# Composing Tree Attributions*

John Boyland† Susan L. Graham†

Computer Science Division – EECS, 571 Evans Hall
University of California, Berkeley, California 94720

## Abstract

Using the *simple tree attributions* described in this paper, attribute values can themselves be trees, enabling attribution to be used for tree transformations. Unlike higher-order attribute grammars, simple tree attributions have the property of *descriptional composition*, which allows a complex transformation to be built up from simpler ones, yet be executed efficiently. In contrast to other formalisms that admit descriptional composition, notably composable attribute grammars, simple tree attributions have the expressive power to handle remote references and recursive syntactic (tree-generating) functions, providing significantly more general forms of attribution and transformation.

## 1 Introduction

Many programming language tools operate on trees that represent a program fragment in some way. For example, early stages of a compiler may represent a program as an abstract syntax tree. Trees might also be used as a lower level intermediate representation. Usually the nodes of a tree are annotated with *attributes* or *attribute values*. For example, type or binding information might be associated with a node designating a variable. A sequence of target machine instructions might be associated with a subtree representing an expression or statement. Typically, attributes are both named and typed. Attribute values may be drawn from a wide variety of types, such as integers, strings, tuples, trees, or graphs.

An *attribution* is an association of the nodes of a tree with appropriately typed attribute values. Attributions can be determined programmatically, by code associated with tree construction algorithms such as parsers, or by more formal systems of attribution rules such as attribute grammars [12]. In attribute grammar formalisms, the rules for calculating attribute values are functionally specified and are associated with the productions of a context-free grammar. Other formal attribution methods vary the attribute grammar approach in two ways. The first is to associate attribution rules directly with the structure of the tree. For example, Farnum's *attribute pattern sets* [5] associate attribution rules with tree patterns. The second kind of variation is in the language in which the attribution rules are expressed. An example is Ballance's *logical constraint grammars* [1, 2]. Kahn's *natural semantics* [10] varies attribute grammars in both these ways.

Although attributions are often used simply to "decorate" the nodes of a tree, they can also be used to construct new trees, computing either a transformation within the same tree language or a translation to another tree language. In this case, the attribution rules are operations on trees that build new trees. A compiler may, for example, use attribution rules to generate intermediate code in tree form. It is possible to build up a sequence of transformation or translation steps by defining a set of attribution rules for each family of trees generated by an earlier attribution. For example, a compiler might use attribution rules to translate the intermediate code into machine code.

The contribution of this paper is to define a new tree attribution formalism for translations, called *simple tree attributions*, that admits *descriptional composition*; namely, the ability to take two attribution specifications, the second of which attributes the trees that

*This research was supported in part by a fellowship from the National Science Foundation to John Boyland, by the Advanced Research Projects Agency (DoD) under Contract N00039-88-C-0292 monitored by Space and Naval Warfare Systems Command, and under Grant MDA972-92-J-1028, and by the National Science Foundation under Infrastructure Grant CDA-8722788. The content of the information does not necessarily reflect the position or the policy of the Government.

†email address: {boyland,graham}@cs.berkeley.edu



POPL 94- 1/94, Portland Oregon,USA

© 1994 ACM 0-89791-636-0/94/001..$3.50

are outputs of the first attribution, and to compose the specifications into one new specification that translates from the inputs for the first attribution to the outputs of the second attribution. Descriptional composition enables a translation to be structured both for modularity and for efficiency. By specifying a tree transformation or a translation as a sequence of smaller "phases", each phase is doing a smaller job. As a result, the specification will be clearer and thus less likely to have major errors. In addition, it is often possible to re-use phases in different combinations. For example, compilers can share intermediate "optimizing" transformation phases, but still have separate target-specific code-generation phases. An important advantage of composition is the elimination of an explicit generation and traversal of the inter-phase representations. Once two translations are composed, the intermediate form that links them together has been eliminated.

There have been several earlier approaches to describing formally such a sequence of translations; among them are attribute coupled grammars [8, 9], higher-order attribute grammars [13, 14], and composable attribute grammars [6]. To achieve descriptional composition, both attribute coupled grammars and composable attribute grammars impose restrictions that limit the power of attributions. In the published reports on these approaches, a compiler written with these restrictions can not use an attribute value to determine a reordering of subtrees of a node, say for instruction scheduling. Neither can it generate multiple copies of a code segment, as in loop unrolling. Neither can it generate an intermediate code tree in which a use of a variable has a remote reference to the definition site in that tree.

Our method, an extension of composable attribute grammars, achieves descriptional composition while providing significantly enhanced expressive power. The added power comes from a combination of the use of conditionals to determine the structure of the tree, the use of computed values to determine the size of the tree, and the ability to access attributes of remote nodes. With this power, the examples mentioned previously can be handled.

The organization of the remaining sections is as follows. We first define simple tree attributions and the tree languages that go along with them, using an example drawn from compilation. We then present our composition algorithm, applying it to the example. Finally, we discuss the properties of our approach and relate it to other methods.

## 2 Simple Tree Attributions for Tree Languages

In attribute grammars, attribution rules are associated with productions of a context-free grammar. Correspondingly, in a simple tree attribution the attribution rules are associated with the kinds of nodes used in the tree. We use tree-based attribution rather than grammar-based attribution in order to get the full power of Farnum's attribute pattern sets[5]. However, in this paper we limit the discussion to specifications in which each kind of node is associated with rules for computing the attributes of that node and its immediate subtrees. There is a simple mapping between such a tree-based specification and a grammar-based specification.

### 2.1 Tree Languages

A *tree language*, namely, a set of trees, is defined by a set of *phyla*, and a set of *constructors*. The names of the constructors label the tree nodes. The constructors (and the node labels) are partitioned into subsets associated with the phyla defining the tree language (i.e., each constructor belongs to exactly one phylum).

A constructor for a node label is a function that creates a subtree rooted at a node with that label. Its parameter list specifies the ordered sequence of subtrees that constitute the children of the node and the values associated with the root node. Following Ganzinger and Giegerich [8], we call the subtree parameters *syntactic* parameters and the other parameters (including remote references to subtrees) *semantic* parameters. The set of trees in a tree language are those that can be built by repeated use of the constructors. In order that there be a base case for starting tree construction, there must always be at least one constructor that has no syntactic parameters. Such a constructor creates a leaf node. Constructors are invoked as ordinary functions.

