Modular Attribute Grammars

G. D. P. DUECK* AND G. V. CORMACK†
Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada N2L 3G1

Attribute grammars provide a formal declarative notation for describing the semantics and translation of programming languages. Describing any real programming language is a significant software engineering challenge. From a software engineering viewpoint, current notations for attribute grammars have two flaws: tedious repetition of essentially the same attribute computations is inevitable, and the various components of the description cannot be decomposed into modules – they must be merged (and hence closely coupled) with the syntax specification. This paper describes a tool that generates attribute grammars from pattern-oriented specifications. These specifications can be grouped according to the separation of concerns arising from individual aspects of the compilation process. Implementation and use of the attribute grammar generator MAGGIE is described.

Received May 1988, revised January 1989

1. INTRODUCTION

Attribute grammars (Knuth, 1968) provide a formalism for describing the syntax, semantics, and translation of programming languages using a declarative specification. One advantage of such a specification is that it provides a formal definition of the programming language being described. Another advantage is that the specification can be converted automatically into a compiler. The formalism itself is simple, yet quite powerful. However, close inspection of even a small attribute grammar will reveal certain drawbacks, namely repetition, overwhelming detail, and the interleaving of many activities. The size and complexity of a specification written as an attribute grammar is such that the notion of correctness, originally an impetus for the formalism, is compromised; the grammar is difficult to read and particularly difficult to debug.

Conventional attribute grammars have no facility to support abstraction; we cannot easily extract from an attribute grammar the nature of individual sets of related computations, or modules. To address this problem, we have developed a mechanism to generate attribute grammars from rules which are grouped together as modular attribute grammars (MAGs). Because a MAG rule describes a class of attribute computations, modular attribute grammars can be significantly less complex than equivalent attribute grammars, and can be structured according to appropriate criteria for modular decomposition (Parnas, 1972).

To introduce MAGS, we first require some terminology from attribute grammars. An attribute grammar consists of a context-free grammar with each production augmented by attribute computations. *Attributes* are values associated with nodes in the derivation trees corresponding to strings in the language generated by the context-free grammar. An *attribute computation* defines an attribute in terms of other attributes in the same or adjacent nodes. Each production may have a number of attribute computations associated with it; because of this, the structure of the context-free grammar, rather than the relationships among the attribute computations, dominates an attribute grammar. We feel it is more appropriate to structure the attribute grammar according to the computation of attributes.

A single MAG is a set of patterns and associated templates. The patterns are applied to a context-free grammar; for those that match, an attribute computation is generated from the associated template. Pattern matching and selection of generated attribute computations are constrained; pattern matching uses *definability* and generated computations are selected by *need*, two ideas which are introduced in this paper.

The attribute computations generated by a MAG collectively define one or more *output* attributes from zero or more *input* attributes. These sets of input and output attributes constitute the interface to the MAG. Separate MAGs can be defined to address separate concerns in the language and compiler specification and can be combined according to their interfaces.

We have built a prototype and gained some experience with modular attribute grammars. The basic tool translates a context-free grammar and several MAGs into a monolithic attribute grammar and produces tables used by another tool to parse program source, build a form of *compound dependency graph* (Jalili, 1983), and evaluate the graph. Attribute computations are written as compound statements in the C programming language, and may reference user-defined functions. We have used the prototype to perform semantic analysis for declarations in Pascal and to develop techniques for using MAGs.

2. RELATED WORK

In several recent papers on attribute grammars, we find research motivated by the need to reduce the complexity of compiler descriptions written as attribute grammars. In particular, Koskimies, Räihä, and Sarjakoski (1982) note that attribute grammars are hard to read and understand, being far from self-documenting; Ganzinger and Giegerich (1984) quote further references in claiming that the few attribute rules which bear semantic significance are often buried in a large number of trivial

---

* On leave from Department of Mathematics and Computer Science, Brandon University, Brandon, Manitoba, Canada R7A 6A9.

† To whom correspondence should be addressed.

rules; and Räihä and Tarhio (1986) mention the difficulty of comprehending the global use pattern of an attribute in the presence of superfluous information. We agree with these concerns.

Koskimies *et al.* (1982) and Räihä (1984) propose that, in designing an attribute grammar, consideration should be given to the objects represented by the non-terminals, not by the productions. This approach, also pursued in the HLP84 system (Koskimies, Nurmi, Paaka, and Sippu, 1988), tends to emphasize single-pass algorithmic computation, and mandates that attribute computations be encapsulated within blocks defining non-terminals. In contrast, our approach emphasizes attributes rather than non-terminals, declarative rather than algorithmic computation, and modular structure independent from the context-free grammar.

GAG (Kastens, Zimmerman, and Hutt, 1982) is a compiler generator that uses monolithic attribute grammars and is necessarily rule based. GAG addresses complexity by providing attribute transfer and remote attribute access facilities. Attribute transfer abbreviates a set of simple-copy attribute computations into one statement. Remote attribute access abbreviates the transfer of attributes over long distances in the derivation tree. In effect, transfer and remote access are specific examples of simple abstractions that the designers have built into GAG. Modular attribute grammars facilitate general user abstractions that subsume both of these facilities. Jullig and DeRemer (1984) and Koskimies *et al.* (1982) also address the problem of automatic propagation of attribute values through the derivation tree.

Attribute coupled grammars (Ganzinger and Giegerich, 1984) and tree transformation grammars (Keller, Perkins, Payton, and Mardinly, 1984) are used to specify compilation phases. We define a phase as a data structure and an algorithm to map the (input) data structure into another (output) data structure. A formalism to specify a phase-structured compiler necessarily requires a way to specify data structures and inter-phase interfaces and we suggest this requirement increases the complexity of the formalism. Modular attribute grammars may be used to specify phases using a single, simpler formalism. However, we do not necessarily agree that phases are the most appropriate approach to module decomposition.

The MUG2 system (Ganzinger, Giegerich, Möncke, and Wilhelm) also provides a mechanism to specify phases of translation. Each phase resembles an attribute grammar, and context-free grammar rules are repeated in each phase. These phases, while resembling our MAGs, differ in two significant ways: (1) MAGs are not phase oriented; no ordering is implied, and MAGs may be mutually dependent and (2) MAGs do not include context-free grammar rules and are much less closely coupled to the concrete syntax.

