To

From

Date

memo

University of Waterloo

Gerald Dueck ~~Karel~~ ~~Dueck~~

Marylene Ledeuff

Sent Oct. 11/89

CS-88-19

IBM Canada Ltd.
Communications
Dept.
5 Place Ville Marie
Montreal, P.Q.
H3B 2G3

UNIVERSITY OF SASKATCHEWAN

SASKATOON, SASKATCHEWAN, CANADA S7N 0W0

INVOICE DATE	DESCRIPTION	PURCHASE ORDER	VOUCHER	INVOICE AMOUNT	DISCOUNT AMOUNT	AMOUNT PAID
88/09/14	CS-88-03		0016003	2.00		2.00 Y
88/09/14	CS-88-19		0016004	2.00		2.00 Y
			TOTALS ▶	4.00		4.00

received & sent reports Oct. 25/88

VENDOR NO. C0001890771 CHEQUE DATE 88/10/11 CHEQUE NO. 21-023103

CS-88-03	Diversity, Accessibility and Adaptability Data Communication Needs For Higher Education The University of Waterloo Experience	D.D. Cowan S.L. Fenton T.M. Stepien A. Pittman	2.00
CS-88-04	Performance Of A Multifrontal Scheme For Partially Seperable Optimization	A.R. Conn N.I.M. Gould M. Lescrenier Ph.L. Toint	2.00
CS-88-05	Optical Mass Storage Systems And Their Performance	S. Christodoulakis K. Elliott D. Ford K. Hatzilemonias E. Ledoux M. Leitch R. Ng	2.00
CS-88-06	Performance Analysis And Fundamental Performance Trade Offs For CLV Optical Disks	S. Christodoulakis Daniel A. Ford	2.00
CS-88-07	Wavefront Elimination and Renormalization	W.P. Tang	2.00
CS-88-08	Orthogonal Decomposition Of Dense And Sparse Matrices On Multiprocessors	E.C-h L. Chu	5.00
CS-88-09	Adaptive Implicit Criteria for Two Phase Flow with Gravity and Capillary Pressure	P.A. Forsyth	2.00
CS-88-10	Exploring the Properties of Uniform B-splines Using the Fourier Integral (in preparation)	Eng-Wee Chionh	?

~~016004~~

Printing Requisition / Graphic Services

19928

1. **Please complete unshaded areas** on form as applicable.
2. **Distribute copies** as follows: White and Yellow to Graphic Services. **Retain Pink Copies for your records.**
3. On completion of order the **Yellow copy will be returned** with the printed material.
4. **Please direct enquiries, quoting requisition number and account number,** to **extension 3451.**

TITLE OR DESCRIPTION: CS-88-19 Modular Attribute Grammars

DATE REQUISITIONED: May 20/88 | DATE REQUIRED: ASAP | ACCOUNT NO.: 126619/41

REQUISITIONER – PRINT | PHONE: 2192 | SIGNING AUTHORITY:

MAILING INFO – NAME: S. DeANGELIS | DEPT.: C.S. | BLDG. & ROOM NO.: DC 2314 | ☑ DELIVER ☐ PICK-UP

Copyright: I hereby agree to assume all responsibility and liability for any infringement of copyrights and/or patent rights which may arise from the processing of, and reproduction of, any of the materials herein requested. I further agree to indemnify and hold blameless the University of Waterloo from any liability which may arise from said processing or reproducing. I also acknowledge that materials processed as a result of this requisition are for educational use only.

NUMBER OF PAGES: 22 | NUMBER OF COPIES: 75

TYPE OF PAPER STOCK: ☑ BOND ☐ NCR ____ PT. ☑ COVER ☐ BRISTOL ☑ SUPPLIED ☐ ____

PAPER SIZE: ☑ 8½ x 11 ☐ 8½ x 14 ☐ 11 x 17 ☐ ____

PAPER COLOUR: ☑ WHITE ☐ ____ | INK: ☑ BLACK ☐ ____

PRINTING: ☐ 1 SIDE ____ PGS. ☑ 2 SIDES ____ PGS. | NUMBERING: FROM ____ TO ____

BINDING/FINISHING: 3 down left side ☑ COLLATING ☑ STAPLING ☐ ____ HOLE PUNCHED ☐ PLASTIC RING

FOLDING/PADDING | CUTTING SIZE

Special Instructions

Math fronts & backs enclosed

COPY CENTRE — OPER. NO. | BLDG. | MACH. NO.

