute remains in the inverse attribute grammar because it is no worse than any other attribute; it may be directly or indirectly used in some condition $g_j(atts_j)$ thereby influencing the translation.

The inverse grammar will have context conditions attached to the productions (see section 2.3) even if the original attribute grammar did not have any attached conditions. These conditions enforce context-sensitivities in the input. For example, according to the grammar T, the inverse grammar T^{-1} should not accept '$(80,5.8,*_i)$' as well-formed input; T would not translate any input string to '$(80,5.8,*_i)$'. The context conditions placed on T^{-1} will accomplish this. Without the conditions '$(80,5.8,*_i)$' would be accepted and translated by T^{-1} to either 'Multiply 80 by 5.8' or 'Multiply 80 to 5.8'. The attached context conditions can also be useful in parsing the input using the techniques of *attributed parsing* [21, 11].

Using the inversion method outlined in this section, it can be shown that if there exists a semantic tree in T translating s to m then there will exist a semantic tree in T^{-1} translating m to s. However, if T is many-to-one (it translates two unique strings s_1 and s_2 into the same output m), then T^{-1} will specify two ways to parse m, one parse tree producing the output s_1 and the other producing the output s_2. Hence if T is many-to-one, T^{-1} will not only be ambiguous, it will not be a function. We will return to the problem of ambiguity in section 4.2. To demonstrate the relationship between trees in the original attribute grammar and trees in the generated inverse attribute grammar, figure 3-3 gives a semantic tree for the string '$(80, 5.8, *_r)$', based on the inverse attribute grammar of figure 3-2. Compare this semantic tree to the semantic tree of figure 2-2.

pI_1: SI ::= (NumI1 , NumI2 , OpI) . <Condition: (OpI.trans = '$*_r$')
or (OpI.trans = '$*_i$')>

SI.trans = Concatenate('(', NumI1.trans, ',', NumI2.trans, ',', OpI.trans, ')');

OpI.type = If (NumI1.type = real) or (NumI2.type = real)
then real else int;

SI.transinv = Concatenate(OpI.transinv, NumI1.transinv, 'by', NumI2.transinv);

pI_2: SI ::= (NumI1 , NumI2 , OpI) . <Condition: (OpI.trans = '$+_r$')
or (OpI.trans = '$+_i$')>

SI.trans = Concatenate('(', NumI1.trans, ',', NumI2.trans,',', OpI.trans, ')');

OpI.type = If (NumI1.type = real) or (NumI2.type = real)
then real else int;

SI.transinv = Concatenate(OpI.transinv, NumI1.transinv, 'to', NumI2.transinv);

pI_3: NumI ::= IntegerI.

NumI.trans = IntegerI.trans;

NumI.type = int;

NumI.transinv = IntegerI.transinv;

pI_4: NumI ::= Decimal_numI.

NumI.translation = Decimal_numI.translation;

NumI.type = real;

NumI.transinv = Decimal_numI.transinv;

pI_{5a}: OpI ::= $+_r$. <Condition: OpI.type = real>

OpI.trans = '$+_r$';

OpI.transinv = 'add';

pI_{5b}: OpI ::= $+_i$. <Condition: NOT(OpI.type = real)>

OpI.trans = '$+_i$';

OpI.transinv = 'add';

