Modularity and Reusability in Attribute Grammars

U. Kastens and W. M. Waite

CU-CS-613-92 September 1992

University of Colorado at Boulder

Technical Report CU-CS-613-92
Department of Computer Science
Campus Box 430
University of Colorado
Boulder, Colorado 80309

Modularity and Reusability in Attribute Grammars

U. Kastens and W. M. Waite

September 1992

Abstract

An attribute grammar is a declarative specification of dependence among computations carried out at the nodes of a tree. Attribute grammars have proven remarkably difficult to decompose into logical fragments. As a result, they have not yet been accepted as a viable specification technique. By combining the ideas of remote attribute access and inheritance, we have been able to define "attribution modules" that can be reused in a variety of applications. As an example, we show how to define reusable modules for name analysis that embody different scope rules.

1 Introduction

A fairly standard decomposition of the compiler construction task has evolved over the past twenty years, and most compilers have very similar structures. Many of the subproblems defined by the standard decomposition can be described as computations over trees, in which information is attached to individual tree nodes and used to control various decisions.[22] For example, consider the subproblem of determining the meaning of an operator that appears in an expression. The Pascal statement "$A := B + C$;" might be represented internally by the tree fragment shown in Figure 1a. Here each node is classified as a *Statement*, *Variable*, *Expression*, *Operator* or *Identifier*. Because there are different ways to construct nodes of each of these classes, the specific rule used is given in parentheses below the node class. (The class can be deduced from the rule, so only an indication of the rule is physically stored in the tree.)

According to the definition of Pascal, the meaning of the the dyadic expression's operator might be integer addition, real addition or set union. In order to determine the meaning, the compiler might *decorate* the tree by attaching additional information to the nodes as shown in Figure 1b. The decoration `AddReal` would then be used in the decision to emit (say) a floating-point add instruction as the translation of the dyadic expression node.

An algorithm for decorating a tree has three distinct components:

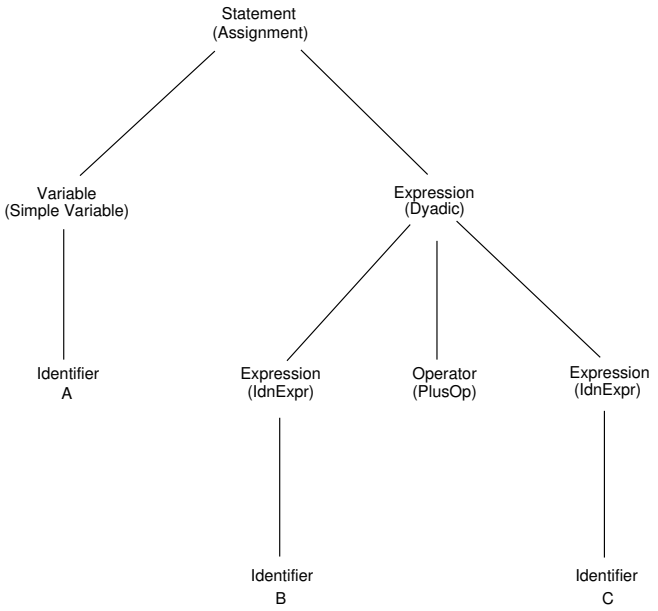
Computations The actual computations that result in individual decorations.

Traversal The order in which the computations must be carried out at the various nodes of the tree.

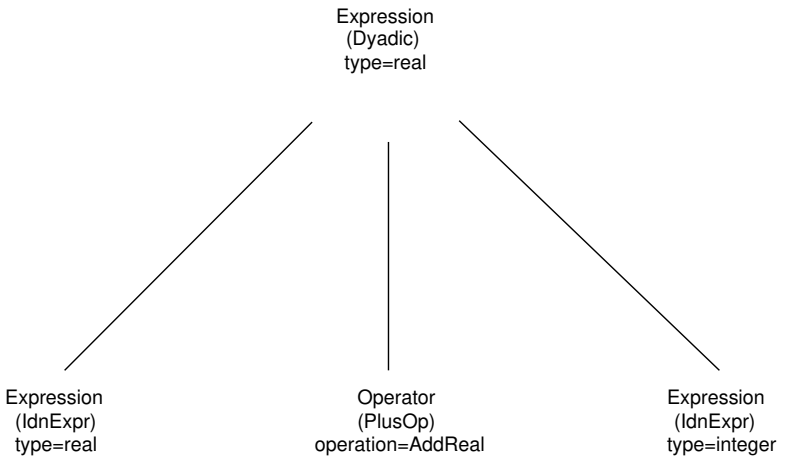
Storage How intermediate results are to be stored during the decoration.

In Figure 1b, a computation determines that the operator is `AddReal`, given the fact that the indication is `plus` and the types of the children are both `real`. The traversal guarantees that this computation takes place after the computations of the indication and the types of the children. If the indication is not used in any other computation, it need not actually be stored in the tree node; the storage component of the algorithm might provide a single global variable that could hold all indications (because their lifetimes would not overlap).

Under certain rather broad conditions, it is possible to mechanically derive both the traversal and the storage components of a decoration algorithm from the dependences among the computations.[10,11] An *attribute grammar* is a specification of computations and dependence based on a formal calculus introduced by Knuth.[16] By specifying only an attribute grammar, and then allowing some tool to derive the decoration algorithm, a designer avoids



a. A Pascal tree fragment



b. The dyadic expression decorated with type information

Figure 1: A Tree Decoration Problem

the need to think about tree traversal strategies and ways to store values used temporarily during decoration. The designer's task is thereby simplified. Despite this advantage, however, attribute grammars have not come into general use as a compiler specification method. We assert that the main reason for their lack of popularity has been the style in which they are normally written – a style that hinders both modular decomposition and reuse of specifications.