DESIGN & PASTE-UP — OPER. NO. | TIME | LABOUR CODE

D 0 1
D 0 1
D 0 1

TYPESETTING — QUANTITY | OPER. NO. | TIME | LABOUR CODE

PAP 0 0 0 0 0 — T 0 1
PAP 0 0 0 0 0 — T 0 1
PAP 0 0 0 0 0 — T 0 1

PROOF

PRF
PRF
PRF

NEGATIVES — QUANTITY | OPER. NO. | TIME | LABOUR CODE

FLM — C 0 1
FLM — C 0 1
FLM — C 0 1
FLM — C 0 1
FLM — C 0 1

PMT

PMT — C 0 1
PMT — C 0 1
PMT — C 0 1

PLATES

PLT — P 0 1
PLT — P 0 1
PLT — P 0 1

STOCK

0 0 1
0 0 1
0 0 1
0 0 1

BINDERY

RNG — B 0 1
RNG — B 0 1
RNG — B 0 1
MIS 0 0 0 0 0 — B 0 1

OUTSIDE SERVICES

$ ____ COST

TAXES – PROVINCIAL ☐ FEDERAL ☐

GRAPHIC SERV. OCT. 85 482-2

Modular Attribute Grammars

Gerald D.P. Dueck
Brandon University

Gordon V. Cormack
Dept. of Computer Science
University of Waterloo

Research Report CS-88-19
May 1988

Faculty of Mathematics

University of Waterloo
Waterloo, Ontario, Canada
N2L 3G1

Modular Attribute Grammars

*Gerald D. P. Dueck**
Gordon V. Cormack†

Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada N2L 3G1

Research Report CS-88-19

ABSTRACT

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 expressions 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.

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

* On leave from Brandon University, Brandon, Manitoba, Canada. Bitnet address: gdueck at water; gery at uofmcc.
† Internet address: gvcormack@waterloo.edu. Bitnet or UUCP: gvcormac at water.

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 algorithms or modules. To address this problem, we have developed a mechanism to generate attribute grammars from templates which are grouped together as modular attribute grammars (MAGs). Because a template describes a class of attribute expressions, 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 expressions. *Attributes* are values associated with nodes in the derivation trees corresponding to strings in the language generated by the context-free grammar. An *attribute expression* defines an attribute in terms of other attributes in the same or adjacent nodes. Each production may have a number of attribute expressions associated with it; because of this, the structure of the context-free grammar, rather than the relationships among the attribute expressions, 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 rule is generated from the associated template. Pattern matching and selection of generated attribute rules are constrained; pattern matching uses *tentative definition* and generated rules are selected by *need*, two ideas which are introduced in this paper.

The rules 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 (see Figure 1) 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 expressions are written as compound statements in C (Kernighan and Ritchie, 1978) and may reference user-defined functions. We have used the prototype to perform semantic analysis of Pascal declarations and to develop techniques for using MAGs.

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; Gansinger and Giegerich (1984) quote further references in claiming that the few