Phyla play the role of nonterminals in a context-free grammar in the sense that phyla denote sets of subtrees, while nonterminals denote sets of substrings. Constructors play the role of grammar productions. Each phylum represents the set of all possible subtrees whose roots are labeled by a constructor associated with that phylum. By analogy with the "start symbol" or "root symbol" of a context-free grammar, one phylum of each tree language may be designated the *root phylum.* A *tree* in the language (as opposed to a subtree) is one in which the node label of its root node belongs to the root phylum. Some tree languages allow all defined subtrees in all contexts; in this case, there is one phylum and all node labels and constructors belong to it. More commonly in the case of abstract syntax trees, there is one phylum for each nonterminal in the abstract syntax.

#### Examples of tree languages

Figure 1 defines two tree languages, a simple arithmetic programming language and a simple low-level virtual machine language.

```
language SIMPLE_PROGRAM begin
  root phylum Program;
  phylum Expression;
  phylum Declaration;
  constructor program(e : Expression) : Program;
  constructor let(d : Declaration; val, body : Expression) : Expression;
  constructor def(name : String) : Declaration;
  constructor plus(e1,e2 : Expression) : Expression;
  constructor minus(e1,e2 : Expression) : Expression;
  constructor constant(i : Integer) : Expression;
  constructor use(d : remote Declaration) : Expression;
end;

language VMACHINE begin
  root phylum Instruction;
  phylum Register;
  constructor register(i : Integer) : Register;
  constructor fp() : Register;   -- frame pointer
  constructor loadi(reg : Register; i : Integer) : Instruction;
  constructor load(dest : Register; address : Register) : Instruction;
  constructor add(dest : Register; s1,s2 : Register) : Instruction;
  constructor addi(dest : Register; s : Register; i : Integer) : Instruction;
  constructor mov(dest : Register; source : Register) : Instruction;
  constructor seq(i1,i2 : Instruction) : Instruction;  -- sequencing
  constructor nop() : Instruction;
end;
```

Figure 1: Example Tree Languages

The first tree language, named SIMPLE_PROGRAM, has three phyla, Program, Expression and Declaration, and seven constructors. The important illustrative characteristics of the language are that the let constructor creates nested scopes and the use constructor creates a remote reference. The second tree language, named VMACHINE, has two phyla and nine constructors. The frame pointer fp will be used in the generated code to access nonlocal scopes.

Figure 2 shows a SIMPLE_PROGRAM tree. The arrows from the use's are remote references; semantic parameters to constructors appear below the node labels.

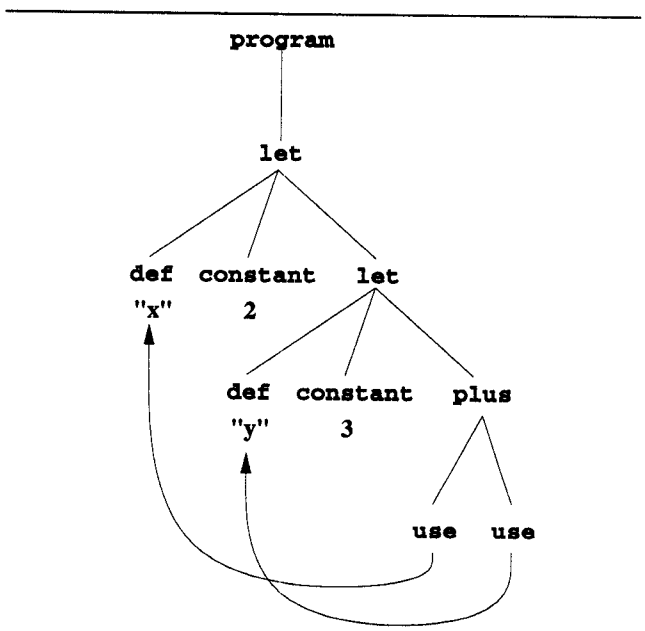


Figure 2: A SIMPLE_PROGRAM tree

## 2.2 Simple Tree Attributions

A *simple tree attribution* is defined over a *source* tree language. For each phylum of the tree language, the tree attribution declares zero or more named and typed *attribute* descriptions. Each attribute is either *synthesized* or *inherited*. With each constructor of the tree language, the simple tree attribution associates a *clause* that defines its attributes.

We use pattern matching to associate constructors with attribute definition clauses. Each clause is gov-

```
attribution realize_as_bytecode[VMACHINE] begin
  synthesized attribute bytecode(i : Instruction) : byte_string;
  synthesized attribute reg_bytecode(reg : Register) : byte_string;

  match ?r=register(?i) begin
    r.reg_bytecode := regnum_to_byte_string(i);
  end;
  match ?r=fp() begin
    r.reg_bytecode := FP;
  end;
  match ?l=loadi(?reg,?i) begin
    l.bytecode := LOADI || reg.reg_bytecode || integer_to_byte_string(i);
  end;
  ... Analogously for load, add, addi, and mov
  match ?s=seq(?i1,?i2) begin
    s.bytecode := i1.bytecode || i2.bytecode;
  end;
  match ?n=nop() begin
    n.bytecode := "";  -- no bytes generated for NOP.
  end;
end;
```

Figure 3: Simple Code Generation

erned by a pattern for the constructor with a pattern variable for the result and one for each parameter. The clause consists of an unordered collection of attribute definitions. Synthesized attributes are defined for the root of the constructed subtree; inherited attributes are defined for the syntactic parameters of the constructor.

We give two examples to explain simple tree attributions, beginning with the simpler bytecode generation phase. The two examples can be viewed as "phases" of a compiler. We will use these examples later to illustrate descriptional composition. As we explain the examples, we describe a number of features of simple tree attributions: local attributes, conditional attribution, syntactic attributes, remote references and functions. Simple tree attributions have other features that will not be discussed here. These include variable-arity constructors, default attribute values, private attributes, attributes defined for every phylum of a tree language, and Farnum's attribute pattern sets [5].

Our first example, given in Figure 3, is a simple tree attribution named `realize_as_bytecode` that creates a string of bytes for the `VMACHINE` language in Figure 1.[1] A synthesized attribute of type `byte_string` is defined for each `VMACHINE` phylum. The patterns provide a case analysis for the constructors, defining the `bytecode` attribute of each subtree of the `Instruction` phylum and the `reg_bytecode` attribute of the `Register` phylum. The cases that are elided in the figure can be inferred from the corresponding constructors.