There are several different notations for describing attribute grammars.[1] Section 2 reviews the formal calculus and illustrates the basic notation used to describe computations and dependence. Experience has shown that this notation must be augmented in order to be suitable for practical applications of attribute grammar specifications. In Section 3 we discuss higher-level constructs that can be mapped to the basic constructs of the calculus, and in Section 4 we apply them to construct modules for a common tree decoration problem.

2 Basic Attibute Grammar Concepts

An attribute grammar is a quadruple AG = (G, A, R, B), where

$$
\begin{aligned}
G &= (T,\ N,\ P,\ Z) \text{ is a reduced context-free grammar,} \\
A &= \bigcup_{X \in T \cup N} A(X) \text{ is a finite set of attributes} \\
R &= \bigcup_{p \in P} R(p) \text{ is a finite set of attribute computations} \\
B &= \bigcup_{p \in P} B(p) \text{ is a finite set of plain computations}
\end{aligned}
$$

$A(X) \cap A(Y) \neq \emptyset$ implies $X = Y$. For each occurrence of X in the derivation of a sentence of $L(G)$, at most one attribute computation is applicable for the computation of each attribute $a \in A(X)$. For each application of a production p in the derivation of a sentence of $L(G)$, all computations of $R(p)$ and $B(p)$ are carried out.

The context-free grammar defines the structure of the tree being decorated. Each production describes a *context* consisting of a node and its children (if any). (Figure 1b is an example of such a context.) Attribute values decorate the nodes, attribute computations specify how those values are related, and plain computations extract information for other processes.

```
SYMBOL Expr: type: DefTableKey;
RULE Dyadic: Expr ::= Expr Op Expr
COMPUTE
   Expr[1].type = SignatureElt(Op.operation,0);
   Op.operation =
      Identify2(Op.indication,Expr[2].type,Expr[3].type);
END;

SYMBOL Op: indication: Delimiter, operation: DefTableKey;
RULE PlusOp: Op ::= '+'
COMPUTE
   Op.indication = plus;
   IF(NotValid(Op.operation)
      message("Invalid operand types for this operator"));
END;
```

Figure 2: Attribute Grammar Fragment Describing Figure 1

Figure 2 shows a fragment of an attribute grammar that describes the decorations of Figure 1b in a typical notation. `SYMBOL` constructs are used to specify the types of attributes in $A(X)$. Each `RULE` corresponds to one production of the reduced context-free grammar. Both the production name and the production itself are specified. (Note that the literal + does not actually appear in the tree; it is used only to aid the user in understanding the specification.) The elements of $R(p)$ (the attribute computations) and $B(p)$ (the plain computations) are given between the keywords `COMPUTE` and `END`. The order in which the computations are written down is totally irrelevant. Attribute computations describe relationships among attributes, *not* an algorithm for computing the values of those attributes.

Our definition is a slight generalization of those usually found in the literature, where the result of $B(p)$ is restricted to Boolean values.[10,23] It is important to emphasize that our generalization abstracts from the domains of attribute values and functions used in the computations, and thus models only the dependence among those computations: Each attribute effectively represents a postcondition reached after completion of the attribute computation of R defining that attribute. Attributes used in a computation of R or B effectively represent preconditions for beginning that computation. If the dependence relationships satisfy certain constraints, it is possible to mechanically derive a complete computation algorithm from them.[23] The derived algorithm is guaranteed to satisfy all pre- and postconditions.

It is important to understand the ramifications of our view of the formal calculus, because it broadens the class of computations in formal specifications.[12,13] Most treatments of attribute grammars use attributes solely for the propagation of values. They demand

that functions appearing in computations of R and B have no side effects, because such side effects cannot be reflected in the attribute values. By considering attributes to represent pre- and postconditions, however, side effects in component functions are easily accounted for.[6] Of course this generalized view of the calculus does not exclude attribute grammar specification languages that require a strictly value passing model; it does, however, allow systematic and efficient implementations of many language processing tasks that are quite tedious for such models. We shall provide examples to support this contention in Section 4.

3 Paradigm Shifts

The two most complete attribute grammars that have appeared in the literature are those for Pascal[25] and Ada.[20] Both of these grammars were developed for the GAG system,[25] a generator that accepts an ordered attribute grammar and produces a Pascal program to carry out attribute evaluation. The grammars are set in fixed-width typewriter font of normal size, with a reasonable but not excessive amount of white space; Pascal requires 63 pages, Ada 395 pages. In each case, the authors have taken pains to structure the specification for the human reader.

Since 1982, when the Pascal and Ada grammars were published, we have accumulated significant experience with attribute grammars as a specification mechanism. On the basis of our experience, we have developed a number of paradigms for notation and usage that simplify compiler descriptions. These paradigms have been employed successfully in a range of compilers built using Eli.[5]

The generalized view of the calculus underlying attribute grammars (described in Section 2) represents one fundamental paradigm shift; the others will be presented in this section. A complete example that shows how they enhance modular decomposition of an attribute grammar and provide reusable components is given in Section 4.

3.1 Remote Attribute Access

The formal calculus of Section 2 imposes the principle of locality in specifications: All of the preconditions and the postcondition for a computation must appear within the context of a single node and its children. Locality of specifications usually supports comprehensibility of attribute grammars, but sometimes one of the preconditions for a computation may be established by another computation that is far away in the tree. The calculus requires such a precondition to be established locally by computations provided in the intermediate

contexts. (This is the source of huge numbers of so called "copy rules" in attribute grammars formulated using only the notation of the underlying calculus.)