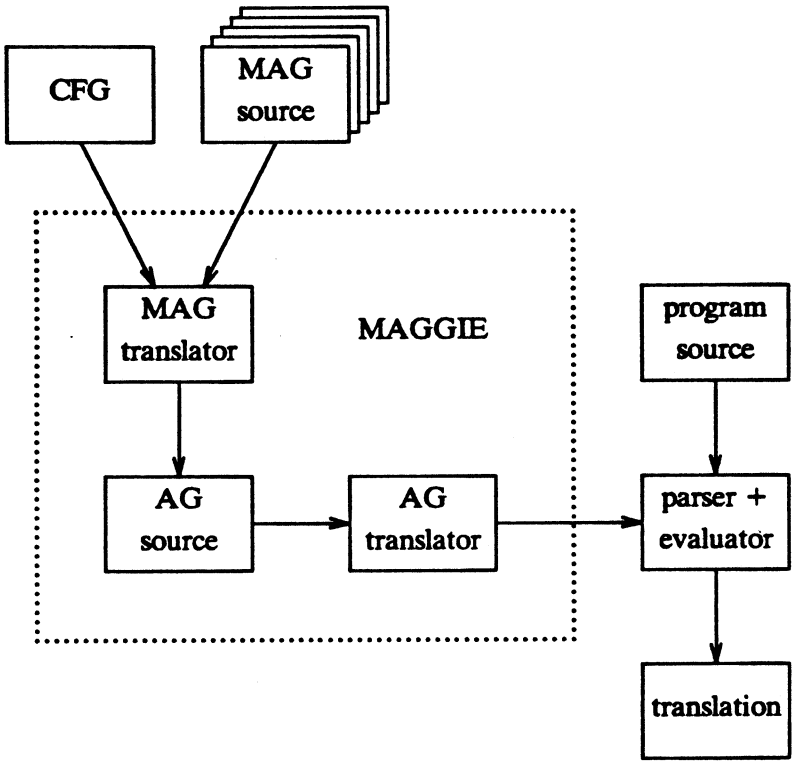

Figure 1: Translation of modular attribute grammars

attribute rules which bear semantic significance are often buried in a large number of trivial 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.

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 expressions 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.

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 expressions. 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 between 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 (Katayam, 1984), 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 (see Figure 1) may serve as input for other systems.

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. The modules developed in the course of experimenting with the tools described here support this view.

Attribute grammars

An attribute grammar (AG) is composed of a context-free grammar G and a set of attribute rules 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, $V=N\cup T$, $S\in T$ is the start symbol, P is a set of productions, A is a set of attributes, and R is a set of rules for the computation of attribute values. Each production $p\in P$ is a sequence $X_0,X_1,\cdots X_n$, 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 rules 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 $X_0\rightarrow X_1X_2\cdots X_n$. Attribute rules pertaining to production p are written adjacent to p. Within an attribute rule (p,D,U,f), each attribute reference (i,a) is denoted $X_in.a$ where $n=|\{X_j:X_i=X_j \wedge j<i\}|$. 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 $p \in P = X_0 \rightarrow X_1 X_2 \cdots X_n$; the node is labelled X_0 and its children are labelled $X_1, X_2 \cdots X_n$ 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 rules. The attribute rule (p, D, U, f) where D is of the form $(0, a)$ specifies the computation of the 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 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.

For example, consider the following attribute grammar.

A → b B c
    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 → e f
    D.a = f_1(D.c)

E → g h
    E.b = f_2(E.c)

In this grammar, we have N={A, B, C, D, E}, T={b, c, e, f, g, h}, S=A, A={a, b, c}, P= {A → b B c, B → C, C → D E, D → e f, E → g h }.

R={	(A → b B c	(0,a)	{(2,a)}	"(0,a) = (2,a);"),
	(A → b B c	(0,b)	{(2,b)}	"(0,b) = (2,b);"),
	(A → b B c	(2,c)	{}	"(2,c) = 0;"),
	(B → C	(0,a)	{(1,a)}	"(0,a) = (1,a);"),
	(B → C	(0,b)	{(1,b)}	"(0,b) = (1,b);"),
	(B → C	(1,c)	{(0,c)}	"(1,c) = (0,c);),
	(C → D E	(0,a)	{(1,a)}	"(0,a) = (1,a);"),
	(C → D E	(0,b)	{(2,b)}	"(0,b) = (2,b);"),
	(C → D E	(1,c)	{(0,c)}	"(1,c) = (0,c);"),
	(C → D E	(2,c)	{(0,c)}	"(2,c) = (0,c);"),
	(D → e f	(0,a)	{(0,c)}	"(0,a) = f_1(0,c);"),
	(E → g h	(0,b)	{(0,c)}	"(0,b) = f_2(0,b);") }

Figure 2 shows the derivation tree and compound dependency graph produced by this attribute grammar for the input string "b d f g h c".