The second example, `compile_simple` shown in Figures 4 and 5, defines a simple tree attribution for the language `SIMPLE_PROGRAM`. The attribution translates its source tree into a tree of `VMACHINE` instructions.

In Figure 4, semantic attribute `decl_depth` is defined for the `Declaration` phylum, and semantic attributes `scope_depth`, `reg_num`, and `reg_use` are defined for the `Expression` phylum. Two syntactic attributes (attributes whose values are used to construct trees) are defined, the attribute `code` for the `Expression` phylum and the attribute `progcode` for the `Program` phylum. Values for syntactic attributes are created by constructors and may freely use the values of semantic attributes.

The simple tree attribution `compile_simple` defines a *group* named `compilation`. A group is a device to associate attributes and functions that collaborate to produce a tree. The attributes and functions in the group `compilation` produce `VMACHINE` trees. A simple tree attribution may define more than one group, including multiple groups over the same output tree language. Each syntactic attribute in a simple tree attribution must be included in a group. The syntactic attributes in the example each produce `VMACHINE` subtrees of the `Instruction` phylum.

[1] Pattern variables are prefixed by '?' in match expressions; '||' denotes string concatenation. We use the syntax *varname.name* to refer to an attribute *name* of a node bound by the pattern match to a pattern variable *varname*. The `integer_to_byte_string` and `regnum_to_byte_string` functions do type conversion.

```
attribution compile_simple[SIMPLE_PROGRAM] begin

        -- scope_depth and decl_depth count surrounding blocks:
  inherited attribute scope_depth(e : Expression) : Integer;
  inherited attribute decl_depth(d : Declaration) : Integer;
        -- reg_num is the register number into which to compute the result
  inherited attribute reg_num(e : Expression) : Integer;
        -- reg_use is the number of registers needed to compute the result
  synthesized attribute reg_use(e : Expression) : Integer;

  group compilation[VMACHINE];

  synthesized attribute code(e : Expression) : Instruction in compilation;
  synthesized attribute progcode(p : Program) : Instruction in compilation

  ...   clause section
end;
```

Figure 4: Generation of Virtual Machine Code (declaration section)

In the SIMPLE_PROGRAM tree language, a use node contains a remote reference to a def node, providing access to the attributes of the variable declaration. In simple tree attributions, attributes of a remote reference may be read but not remotely defined. The syntactic attributes of a remote node may not be passed as syntactic parameters to constructors.

Syntactic attributes can be used for at least two conceptually different purposes: to provide a simplified view of the tree for another attribution purpose (such as name resolution) or to compute a transformation (such as for code generation, as in this example). Syntactic attributes are used for the second purpose in attribute coupled grammars. Composable attribute grammars use syntactic attributes for both purposes.

Each simple tree attribution may declare helper functions used in the attribute definitions. Like attributes, functions are either semantic or syntactic. Syntactic functions also must be part of a group. The body of the function is simply an expression of the same form as those used in attribute definitions. Functions derive their expressiveness from recursion.

In the SIMPLE_PROGRAM language, a bound variable may be used in scopes nested within its declaration scope. Every let is implemented as a full stack frame, using a frame pointer and static links. Therefore the code for a variable use may need to traverse multiple static links.[2] The link traversal code is generated using the syntactic function follow_link defined in Figure 5. Function follow_link takes two semantic parameters (times and reg) and generates a tree of VMACHINE instructions. The value of times is the difference in scope depths; reg is the register into which to place the result. The size of the generated tree is determined by the semantic parameter times.

Figure 5 gives the clauses defining the attributes for the constructors of SIMPLE_PROGRAM. Again the patterns provide a case analysis for the constructors of the tree language. The clauses matching subtrees rooted by plus nodes, and those rooted by use nodes illustrate several features of the attribution language.

A simple tree attribution clause may define an attribute local to that clause. Local attributes allow the programmer to factor out common parts of an expression, and are often initialized at the definition site. In the clause for plus, a local attribute addition is used to factor out a common part of the subtree being generated. A local attribute can be considered a synthesized attribute that cannot be defined or used outside of the context of a simple tree attribution clause.

A simple tree attribution clause may have attribute definitions inside if statements. Such conditional attributions are well-defined if every attribute defined within one branch of an if statement is defined in the other branch(es), and is not defined outside the if.[3] In the plus clause, the definitions of e.reg_use, e1.reg_num, e2.reg_num and e.code are all in if statements. Note that if's may be nested arbitrarily. Attribute values also can be defined by if expressions, as illustrated by the second definition of e.reg_use. It is possible to transform a simple tree attribution with conditional attributions into one without it by transforming all if

[2]This situation arises in programming languages with nested functions that can be passed as arguments to other functions.

[3]We make the safe assumption that each (non-identical) conditional expression is independent.

```
-- generate code to follow the static link in register "reg" "times" times.
function follow_link(times : Integer; reg : Integer) : Instruction in compilation is
  if times = 0 then
    nop()
  else
    seq(follow_link(times-1,reg),seq(addi(register(reg),register(reg),SL_OFFSET),
                                     load(register(reg),register(reg))))
  endif;
end;

match ?p=program(?e) begin
  e.scope_depth := 0;
  e.reg_num := 1;  -- start using registers from register 1
  p.progcode := e.code;
end;
match ?l=let(?d,?v,?b) begin
  local depth : Integer := l.scope_depth+1;
  d.decl_depth := depth; v.scope_depth := depth; b.scope_depth := depth;
  l.code := Establish a frame, elaborate declaration, perform b.code, pop frame;
  l.reg_use := max(v.reg_use,b.reg_use);
  v.reg_num := l.reg_num;  b.reg_num := l.reg_num;
end;
match ?d=def(?n) begin
  -- nothing to do here
end;
match ?e=plus(?e1,?e2) begin
  local addition : Instruction in compilation
      := add(register(e.reg_num),register(e1.reg_num),register(e2.reg_num));
  if e2.reg_use > e1.reg_use then            -- do e2 first
    e.reg_use := e2.reg_use;
    e1.reg_num := e.reg_num+1;
    e2.reg_num := e.reg_num;
    e.code := seq(seq(e2.code, e1.code), addition);
  else                                       -- do e1 first
    e.reg_use := if e1.reg_use = e2.reg_use then e1.reg_use+1 else e1.reg_use endif;
    e1.reg_num := e.reg_num;
    e2.reg_num := e.reg_num+1;
    e.code := seq(seq(e1.code, e2.code), addition);
  endif;
  e1.scope_depth := e.scope_depth;  e2.scope_depth := e.scope_depth;
end;
match ?e=minus(?e1,?e2) begin ... (Analogous to plus) end;
match ?e=constant(?i) begin
  e.reg_use := 1;
  e.code := loadi(register(e.reg_num),i);
end;
match ?u=use(?d) begin
  u.reg_use := 1;
  local depthdiff : Integer := u.scope_depth - d.decl_depth;
  u.code := seq(seq(move(register(u.reg_num),fp()),
                    follow_link(depthdiff,u.reg_num)),
                load(register(u.reg_num),register(u.reg_num)));
end;
```