Regular expressions improve the conciseness of the context-free portion of an attribute grammar (cf. Kastens *et al.* (1982) and Jullig and DeRemer (1984)). However, they introduce the additional notions of alternation and repetition into attribute computations. They provide no fundamental alternative to the monolithic structure of conventional attribute grammars.

Extended attribute grammars (EAGs) are another notation based on attribute grammars (Watt and Madsen (1982), Watt (1986)). In EAGs, the notation for expressing relationships betwen attributes is embedded within the notation for expressing the context-free grammar. The benefit of this approach is to allow attribute relationships to be stated implicitly. Our approach, in contrast, is based on decoupling attribute computation and context-free productions.

Work on attribute evaluators that are efficient in terms of execution time (left to right evaluators (Bochmann, 1976), ordered attribute grammars (Kastens, 1980), translation for direct execution (Katayama, 1985), one-pass evaluators (Koskimies, 1984)) and storage management (Jazayeri and Pozefsky, 1981) has progressed so that they can be realistically engineered for production compilers (Kastens *et al.*, 1982). Our work complements this technology; the monolithic attribute grammar produced in an intermediate stage of our prototype may serve as input for other systems.

3. MODULAR ATTRIBUTE GRAMMARS

A conventional attribute grammar consists of a number of context-free grammar rules; textually associated with each grammar rule are a number of attribute computations. In our system, the attribute computations are specified separately from the CFG in a number of MAGs (see Fig. 1). A MAG resembles an attribute grammar: it consists of a number of *patterns*; with each pattern is associated one or more templates. A pattern matches a set of rules in the CFG, and a template specifies the attribute computations to be generated for each matching CFG rule. Whereas a CFG rule contains only vocabulary symbols, a MAG pattern contains: variable symbols, which match *any* vocabulary symbol; quoted symbols, which match one vocabulary symbol; and ellipses, which match zero or more vocabulary symbols. Attribute references within the template are of the form P.x, where P is the name of a variable or quoted symbol,

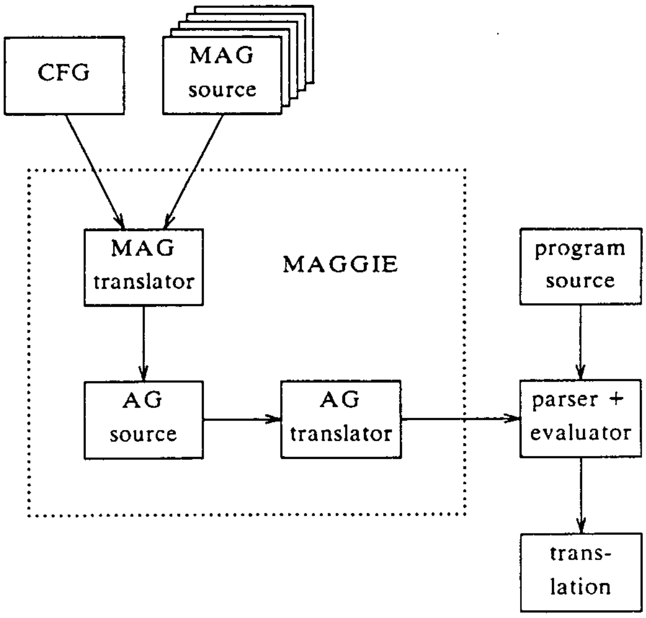

Figure 1. Translation of modular attribute grammars.

and x is an attribute name. (If P is repeated in the pattern, successive occurrences are denoted P, P1, P2, etc. For example, the MAG rule

```
P → P 'B ... Q
  P.w = P1.x + B.y + Q.z
```

matches the CFG rule

```
A → A B C D E
```

and generates the attribute computation

```
A.w = A1.x + B.y + E.z
```

The attribute computation is meaningful only if the attributes A.x, B.y, and E.z are defined (by some other computation). The computation defines A.w and is useful only if A.w is used elsewhere in an attribute computation. The matching of MAG rules to CFG rules is constrained to generate only useful and meaningful attribute computations. The generated attribute computations must comprise a well-formed attribute grammar (Knuth, 1968), but this is not necesary to the operation of the MAG translator; the translator does, however, perform some *post hoc* consistency checks.

An Example

In this section we use an example from Knuth (1968). The problem is to create modules that generate an attribute grammar to recognize and evaluate expressions on binary numbers. At the end of this section, we discuss how changes to the problem affect the modules.

The syntax is described by the following context-free grammar.

```
1 goal → expr
2 expr → term
3 expr → expr addop term
4 term → factor
5 term → term mulop factor
6 factor → int
7 factor → (expr)
8 int → digit
9 int → int digit
10 digit → 0
11 digit → 1
12 addop → +
13 mulop → *
```

The value of a binary number is determined using $\Sigma_{p=1}^{n} (d_p)^{p-1}$ where p denotes the position of a binary digit d with respect to the right boundary of a string of n digits.

The following MAG rules describe the synthesized attribute *val*, used to compose the value of a binary number or expression.

```
module val
1 'digit → '0
    'digit.val = 0;
2 'digit → '1
    'digit.val = 2 ^ 'digit.scale;
3 binexpr → Lopnd op Ropnd
    binexpr.val = callop(op.operator, Lopnd.val,
    Ropnd.val);
4 compose → valA valB
    compose.val = valA.val + valB.val;
5 A → · · · B · · ·
    A.val = B.val;
```

Considering only textual pattern matching, the first two MAG rules match productions 10 and 11; rule 3 matches productions 3, 5 and 7; rule 4 matches production 9; and rule 5 matches productions 1–13 in 20 different ways.

Rule 5 textually matches any production with one or more right-part symbols. An attribute computation will only be generated from this rule if, for a production that matches, (a) the right-part production symbol matching B is able to synthesize the attribute *val* and (b) an occurrence of the left-part production symbol matching A appears in some other production on the right-hand side and *needs* the attribute *val* in that context. For example, although the variable symbol B in rule 5 could match the symbol "(" in production 7, there is no opportunity for "(" to have synthesized this attribute.