In most cases, remote dependence follows one of three simple patterns:

- A computation at the root of a subtree containing the local context establishes the precondition.
- The precondition is the union of the postconditions for some set of computations at nodes that are descendants of the subtree rooted in the local context.
- The dependence involves an invariant for some iterative computation visiting nodes in (depth-first) left-to-right order.

Each of the three patterns can be directly formulated by an attribute grammar specification construct, as shown below. Those constructs can be transformed into the notation of the underlying calculus by generating all the intermediate computations mechanically, so they do not alter the theoretical properties of the attribute grammar.

Direct formulation of remote dependence reduces the size of an attribute grammar, but an even more important characteristic is that it abstracts from the intermediate tree structure. It is invariant with respect to modifications of that structure, a requirement for reuse of attribute grammar modules from libraries. In the remainder of this subsection we give a simple example of each of the three remote dependence patterns.

As an example of a computation whose precondition is a computation at the root of a subtree containing the local context, consider the lexical nesting depth of a block. (The block nesting depth is used in forming addresses for variables declared within the block.[23]) Suppose that the nesting depth of a block is represented by an attribute `Depth` at the node corresponding to the block. The precondition for computation of the nesting depth is that a value be available for the nesting depth of the immediately enclosing block. If we assume that a block is one form of a statement, then this relationship might be expressed by the following attribute grammar rules:

```
RULE Program:   Root ::= Block COMPUTE
     Block.Depth = 0;
END;
RULE Inner:   Statement ::= Block COMPUTE
     Block.Depth = ADD(INCLUDING Block.Depth, 1);
END;
```

Here the keyword `INCLUDING` indicates that the precondition is the first instance of the specified attribute found when walking up the tree from the current node. Thus the first argument of `ADD` is the nesting depth of the immediately enclosing block. If the single attribute name in this example is replaced by a parenthesized list, the precondition is the first element of that list encountered when walking up the tree from the current node.

A precondition that is the union of the postconditions for some set of computations at descendant nodes is illustrated by the problem of determining the cost of each statement in a program. Suppose that each operator has an associated cost (e.g. the length of the code generated for that operator), and that the cost of a statement is the sum of the costs of its component operators. If the cost of any operator or statement is represented by an attribute `Cost` at nodes corresponding to operators or statements, then the precondition for computation of the cost of a statement is that values be available for the costs of all of its component operators:

```
RULE Calculation:  Statement ::= Expression COMPUTE
    Statement.Cost = CONSTITUENTS Operator.Cost
                  WITH (int, ADD, IDENTICAL, ZERO);
END;
```

Here the keyword `CONSTITUENTS` indicates that the precondition is the union of the instances of the given attribute appearing in the subtree rooted in the current node. Thus the cost of the statement depends on the union of the costs of the operators used in that statement. If the single attribute name in this example is replaced by a parenthesized list, the precondition is the union of all instances of all attributes on the list appearing in the subtree rooted in the current node.

In this example, `CONSTITUENTS` must yield a value: the sum of the operator costs. The `WITH` clause specifies the type of the "union" value (`int`) and names three user-defined functions used in its computation. The first function (`ADD`) combines two "union" values to yield a "union" value, the second (`IDENTICAL`) creates a "union" value from an `Operator.Cost` value, and the third (`ZERO`) creates a "union" value from nothing.

An invariant for a left-to-right iteration is illustrated by the computation of enumerated constant values in Pascal. Pascal enumeration constants are defined by a list of identifiers, and the value represented by each identifier in the list is just the number of identifiers preceding it. Thus the first identifier represents 0 because there are no identifiers to its left, the second represents 1, and so on.

A user describes an invariant by declaring a special kind of attribute called a "chained attribute", or simply a "chain". The invariant is established initially by a `CHAINSTART`

directive at the root of some subtree. Computations updating the invariant may be associated with any node in that subtree.

If `Count` represents the number of identifiers to the left of the current enumerated constant, the appropriate computations are expressed by the following rules:

```
CHAIN Count: int;
RULE Enumeration:  Type ::= '(' EnumConstList ')'  COMPUTE
   CHAINSTART EnumConstList.Count = 0;
END;
RULE ConstListElt: EnumConst ::= Identifier COMPUTE
   EnumConst.Count = ADD(EnumConst.Count, 1);
END;
```

Invariants can be expressed in terms of the underlying calculus by using two attributes per node. One is an inherited attribute representing the precondition for computations in the context of that node and its children, and the other is a synthesized attribute representing the postcondition established by those computations. Each nonterminal node in the subtree has such an attribute pair for each distinct invariant.

3.2 Symbol Computations

The block nesting depth computation discussed in Section 3.1 had two interesting properties:

- It only depended on attributes of a single symbol or on remote attributes.
- It should be applied at (almost) every instance of that symbol in the tree.

When a computation has these properties, it can be associated with the symbol itself rather than with each of the different contexts in which the symbol appears.

In Section 3.1, we assumed that a program was a block, and that a block was one kind of statement. One computation (setting the nesting depth to 0) was associated with the former context and another (incrementing the nesting depth) was associated with the latter. Suppose that, as in ALGOL 60, one possible implementation of a procedure body were also a block. This would result in two rules with identical computations:

```
RULE Inner:   Statement ::= Block COMPUTE
```

```
     Block.Depth = ADD(INCLUDING Block.Depth, 1);
END;
RULE Proc:   Body ::= Block COMPUTE
     Block.Depth = ADD(INCLUDING Block.Depth, 1);
END;
```

It would then pay to associate the nesting depth increment with the symbol `Block` instead of each context in which a block appears:

```
SYMBOL Block COMPUTE
   INH.Depth = ADD(INCLUDING Block.Depth, 1);
END;
```

Here the keyword `INH` indicates that `Depth` is an inherited attribute and that its computation has to be associated with each context having `Block` on the right-hand side. The computation of a synthesized `Block.i` would be denoted `SYNT.i` and would be associated with each context having `Block` on the left-hand side. A computation that does not define an attribute is associated with each context having its symbol on the left-hand side. There can be several computations of either class in a single symbol attribution.