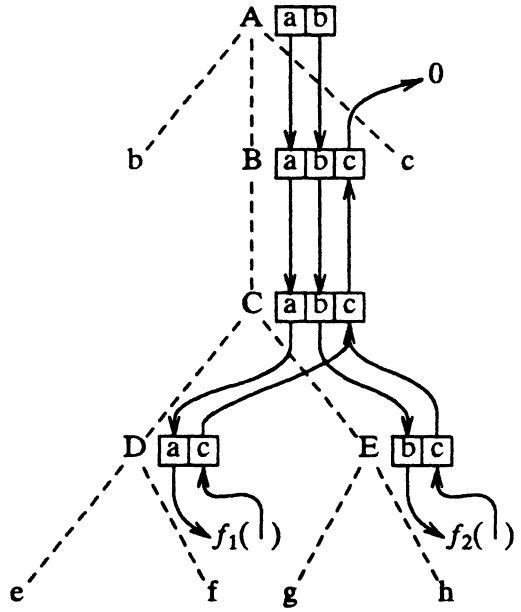

Figure 2: Attribute Relations

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 an attribute grammars, and the templates are the analogues of attribute rules. 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 $Y_0 Y_1 \cdots Y_n$, where $Y_0 \in {}'N \cup W$ and $Y_i \in {}'V \cup W \cup \{\cdots\}$. $'N$ (the *stropped* set of nonterminals) is $\{'x : x \in N\}$, and $'V$ (the *stropped* set of vocabulary symbols) is $\{'x : x \in V\}$. W is the set of all words. 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 $m = M_0 M_1 \cdots M_{|pt|-1}$ $(0 \leq M_i \leq |p|)$. $mt \in MT$ if and only if the following constraints hold: $M_i < M_{i+1}$ $(0 \leq i < |pt|-1)$; $M_i + 1 = M_{i+1}$ if $Y_i \neq \cdots$ $(0 \leq i < |pt|-1)$; $Y_i = 'X_{M_i}$ if $Y_i \in 'V$ $(0 \leq i < |pt|)$. Finally, we impose a total ordering on MT with the following relations: $(p,pt,m) < (p',pt',m')$ if p appears before p' in P; $(p,pt,m) < (p,pt',m')$ 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 rules, denoted $gen(mt)$; the union of all $gen(mt \in MT)$ is denoted RM. For $mt = (p,pt,M_0 M_1 \cdots M_{|pt|-1}) \in MT$, each corresponding template (pt,D,U,f) generates an attribute rule $r = (p,D',U',f')$, 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 values denoted by U do 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 tentatively defined 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 smallest 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 smallest 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 $r < r'$. 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 = f_1('D.c);
P → · · · Q · · ·
    P.a = Q.a;
'E → · · ·
    'E.b = f_2('E.c);
P → · · · Q · · ·
    P.b = Q.b;
P → · · · 'B · · ·
    'B.c = 0;
P → · · · Q · · ·
    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 ∪ {'b, 'c, 'e, 'f, 'g, 'h}, and the relevant subset of W is {P, Q}. The set TM={(pt,D,U,f)} of templates, with attribute references of the form $X_i n.a$ replaced by (i,a), is

TM={ ('D → · · · (0,a) {(0,c)} "(0,a) = f_1(0,c);"),
(P_0→ · · · Q_0 · · · (0,a) {(2,a)} "(0,a) = (2,a);"),
('E → · · · (0,b) {(0,c)} "(0,b) = f_2(0,b);"),
(P_1→ · · · Q_1 · · · (0,b) {(2,b)} "(0,b) = (2,b);"),
(P_2→ · · · 'B · · · (2,c) {} "(2,c) = 0;"),
(P_3→ · · · Q_3 · · · (2,c) {(0,c)} "(2,c) = (0.c);") }