The template in rule 3 invokes the function *callop* with two values and an operation indicator bound to the attribute *operator.*

```
module operator
6 op → '+
    op.operator = add;
7 op → '*
    op.operator = mul;
```

The inherited attribute *scale* (required by rule 2) is initially zero for the right-most digit of a binary number and incremented to the left.

```
module scale
8 binary → left right
    right.scale = binary.scale;
9 binary → left right
    left.scale = binary-scale + 1;
10 A → 'int
    'int.scale = 0;
11 A → B
    B.scale = A.scale;
```

Although rules 8 and 9 have the same pattern, their respective templates generate different computations. We have chosen not to condense these two into one MAG rule with two templates because of the implication that two attribute computations would then be generated simultaneously.

The result of applying the generator to the context-free grammar and the MAG rules illustrated above is the following attribute grammar.

```
1 goal → expr
    goal.val = expr.val;
2 expr → term
    expr.val = term.val;
3 expr → expr addop term
    expr.val = callop   (addop.operator,   expr1.val,
    term.val);
4 term → factor
    term.val = factor.val;
```

```
5 term → term mulop factor
   term.val = callop (mulop.operator, term1.val, factor.val);
6 factor → int
   int.scale = 0;
   factor.val = int.val;
7 factor → (expr)
   factor.val = expr.val;
8 int → digit
   digit.scale = int.scale;
   int.val = digit.val;
9 int → int digit
   int1.scale = int.scale + 1;
   digit.scale = int.scale;
   int.val = int1.val + digit.val;
10 digit → 0
   digit.val = 0;
11 digit → 1
   digit.val = 2 ^ digit.scale;
12 addopt → +
   addop.op = add;
13 mulop → *
   mulop.op = mul;
```

Discussion of modularity. To measure the degree of abstraction achieved by the three modules above, let us propose some syntactic and semantic changes to the example language and discuss how these affect the modular specification.

Module *val* is immune to all changes except those that directly affect the computation of *val*. For example, MAG rules 1 and 2 each handle unique digits. If the number-base changes from binary to some other base, the number of digits will increase and each will require a MAG rule similar to 1 and 2. This will change module *val* directly in proportion to the number base change. On the other hand, if more operators or more operator priority levels are added, rule 3 will remain unchanged. Rule 5 is a copy rule that simply propagates *val* up the tree. The language could also be changed so that *val* is required in more places; rule 5 guarantees that it is always available.

Module *operator* abstracts the individual operators. It is immune to number base and priority changes but is clearly affected by the addition of more operators. The grammar could be modified by incorporating the unit productions for *addop* and *mulop* into productions where they are referenced. In this case, the rules in module *operator* could be modified to recognize the operators + and * *in situ*, using patterns such as A → ... '+ ... or the more restrictive A → A '+ C, with no further changes to the modules required.

Module *scale* is unaffected by any of the proposed changes.

4. USING MODULAR ATTRIBUTE GRAMMARS

In this section we discuss different approaches to modularity in compiler construction and show how MAGs may be used to achieve module decomposition.

Bucket brigade. In a compiler, it is often the case that the name space or environment is propagated down the derivation tree and an environment augmented with new definitions is propagated up the tree (Koskimies *et al.*, 1982) (see Fig. 2).

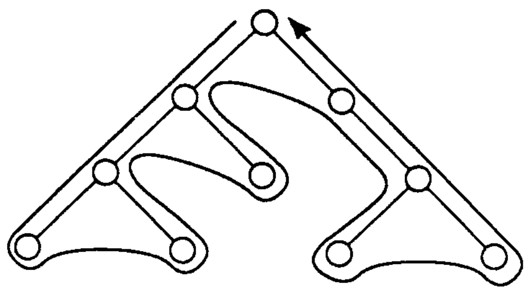

Figure 2. Left-to-right tree traversal.

The *bucket brigade* operator of regular-right part attribute grammars (Jullig and DeRemer, 1984) was introduced for this purpose. The following sets of MAG rules accomplish the same effect. We have subdivided them into two modules: the first produces the output attribute *env* intended to represent the environment propagated down the derivation tree and the second module produces the attribute *def* intended to represent the (modified) environment passed up the tree. We assume the start symbol in the context-free grammar is *goal*. In terms of inputs and outputs, the two modules are dependent both on themselves and on each other. The example illustrates a valid modular decomposition that that cannot be expressed as a phase oriented decomposition.

module *env*
(1) 'goal → A ...
  A.env = 0;
(2) A → B ...
  B.env = A.env;
(3) A → ... B C ...
  C.env = B.def;
module *def*
(4) A → ... B ...
  B.def = B.env;
(5) A → ... B
  A.def = B.def;
(6) A →
  A.def = A.env;

In module *env*, (2) the left symbol of a right part inherits *env* from the left part and (3) other symbols in the right part inherit *env* in terms of *def* synthesized by their left neighbours.

In module *def*, an empty production (6) sends *env* back to its parent as *def*. In a non-empty production, (5) *def* is synthesized from the right-most symbol of the right part. In the case (4) that a right-part symbol is not a non-terminal and therefore *def* cannot be synthesized by (5) or (6) when the symbol appears as a left part, *env* is passed along as *def*.

The modules shown above will generate a top-down left-to-right traversal for any derivation tree of any context-free grammar. The modules demonstrate that a constant number of rules can be used to describe an abstraction – namely, bucket brigade – that applies to any size attribute grammar; we feel this is a significant reduction in complexity. In a compiler application, the *def* module will be augmented by rules that recognize defining occurrences of identifiers and generate attribute definitions that modify the environment.

To illustrate the flexibility of modular attribute grammars, we show how to perform a top-down *right-to-left* traversal of the derivation tree with the following modules.

```
module env
  'goal→ ... A
    A.env = 0;
  A→ ... B
    B.env = A.env;
  A→ ... B C ...
    B.env = C.def;
module def
  A→ ... B ...
    B.def = B.env;
  A→ B
    A.def = B.def;
  A→
    A.def = A.env;
```

Abstract MAGs. A common technique in compiler construction is to transform the parse tree into an *abstract syntax tree*, a representation of the parse with less irrelevant detail. Some notation is necessary to specify the relationship of the parse tree to the abstract syntax tree. A MAG could be used to specify the translation from concrete to abstract syntax tree, whose shape is specified by a second context-free grammar (the abstract grammar). A second MAG, based on the abstract grammar, could specify the evaluation of attributes in the abstract syntax tree.