The nesting depth increment must be replaced by a nesting depth initialization when the block is the entire program:

```
RULE Program:   Root ::= Block COMPUTE
     Block.Depth = 0;
END;
```

Here the computation given explicitly in the rule overrides the computation introduced implicitly by the use of `Block` on the right-hand side.

3.3 Inheritance

Several syntactic constructs of a language often share some set of semantic properties. For example, in ALGOL 68 there are three constructs that serve as scopes for identifier declarations: programs, serial clauses and procedures.[24] To simplify the description of the semantics, ALGOL 68's designers introduced an additional symbol, "range", defined in the grammar as representing all of these three constructs. "Range" could not be derived from

the axiom of the grammar, so there was never any phrase corresponding to that symbol in a program; it was used only in descriptions of the semantics to avoid having to make the same assertions about scope rules for identifiers in programs, serial clauses and procedures. The technique of inheritance allows one to obtain the same benefits in an attribute grammar.

The block nesting depth we have been discussing is used to access objects stored in a particular activation record at execution time. In ALGOL 60, distinct activation records may be associated with each procedure and each block. Thus procedures and blocks share the semantic property of being associated with distinct activation records at execution time. To simplify the description of this semantic property, we might introduce an additional symbol, `Contour`, representing it:

```
SYMBOL Contour COMPUTE
   INH.Depth = ADD(INCLUDING Contour.Depth, 1);
END;
```

Note that this computation is now completely independent of the symbols used in a particular language definition to denote constructs associated with distinct activation records. It abstracts the semantic concept of activation records addressed at execution time via a static chain.[23]

An abstract computation can be inherited by some set of symbols representing nodes in the tree:

```
SYMBOL Block INHERITS Contour END;
SYMBOL Procedure INHERITS Contour END;
```

Each symbol may also inherit computations from several other symbols, possibly through several levels of inheritance. Inherited computations may be overridden by including explicit computations in `SYMBOL` declarations as well as by including them in `RULE` declarations.[14]

3.4 Cumulative Attribution

Cumulative attribution is a simple and effective notational technique for decomposition of large attribute grammar specifications: The user is allowed to write an arbitrary number of rules with the same production. Since each context in a tree is uniquely defined by a single production in a reduced context-free grammar, the total set of computations for that context is the union of the sets of computations described by each rule with that production.

This makes it possible to decompose a specification into components, each covering one aspect of the total problem. For example, a compiler writer might provide separate descriptions of name analysis, overload resolution and translation into a target representation. Each of these tasks can be described by a few computations in a small number of contexts, using the technique of remote attribute access discussed above.

The technique can also be applied to symbol computations, allowing an arbitrary number of specifications describing different computations for the same symbol.

A physical decomposition of an attribute grammar can be easily achieved by storing each component in a distinct file. The input to the attribute grammar processor is then a set of files, which cumulative attribution merges into a single specification in the underlying calculus. Some of those files might belong to libraries of reusable specification modules, each of which describes a single aspect of the problem to be solved. Examples of such specification modules are given in Section 4.

4 Attribution Modules for Scope Rules

Consistent renaming,[23] the task of determining which source language entity is denoted by each identifier occurrence in a program, is defined by the scope rules of the source language. In this section we show how the consistent renaming task can be embodied in scope-rule specific attribution modules that use a scope-rule independent module[15] for their computations. When developing an attribute grammar for a new language, an appropriate one of these attribution modules can easily be used to implement consistent renaming.

Consistent renaming is often merged with the solution of other compiler subproblems, such as that of converting symbol strings to an internal representation and that of associating properties with defined entities. Such solutions cannot easily be reused in other compiler implementations. Our approach separates consistent renaming from other tasks by providing distinct modules for each.

Source language entities are described by sets of *properties* (e.g. the lexical level of a variable, the value of a constant, or the amount of storage required for objects of a type). Each entity can be characterized by a unique *key* that allows the compiler to access these properties. The consistent renaming task therefore determines the appropriate key for each identifier occurrence in the program: It effectively changes the programmer names (identifiers) to compiler names (keys). Access to the properties of an entity are provided by a different module, which is not concerned with relationships between identifiers and keys.

Computations for consistent renaming can be defined in terms of three abstract data types:

Symbol The compiler's internal representation of an identifier.

Environment The compiler's internal representation of a scope.

DefTableKey The compiler's internal representation of a key for a program entity.

We assume that the `Identifier`-class leaves of the tree have been decorated with appropriate `Symbol` values when the tree representing a program is built, and therefore the computation of `Symbol` values is beyond the scope of this paper.[23]

The computations associated with the `Environment` abstract data type can be provided efficiently by a module that is independent of any specific scope rules.[15] Figure 3 defines the part of the module interface that is relevant for this paper.

No operations of the `DefTableKey` abstract data type are used explicitly during consistent renaming (the `DefineIdn` operation of the `Environment` abstract data type obtains `DefTableKey` values when needed). Therefore we will not consider details of the `DefTableKey` abstract data type in this paper.

4.1 ALGOL 60 Scope Rules

ALGOL 60[19] was the first language that defined scope rules for definitions in nested ranges. The basic concept can be summarized by the following rule:

> A definition of an identifier `a` is visible in the smallest enclosing range with the exception of inner ranges that also contain a definition of `a`.