Figure 3-2: The inverse AG generated from the example AG

3.4. Extending the Inversion Paradigm

In the last section we showed how any AG in *restricted inverse form* can be inverted. However, it is not always apparent how to express translations in this restricted form; many attribute grammars make use of constructs which violate these constraints. In section 4.1 we show how we were able to transform an attribute grammar which was not in restricted inverse form into one which was. However, this may not always be possible. In this section we suggest another alternative: extending RIF grammars to express a wider
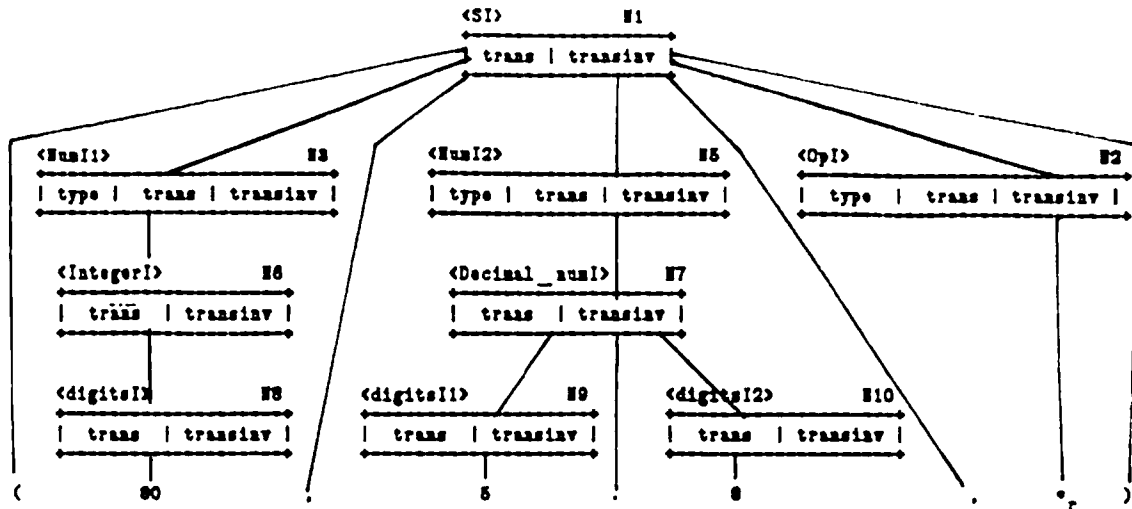
Figure 3-3: A typical semantic tree for the inverse AG

variety of translations yet still retain invertibility.

In our current work, using RIF grammars to express translations between programming languages, we have found that it often requires more than a simple token permuting function to define the translation of a subtree. For example, consider the production of figure 3-4. In this case the attribute if_stmt.trans is not defined by a token permuting function, and hence the production is not invertible by the inversion algorithm of the last section. In this example, genLab is a function from the domain of *integers* to the domain of *labels*, and genLab(i) = 'Li', where i is an integer and 'Li' is a string.

```
p: if_stmt ::= IF expression THEN stmt.

    if_stmt.trans ⟹ Concat[expression.trans,

                        'FJP', genLab(if_stmt.labnum),

                        stmt.trans,

                        'LAB', genLab(if_stmt.labnum)];
```

Figure 3-4: A semantic function using a non- *token permuting function*

The problem then is how the inversion algorithm can be expanded to deal with such constructs. Intuitively, the syntax of the inverse production should have the following form: [pI: if_stmtI ::= expressionI FJP X stmtI LAB X] where X represents a label. Assuming that we provide the inversion algorithm with knowledge about the primitive types (domains) employed by the semantic functions of the RIF grammar, there is no reason why it cannot also deduce this syntax for the inverse production. In particular, to invert this production the inversion algorithm would need to know

1. the syntax of a *label* and that

2. genLab is a function from integers to *labels*.

Using this information, it could invert the production p, producing the inverse productions pI and pI' given in figure 3-4. In this figure *label* is a nonterminal deriving a label. This nonterminal has the distinguished attribute, *label.value*, which gives the string derived by this nonterminal (e.g., if label derives 'Li', then the value of label.value is 'Li'). The condition attached to production pI enforces the relationship that the label derived from this nonterminal (given in label.value) must equal genLab(if_stmtI.labnum), as required by the original production p.

pI: if_stmtI ::= expressionI FJP label1 stmtI LAB label2.

$$<\text{Condition: (label1.value} = \text{genLab(if\_stmtI.labnum) AND}$$
$$\text{(label2.value} = \text{genLab(if\_stmtI.labnum)}>$$

if_stmtI.transinv = Concat['IF', expressionI.transinv, 'THEN',
                                                    stmtI.transinv];

pI': label :: = Li.

    label.value = Concat['L', 'i'];

<center>**Figure 3-5:** The inverse productions</center>

The technique illustrated by this example can be formalized and generalized, allowing RIF grammars to express a greater variety of constructs that arise naturally in AGs. Yet, this is only one out of several techniques that can be used to extend RIF grammars and the inversion algorithm. Part of our current work is aimed at finding a general version of RIF grammars and the inversion algorithm that will enable RIF grammars to express, without too much difficulty, most translations that arise in practice.