Figure 5: Generation of Virtual Machine Code (clause section)

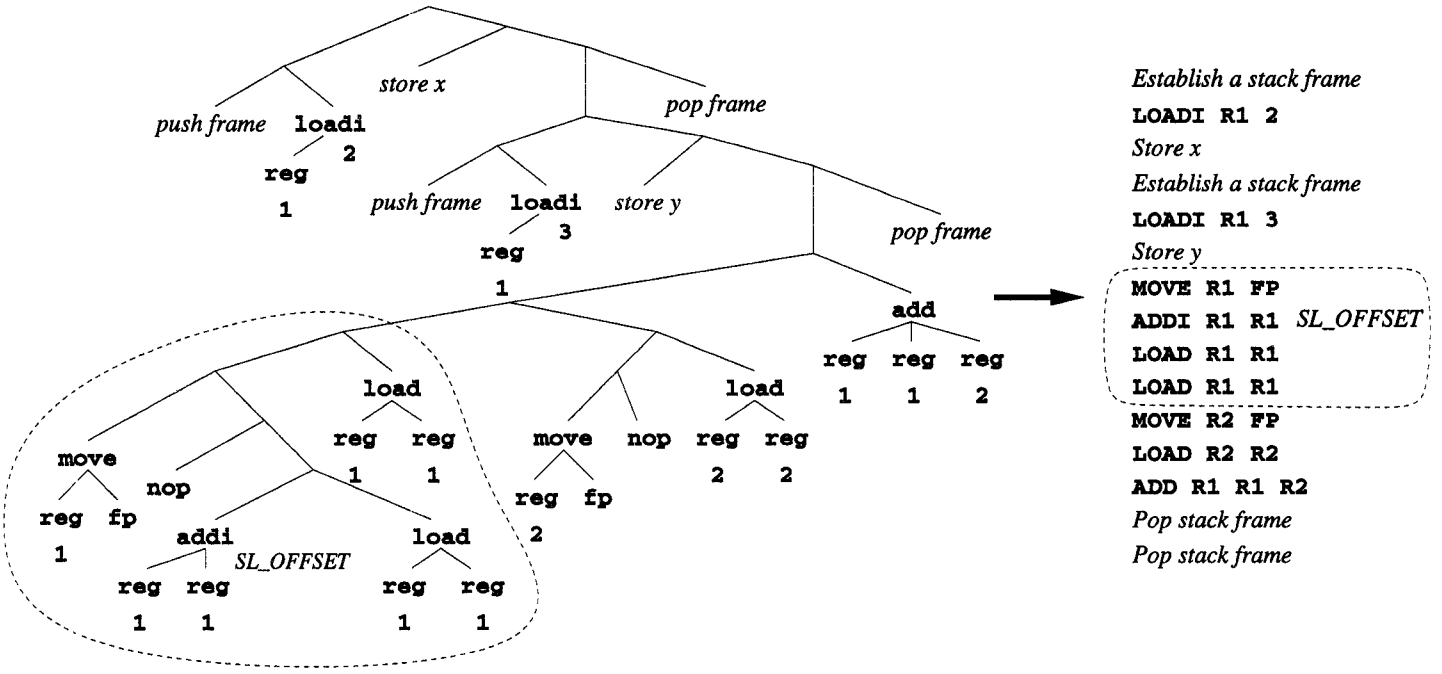


Figure 6: The VMACHINE version of Figure 2, and its byte_string realization

statements to equivalent uses of if expressions. Alternatively, any use of an if expression can be replaced by conditional attribution. In the example, the if's are used so that the definitions of the attributes for plus minimize register pressure by scheduling the subtree that needs more registers before the other subtree.

Figure 6 shows the result of applying compile_simple to the example in Figure 2 and the result of applying realize_as_byte_code to that result. The regions enclosed in dotted lines show the various stages for the use of x in the input.

In the VMACHINE tree, we omit the internal node labels for seq. The byte code is written so that each sequence of bytes for an instruction is on one line.

## 3 Descriptional Composition

Descriptional composition takes a simple tree attribution A1 defined over a source tree language $T_{A1}$ with a group G defined on some target tree language $T_{A2}$ and composes it with a simple tree attribution A2 defined over $T_{A2}$.[4] The resulting simple tree attribution has the same "effect" as the functional composition of A1 and A2 but the intermediate tree in $T_{A2}$ is never produced. Moreover, since the composed simple tree attribution exposes all the computation of both A1 and A2, it can be optimized using partial evaluation.

If we descriptionally compose the simple tree attribution compile_simple in Figures 4 and 5 with the simple tree attribution realize_as_bytecode in Figure 3, we get a simple tree attribution over the language SIMPLE_PROGRAM that computes the bytecode directly. The composition is shown in Figure 7. It will be explained in the following section.

### 3.1 Achieving Descriptional Composition

In this section, we describe an algorithm for descriptional composition. This algorithm works only for tree attributions satisfying certain properties, which will be introduced as the need for them arises.

In the composed simple tree attribution A1_A2, we include unchanged every semantic attribute of A1, every semantic function of A1, and every group other than G. Moreover, *every* function in A2 (semantic or syntactic) is carried over unchanged. Every syntactic attribute and function in G is replaced by composed attributes and functions.