In a number of instances, the same abstraction can be achieved without using the two-phase approach described above. Patterns can be used in place of two common mappings from concrete to abstract syntax: *grouping* and *elision.* Grouping involves building nodes of a common type for subtrees derived from a number of nonterminal symbols or rules with similar meanings. In the binary-numbers example given above, the nonterminals *term*, *factor*, and *expr* all represent expressions; the nonterminals *addop* and *mulop* both represent operators. The rules *expr*→ *expr addop term* and *term*→ *term mulop factor* both represent binary-expression tree nodes. MAG rules conveniently express this abstract grouping: attributes are assigned to concrete syntax elements using MAG rules containing quoted symbols; abstract attribute computations are expressed in terms of variable symbols in MAG rules – the binding of these computations to the concrete syntax is controlled by the availability of attribute values. That is, the set of definable attributes is used to label nodes according to the abstract syntax. Elision, the removal of irrelevant detail, is accomplished in two ways: patterns containing . . . are used to match strings of symbols that are semantically irrelevant, and unit rules can be handled by the general *copy rule* paradigm presented above.

Several layers of abstraction may be built using attribute-controlled MAGs – each successive layer is built from attribute values generated by the previous one. Also, several different abstract views may be imposed on the parse tree at the same time: each abstract view needs to deal with only the sorts of information of interest to it. In our evolving methodology for the use of MAGs, we are attempting to write reusable modules such as symbol table routines, expression evaluators, overload resolution algorithms, and code generators. One of our first motivating examples was to express using reusable MAGs an operator selection algorithm akin to that of Ada; this algorithm has been described informally in terms of attributes by Cormack and Wright (1987). The general approach is that each such module would apply to any parse tree decorated by some specific set of attributes. The user of the module would be responsible to ensure (possibly by writing an interface MAG) that the input attributes decorate the parse tree in the appropriate manner. These input attributes will be often computed from the outputs of other modules. For example the expression evaluator in the example could be applied to a variety of concrete grammars, provided the operator nodes and value-generating nodes were assigned the attributes *op* and *val* respectively.

5. A FORMAL SPECIFICATION

Attribute grammars. An attribute grammar (AG) is composed of a context-free grammar G and a set of attribute computations R. The attribute rules describe how an *attributed parse tree* is derived from any string in the language L generated by G.

More formally, an AG is a sextuple (N,T,S,P,A,R), where N is the set of non-terminal symbols, T is the set of terminal symbols, $V = N \cup T$, $S \in N$ is the start symbol, P is a set of productions, A is a set of attributes, and R is a set of attribute computations. Each production $p \in P$ is a sequence $X_0,X_1, \ldots X_n$, where $X_0 \in N$ and $X_i \in V$ $(1 \leq i \leq n)$. A is the set of symbols used to denote attribute values within the attributed parse tree. R is a set of attribute computations of the form (p,D,U,f) where $p \in P$, D is an attribute reference of the form (i,a) $(0 \leq i < |p|,\ a \in A)$, U is a set of attribute references of the same form as D, and f is a function that defines the attribute referenced by D in terms of those referenced by U.

In this exposition, we denote members of N and A by words or letters, and members of T by words, letters, or single symbols. Membership in N, T, or A may be deduced from context. A production p is denoted $X_0 \rightarrow X_1X_2 \ldots X_n$. Attribute computations pertaining to production p are written adjacent to p. Within an attribute computation (p,D,U,f), each attribute reference (i,a) is denoted $X_i n.a$ where n selects from duplicate occurrences of the symbol X_i within p: $n = |\{X_j : X_i = X_j \wedge j < i\}|$. The special case of $n = 0$ is abbreviated $X_i.a$. The function f is a sequence of statements in the C programming language that computes D in terms of the elements of U.

The derivation of any string s from G may be described as a parse tree, with each leaf labelled by a terminal symbol such that, when concatenated from left to right, these labels form s. Each interior node represents the expansion of some production $p \in P = X_0 \rightarrow X_1X_2 \ldots X_n$; the node is labelled X_0 and its children are labelled $X_1,X_2 \ldots X_n$ in order. Each node also has a set of attributes and attribute values associated with it; the computation of these attributes is specified by the attribute computations. The attribute rule (p,D,U,f) where D is of the form $(0,a)$ specifies the computation of the *synthesized* attribute a for each node n representing an expansion of p. This value is obtained by applying f to the values denoted by U; each $(i,b) \in U$ denotes the value of attribute b of n if $i = 0$, otherwise of the ith child of n. If D is of the form (i,a) for $i \neq 0$, it specifies the computation of the *inherited* attribute a for each node which is the ith child of a node n representing an expansion of p. This value is obtained by applying f to the values denoted by U, as defined above. To ensure that the attribute computation is well defined for all parse trees, we may apply th consistency constraints of Knuth (1968). (However, the following description of modular attribute grammars neither depends on nor enforces these constraints.)

For example, consider the following attribute grammar.

A → u B z
  A.a = B.a
  A.b = B.b
  B.c = 0
B → C
  B.a = C.a
  B.b = C.b
  C.c = B.c
C → D E
  C.a = D.a
  C.b = E.b
  D.c = C.c
  E.c = C.c
D → v w
  D.a = f_1(D.c)
E → x y
  E.b = f_2(E.c)

Using the formalism to represent this grammar, we have $N = \{A, B, C, D, E\}$, $T = \{u, v, w, x, y, z\}$, $S = A$, $A = \{a, b, c\}$, $P = \{A \rightarrow u\ B\ z,\ B \rightarrow C,\ C \rightarrow D\ E,\ D \rightarrow v\ w,\ E \rightarrow x\ y\}$; the set $R = \{(p,D,U,F)\}$ is given in Table 1.