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={ (A → b B c $P_0 \rightarrow \cdots Q_0 \cdots$ (0, 1, 1, 2)),
(A → b B c $P_0 \rightarrow \cdots Q_0 \cdots$ (0, 1, 2, 3)),
(A → b B c $P_0 \rightarrow \cdots Q_0 \cdots$ (0, 1, 3, 4)),
(A → b B c $P_1 \rightarrow \cdots Q_1 \cdots$ (0, 1, 1, 2)),
(A → b B c $P_1 \rightarrow \cdots Q_1 \cdots$ (0, 1, 2, 3)),
(A → b B c $P_1 \rightarrow \cdots Q_1 \cdots$ (0, 1, 3, 4)),
(A → b B c $P_2 \rightarrow \cdots$ 'B $\cdots$ (0, 1, 2, 3)),
(A → b B c $P_3 \rightarrow \cdots Q_3 \cdots$ (0, 1, 1, 2)),
(A → b B c $P_3 \rightarrow \cdots Q_3 \cdots$ (0, 1, 2, 3)),
(A → b B c $P_3 \rightarrow \cdots Q_3 \cdots$ (0, 1, 3, 4)),
(B → C $P_0 \rightarrow \cdots Q_0 \cdots$ (0, 1, 1, 2)),
(B → C $P_1 \rightarrow \cdots Q_1 \cdots$ (0, 1, 1, 2)),
(B → C $P_3 \rightarrow \cdots Q_3 \cdots$ (0, 1, 1, 2)),
(C → D E $P_0 \rightarrow \cdots Q_0 \cdots$ (0, 1, 1, 2)),
(C → D E $P_0 \rightarrow \cdots Q_0 \cdots$ (0, 1, 2, 3)),
(C → D E $P_1 \rightarrow \cdots Q_1 \cdots$ (0, 1, 1, 2)),
(C → D E $P_1 \rightarrow \cdots Q_1 \cdots$ (0, 1, 2, 3)),
(C → D E $P_3 \rightarrow \cdots Q_3 \cdots$ (0, 1, 1, 2)),
(C → D E $P_3 \rightarrow \cdots Q_3 \cdots$ (0, 1, 2, 3)),
(D → e f 'D → $\cdots$ (0, 1)),
(D → e f $P_0 \rightarrow \cdots Q_0 \cdots$ (0, 1, 1, 2)),
(D → e f $P_0 \rightarrow \cdots Q_0 \cdots$ (0, 1, 2, 3)),
(D → e f $P_1 \rightarrow \cdots Q_1 \cdots$ (0, 1, 1, 2)),
(D → e f $P_1 \rightarrow \cdots Q_1 \cdots$ (0, 1, 2, 3)),
(D → e f $P_3 \rightarrow \cdots Q_3 \cdots$ (0, 1, 1, 2)),
(D → e f $P_3 \rightarrow \cdots Q_3 \cdots$ (0, 1, 2, 3)),
(E → g h 'E → $\cdots$ (0, 1)),
(E → g h $P_0 \rightarrow \cdots Q_0 \cdots$ (0, 1, 1, 2)),
(E → g h $P_0 \rightarrow \cdots Q_0 \cdots$ (0, 1, 2, 3)),
(E → g h $P_1 \rightarrow \cdots Q_1 \cdots$ (0, 1, 1, 2)),
(E → g h $P_1 \rightarrow \cdots Q_1 \cdots$ (0, 1, 2, 3)),
(E → g h $P_3 \rightarrow \cdots Q_3 \cdots$ (0, 1, 1, 2)),
(E → g h $P_3 \rightarrow \cdots Q_3 \cdots$ (0, 1, 2, 3)) }

The textual matches MT and templates TM generate $RM=\{(p,D',U',f'i\}$, a set of attribute rules. For example, ($P_0 \rightarrow \cdots Q_0 \cdots$, (0,a), {(2,a)}, "(0,a) = (2,a);") $\in TM$ and (A → b B c, $P_0 \rightarrow \cdots Q_0 \cdots$, (0, 1, 1, 2)) $\in MT$ generate (A → b B c, (0,a), {(1,a)}, "(0,a) = (1,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 b in p.

RM={ (A → b B c (0,a) {(1,a)} "(0,a) = (1,a);"),
(A → b B c (0,a) {(2,a)} "(0,a) = (2,a);"),
(A → b B c (0,a) {(3,a)} "(0,a) = (3,a);"),
(A → b B c (0,b) {(1,b)} "(0,b) = (1,b);"),
(A → b B c (0,b) {(2,b)} "(0,b) = (2,b);"),
(A → b B c (0,b) {(3,b)} "(0,b) = (3,b);"),
(A → b B c (2,c) {} "(2,c) = 0;"),
(A → b B c (1,c) {(0,c)} "(1,c) = (0,c);"),
(A → b B c (2,c) {(0,c)} "(2,c) = (0,c);"),
(A → b B c (3,c) {(0,c)} "(3,c) = (0,c);"),
(B → C (0,a) {(1,a)} "(0,a) = (1,a);"),
(B → C (0,b) {(1,b)} "(0,b) = (1,b);"),
(B → C (1,c) {(0,c)} "(1,c) = (0,c);"),
(C → D E (0,a) {(1,a)} "(0,a) = (1,a);"),
(C → D E (0,a) {(2,a)} "(0,a) = (2,a);"),
(C → D E (0,b) {(1,b)} "(0,b) = (1,b);"),
(C → D E (0,b) {(2,b)} "(0,b) = (2,b);"),
(C → D E (1,c) {(0,c)} "(1,c) = (0,c);"),
(C → D E (2,c) {(0,c)} "(2,c) = (0,c);"),
(D → e f (0,a) {(0,c)} "(0,a) = f_1(0,c);"),
(D → e f (0,a) {(1,a)} "(0,a) = (1,a);"),
(D → e f (0,a) {(2,a)} "(0,a) = (2,a);"),
(D → e f (0,b) {(1,b)} "(0,b) = (1,b);"),
(D → e f (0,b) {(2,b)} "(0,b) = (2,b);"),
(D → e f (1,c) {(0,c)} "(1,c) = (0,c);"),
(D → e f (2,c) {(0,c)} "(2,c) = (0,c);"),
(E → g h (0,a) {(1,a)} "(0,a) = (1,a);"),
(E → g h (0,a) {(2,a)} "(0,a) = (2,a);"),
(E → g h (0,b) {(0,c)} "(0,b) = f_2(0,c);"),
(E → g h (0,b) {(1,b)} "(0,b) = (1,b);"),
(E → g h (0,b) {(2,b)} "(0,b) = (2,b);"),
(E → g h (1,c) {(0,c)} "(1,c) = (0,c);"),
(E → g h (2,c) {(0,c)} "(2,c) = (0,c);") }