To simplify the composition constructions, we simplify the forms of the attribute definitions being transformed. Nested calls to constructors or functions are eliminated by introducing local attributes. All if expressions are transformed to the corresponding if state-

[4] Although $T_{A1}$ and $T_{A2}$ are different tree languages in our example, they need not be in general.

```
attribution compile_simple_realize_as_bytecode[SIMPLE_PROGRAM] begin
  ... Semantic attributes of compile_simple remain as before.
          -- composed attributes:
  synthesized attribute code_bytecode(e : Expression) : byte_string;
  synthesized attribute progcode_bytecode(p : Program) : byte_string;
          -- composed function:
  function follow_link_bytecode(times : Integer; reg : Integer) : byte_string is
    if times = 0 then
      ""
    else
      follow_link_bytecode(times-1,reg) || ADDI || regnum_to_byte_string(reg) ||
      ... || LOAD || regnum_to_byte_string(reg) || regnum_to_byte_string(reg));
    endif
  end;

  match ?p=program(?e) begin
    ...
    p.progcode_bytecode := e.code_bytecode;
  end;
  match ?l=let(?d,?v,?b) begin
    ...
    l.code_bytecode := ...;
  end;
  match ?d=def(?n) begin end;
  match ?e=plus(?e1,?e2) begin
    local addition_bytecode : byte_string :=
          ADD || regnum_to_byte_string(e.reg_num) || regnum_to_byte_string(e1.reg_num)
              || regnum_to_byte_string(e2.reg_num);
    if e2.reg_use > e1.reg_use then
      ...
      e.code_bytecode := e2.code_bytecode || e1.code_bytecode || addition_bytecode;
    else
      ...
      e.code_bytecode := e1.code_bytecode || e2.code_bytecode || addition_bytecode;
    endif;
  end;
  match ?e=minus(?e1,?e2) begin
    ... Analogous to plus
  end;
  match ?e=constant(?i) begin
    ...
    e.code_bytecode := LOADI || regnum_to_byte_string(e.reg_num)
                             || integer_to_byte_string(i);
  end;
  match ?u=use(?d) begin
    ...
    u.code_bytecode := MOV || regnum_to_byte_string(u.reg_num) || FP ||
        follow_link_bytecode(depthdiff, u.reg_num) ||
        LOAD || regnum_to_byte_string(u.reg_num) || regnum_to_byte_string(u.reg_num);
  end;
end;
```

Figure 7: Generated Byte String Compiler

ments. Then the attribute definitions have the form

```
if cond1 then
  if cond2 then
    . . .
      definition
    . . .
  endif;
endif;
```

where *definition* has one of the forms $attr_0$ := $attr_1$; or $attr_0$ := *constructor*($attr_1$, $attr_2$, ...); or $attr_0$ := *function*($attr_1$, $attr_2$, ...);.

In A1_A2, each attribute in G is composed with each attribute in A2 for the same phylum. Since syntactic functions build their subtrees bottom-up, for each function in G, a composed function is introduced for each synthesized attribute of the result phylum. In order to incorporate the inherited attributes of the constructed subtree, for each syntactic formal parameter of a function in G, a composed function is added for each inherited attribute of its phylum in A2. We consider each case in turn.

Since the introduction of composed functions also affects their calls, consider first the two forms of attribute definition that make no calls on syntactic functions. Suppose the definition has the form

$attr_0$ := $attr_1$;

where $attr_0$ is defined in G and the value of attribute $attr_1$ belongs to some phylum of $T_{A2}$. For each attribute *name* defined in A2 for that phylum, if *name* is a synthesized attribute, the corresponding composed definition is of the form

$attr_0$.*name* := $attr_1$.*name*;

If *name* is an inherited attribute, the corresponding composed definition is of the form

$attr_1$.*name* := $attr_0$.*name*;

For instance, `compile_simple` has the definition

```
p.progcode := e.code;
```

When we compose this definition with the single synthesized attribute `bytecode` for phylum `Instruction`, we get the definition

```
p.progcode.bytecode := e.code.bytecode;
```

Suppose the attribute definition in G has the form

$attr_0$ := *constructor*($attr_1$, $attr_2$, ...);

In A1_A2, we compose this definition with the attribute definitions for that constructor in A2, replacing each pattern variable for that simple tree attribution clause by the corresponding attribute instance in the definition. For example, the `plus` clause in `compile_simple` includes the definition

```
e.code := seq(seq(e2.code, e1.code), addition);
```

which is replaced by two definitions such as

```
loc341 := seq(e2.code, e1.code);
e.code := seq(loc341,addition);
```

when nested constructors are eliminated. (Here `loc341` is the name of the local attribute that is introduced.) When we compose these definitions with the definition of attribute `bytecode` in `realize_as_bytecode`, we get the definitions

```
loc341.bytecode := e2.code.bytecode ||
                   e1.code.bytecode;
e.code.bytecode := loc341.bytecode ||
                   addition.bytecode;
```

Finally, we name all attributes of A1_A2 by applying the rules:

$name_1.name_2 \Rightarrow name_1\_name_2$
$node.name_1.name_2 \Rightarrow node.name_1\_name_2$

For example, the preceding definitions are rewritten as

```
p.progcode_bytecode := e.code_bytecode;
loc341_bytecode := e2.code_bytecode ||
                   e1.code_bytecode;
e.code_bytecode := loc341_bytecode ||
                   addition_bytecode;
```

(For simplicity, we assume no name conflicts ensue with this rule. If necessary, unique names can be created.)

In the composition process, for each (syntactic) function *function* of G, a composed function is created for each synthesized attribute *name* of A2. The composed functions compute the values those attributes would have for the subtree built by the original syntactic function. We create names *function_name* for these functions. Since in A2, a synthesized attribute of a node may depend (directly or indirectly) on the inherited attributes of that node, the composed functions must be passed some of the inherited attributes of the subtree generated by *function*. Furthermore, if *function* takes syntactic parameters, the synthesized attributes of each such parameter may affect the synthesized attributes of the generated subtree. Therefore, the composed functions must be passed some of the synthesized attributes of each parameter.