Table 1. The set of attribute computations R

	p	D	U	f	
{	(A → u B z	(0,a)	{(2,a)}	"A.a = B.a;"),	
	(A → u B z	(0,b)	{(2,b)}	"A.b = B.b;"),	
	(a → u B z	(2,c)	{}	"B.c = 0;"),	
	(B → C	(0,a)	{(1,a)}	"B.a = C.a;"),	
	(B → C	(0,b)	{(1,b)}	"B.b = C.b;"),	
	(B → C	(1,c)	{(0,c)}	"C.c = B.c;),	
	(C → D E	(0,a)	{(1,a)}	"C.a = D.a;"),	
	(C → D E	(0,b)	{(2,b)}	"C.b = E.b;"),	
	(C → D E	(1,c)	{(0,c)}	"D.c = C.c;"),	
	(C → D E	(2,c)	{(0,c)}	"E.c = C.c;"),	
	(D → v w	(0,a)	{(0, c)}	"D.a = f_1D.c;"),	
	(E → x y	(0,b)	{(0,c)}	"E.b = f_2E.c;")	}

Figure 3 shows the derivation tree and compound dependency graph produced by this attribute grammar for the input string "u v w x y z".

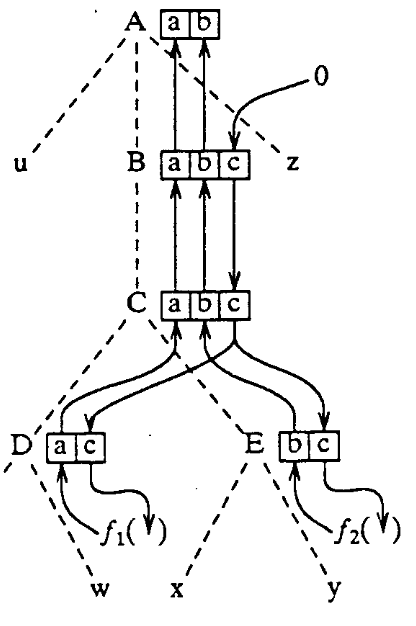

Figure 3. Attribute relations.

Modular attribute grammars. A modular attribute grammar (MAG) consists of an ordered set of patterns PT and a set of templates TM. The patterns are the analogues of productions in atribute grammars, and the templates are the analogues of attribte computations. One or more MAGs is applied to a context-free grammar G to create an attribute grammar as defined above. That is, the MAG (PT,TM) specifies the construction of R in terms of N, T, S, P, and A.

Each $pt \in PT$ has the form $Y_0,Y_1, \ldots Y_n$, where $Y_0 \in {'N} \cup W$ and $Y_i \in {'V} \cup W \cup \{\ldots\}$. $'N$ (the set of *quoted* nonterminals) is $\{'x : x \in N\}$, and $'V$ (the set of *quoted* vocabulary symbols) is $\{'x : x \in V\}$. W is the set of all variable symbols used in MAG rules. Each $tm \in TM$ is a quadruple (pt,D,U,F), where D is a pattern attribute reference (i,a) where $a \in A$ and $0 \leq i < |pt|$. U is a set of pattern attribute references of the same form as D. The denotation used for PT and TM is identical to that used for P and R in the attribute grammar.

From PT and P we compute MT, the set of textual matches between patterns and productions. Each element $mt \in MT$ has the form (p,pt,m), where $p \in P$, $pt \in PT$, and m provides a mapping from symbols of pt onto symbols of p: $m = M_0,M_1, \ldots M_{|pt|-1}$ $(0 \leq M_i \leq |p|)$. $mt \in MT$ if and only if the following constraints hold: $M_i \leq M_{i+1}$ $(0 \leqslant i < |pt|-1)$; $M_i + 1 = M_{i+1}$ if $Y_i \neq \ldots$ $(0 \leq i < |pt|-1)$; $Y_i = {'X_{M_i}}$ if $Y_i \in {'V}$ $(0 \leq i < |pt|)$. Finally, we impose a partial ordering on MT with the following relations: $(p,pt,m) < (p,pt',m')$ if pt appears before pt' in PT; $(p,pt,m) < (p,pt,m')$ if $M_i = M'_i$ $(0 \leq i < j)$ and $M_j < M'_j$ for some j.

Each element $mt \in MT$ generates a set of attribute computations, denoted $gen(mt)$; the union of all $gen(mt \in MT)$ is denoted RM. For $mt = (p,pt,M_0M_1 \ldots M_{|pt|-1}) \in MT$, each corresponding template (pt,D,U,f) generates an attribute computation $r = (p,D',U',f')$, where D', U', and f' are computed from D, U, and f by uniform replacement of pattern references of the form (i,a) by attribute references

(M_i,a). $gen(mt)$ is the set of all such r. RM is ordered according to the relation: $r < r'$ if r is generated from $s \in MT$ and r' is generated from $s' \in MT$ and $s < s'$.

If the entire set RM were used as R as described above, R would contain many meaningless, useless, and ambiguous attribute rules. A rule $r = (p,D,U,f)$ is meaningless if any of the attribute value denoted by U does not exist; r is useless if the attribute value denoted by D is not used as an input to some other rule; r and r' are ambiguous if they both define the same attribute for some node in the parse tree.

The set of definable attributes $TD \subseteq V \times A$ is the set of pairs (w,a) (denoted w.a) for which a meaningful rule $r = (p,(i,a),U,f) \in RM$ exists, where $X_i = w$. TD is the set defined by the recursive rule $(w,a) \in TD$ if $\exists_{r=(p,(i,a),U,f)\in RM}\ X_i = w \wedge \forall_{(j,b)\in U}\ (X_j,b) \in TD$. TD may be computed assuming $TD = \{\}$ initially, and repeatedly testing $(w,a) \in V \times A$ for membership in TD, iterating to convergence.

The set of needed attributes $TN \subseteq V \times A$ is computed in a similar fashion: it is the set that satisfies the two constraints: $TN \supseteq \{(S,a):(S,a) \in TD\}$; $(w,a) \in TN$ if $\exists_{r=(p,(i,b),U,f)\in RM}\ (X_i,b) \in TN \wedge \exists_{(j,a)\in U} X_j = w$.

The (possibly ambiguous) set of rules RA is generated by constraining RM using TD and TN: $(RA = \{r = (p(i,a),U,f):(X_i,a) \in TN \forall_{(j,b)\in U}(X_j,b) \in TD\}$. Two rules $r = (p,D,U,f)$ and $r' = (p',D',U',f')$ are ambiguous if $p = p'$ and $D = D'$. In this case, r is selected if $r < r'$. This arbitrary choice ensures that each attribute of a symbol within a production rule is defined by only one attribute computation, and allows the user to specify MAG rules in order of importance. The resulting attribute grammar must still be checked for consistency; in particular, this construction may yield an incomplete or circular attribute grammar. The final set of rules of the attribute grammar is $R = \{(p,D,U,f) \in RA: \nexists_{(p,D,U',f')\in RA}(p,D,U',f') < (p,D,U,f)\}$.