3.5. Efficiency

Although the inverted attribute grammar T^{-1} generated by inversion algorithm is guaranteed to be the inverse of the original attribute grammar T, it may be a very inefficient version of it. We can 'clean up' the attribute grammar T^{-1} by removing all *useless* attributes- those which cannot possibly contribute to the translation. A prime suspect as a useless attribute is the trans attribute; although it is essential in the original attribute grammar T, it probably (but not necessarily) contains unneeded information in the inverse attribute grammar T^{-1}. If we look at figure 3-2, we see that the attributes SI.trans, NumI.trans, Decimal_numI.trans and IntegerI.trans are useless and can be removed but that OpI.trans does contribute to the translation and cannot be removed. This 'cleaning up' of the attribute grammar can also be done automatically.

4. Using Attribute Grammar Inversion To Build An Interface For SQL

Attribute grammar technology is used in the PERFORM (Paraphrase and ERror message for FORMal languages) system, developed at the IBM Thomas J. Watson Research Center [17]. The PERFORM system is currently implemented to generate paraphrases and error

<center>13</center>

messages for a relational database querying language (SQL). It serves SQL users as a feedback device to make sure their queries are semantically correct from their point of view and from the system's point of view. It is an aid for the novice user in learning SQL and serves the occasional user as a documentation device for SQL queries. The paraphrases are designed in one to one correspondence to SQL expressions, preserving the SQL structure yet obeying natural language rules. The number of different natural language constructions employed is relatively small (essentially the same number as there are SQL constructions), and so is the basic vocabulary. Figure 4-1 gives an example of a SQL query and the English paraphrase generated by the PERFORM system.

SELECT DIVISION, ID, LOCATION, NAME FROM STAFF
WHERE DIVISION = "EASTERN" AND JOB = "CLERK";

What is the division, id number, city and last name for employees in
division "EASTERN", and with the job description "CLERK".

Figure 4-1: A SQL query and its English paraphrase

With PERFORM, users are still expected to construct their queries in SQL. To make the query construction itself easier for users, a guided natural language interface has been designed. It displays template queries in natural language on the screen with windows for the selection of specific items. The natural language constructs are based on the same language as PERFORM, consistent with the lexicon and syntax. The interface frees users from formal language requirements such as variable binding, or in the case of SQL, joining of tables. To assure the correct translation of the natural language input back into SQL, an "inverse" attribute grammar is needed [18].

To examine the feasibility of attribute grammar inversion, we decided to take a subset of the PERFORM attribute grammar (translating a subset of all SQL queries into an English paraphrase) and to apply the techniques given above to invert this subset attribute grammar. We performed this process by hand, but were faithful to the principles given above. The inverted attribute grammar translates simple English queries (paraphrases) into SQL queries and will become part of a larger system built around the PERFORM attribute grammar.

The original PERFORM attribute grammar was written without any thought of inversion and without any consideration to the principles of sections 3.2 and 3.3. For this reason we encountered several difficulties when we attempted the inversion process. Some of these difficulties were overcome by making small changes to the original attribute grammar. Other problems proved more stubborn and forced us to develop richer techniques of inversion to deal with specialized cases.

4.1. Non-invertible function constructs

Our first job in inverting the PERFORM AG was to put it into restricted inverse form. For most productions of the AG this was quite easy, requiring only small syntactic changes to the function computing the trans attribute. Sometimes, however, the function computing the trans attribute was semantically very different than a token permuting function and stronger techniques were required. An example of this sort of production is given in figure 4-2.

p: EXPR ::= FIELD_NAME.

 EXPR.trans = if (EXPR.plural = true)
   then make_plural(FIELD_NAME.trans) else FIELD_NAME.trans;

q: FIELD_NAME ::= location.