The inherited attributes of each syntactic parameter must be computed in some way, since the structure built around each parameter would define its inherited attributes. Therefore, A1_A2 must include functions for each of the inherited attributes of A2 that would be

defined for that syntactic parameter. We name these functions using the names of the function, the formal syntactic parameter and the inherited attribute: *function_formal_name*. Again these functions must passed some of the inherited attributes of the generated subtree and some of the synthesized attributes of each syntactic parameter. The set of attribute values that must be passed to these functions can be computed statically from the functions and A2, using analysis similar to that for computing whether an attribute grammar is strongly non-circular [4].

Consider an attribute definition of the form

$attr_0$ := *function*($attr_1$, $attr_2$, ...);

The subtree returned by the function will be of some phylum of $T_{A2}$. For each synthesized attribute *name* defined in A2 for this phylum, we add a composed definition of the form

$attr_0$.*name* := *function_name*(...);

The needed composed attributes are passed to the composed function.

Each syntactic parameter of the function belongs to some phylum. For each inherited attribute *name* defined in A2 for this phylum, we add a composed definition of the form

$attr_j$.*name* := *function_formal*$_j$*_name*(...);

Again, we must pass needed composed attributes.

For example, in the clause for the `use` constructor in `compile_simple`, after nesting has been replaced by introduction of local attributes, our example would have a definition such as:

```
loc172 := follow_link(depthdiff,u.reg_num);
```

In the composed simple tree attribution, this definition would be replaced eventually by

```
  loc172_bytecode :=
follow_link_bytecode(depthdiff,u.reg_num);
```

The composed function needs no additional parameters because `follow_link` does not take any syntactic parameters, and `realize_as_bytecode` does not have any inherited attributes.

Next we must define the new functions for the composed description. To reuse the composition rules just described, we treat functions as if they were simple tree attribution clauses for pseudo constructors of $T_{A2}$. Each of the formal parameters to the function becomes a parameter to the pseudo constructor, and the pattern variable bound to the result of the constructor is used as a local attribute. By introducing local attributes, the clause can be put in the simple form used previously. The result is *not* a well-defined simple tree attribution clause; we use it only for descriptional composition.

For example, if we treat `follow_link` in Figure 5 in this way, we have the clause

```
match ?follow_link=pc197(?times,?reg) begin
  local loc198 : Integer;
  local loc199 : Register;
  ...
  if times = 0 then
    follow_link := nop();
  else
    loc198 := times-1;
    loc199 := follow_link(loc198,reg);
    loc200 := register(reg);
    loc201 := register(reg);
    loc202 := addi(loc200,loc201,SL_OFFSET)
    loc203 := register(reg);
    loc204 := register(reg);
    loc205 := load(loc203,loc204);
    loc206 := seq(loc202,loc205);
    follow_link := seq(loc199,loc206);
  endif;
end;
```

(Note that the name `follow_link` is used both as a local attribute and as a function.) We then perform composition on all attributes that compute tree nodes. In our example, this gives us the composed clause

```
match ?follow_link=p197(?times,?code) begin
  local loc198 : Integer;
  local loc199_bytecode : byte_string;
  ...
  if times = 0 then
    follow_link_bytecode := "";
  else
    loc198 := times-1;
    loc199_bytecode :=
      follow_link_bytecode(loc198,reg);
    ...
    follow_link_bytecode :=
      loc199_bytecode || loc206_bytecode
  endif;
end;
```

Then we convert the result back to a function. Since function bodies are expressions, `if` statements are converted to `if` expressions, local attributes that name composed functions are converted into function definitions, and local attribute uses are replaced by their definitions. The converted function `follow_link_bytecode` is shown in Figure 7.

Remote node references, in which a designation of a node may be passed through the attribute system and then queried for (inherited or synthesized) attributes, require special attention in the algorithm. Attributes

or functions that have node references as values and function parameters that take node references as values must be declared in the group of the nodes being [referred to]. The algorithm treats them much the same as syntactic attributes or functions, except that since attributes of node references can only be read, never written, for the purposes of composition all the attributes of the node reference phylum in A2 can be treated as if they were synthesized.

## 3.2 Complications

In this section, we discuss the ways in which the composition algorithm may fail, producing a simple tree attribution that is circular or not well-formed.

### Graphs

If A1 builds a (directed acyclic) graph, rather than a tree, and A2 has inherited attributes, the resulting simple tree attribution will be ill-formed because there will be two competing definitions for the composed inherited attributes. Moreover, it is assumed that whenever a subtree is assigned to a syntactic attribute, or passed as a syntactic parameter to a constructor or function, the attribute or parameter is the sole consumer of that subtree. If a subtree is used syntactically in more than one place (other than in the two branches of an `if`), the algorithm will fail, again by producing two competing definitions for inherited attributes.

We term the restriction that such a subtree may only be used once the *syntactic at most one dynamic use requirement* echoing the *syntactic single use requirement* of Ganzinger and Giegerich [8]. We say *syntactic* because the restriction is only on syntactic uses. We say *at most one* because we permit a subtree to be unused, in which case the composed inherited attributes will get default values. We say *dynamic use* because our restriction reflects the fact that only one branch of an `if` will be active each time it is used dynamically. We call the restriction by its acronym (SAMODUR).

### Attributes Needed by Composed Functions

The analysis we do to determine what parameters must be passed to a composed function cannot distinguish between values that are used in *some* circumstances, and values that are *always* used. Sometimes, even if both A1 and A2 are well-formed, a composed function may (directly or indirectly) require its own result as a parameter, in which case, the result of descriptional composition is circular.

If we restrict A2 to belong to an evaluation class corresponding to the attribute grammar evaluation class SNC [4] (also known as ANC [11]), the summary graphs for syntactic functions will not result in cycles in the composed simple tree attribution. If the functions take syntactic parameters, then, in general, A2 must belong to an evaluation class corresponding to the doubly non-circular class DNC [7].

Intuitively, a simple tree attribution is SNC if it can be implemented as a set of mutually recursive strict functions, each of which computes a single synthesized attribute of a phylum as a function of a subset of its inherited attributes. A simple tree attribution is DNC if inherited attributes can be computed that way as well.

### Remote Node References

Recall that in constructing the composed functions, the use of a local attribute is replaced by its definition. If the definition invokes a constructor, then in the presence of remote references, this transformation may no longer be correct, since a reference is a designation of a particular node, and each call to a constructor returns a new node.