Hence, an applied occurrence of an identifier `a` identifies a definition of `a` in the smallest enclosing range. Definitions of `a` in outer ranges are hidden. Figure 4 states this concept in terms of symbol attributes.

Four symbols are used in Figure 4 to describe the syntactic constructs involved in the scope rules:

Range A region containing definitions

NewEnv() Creates and returns a new `Environment` value representing an outermost scope.

NewScope(e) Creates and returns a new `Environment` value representing a scope nested within the scope represented by `e`.

DefineIdn(e,i) If the `Symbol` value `i` is not associated with a `DefTableKey` value in the scope represented by `e`, then a new `DefTableKey` value is obtained and associated with `i` in `e`. The `DefTableKey` value associated with `i` in `e` is returned.

AddIdn(e,i,k) If the `Symbol` value `i` is not associated with a `DefTableKey` value in the scope represented by `e`, then the `DefTableKey` value `k` is associated with `i` in `e` and the Boolean value `true` is returned. Otherwise no associations are changed and the Boolean value `false` is returned.

KeyInEnv(e,i) If the `Symbol` value `i` is associated with a `DefTableKey` value in the scope represented by `e` then that value is returned. Otherwise, if `e` represents the outermost scope then the distinguished `DefTableKey` value `NoKey` is returned. Otherwise the value of `KeyInEnv(e',i)`, where `e'` is the `Environment` value representing the scope immediately enclosing `e`, is returned.

KeyInScope(e,i) If the `Symbol` value `i` is associated with a `DefTableKey` value in the scope represented by `e` then that value is returned. Otherwise the distinguished `DefTableKey` value `NoKey` is returned.

Figure 3: Operations of the Standard Environment Module

```
SYMBOL Range: Env: Environment, GotLocalKeys, GotAllKeys: VOID;
SYMBOL IdDef, IdUse: Sym: int, Key: DefTableKey;

SYMBOL Root INHERITS Range COMPUTE
  INH.Env = NewEnv();
  INH.GotAllKeys = THIS.GotLocalKeys;
END;

SYMBOL Range COMPUTE
  INH.Env = NewScope(INCLUDING Range.Env);
  INH.GotAllKeys = THIS.GotLocalKeys
    DEPENDS_ON INCLUDING Range.GotAllKeys;
  SYNT.GotLocalKeys = CONSTITUENTS IdDef.Key;
END;

SYMBOL IdDef COMPUTE
  SYNT.Key = DefineIdn(INCLUDING Range.Env, THIS.Sym);
END;

SYMBOL IdUse COMPUTE
  SYNT.Key = KeyInEnv(INCLUDING Range.Env, THIS.Sym)
    DEPENDS_ON INCLUDING Range.GotAllKeys;
END;
```

Figure 4: An Attribution Module for ALGOL 60-like Scope Rules

Root The outermost region containing definitions

IdDef A defining occurrence of an identifier

IdUse An applied occurrence of an identifier

`Key` is the definition table key associated with an identifier occurrence, and also represents the condition "this identifier occurrence is associated with a definition table key". `CONSTITUENTS IdDef.Key` therefore represents the condition "all defining occurrences in the subtree rooted here are associated with definition table keys". Because the `CONSTITUENTS` construct computes a synthesized attribute of the symbol `Range`, however, the set over which the union is taken does not include any occurrences of `IdDef` lying inside subtrees that are themselves rooted in `Range` nodes.[12] The attribute `GotLocalKeys` therefore represents the condition "all defining occurrences in this range are associated with definition table keys".

"All defining occurrences in the current range and all enclosing ranges are associated with definition table keys" is a precondition for obtaining the key associated with an applied occurrence. The attribute `GotAllKeys` represents that precondition: Its computation combines "all defining occurrences in enclosing ranges are associated with definition table keys" with "all defining occurrences in the current range are associated with definition table keys".

Figure 4 is completely independent of the abstract syntax of a particular language. In order to use it as a module for a particular language analysis, its symbols must be mapped to phrases of an abstract syntax for that language. For example, suppose that an abstract syntax for ALGOL 60 were to be developed from the grammar of the Revised Report.[19] To use the module, we need distinct symbols corresponding to defining and applied occurrences of identifiers. Unfortunately, the Revised Report does not make this distinction. Instead, all occurrences of identifiers in rules from Section 5 of the report are defining occurrences, while occurrences of identifiers in rules from other sections of the report are applied occurrences.

To solve this problem, define two new symbols for each symbol that represents an identifier in the original grammar. Replace all instances of the original symbol that represent defining occurrences with one of the new symbols, and all instances of the original symbol that represent applied occurrences with the other. (For example, `VarIdDef` and `VarIdUse` might be defined for the symbol `variable_identifier`.) The result of this modification is to move the specification of defining and applied occurrences from the text of the report to the abstract grammar.