For example, the attribute grammar given in the previous section is specified by the following MAG.

```
'D→...
  D.a = f1(D.c);
P→...Q...
  Q.a;
'E→...
  E.b = f2(E.c);
P→...Q...
  P.b = Q.b;
P→...'B...
  B.c = 0;
P→...Q...
  Q.c = P.c;
```

Table 2. The set of templates *TM*

p	*D*	*U*	*f*
{ ('D → . . .	(0,a)	{(0,c)}	"D.a = f_1(D.c);"),
(P^0 → . . . Q^0 . . .	(0,a)	{(2,a)}	"P.a = Q.a;"),
('E → . . .	(0,b)	{(0,c)}	"E.b = f_2(E.c);"),
(P^1 → . . . Q^1 . . .	(0,b)	{(2,b)}	"P.b = Q.b;"),
(P^2 → . . . 'B . . .	(2.c)	{}	"B.c = 0;"),
(P^3 → . . . Q^3 . . .	(2,c)	{(0,c)}	"Q.c = P.c;") }

When this MAG is applied to the context free grammar from the previous example, we have the set of patterns $PT = \{$'D → . . ., P^0 → . . . Q^0 . . ., 'E → . . ., P^1 → . . . Q^1 . . ., P^2 → . . . 'B . . ., P^3 → . . . Q^3 . . .$\}$, $'N = \{$'A, 'B, 'C, 'D, 'E$\}$, $'V = 'N \cup \{$'b, 'c, 'e, 'f, 'g, 'h$\}$, and $W = \{$P, Q$\}$. The set of templates $TM = \{(pt,D,U,f)\}$ is shown in Table 2.

The set of textual matches between patterns and productions is $MT = \{(p,pt,m)\}$ where m is a tuple of elements that correspond positionally to elements of a pattern pt and that map elements of pt onto elements of a production p; MT is shown in Table 3.

The textual matches MT and templates TM generate $RM = \{(p,D',U',f')\}$, a set of attribute computations. For example, (P^0 → . . . Q^0 . . ., (0,a), {(2,a)}, "P.a = Q.a;") $\in TM$ and (A → u B z, P^0 → . . . Q^0 . . ., (0, 1, 1, 2)) $\in MT$ generate (A → u B z, (0,a), {(1,a)}, "A.a = u.a;") $\in RM$ because P^0 corresponds to 0 in m which corresponds to A in p, and Q^0 corresponds to the second 1 in m which corresponds to u in p. RM is shown in Table 4. For the purpose of this exposition attribute definitions and references of the form (i,a), as required by the formalism, are rewritten inthe form $X_i n.a$ in Table 4.

RM is used to generate TD, the set of definable attributes. TD is initially empty: B.c can be added to TD because $\exists_{(p,D,U,f)\in RM} D =$ B.c and $U = \{\}$. Now the set $\{D:\forall_{(p,D,U,f)} U \subseteq \{\text{B.c}\}\} = \{\text{C.c}\}$ can be added to TD.

Table 3. The set of textual matches *MT*

p	*pt*	*m*
{ (A → u B z	P^0 → . . . Q^0 . . .	(0, 1, 1, 2)),
(A → u B z	P^0 → . . . Q^0 . . .	(0, 1, 2, 3)),
(A → u B z	P^0 → . . . Q^0 . . .	(0, 1, 3, 4)),
(A → u B z	P^1 → . . . Q^1 . . .	(0, 1, 1, 2)),
(A → u B z	P^1 → . . . Q^1 . . .	(0, 1, 2, 3)),
(A → u B z	P^1 → . . . Q^1 . . .	(0, 1, 3, 4)),
(A → u B z	P^2 → . . . 'B . . .	(0, 1, 2, 3)),
(A → u B z	P^3 → . . . Q^3 . . .	(0, 1, 1, 2)),
(A → u B z	P^3 → . . . Q^3 . . .	(0, 1, 2, 3)),
(A → u B z	P^3 → . . . Q^3 . . .	(0, 1, 3, 4)),
(B → C	P^0 → . . . Q^0 . . .	(0, 1, 1, 2)),
(B → C	P^1 → . . . Q^1 . . .	(0, 1, 1, 2)),
(B → C	P^3 → . . . Q^3 . . .	(0, 1, 1, 2)),
(C → D E	P^0 → . . . Q^0 . . .	(0, 1, 1, 2)),
(C → D E	P^0 → . . . Q^0 . . .	(0, 1, 2, 3)),
(C → D E	P^1 → . . . Q^1 . . .	(0, 1, 1, 2)),
(C → D E	P^1 → . . . Q^1 . . .	(0, 1, 2, 3)),
(C → D E	P^3 → . . . Q^3 . . .	(0, 1, 1, 2)),
(C → D E	P^3 → . . . Q^3 . . .	(0, 1, 2, 3)),
(D → v w	'D → . . .	(0, 1)),
(D → v w	P^0 → . . . Q^0 . . .	(0, 1, 1, 2)),
(D → v w	P^0 → . . . Q^0 . . .	(0, 1, 2, 3)),
(D → v w	P^1 → . . . Q^1 . . .	(0, 1, 1, 2)),
(D → v w	P^1 → . . . Q^1 . . .	(0, 1, 2, 3)),
(D → v w	P^3 → . . . Q^3 . . .	(0, 1, 1, 2)),
(D → v w	P^3 → . . . Q^3 . . .	(0, 1, 2, 3)),
(E → x y	'E → . . .	(0, 1)),
(E → x y	P^0 → . . . Q^0 . . .	(0, 1, 1, 2)),
(E → x y	P^0 → . . . Q^0 . . .	(0, 1, 1, 2)),
(E → x y	P^0 → . . . Q^0 . . .	(0, 1, 2, 3)),
(E → x y	P^1 → . . . Q^1 . . .	(0, 1, 1, 2)),
(E → x y	P^1 → . . . Q^1 . . .	(0, 1, 2, 3)),
(E → x y	P^3 → . . . Q^3 . . .	(0, 1, 1, 2)),
(E → x y	P^3 → . . . Q^3 . . .	(0, 1, 2, 3)) }