Our expression language permits `let` bindings, which can be used instead of local attributes, so this problem does not arise unless A2 has more than one *node-valued attribute* (that is, a syntactic attribute or an attribute carrying a remote reference) that share a certain node constructed locally. Even if there are multiple synthesized node-valued attributes, but they could be evaluated in one pass, a modified version of the algorithm could compute them all together in a single composed function that returns a tuple of values. Similarly if the node-valued attributes cannot be evaluated in one pass, but can be ordered in multiple passes, we can construct tuples of anonymous functions that share local attribute bindings and take the necessary inherited attributes.

In addition, the use of remote node references can lead to circularities, or at least make it very difficult to determine attribute dependencies statically. Strictly speaking, a simple tree attribution that reads attributes through remote node references cannot be SNC because not all data dependencies into a subtree are given by the inherited attributes. This fact does not cause problems for the algorithm. If a function of A1 creates a tree with a node reference used by A2, it must have created the corresponding node itself, or have the reference passed to it as a parameter. In either case, the dependency can be tracked easily.

### Nonseparability

In simple tree attributions, as in composable attribute grammars, attribute definitions may use the values of attributes of a generated node (attributes that are defined by a later attribution), even if an 'out-of-order' attribution evaluation is required. In other words, A1

may use the values of attributes of A2 on nodes created in the group G.

However, in this case circularity may result, because A1 may use the values read to determine the structure to be used for the tree being built. Descriptional composition will only introduce circularity if the (functional) composition is intrinsically circular. Farrow, Marlowe and Yellin [6] define a property of A1 that is sufficient for preventing circularity in the result. They term this property *separability*.[5] A simple tree attribution is separable provided no construction of a subtree depends directly or indirectly on an attribute read from any other node in the same tree.

#### Summary

We can give a safe domain for our algorithm: descriptional composition of A1 and A2 is possible when the following conditions are met:

- both A1 and A2 are noncircular
- A1 is separable.
- A1 satisfies SAMODUR *or* A2 has no inherited attributes.
- A1 has no syntactic functions *or* the following hold:
  - A2 is strongly non-circular (SNC)
  - A1 has no syntactic parameters *or* A2 is doubly non-circular (DNC)
  - the node-valued attributes of A2 can be ordered.

In section 3.4, we put forward some ideas for removing the last set of conditions.

### 3.3 Properties of Composed Tree Attributions

To compose a whole series of simple tree attributions into a single simple tree attribution, one needs to be able to use the composition algorithm with composed simple tree attributions. Since the algorithm can fail outside of the safe domain mentioned above, it is important to see what properties of the resulting simple tree attribution can be guaranteed. As it turns out, even when all the tree attributions individually have very simple dependencies, the composition may be much more complex.

[5]For Farrow, Marlowe and Yellin, separability is actually defined as a property of a collection of composable attribute grammars, not an individual CAG.

#### Superficial Circularity

SAMODUR takes into account that the two branches of an `if` are mutually exclusive, but the standard definitions of evaluation classes for attribute grammars do not. As a result, the composition may be superficially circular. For example[6], if A1 has an equation such as

```
p.out := if cond then
           construct(c1.out,c2.out)
         else
           construct(c2.out,c1.out)
         endif;
```

and A2 has a simple dependency thread for `construct` with inherited attributes:

```
match ?a=construct(?b,?c) begin
  b.i := a.i;
  c.i := b.s;
  a.s := c.s;
end;
```

(where `i` is always used to compute `s`), then A1_A2 would be considered circular under the traditional definition, although it may not actually be circular:

```
c1.out_i := if cond then p.out_i
                    else c2.out_s endif;
c2.out_i := if cond then c1.out_s
                    else p.out_i endif;
p.out_s := if cond then c2.out_s
                   else c1.out_s endif;
```

(where `out_i` is used to compute `out_s`).

We believe that we can redefine the standard attribute grammar classes (such as SWEEP, $l$-ORD, DNC and SNC) to take into account the nonstrictness of `if`. Basically, each constructor can be "split" into several otherwise identical constructors except that each handles the case for a different value of a conditional used in an attribute definition.

According to this redefinition, the example composed attribution above would be in the SWEEP class (one pass, either left to right or right to left depending on the constructor; that is, depending on the condition).

#### Syntactic Parameters

If a syntactic function in A1 takes syntactic or remote node reference parameters, then in the composed simple tree attribution, we will have a composed function for each of the appropriate synthesized attributes in A2. If some of these synthesized attributes are syntactic (i.e., A2 builds trees also), then it is possible

[6]This example is a simplified version of the definition for `e.code` in the clause for `plus` in Figure 5.

that SAMODUR will be violated in the composed simple tree attribution. For example, two composed functions could each take the same syntactic attribute as a parameter, and each use the parameter under mutually exclusive conditions. Even if both A1 and A2 satisfy SAMODUR, even though A1_A2 actually still builds a tree; the composition algorithm as given cannot compose it any further because SAMODUR is violated. SAMODUR assumes that every syntactic parameter is always used by the function to which it is passed.

The problem can be traced back to the fact that when we compose the function with A2, we pull apart the composed attribution clause for the pseudo-constructor into separate composed functions. When we determine what parameters are used by the body of each function, we must of necessity include every parameter that *might* be used by the composed function.

This analysis suggests two lines of attack for avoiding this problem. First, it seems that the condition that determines whether the parameter is used could be computed in advance of a call, and used to condition the actual parameter. Second, the algorithm could be modified so that the composed attribution clause for the pseudo-constructor does not need to be separated into functions with parameters, and instead all the composed functions would be defined together. This second idea is explored briefly in Section 3.4.

#### Inherited Syntactic Attributes

Giegerich gives a thorough examination of the closure properties of descriptional composition for attribute coupled grammars [9]. Among other results, he gives an example in which descriptionally composing three serial L-attributed attribute coupled grammars gives a composed grammar that is not only not L-attributed, but is not even strongly non-circular (SNC). However, if inherited syntactic attributes are not permitted, closure does result for SNC attribute coupled grammars.

In order to know that repeated composition is possible in the presence of syntactic functions, we need the result to be SNC. Therefore, if we want guaranteed closure, the simple tree attributions cannot have inherited syntactic or remote node reference attributes.