The module of Figure 4 is linked to the modified abstract syntax by providing a "linkage" specification that resides in a distinct file:

```
SYMBOL VarIdDef INHERITS IdDef END;
```

```
SYMBOL VarIdUse INHERITS IdUse END;
...
```

Additional restructuring of the abstract grammar is needed to deal with the `Range` symbol. An ALGOL 60 `block` symbol represents a phrase that has exactly the semantics of a `Range`, but the `procedure_declaration` does not. Here is one of the productions for a `procedure_declaration`:

```
procedure_declaration:
  'procedure' procedure_heading procedure_body
```

The problem is that the `procedure_heading` phrase contains both the procedure identifier (which is a defining occurrence in the `Range` containing the procedure declaration) and the formal parameters (which are defining occurrences in the `Range` constituting the procedure declaration). In order to place the identifier occurrences in their proper ranges, define a new symbol `ProcRange` to inherit the computations of a `Range`, and alter the abstract syntax to:

```
procedure_declaration: 'procedure' ProcIdDef ProcRange
ProcRange: formal_parameter_part ';'
  value_part specification_part procedure_body
```

This formulation separates the procedure identifier from the formal parameters, placing them in distinct phrases. The proper association is then obtained by including two lines in the linkage specification:

```
SYMBOL ProcIdDef INHERITS IdDef END;
SYMBOL ProcRange INHERITS Range END;
```

4.2 C Scope Rules

The basic scope rule concept of C deviates from that of ALGOL 60:

> A definition of an identifier `a` is visible from its definition point up to the end of the smallest enclosing range with the exception of inner ranges that also contain a definition of `a`.

To see the effect of this rule, consider the following example:

```
{   int  a = 1;
    {    int b = a;
         float a;
         a = b / 3;
    }
}
```

Here `b` is initialized with the value of the `int` variable `a`. The last statement assigns to the `float` variable `a`. ALGOL 60 scope rules would cause an undefined result because `b` would be initialized to the (uninitialized) value of the `float` variable `a`.

C scope rules are most easily implemented by an iterative process whose invariant is "all identifier occurrences preceding this point are associated with definition table keys", as shown in Figure 5. This invariant is represented by the chain `Env`, which also provides access to the current environment. The `CHAINSTART` computation will be introduced into each context defined by a production with `Root` or `Range` on the left-hand side. `HEAD` denotes the first symbol on the right-hand side of that production.

A new environment is established within each subtree rooted in a `Root` or `Range` symbol. In the case of a `Range` symbol, the new environment cannot be established within the subtree until the value of the `Env` environment is available at that `Range` node. Establishment of the new environment within the subtree does not affect the value of the environment made available to nodes following the `Range` node in the iteration, but all possible identifier uses within that `Range` must be identified before any definitions following the `Range` are entered into the environment.

The invariant of the iteration must be re-established at each identifier occurrence. This is done by associating a definition table key with the identifier occurrence, and making the invariant depend upon that key. Defining occurrences are distinguished from applied occurrences by the function invoked.

The module of Figure 5 defines the same symbols as the module of Figure 4, and interacts with the abstract syntax in the same way.

4.3 Modula-2 Scope Rules

Modula-2 follows ALGOL 60 scope rules for nested procedures (Figure 6), but also uses the concept of a *module* that may have *import* and *export* lists. A module is a phrase that may occur within a procedure declaration. Normally, none of the identifiers visible in the procedure body are visible in the module, and none of the identifiers declared in the module

```
CHAIN Env: Environment;
SYMBOL IdDef, IdUse: Sym: int, Key: DefTableKey;

SYMBOL Root COMPUTE
  CHAINSTART HEAD.Env = NewEnv();
END;

SYMBOL Range COMPUTE
  CHAINSTART HEAD.Env = NewScope(THIS.Env);
  THIS.Env = THIS.Env DEPENDS_ON TAIL.Env;
END;

SYMBOL IdDef COMPUTE
  SYNT.Key = DefineIdn(THIS.Env, THIS.Sym);
  THIS.Env = THIS.Env DEPENDS_ON THIS.Key;
END;

SYMBOL IdUse COMPUTE
  SYNT.Key = KeyInEnv(THIS.Env, THIS.Sym);
  THIS.Env = THIS.Env DEPENDS_ON THIS.Key;
END;
```

Figure 5: An Attribution Module for C-like Scope Rules

```
SYMBOL Range: Env: Environment, GotLocalKeys, GotAllKeys: VOID;
SYMBOL IdDef, IdUse: Sym: int, Key: DefTableKey;

SYMBOL Root INHERITS Range COMPUTE
  INH.Env = NewEnv();
  INH.GotAllKeys = THIS.GotLocalKeys;
END;

SYMBOL Range COMPUTE
  INH.Env = NewScope(INCLUDING Range.Env);
  INH.GotAllKeys = THIS.GotLocalKeys
    DEPENDS_ON INCLUDING Range.GotAllKeys;
  SYNT.GotLocalKeys = CONSTITUENTS (IdDef.Key, Module.Exports);
END;

SYMBOL IdDef COMPUTE
  SYNT.Key = DefineIdn(INCLUDING Range.Env, THIS.Sym);
END;

SYMBOL IdUse COMPUTE
  SYNT.Key = KeyInEnv(INCLUDING Range.Env, THIS.Sym)
    DEPENDS_ON INCLUDING Range.GotAllKeys;
END;
```

Figure 6: An Attribution Module for Modula-2 Ranges

are visible in the procedure body. If an identifier visible in the procedure body occurs on an import list in the module, however, that identifier becomes visible in the module. Similarly, if an identifier declared in the module occurs on an export list in the module, that identifier becomes visible in the procedure body. This last point gives rise to the only difference between Figure 6 and Figure 4: The inclusion of symbols exported by local modules as locally-defined symbols in the surrounding range.

Figure 7 defines the behavior of a module in Modula-2. By overriding the computation of the environment, Figure 7 makes identifiers from the surrounding environment invisible unless they are explicitly imported. (`GotAllKeys`, the condition "all defining occurrences in the current range and all enclosing ranges are associated with definition table keys", is also overridden to ensure that the necessary identifiers have been imported.)