The elements of RM are more useful for the purpose of this exposition if attribute references (i,a) are rewritten in the form $X_i n.a$.

RM={	(A → b B c	A.a	{b.a}	"A.a = b.a;"),
	(A → b B c	A.a	{B.a}	"A.a = B.a;"),
	(A → b B c	A.a	{c.a}	"A.a = c.a;"),
	(A → b B c	A.b	{b.b}	"A.b = b.b;"),
	(A → b B c	A.b	{B.b}	"A.b = B.b;"),
	(A → b B c	A.b	{c.b}	"A.b = c.b;"),
	(A → b B c	B.c	{}	"B.c = 0;"),
	(A → b B c	b.c	{A.c}	"b.c = A.c;"),
	(A → b B c	B.c	{A.c}	"B.c = A.c;"),
	(A → b B c	c.c	{A.c}	"c.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 → e f	D.a	{D.c}	"D.a = f_1D.c;"),
	(D → e f	D.a	{e.a}	"D.a = e.a;"),
	(D → e f	D.a	{f.a}	"D.a = f.a;"),
	(D → e f	D.b	{e.b}	"D.b = e.b;"),
	(D → e f	D.b	{f.b}	"D.b = f.b;"),
	(D → e f	e.c	{D.c}	"e.c = D.c;"),
	(D → e f	f.c	{D.c}	"f.c = D.c;"),
	(E → g h	E.a	{g.a}	"E.a = g.a;"),
	(E → g h	E.a	{h.a}	"E.a = h.a;"),
	(E → g h	E.b	{E.c}	"E.b = f_2E.c;"),
	(E → g h	E.b	{g.b}	"E.b = g.b;"),
	(E → g h	E.b	{h.b}	"E.b = h.b;"),
	(E → g h	g.c	{E.c}	"g.c = E.c;"),
	(E → g h	h.c	{E.c}	"h.c = E.c;") }

RM is used to generate *TD*, the set of tentatively defined 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 $\{D:\forall_{(p,D,U,f)} U \subseteq \{B.c\}\}$ = {C.c} can be added to *TD*. 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 tentatively defined on the goal symbol. *TN* is initially {A.a, A.b}. By repeatedly adding to *TN* the set $\{U:\exists_{(p,D,U,f)\in RM} D \in TN$ and $U \notin TD\}$, 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 rules R, specified in the attribute grammar of the previous section, may now be derived from RM, TD, and TN.

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 $\sum_{p=1}^{n}(d_p)^{p-1}$ where p denotes the position of a binary digit d with respect to the right boundary of a string of n digits.

The following patterns 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 binop → Lopnd op Ropnd
    binop.val = callop(op.op, 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 patterns match productions 10 and 11; pattern 3 matches productions 3, 5 and 7; pattern 4 matches production 9; and pattern 5 matches productions 1 – 13 in 20 different ways.

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

The template in pattern 3 invokes a function that has been bound to the attribute *op*.

module *op*

```
 6 op → '+
      op.op = (int) add;
 7 op → '*
      op.op = (int) mul;
```

The inherited attribute *scale* (required by pattern 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  factor → 'int
      'int.scale = 0;
11  A → B
      B.scale = A.scale;
```

The result of applying the generator to the context-free and pattern grammars illustrated above is the following attribute grammar. (The numeric suffixes on vocabulary symbols disambiguate multiple occurrences of the symbol within a production.)