 FIELD_NAME.trans = 'city';

Figure 4-2: A non-invertible function construct

In this example EXPR derives the nonterminal FIELD_NAME. FIELD_NAME in turn can derive several terminal strings (SQL field names). EXPR.trans is set to the value of FIELD_NAME.trans with one qualification: if it has been determined elsewhere that this value, a noun which is the English equivalent of the SQL field name, is to be made plural, then first a function make_plural is called which finds the plural form of the noun. This function is not a token permuting function and cannot be inverted according to the paradigm of section 3.3. Conceptually, production p and productions of type q should invert to a set of productions $\{p_1, p_1', p_2, p_2', \ldots \}$ where p_i is of the form $[p_i: \text{EXPR} ::= \text{fname\_singular}_i]$ and $[p_i': \text{EXPR} ::= \text{fname\_plural}_i]$, where fname_singular_i and fname_plural_i are singular and plural terminal strings representing English field names. Besides many technical difficulties in deriving such an inverse set of productions, to do so would require an amount of semantic knowledge concerning the function make_plural which is beyond the scope of our paradigm. Instead we chose to rewrite the attribute grammar as in figure 4-3.

p: EXPR ::= FIELD_NAME.

 FIELD_NAME.plural = EXPR.plural;

 EXPR.trans = FIELD_NAME.trans;

q: FIELD_NAME ::= location.

 FIELD_NAME.trans = if FIELD_NAME.plural then 'cities' else 'city';

Figure 4-3: Figure 4-2 changed to restricted inverse form

By adding the attribute FIELD_NAME.plural we transmit the information as to whether the noun should be plural or singular further down the tree to the point where the translation for the field name is generated. We then explicitly choose either the plural or singular form based upon this information. The rewritten attribute grammar is equivalent

15

to the initial one and it is in restricted inverse form. It is less efficient since we had to make explicit the generation of different noun forms instead of performing this act in an efficient semantic function. Yet perhaps for this very reason the attribute grammar also becomes easier to read and understand.

In a similar fashion we rewrote the attribute grammar to accommodate another non-invertible function construct given in 4-4.

r: PRED ::= EXPR$_1$ COMP_OP EXPR$_2$.

    PRED.trans = if g(...)
        then concatenate(EXPR$_1$.trans, head(COMP_OP.trans), EXPR$_2$.trans)
        else concatenate(EXPR$_1$.trans, head(tail(COMP_OP.trans)), EXPR$_2$.trans);

s: COMP_OP ::= <.

    COMP_OP.trans = {'less than', 'is less then'};

<div align="center">Figure 4-4:   Another non-invertible function construct</div>

r: PRED ::= EXPR$_1$ COMP_OP EXPR$_2$.

    COMP_OP.value1 = g(...);

    PRED.trans = concatenate(EXPR$_1$.trans, COMP_OP.trans, EXPR$_2$.trans);

s: COMP_OP ::= <.

    COMP_OP.trans = if COMP_OP.value1 then 'less than'
        else 'is less then';

<div align="center">Figure 4-5:   Figure 4-4 changed to restricted inverse form</div>

In production s of this figure COMP_OP.trans was set equal to two possible values. The correct one was chosen higher up in the tree (at production r) depending on information available there. Once again the function defining PRED.trans is not in restricted inverse form due to the functions "head" (first element of list) and "tail" (all but the first element of list). We got around this problem by introducing a new attribute COMP_OP.value1 as given in figure 4-5. With these changes the productions were in restricted inverse form and the attribute grammar computed the same translation. Once again a little extra expense was entailed (introduction of the additional attribute COMP_OP.value1) but the attribute grammar became invertible. The attribute grammar also became cleaner in that we no longer assign two possible translations to a single node passing these values up the tree until there is enough information present to choose between them but we instead passed enough information down the tree to correctly choose the proper value initially.

Although several other problems were encountered, the examples presented above should suffice to give a flavour of the method of resolving these difficulties. In general we found that with a little effort most non-invertible constructs could be rewritten into an invertible format. Some of our solutions could be stated in more general terms and brought into the

paradigm of automatic inversion (such as the solution to the "head" and "tail" functions). A practical system might also employ special techniques to invert non-invertible function constructs which occur frequently in attribute grammars (such as the make_plural semantic function). To do so, more data needs to be collected concerning typical attribute grammars and the type of semantic functions they use.