#### Closure

In summary, descriptional composition is closed over the set of simple tree attributions $A$ where

- $A$ is separable and satisfies SAMODUR
- $A$ is SNC
- $A$ has no node-valued function parameters
- $A$ has no inherited node-valued attributes

where SNC is defined by "splitting" constructors and by ignoring attributes read from remote references built into the tree being attributed. Giegerich's proof for attribute coupled grammars [9] should carry over to simple tree attributions, with changes to account for the `if`'s, functions, and remote node references.

### 3.4 Further Work

The restrictions given in Section 3.2 on the use of syntactic functions (and syntactic function parameters) are somewhat unsatisfying. In this section, we suggest some ways to modify the algorithm so that it operates correctly without these restrictions.

In the composition algorithm shown previously, we convert a function body into an attribution clause, compose it with A2 and then convert it back into a set of composed functions. The body of a function in our system is simply an expression and thus there is no way to express a function that needs multiple passes to completely define its return value.

However, for computing attributes in a simple tree attribution, the programmer is not forced to compute all the synthesized attributes of a node as a function of the inherited ones in one pass. Instead, the programmer gives the definitions and a simple tree attribution compiler finds an evaluation mechanism that accommodates the dependencies. The question comes to mind as to whether functions could have this flexibility as well.

Instead of parameters and a return value, these more powerful "functions" would have inherited and synthesized attributes. This expressive power is similar to that of lazy evaluation. However, as with attribute grammars, one can define evaluation classes for these functions that restrict the generality and that can be checked statically.

We believe all the restrictions for syntactic functions given in Section 3.2 would go away if the system used these more powerful functions.

## 4 Comparison with Related Work

Attribute coupled grammars allow serial phases of a compiler to be defined separately, and then either executed separately or descriptionally composed into a monolithic attribute grammar. Our interest in composition is for the construction of dynamic compilers, that is, incremental compilers that patch running programs [3]. Our method of constructing a dynamic compiler requires that the entire task of compilation be described in a single simple tree attribution.

One can consider composable attribute grammars and higher order attribute grammars to be extensions of attribute coupled grammars. In composable attribute

grammars, a "glue" attribute grammar may use "output" attributes of a computed tree. The glue grammar is aware of the attributes defined by the "component" grammars, and uses them to compute the results of the composition.

With higher order attribute grammars, there is no distinction between syntactic and semantic values, and subtrees may be passed in arbitrary ways in arbitrary data structures to arbitrary functions. Resulting trees may be grafted into the original tree, and the "host" attribute grammar can then use synthesized attributes of the root of the grafted tree. The system described here could be expanded easily to accommodate these facilities. However, simple tree attributions that use this power are likely to run afoul of the SAMODUR restriction. Furthermore, since for higher order attribute grammars, subtrees do not obey object identity, there is no way to fetch the inherited attribute of a node at the root of a subtree. In a sense, higher order attribute grammars provide so much power in the ways in which tree values are used that any interesting descriptional composition is impossible.

## 5 Acknowledgements

We thank Robert Giegerich for sharing his insights with us and for his suggested improvements to the paper. The comments of David Bacon and William Maddox were also most valuable.

## References

[1] BALLANCE, R. A. *Syntactic and semantic checking in language-based editing systems.* PhD thesis, Computer Science Division—EECS, University of California, Berkeley, Dec. 1989. Technical report UCB/CSD 89/548.

[2] BALLANCE, R. A., AND GRAHAM, S. L. Incremental consistency maintenance for interactive applications. In *Proc. of the Eighth International Conference on Logic Programming*, K. Furukawa, Ed. The MIT Press, Cambridge, Massachusetts and London, England, 1991, pp. 895–909.

[3] BOYLAND, J., FARNUM, C., AND GRAHAM, S. L. Attributed transformational code generation for dynamic compilers. In *Code Generation - Concepts, Tools, Techniques. Workshops in Computer Science*, R. Giegerich and S. L. Graham, Eds. Springer-Verlag, Berlin, Heidelberg, New York, 1992, pp. 227–254.

[4] COURCELLE, B., AND FRANCHI-ZANNETTACCI, P. Attribute grammars and recursive program schemes. *Theoretical Computer Science 17* (1982), 163–191,235–257.

[5] FARNUM, C. Pattern-based tree attribution. In *Conference Record of the Nineteenth Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages* (Jan. 1992), pp. 211–222.

[6] FARROW, R., MARLOWE, T. J., AND YELLIN, D. M. Composable attribute grammars: Support for modularity in translator design and implementation. In *Conference Record of the Nineteenth Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages* (Jan. 1992), pp. 223–234.

[7] FILÉ, G. Classical and incremental attribute evaluation by means of recursive procedures. In *11th. Coll. on Trees in Algebra and Programming (CAAP '86), LNCS vol. 214* (Mar. 1986), P. Franchi-Zannettacci, Ed., Springer-Verlag, pp. 112–126.

[8] GANZINGER, H., AND GIEGERICH, R. Attribute coupled grammars. In *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction* (June 1984), pp. 157–170.

[9] GIEGERICH, R. Composition and evaluation of attribute coupled grammars. *Acta Inf. 25* (1988), 355–423.

[10] KAHN, G. Natural semantics. In *STACS '87: Fourth Annual Symposium on Theoretical Aspects of Computer Sciences* (Berlin, Heidelberg, New York, 1987), F. Brandenburg, G. Vidal-Nacquet, and W. Wirsig, Eds., Lecture Notes in Computer Science, Springer-Verlag, pp. 22–39.

[11] KENNEDY, K., AND WARREN, S. K. Automatic generation of efficient evaluators for attribute grammars. In *Conference Record of the Third ACM Symposium on Principles of Programming Languages* (1976), pp. 32–49.

[12] KNUTH, D. E. Semantics of context free languages. *Math Systems Theory 2*, 2 (June 1968), 127–145. Errata *Math Systems Theory* 5(1):95–96(1971).

[13] SWIESTRA, D., AND VOGT, H. Higher order attribute grammars. In *Attribute Grammars, Applications and Systems*, H. Albas and B. Melichar, Eds., no. 545 in Lecture Notes in Computer Science. Springer-Verlag, Berlin, Heidelberg, New York, 1991, pp. 256–296.

[14] VOGT, H. H., SWIESTRA, S. D., AND KUIPER, M. F. Higher order attribute grammars. In *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation* (June 1989), pp. 131–145.