```
 1 goal → expr
    goal.val = expr.val ;
 2 expr → term
    expr.val = term.val ;
 3 expr → expr addop term
    expr.val = callop ( addop.op , expr1.val , term.val ) ;
 4 term → factor
    term.val = factor.val ;
 5 term → term mulop factor
    term.val = callop ( mulop.op , 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 addop → +
    addop.op = ( int ) "add" ;
13 mulop → *
    mulop.op = ( int ) "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 this affects the modular specification.

Module *val* is immune to all changes except those that directly affect the computation of *val*. For example, patterns 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 pattern and template 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, pattern 3 will remain unchanged. Pattern 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; pattern 5 guarantees that it is always available.

Module *op* 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 patterns in module *op* could be modified to recognize the operators + and * *in situ*, with no further changes to the modules required.

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

Implementation Concerns

Figure 1 shows the components of the implementation. 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. Should performance become an issue, we note 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.

MAG templates are written as C statements using a special denotation for attributes. These denotations are recognised by the AG translator, which replaces attribute references with references to elements of an activation record and emits the C statement as a procedure parameterized by the activation record. The activation record is built by the parser from an attribute-record descriptor provided by the AG translator.

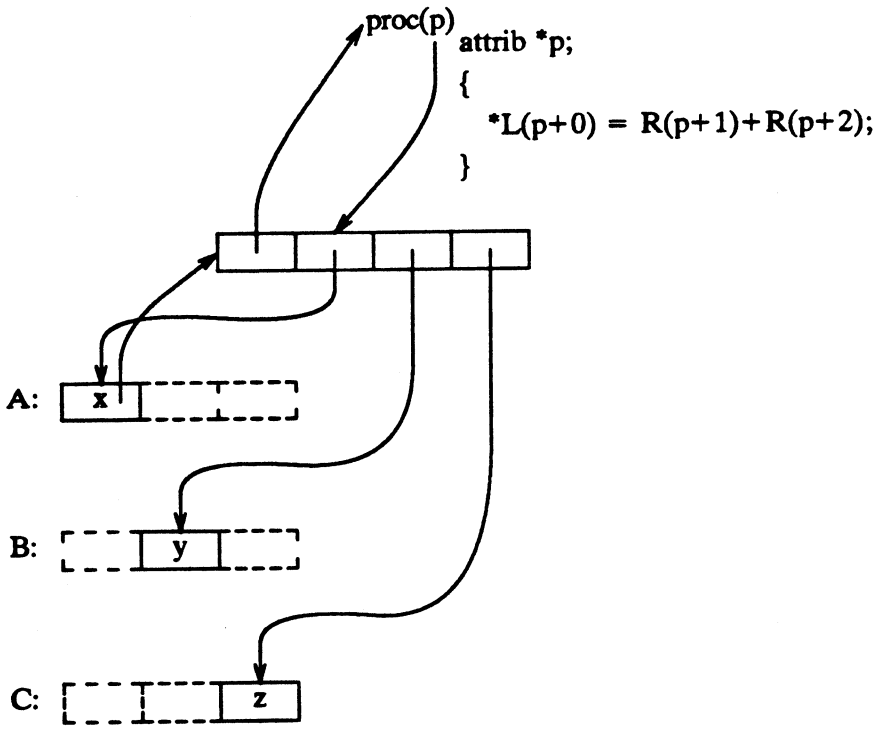

Figure 3: Evaluation of attribute expression
A.x = B.y + C.z

Figure 3 shows the structure built to evaluate a representative attribute expression. Each node in the derivation tree has an associated attribute record. An unevaluated attribute is initialized by the graph builder to be a pointer to an activation record prefixed with a field containing the address of a procedure. Upon the first evaluation of an attribute, the procedure is invoked. The body of the routine is generated from an attribute expression with routine calls in place of attribute references. Routine *R* evaluates an attribute and returns the result; routine *L* returns an L-value. In the example, the initial value of *A.x* will be replaced by a computed value. A tag is maintained by *L* so that future references to an evaluated attribute via routine *R* simply return the value.

The implementation has the following implications.

1) Module decomposition is enhanced because well-known data structures and algorithms can be used to implement abstractions that can be accessed easily by C function calls from within attribute rules.

2) There is no constraint imposed by the evaluator on the interdependencies of attribute expressions.

3) The compound dependency graph is executed directly instead of using an interpretive evaluator.

4) Because the parser builds the compound dependency graph from activation-record descriptors and procedure addresses provided by the generator, the derivation tree is not required and is not built.