4.2. Ambiguity

One other problem which we encountered in our inversion of the PERFORM subset deserves mention. In Figure 4-6, although p_a and p_b are unique context-free productions, pI_a and pI_b are the same context-free productions but with different semantics. This is due to the fact that the original grammar allows two pseudonyms (prodno and prodnum) to express the same meaning ('product number'). It results in an ambiguous grammar since we do not know which production applies on the input 'product number'. Fortunately this can be resolved by *collapsing* the two productions into a single production pI_{ab}. In this production, FIELD_NAME derives the terminal 'prodno' and is assigned the translation {'prodno', 'product number'}, meaning that either translation is acceptable.

p_a: FIELD_NAME ::= prodno.
       FIELD_NAME.trans = 'product number';

p_b: FIELD_NAME ::= prodnum.
       FIELD_NAME.trans = 'product number';

pI_a: FIELD_NAMEI ::= product number.
        FIELD_NAMEI.transinv = 'prodno';

pI_b: FIELD_NAMEI ::= product number.
        FIELD_NAMEI.transinv = 'prodnum';

pI_{ab}: FIELD_NAMEI ::= product number.
          FIELD_NAMEI.transinv = {'prodno', 'prodnum'};

Figure 4-6: Two unique productions inverting to identical ones

This technique of collapsing multiple productions into a single one can be more involved then demonstrated above if the semantic functions are more complicated or if there are context conditions on the productions. For example, consider production pI_1 and pI_2 of figure 3-2. Here, once again, the context-free portion of the productions are the same but the semantics are different. In this case, the productions also have different conditions attached. Once again we can collapse these productions into a unique production, pI_{12}, given in figure 4-7. Notice how the conditions attached to the productions get introduced into the semantic function defining SI.transinv. Using this single production instead of the two productions pI_1 and pI_2, the inverse RIF grammar no longer has an ambiguous underlying context-free grammar.

In the cases given above we were able to solve the ambiguity of the inverse attribute grammar by collapsing several productions into one. Unfortunately, often the ambiguity is spread out over several productions and can be hard to detect and remove. In general, if

pI_{12}: SI ::= (NumI1 , NumI2 , OpI) .
<Condition: (OpI.trans = '$*_r$') or (OpI.trans = '$*_i$') or
(OpI.trans = '$+_r$') or (OpI.trans = '$+_i$') >

SI.trans = Concatenate('(', NumI1.trans, ',', NumI2.trans, ',',
OpI.trans, ')');

OpI.type = If (NumI1.type = real) or (NumI2.type = real)
then real else int;

SI.transinv = if (OpI.trans = '$*_r$') or (OpI.trans = '$*_i$')

then Concatenate(OpI.transinv, NumI1.transinv, 'by', NumI2.transinv)

else Concatenate(OpI.transinv, NumI1.transinv, 'to', NumI2.transinv);

Figure 4-7: Two productions collapsing into one

the original translation is many-to-one, the inverse grammar will be one-to-many. This means that, if in the original attribute grammar two unique inputs produce the same output m, then in the inverse attribute grammar the input m will have two unique parse trees each producing a different output. The problem is which one should be selected? We have not yet been able to solve this problem to our satisfaction. One solution is to choose during run-time one of the parse trees. This choice could be based on some notion of a "best" translation or could be made arbitrarily. A better but much more difficult solution is to statically detect and remove the ambiguity from the inverse grammar.

5. Conclusion

This paper has introduced the technique of attribute grammar inversion. Given an attribute grammar in restricted inverse form, describing a translation T: $L_1 \rightarrow L_2$, the inversion algorithm presented in this paper will automatically synthesize the inverse attribute grammar T^{-1}: $L_2 \rightarrow L_1$.

The inversion process is highly modular; each production of the original attribute grammar gives rise to one or more productions in the inverse attribute grammar. Even if one production is not in restricted inverse form and is not invertible, the rest of the productions of the attribute grammar may still be invertible. And even within a non-invertible production, the construct causing the problem can be easily identified. An interactive inversion system could take advantage of this fact by automatically inverting as much of the attribute grammar as it can and then prompting the user for help where it encounters non-invertible constructs.