The computations for occurrences of identifiers on the import and export lists make definitions in one scope visible in another. An exported identifier must be defined in the

```
SYMBOL Module: Surround: Environment, Exports, GotAllSurroundingKeys: VOID;

SYMBOL Module INHERITS Range COMPUTE
  INH.Env = NewScope(INCLUDING Root.Env);
  INH.GotAllKeys = THIS.GotLocalKeys
    DEPENDS_ON
      CONSTITUENTS ImportedId.Imported;
  INH.Surround = INCLUDING Range.Env;
  SYNT.Exports = CONSTITUENTS ExportedId.Exported;
  INH.GotAllSurroundingKeys = INCLUDING Range.GotAllKeys;
END;


SYMBOL ExportedId: Exported, Sym: int, Key: DefTableKey;
SYMBOL ImportedId: Imported, Sym: int, Key: DefTableKey;

SYMBOL ExportedId COMPUTE
  SYNT.Key = KeyInScope(INCLUDING Range.Env, THIS.Sym)
    DEPENDS_ON INCLUDING Range.GotLocalKeys;
  SYNT.Exported =
    AddIdn(INCLUDING Module.Surround, THIS.Sym, THIS.Key);
END;

SYMBOL ImportedId COMPUTE
  SYNT.Key = KeyInEnv(INCLUDING Module.Surround, THIS.Sym)
    DEPENDS_ON INCLUDING Module.GotAllSurroundingKeys;
  SYNT.Imported =
    AddIdn(INCLUDING Range.Env, THIS.Sym, THIS.Key);
END;
```

Figure 7: Modules, Export and Import in Modula-2

scope from which it is being exported, so `KeyInScope` is used to obtain the key. Once the key is available, `AddIdn` associates that key with the identifier in the surrounding environment. `KeyInEnv` is used to obtain the key for an imported identifier, since any identifier valid in the surrounding scope can be legally imported.

5 Related Work

The techniques described in this paper result from fifteen years' experience using attribute grammars for compiler construction. All of them are implemented by LIGA,[12] one of the tools integrated into the Eli compiler construction system.[5] This section gives a brief summary of their origins, and compares them with other approaches to achieving modularity and reusability in attribute grammars.

Our first attribute grammar processor, GAG,[25] was a direct implementation of the fundamental calculus described in Section 2. The idea of abstracting from the domain of attribute values and using attribute dependence to control side effects of computations was introduced in 1982,[21] and led to the initial design of LIGA in 1986. Remote attribute access using `INCLUDING` and `CONSTITUENTS` was first introduced in GAG. The importance of chain attribution was recognized early, but special notation was used only in very restricted situations (e.g. the "bucket brigade"[9]). `CHAIN` and cumulative attribution were introduced into LIGA in 1990;[12] symbol computations and inheritance appear for the first time in this paper.

We have emphasized that our approach supports decomposition of an attribute grammar into specifications of the individual tasks to be carried out. Other approaches reported in the literature decompose the grammar along structural lines, independent of function. For example, in the decompositions based on object-oriented extensions [7,8,17,18] inheritance of attribution is tied to chain productions in the underlying concrete or abstract syntax. This means that computations can be reused only in contexts that are immediate descendants in the tree, or in which certain substructures coincide. Our mechanism, on the other hand, abstracts from the stucture so that computations can be reused with almost any abstract syntax. (Hedin's abstract productions[8] have an effect similar to our remote attribute access.)

We advocate association of computations with symbols, but reuse can also be achieved by associating computations with "rule patterns".[2] Unfortunately this approach does not support hierarchical organization of the specifications, and it needs rather complicated overriding rules to describe the effect of patterns which usually overlap.

All the techniques mentioned so far map the specifications to basic concepts of attribute grammars as our approach does. Any of the well-known implementation techniques for attribute evaluators can therefore be applied to generate code from them. Even though the

original specification is modular, the final evaluator is monolithic. Another approach introduces a new evaluation model that decomposes the evaluator according to the specification structure.[3] Reusability of such modules is restricted to grammars that match in their corresponding parts. The components interact at 1-visit boundaries, a hard restriction for the specification of language properties like scope rules that involve several subgrammars and often need more than one visit.

Finally it should be mentioned that a modular decomposition of attribute grammars was suggested by attribute coupled grammars.[4]. That approach is aimed primarily at the specifications of attributed tree transformation decomposed into sequences of transformation phases: Each module operates on a specific tree computed by the preceding phase. Modules can thus only be reused for specifications that have the same tree structure.

6 Conclusion

Attribute grammars provide a notation for specifying relationships among computations on a tree. The designer describes only individual computations, and a tool deduces both a traversal strategy and a strategy for storing computed values. Since the designer understands and maintains the specification, not the algorithm, the tool is free to use any implementation satisfying the given relationships. Logically distinct computations may be interleaved by the tool to minimize the number of traversals and the amount of temporary storage. Thus the tool may be able to significantly improve performance over an algorithm for the same computations that is understandable and maintainable.

We have given an interpretation of the fundamental calculus on which attribute grammars are based that abstracts from the value domain of the computations, providing control of computations with side effects. Using this interpretation, we have characterized patterns of dependence that allow us to abstract from the details of the tree structure as well. This abstraction permits us to construct modules defining abstract semantics that are applicable to a variety of situations. We illustrated the process with specifications of consistent renaming semantics for several languages.

Descriptions like those in Section 4 are formal, yet easily understood. Proofs of their correctness can be constructed by available means,[6] and they can be used to produce code without human intervention. Thus they can be used unchanged as the basis for both a language definition and a compiler. This means that the compiler is guaranteed to implement the language definition with respect to the feature described.