Table 4. The set of generated attribute computations *RM*

p	*D*	*U*	*f*
{ (A → u B z	A.a	{u.a}	"A.a = u.a;"),
(A → u B z	A.a	{B.a}	"A.a = B.a;"),
(A → u B z	A.a	{z.a}	"A.a = z.a;"),
(A → u B z	A.b	{u.b}	"A.b = u.b;"),
(A → u B z	A.b	{B.b}	"A.b = B.b;"),
(A → u B z	A.b	{z.b}	"a.b = z.b;"),
(A → u B z	B.c	{}	"B.c = 0;"),
(A → u B z	u.c	{A.c}	"u.c = A.c;"),
(A → u B z	B.c	{A.c}	"B.c = A.c;"),
(A → u B z	z.c	{A.c}	"z.c = A.c;"),
(B → C	B.a	{C.a}	"B.a = C.a;"),
(B → C	B.b	{C.b}	"B.b = C.b;"),
(B → C	C.c	{B.c}	"C.c = B.c;"),
(C → D E	C.a	{D.a}	"C.a = D.a;"),
(C → D E	C.a	{E.a}	"C.a = E.a;"),
(C → D E	C.b	{D.b}	"C.b = D.b;"),
(C → D E	C.b	{E.b}	"C.b = E.b;"),
(C → D E	D.c	{C.c}	"D.c = C.c;"),
(C → D E	E.c	{C.c}	"E.c = C.c;"),
(D → v w	D.a	{D.c}	"D.a = f_1(D.c);"),
(D → v w	D.a	{v.a}	"D.a = v.a;"),
(D → v w	D.a	{w.a}	"D.a = w.a;"),
(D → v w	D.b	{v.b}	"D.b = v.b;"),
(D → v w	D.b	{w.b}	"D.b = w.b;"),
(D → v w	v.c	{D.c}	"v.c = d.c;"),
(D → v w	w.c	{D.c}	"w.c = D.c;"),
(E → x y	E.a	{x.a}	"E.a = x.a;"),
(E → x y	E.a	{y.a}	"E.a = y.a;"),
(E → x y	E.b	{E.c}	"E.b = f_2(E.c);)"),
(E → x y	E.b	{x.b}	"E.b = x.b;"),
(E → x y	E.b	{y.b}	"E.b = y.b;"),
(E → x y	x.c	{E.c}	"x.c = E.c;"),
(E → x y	y.c	{E.c}	"y.c = E.c;") }

The set *TD* finally becomes {B.c, C.c, D.c, D.a, E.c, E.b, E.c, e.c, f.c, g.c, h.c, C.b, C.a, B.b, B.a, A.b, A.a}.

The set *TN* of needed attributes is initially composed of those attributes definable on the goal symbol. *TN* is initially {A.a, A.b}. By repeatedly adding to *TN* the set $\{U{:}\exists_{(p,D,U,f)\in RM} D \in TN \text{ and } U \in TD\}$, the final set $TN = \{$A.a, A.b, B.a, C.a, D.a, D.c, C.c, B.c, B.b, C.b, E.c, E.c$\}$ is created.

The set of attribute computations *R*, specified in the attribute grammar of the previous section, may now be derived from *RM*, *TD*, and *TN* by selecting those computations in *RM* (Table 4) that define needed attributes (members of *RN*) in terms of definable atributes (members of *TD*).

6. IMPLEMENTATION

Figure 1 shows the components of the implementation. The following algorithm may be used to generate an attribute grammar *AG* from a context free grammar *G* and a set of modular attribute grammars *MAG*.

1. Generate the set of all attributes *X.a* where *X* is a symbol in the vocabulary of *G* and *a* is any attribute name used in *MAG*. Initially, all attributes are marked as not needed and not defined.
2. For each attribute computation *c* generated by matching some pattern *pt* to some production *p*, if all inputs of *c* are definable mark the output of *c* as definable. Perform this procedure until no more attributes can be marked.
3. Mark all definable attributes *S.a* as needed, where *S* is the goal symbol of G.
4. For each attribute computation *c* generated by matching some pattern *pt* to some production *p*, if the output X.a of *c* is needed and no attribute computation with the output X.a has yet been associated with *p*, associate *c* with *p* and mark all the inputs of *c* as needed. Repeat this procedure until no more attribute computations can be associated with productions.
5. Check to ensure that
   - (*a*) no attribute is both synthesized and inherited,
   - (*b*) every synthesized attribute X.a is defined in all attribute computations associated with each production that has X as a left-part symbol.
   - (*c*) every inherited attribute Y.b is defined in all attribute computations associated with each production that has Y as a right-part symbol.

In the conventional view of an attribute grammar evaluator, the derivation tree is decorated with attributes connected to form a graph. The graph, traditionally called a compound dependency graph, reflects both the structure of the derivation tree and relationships specified in the attribute grammar. The graph represents an expression that can be evaluated and, if the attribute grammar specifies a translation, the result of the evaluation is the desired translation. The size of the graph and the time to traverse it is linear with respect to the size of the input and the size of the attribute grammar. For our research, a straightforward graph evaluator is sufficient to provide adequate performance. Circularity in the compound dependency graph is detected by the evaluator. Should performance become an issue, we have noted previously that tools for generating efficient evaluators exist and that the MAG translator has been designed so that such evaluators can be interfaced with the monolithic attribute grammar produced by our system.

7. EXPERIENCE

We have written MAGs to perform semantic analysis for Pascal declarations. Our implementation uses a 158 line context-free grammar and 12 modules comprising 101 MAG rules. The generated attribute grammar contains 525 attribute definitions that require 50 unique attribute computations.

From this experience we note that module usage may be partitioned into *creation*, *combination*, and *distribution*. Modules that are not partitioned strictly according to these categories generally contain some aspect of each. Creation modules use rules to recognise syntactic constructs that uniquely identify semantic constructs – syntactic recognition is enhanced by availability of definable attributes. Combining modules recognise situations where multiple threads of similar information are combined into a single thread. For example, a list is composed by appending elements to another list or to an empty list. Combining situations can be recognised solely through attribute availability but may be triggered by cues in the concrete syntax. Distribution modules use bucket-brigade rules to distribute information throughout a derivation tree. These modules may also use syntactic cues to terminate distribution.