The modular input to the MAG translator is typically smaller in terms of numbers of attribute definitions than the resulting monolithic attribute grammar. It would not be reasonable to generate a unique C procedure for each attribute definition. For example, many generated attribute definitions are just copy rules. The C procedure for a copy rule is quite simple, and our implementation allows us to generate just one version of the copy procedure that serves for all occurrences of copy rules. We have generalized this optimization to generate unique C procedures for sets of isomorphic attribute expressions.

For a given production, there may be a number of attribute expressions that compute information used only by other attribute expressions in the same rule and not at all by attribute expressions in any other production. Using textual substitution, an attribute reference may be replaced by the expression used in the attribute's definition. This optimization, in conjunction with the one described in the previous paragraph, has yielded a 90 per cent reduction in the number of procedures generated.

As a consequence of the textual substitution optimization, it is possible that certain attribute definitions associated with a production, which had been originally generated because of need, become unreferenced. If this is the situation for all definitions of an attribute of a particular grammar symbol, the attribute becomes redundant for that symbol and may be removed from the symbol's attribute record.

In development work, it is useful to have automatic checks for undefined attributes. This is unnecessary in our system because we perform static consistency checks on the generated attribute grammar. Starting from the goal production,

a) for each synthesized attribute $X.a$ that is referenced, the attribute must be defined for all derivations of X, and

b) for each inherited attribute $X.a$ that is referenced, the attribute must be defined for all productions that contain X in the right part.

This simple test guarantees that all attributes reachable from the goal are defined for any derivation tree. It can also be used to detect attribute definitions that are not referenced at all, indicating errors in the modular grammars.

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 Figure 4).

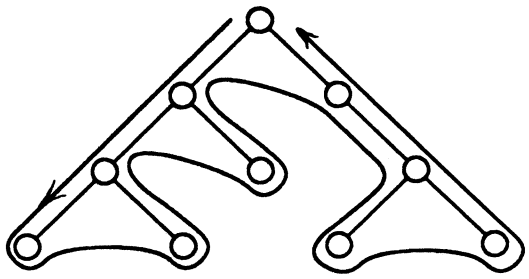

Figure 4: left-to-right tree traversal

The *bucket brigade* operator of regular-right part attribute grammars (Jullig and DeRemer, 1984) was introduced for this purpose. The following sets of patterns and templates 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 by 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 patterns 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 patterns 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. Patterns and templates conveniently express this abstract grouping: attributes are assigned to concrete syntax elements using templates and patterns containing stropped symbols; abstract attribute computations are expressed in terms of unstropped patterns and templates – the binding of these computations to the concrete syntax is controlled by the availability of attribute values. That is, the set of tentatively defined 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 may be computed from concrete patterns, or 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.

Experience. We have written MAGs to perform semantic analysis for the declaration statements of Pascal. Our implementation uses a 158 line context-free grammar and 12 modules comprising 101 patterns and templates. The generated attribute grammar contains 525 attribute definitions that require 50 unique procedures, given the optimizations described in the implementation section.

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 patterns to recognise syntactic constructs that uniquely identify semantic constructs – syntactic recognition is enhanced by availability of tentatively defined 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 patterns 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 easy as we would like. Perhaps due to unfamiliarity 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 work exclusively at the MAG source level. This aim is partly addressed with the use of a strict methodology like that outlined above.

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 expressions. 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 are 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 *tentative definition* 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 patterns 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 addressed by a more general solution. We find many modules, not necessarily concerned solely with attribute propagation by copy rule, that share a similar overall appearance. A possible solution is to provide generic modules, parameterized 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

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

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

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

Ganzinger, H., and Giegerich, R., Attribute Coupled Grammars, *Proc. SIGPLAN '84 Symposium on Compiler Construction*, 157–170.

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

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

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

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

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

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

Keller, S. E., Perkins, J. A., Payton, T. F., and Mardinly, S. P., Tree transformation techniques and experiences, *Proc. SIGPLAN '84 Symposium on Compiler Construction*, 190–201.

Kernighan, B. W. and Ritchie, D. M., *The C programming language*, Prentice-Hall.

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

Koskimies, K., Räihä, K-J., and Sarjakoski, M., Compiler construction using attribute grammars, *Proc. SIGPLAN '82 Symposium on Compiler Construction*, 153–159.

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

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

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

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

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

Watt, D. A. and Madsen, O. L., Extended attribute grammars, *Comp. Journal* **26**, 2 (1982), 142–153.

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