The Eli system[5] supports the techniques described in this paper, and has been used to construct compilers for a variety of languages. Tree decoration processes can also characterize

a number of problems in the processing of structured text and other applications. In each case, attribute grammars are useful in specifying the computation without having to provide details of traversal strategy and intermediate storage. Eli is capable of producing standalone modules that implement tree decoration algorithms for use as components of other systems. These modules are competitive in performance with hand-coded modules solving the same problems; if the decoration process has several logical parts, the generated modules are usually superior.

By enabling modularity and reusability of specifications, we belive that the techniques in this paper should overcome the disadvantages of classical attribute grammars, making their use as common as the use of context-free grammars. This should help to shorten the development time for new languages, and encourage the introduction of special problem-oriented languages to simplify repetitive programming tasks.

Acknowledgements

This work was partially supported by the US Army Research Office under grant DAAL03-92-G-0158, and by the Ministry of Science and Technology of Nordrhein-Westfalen, Germany, under grant IVA3-10701691.

7 References

1. Deransart, P., Jourdan, M. & Lorho, B., *Lect. Notes in Comp. Sci.* #**323**, Springer Verlag, New York–Heidelberg–Berlin, 1988.

2. Dueck, G. D. P. & Cormack, G. V., "Modular Attribute Grammars," *The Computer Journal* **33** (1990), 164–172.

3. Farrow, R., Marlowe, T. J. & Yellin, D. M., "Composable Attribute Grammars: Support for Modularity in Translator Design and Implementation," *Proc. of ACM Symposium on Principles of Programming Languages* (1992).

4. Ganzinger, H. & Giegerich, R., "Attribute Coupled Grammars," *SIGPLAN Notices* **19** (June 1984), 157–170.

5. Gray, R. W., Heuring, V. P., Levi, S. P., Sloane, A. M. & Waite, W. M., "Eli: A Complete, Flexible Compiler Construction System," *Communications of the ACM* **35** (Feb. 1992), 121–131.

6. Gries, D., *The Science of Programming*, Springer Verlag, 1981.

7. Grosch, J., "Ag - An Attribute Evaluator Generator," GMD Forschungsstelle Karlsruhe, Germany, Report 16, 1989.

8. Hedin, G., "An Object-Oriented Notation for Attribute Grammars," *Proc. of the European Conf. on Object-Oriented Programming, ECOOP'89* (1989).

9. Jullig, R. K. & DeRemer, F., "Regular Right-Part Attribute Grammars," *SIGPLAN Notices* **19** (June 1984), 171–178.

10. Kastens, U., "Ordered Attribute Grammars," *Acta Informatica* **13** (1980), 229–256.

11. Kastens, U., "Implementation of Visit-Oriented Attribute Evaluators," in *Proceedings of the International Summer School on Attribute Grammars, Application and Systems*, Lect. Notes in Comp. Sci. #**545**, Springer Verlag, New York–Heidelberg–Berlin, 1991, 114–139.

12. Kastens, U., "Attribute Grammars as a Specification Method," in *Proceedings of the International Summer School on Attribute Grammars, Application and Systems*, Lect. Notes in Comp. Sci. #**545**, Springer Verlag, New York–Heidelberg–Berlin, 1991, 16–47.

13. Kastens, U., "Attribute Grammars in a Compiler Construction Environment," in *Proceedings of the International Summer School on Attribute Grammars, Application and Systems*, Lect. Notes in Comp. Sci. #**545**, Springer Verlag, New York–Heidelberg–Berlin, 1991, 380–400.

14. Kastens, U., "LIDO - Short Reference," University of Paderborn, FRG, Documentation of the LIGA System, 1992.

15. Kastens, U. & Waite, W. M., "An Abstract Data Type for Name Analysis," *Acta Informatica* **28** (1991), 539–558.

16. Knuth, D. E., "Semantics of Context-Free Languages," *Mathematical Systems Theory* **2** (June 1968), 127–146.

17. Koskimies, K., "Object-Orientation in Attribute Grammars," University of Tampere, Finland, A-1991-1, April 1991.

18. Magnusson, B., Bengtsson, M., Dahlin, L. O., Fries, G., Gustavsson, A., Hedin, G., Minor, S., Oscarsson, D. & Taube, M., "An Overview of the Mjolner/Orm Environment: Incremental Language and Software Development," Lund University, Report LU-CS-TR:90:57, 1990.

19. "Revised Report on the Algorithmic Language ALGOL 60," *Communications of the ACM* **6** (Jan. 1963), 1–17.

20. Uhl, J., Drossopoulou, S., Persch, G., Goos, G., Dausmann, M., Winterstein, G. & Kirchgässner, W., *An Attribute Grammar for the Semantic Analysis of ADA*, Lect. Notes in Comp. Sci. #**139**, Springer Verlag, Berlin, 1982.

21. Waite, W. M., "Generator for Attribute Grammars - Abstract Data Type," Gesellschaft für Mathematik und Datenverarbeitung, Arbeitspapier 219, Karlsruhe, BRD, Sept. 1986.

22. Waite, W. M., "Use of Attribute Grammars in Compiler Construction," in *Attribute Grammars and their Applications*, Pierre Deransart & Martin Jourdan, eds., Lect. Notes in Comp. Sci. #**41**, Springer Verlag, Berlin, 1990, 255–265.

23. Waite, W. M. & Goos, G., *Compiler Construction*, Springer Verlag, New York, NY, 1984.

24. Wijngaarden, A. van, Mailloux, B. J., Peck, J. E. L. & Koster, C. H. A., "Report on the Algorithmic Language ALGOL 68 ," *Numerische Mathematik* **14** (1969), 79–218 .

25. Zimmermann, E., Kastens, U. & Hutt, B., *GAG: A Practical Compiler Generator*, Lect. Notes in Comp. Sci. #**141**, Springer Verlag, Heidelberg, 1982.