From the Pascal implementation we also note that using MAGs is not as easy as we would like. Perhaps due to unfamiliary with the pattern matching process, we observe that the user occasionally must resort to inspecting the generated AG to determine the effect of pattern matching. This is analogous to object-level debugging of a program written in a high level language; we hope to find methods and tools to allow us to wrok exclusively at the MAG source level. This aim is partly addressed with the use of a strict methodology like that outlined above.

8. CONCLUSIONS AND FUTURE RESEARCH

The complexity of attribute grammars is due to the dominance of the structure of the context-free grammar over the structure of information flow through attribute computations. This paper has suggested that the specification of the computation and flow of information through attributes can be decoupled from concrete syntax and rearranged as an appropriate module decomposition. A tool to generate attribute grammars from modular specifications has been used to investigate how decomposition should apply to attribute grammars. We have gained enough experience to conclude that modular attribute grammars represent an improvement over monolithic attribute grammars in reducing the complexity of attribute specifications; we have now in a position to make suggestions for further improvements.

The textual patterns introduced in this paper are intentionally designed for simplicity. While more sophisticated patterns are possible, we are not yet convinced that individual improvements in this area will significantly reduce complexity. In contrast, the contribution of *definability* and of *need* in constraining textual pattern matching cannot be overstressed. If a better technique for MAG translation is to be found, we believe it will be coupled with an increase in the power of attribute-constrained pattern matching.

Our experience with modular attribute grammars shows that some form of bucket-brigade rules is incorporated into most modules. We hesitate to incorporate automatic generation of copy rules into the MAG translator because this is a particular solution for a particular problem that may be addressed by a more general solution. We find many modules, not necesarily concerned solely with attribute propagation by copy rule, that share a similar overall appearance. A possible solution is to provide generic modules, parametrized by attribute and production symbols, that can be used to produce module instances. This would make it possible to incorporate an instance of a generic general-purpose bucket-brigade module as a sub-module of any module, or, indeed, to compose modules entirely of instances of generic modules.

MAGGIE was designed to address shortcomings in attribute grammars that became apparent because of considerable experience (our own and others) in using attribute grammars to specify programming languages and compilers. Before making any changes to MAGGIE, we need a larger body of expertise in creating reusable MAGs using the existing tool. Only with this expertise can we properly evaluate possible enhancements.

REFERENCES

G. V. Bochmann, Semantic evaluation from left to right. *Comm. ACM* **19** (2), 55–62 (1976).

G. V. Cormack and A. K. Wright, Polymorphism in the Compiled Language Force One. *Proc. 20th Annual Hawaii International Conference on System Sciences*, pp. 284–292 (1987).

R. Farrow, LINGUIST-86: yet another translator writing system based on attribute grammars. *Proc. SIGPLAN '82 Symposium on Compiler Construction, SIGPLAN Notices*, **17** (6), 160–171 (1982).

H. Ganzinger, R. Giegerich, U. Möncke and R. Wilhelm, A truly generative semantics-directed compiler generator. *Proc. SIGPLAN '82 Symposium on Compiler Construction, SIGPLAN Notices*, **17** (6), 173–184 (1982).

H. Ganzinger and R. Giegerich, Attribute Coupled Grammars, *Proc. SIGPLAN '84 Symposium on Compiler Construction, SIGPLAN Notices*, **19** (6), 157–170 (1984).

F. Jalili, A general linear-time evaluator of attribute grammars. *SIGPLAN Notices*, **18** (9), 35–44

M. Jazayeri and D. Pozefsky, Space-efficient storage management in an attribute grammar evaluator. *IACM Trans. Prog. Lang. Syst*, **3** (4), 388–404 (1981).

R. K. Jullig and F. DeRemer, Regular right-part grammars. *Proc. SIGPLAN '84 Symposium on Compiler Construction, SIGPLAN Notices*, **19** (6), 171–178 (1984).

U. Kastens, Ordered attribute grammars. *Acta. Inf.*, **13**, 229–256 (1980).

U. Kastens, E. Zimmerman and B. Hutt, *GAG – a Practical Compiler Generator*. Lecture Notes in Computer Science 141, Berlin, Springer (1982).

T. Katayama, Translation attribute grammars into procedures. *ACM Trans. Prog. Lang. Syst.*, **6** (3), 345–369 (1984).

S. E. Keller, J. A. Perkins, T. F. Payton and S. P. Mardinly, Tree transformation techniques and experiences. *Proc. SIGPLAN '84 Symposium on Compiler Construction, SIG-PLAN Notices*, **19** (6), 190–201 (1984).

D. E. Knuth, Semantics of context-free languages. *Math. Syst. Theory*, **2** (2), 127–145 (1968); correction in *Math. Syst. Theory*, **5** (1), 95–96 (1971).

K. Koskimies, K-J. Räihä and M. Sarjakoski, Compiler construction using attribute grammars. *Proc. SIGPLAN '82 Symposium on Compiler Construction, SIGPLAN Notices*, **17** (6), 53–159 (1982).

K. Koskimies, A note on one-pass evaluation of attribute grammars. *BIT*, **25**, 439–450 (1985).

K. Koskimies, O. Nurmi, J. Paakki and S. Sippu, The design of a language processor generator. *Soft. Pract. Exp.*, **18** (2), 107–135 (1988).

D. L. Parnas, On the criteria to be used in decomposing systems into modules. *Comm. ACM*, **15** (10), 1053–1058 (1972).

K-J. Räihä, M. Saarinen, E. Soisalon-Soininen and M. Tienari, The compiler writing system HLP, report A-1978-2, Department of Computer Science, University of Helsinki (1978).

K-J. Räihä, Attribute Grammar design using the compiler writing system HLP. *Methods and Tools for Compiler Construction*, Cambridge UP, pp. 183–206 (1984).

K-J. Räihä and J. Tarhio, A globalizing transformation for attribute grammars, *Proc. SIGPLAN '86 Symposium on Compiler Construction, SIGPLAN Notices*, **21** (7), 74–83 (1986).

D. A. Watt and O. L. Madsen, Extended attribute grammars. *Comp. Journal* **26** (3), 142–153 (1962).

D. A. Watt, Executable Semantic Descriptions. *Soft. Pract. Exp.*, **16** (1), 13–43 (1986).