In this paper we also related our experience in inverting a subset of the PERFORM attribute grammar. This experiment was very successful. It proved that automatic inversion of attribute grammars is feasible and useful. It required surprisingly little effort; we believe that manual generation of the inverse attribute grammar PERFORM[1] from scratch would have required significantly more resources besides the fact that it would probably not be the true inverse of PERFORM. Our experience with PERFORM also indicates that even without a completely automated system for inversion, the principles of

section 3.3 provide useful guidelines on how to generate an inverse attribute grammar. In the worse case, it will provide users with a rough draft of the inverse attribute grammar which can then be further refined.

Our future research is aimed at building an automated system for translating between programming languages, based upon the idea of AG inversion, as outlined in section 1. The concepts introduced in this paper and the experience gained from our inversion of the PERFORM AG makes us optimistic on the success of this task.

ACKNOWLEDGEMENT

References

[1] A. V. Aho and J. D. Ullman.
Syntax Directed Translations and the Pushdown Assembler.
Journal of Computer and System Sciences 3(1):37-58, February, 1969.

[2] A. V. Aho and J. D. Ullman.
Properties of Syntax Directed Translations.
Journal of Computer and System Sciences 3(3):319-334, August, 1969.

[3] G.V. Bochmann.
Semantic evaluation from left to right.
Communications of the ACM 19, 1976.
pp. 55-62.

[4] 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.

[5] Rodney Farrow and Daniel Yellin.
Generating Bi-Directional Translators from RIF Grammars.
Technical Report, Department of Computer Science, Columbia University, New York,
New York 10027, August, 1985.

[6] Harald Ganzinger and Robert Giegerich.
Attribute Coupled Grammars.
In *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction.* ACM-SIGPLAN,
June, 1984.
Published as Volume 19, Number 6, of *SIGPLAN Notices.*

[7] H. Ganzinger, R. Giegerich, U. Moncke and R. Wilhelm.
A Truly Generative Semantics-Directed Compiler Generator.
In *Proceedings of the SIGPLAN Symposium on compiler construction.* ACM, June, 1982.

[8] E. Irons.
A Syntax Directed Compiler for ALGOL-60.
CACM 4:51-55, 1961.

[9] M. Jazayeri, W.F. Ogden, and W.C. Rounds.
The intrinsically exponential complexity of the circularity problem for attribute
grammars.
Communications of the ACM 18, 1975.

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

[11] Neil D. Jones and C. Michael Madsen.
      Attribute-Influenced LR Parsing.
      In *Lecture Notes in Computer Science 94*, pages 393-407. Springer-Verlag, Berlin-
          Heidelberg-New York, 1980.

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

[13] 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.

[14] 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.

[15] D. E. Knuth.
      Semantics of context-free languages.
      Mathematical Systems Theory 2:127-145, 1968.
      correction in volume 5, number 1.

[16] B. Lorho.
      Semantic attribute processing in the system DELTA.
      In A. Ershov and C.H.A. Koster (editor), *Methods of Algorithmic Language
          Implementation*. Springer-Verlag, Berlin-Heidelberg-New York, 1977.

[17] Eva-Maria M. Mueckstein.
      Q-TRANS: Query Translation Into English.
      In *Proceedings of the Eight International Joint Conference on Artificial Intelligence*, pages
          660-662. IJCAI-83, August, 1983.

[18] Eva-Maria M. Mueckstein.
      Controlled Natural Language Interfaces: The Best of Three Worlds.
      In *Proceedings of the ACM Computer Science Conference 1985*. ACM, March, 1985.

[19] Kari-Jouko Raiha, M. Saarinen, E. Soisalon-Soininen and M. Tienari.
      The Compiler Writing System HLP (Helsinki Language Processor).
      Technical Report A-1978-2, Dept. of Computer Science, Univ. of Helsinki, 1978.

[20] David A. Watt and Ole Lehrmann Madsen.
      Extended Attribute Grammars.
      The Computer Journal 26(2):142-153, 1983.

[21] David A. Watt.
      Rule splitting and attribute-directed parsing.
      In *Lecture Notes in Computer Science 94*, pages 363 - 392. Springer-Verlag, Berlin-
          Heidelberg-New York, 1980.

[22] 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.

[23] Daniel M. Yellin.
      Thesis Proposal: Restricted Inverse Form Grammars and Bi-Directional Translators.
      Technical Report, Department of Computer Science, Columbia University, New York,
          New York 10027, June, 1985.