The Synthesizer Generator

Thomas Reps and Tim Teitelbaum
Cornell University

Abstract

Programs are hierarchical compositions of formulae satisfying structural and extra-structural relationships. A program editor can use knowledge of such relationships to detect and provide immediate feedback about violations of them. The Synthesizer Generator is a tool for creating such editors from language descriptions. An editor designer specifies the desired relationships and the feedback to be given when they are violated, as well as a user interface; from the specification, the Synthesizer Generator creates a full-screen editor for manipulating programs in the language.

1. Introduction

With the Cornell Program Synthesizer, we demonstrated the power of full-screen, syntax-directed editing for block-structured languages, especially when coupled with incremental compilation and structured interpretation and debugging [Teitelbaum & Reps 1981]. The success of the initial, hand-crafted Synthesizer encouraged us to create the Synthesizer Generator, a system for building such environments from language descriptions.

At the same time, our goal was to explore methods for integrating additional program analysis and translation tools into interactive program development systems. The desire to enforce context-sensitive syntactic constraints, perform incremental translation, and detect data-flow anomalies implied we needed a way of incorporating knowledge of a language's semantic relations.

As argued in our earlier paper [Demers et al. 1981], attribute grammars are an attractive underlying formalism for systems such as the Synthesizer Generator because:

- They extend the descriptive power of context-free grammars, thereby permitting expression of context-sensitive relationships, such as type consistency in a program.
- They are declarative definitions of relations that must hold among the parts of a program, thereby ensuring that context-sensitive analyses defined with attribute grammars cannot depend on the order in which a program is developed. Consequently, a system designer can never create an editor having order-dependent errors.
- They allow automatic reestablishment of consistent relationships among attributes when a syntax tree is modified, without explicit "undoing" or "rollback" actions, for propagation of semantic information through the tree is implicit in the formalism. Furthermore, this updating process can be performed in an asymptotically time-optimal manner [Reps 1984, Reps et al. 1983].

Because of these properties, we have based the Synthesizer Generator on attribute grammars.

The Synthesizer Generator is a tool for specifying how structured objects may be manipulated in the presence of context-sensitive relationships. The editor designer prepares an attribute-grammar specification that includes rules defining abstract syntax, attribution, display format, and concrete input syntax. From this specification, the Generator creates a full-screen editor for manipulating objects according to these rules.

In an editor generated with the Synthesizer Generator, a program is represented as a consistently attributed derivation tree. Programs are modified by operations that restructure the derivation tree, such as pruning, grafting, and deriving. Restructuring a derivation tree directly affects the values of the attributes at the modification point; some of the attributes may no longer have consistent values. Incremental analysis is performed by updating attribute values throughout the tree in response to modifications.

Apart from its use to generate programming-language editors, the Synthesizer Generator has also been used for creating a variety of other tools that manipulate structured data, including a desk calculator, a proof checker, and several text-formatting editors.

The use of attribute grammars for the underlying formalism of the Synthesizer Generator distinguishes our approach from the ones used in MENTOR [Donzeau-Gouge et al. 1975] and GANDALF [Medina-Mora & Notkin 1981]. The use of attribute grammars for specifying interactive environments distinguishes our work from compiler writing systems such as GAG [Kastens et al. 1983]. Other projects currently using attribute grammars in interactive environments include TRIAD [Ramanathan & Soni 1980] and POE [Johnson 1983].

This work was supported in part by the National Science Foundation under grants MCS80-04218, and MCS82-02677.

Authors' address: Department of Computer Science, Upson Hall, Cornell University, Ithaca, N.Y. 14853.

The foundations of the attribute-grammar approach to building language-based environments are discussed in [Reps 1984, Reps et al. 1983]. Applications of the Synthesizer Generator are discussed in [Reps 1983] and in [Reps & Alpern 1984]. This paper gives an overview of the Synthesizer Generator's facilities for specifying language-based editors.

2. Abstract syntax

The core of an editor specification for a given language is the definition of the language's abstract syntax, given as a set of grammar rules. An object being edited is represented by its derivation tree with respect to the grammar, and regardless of user interface, be it textual or structural, the effect of each editing modification is to change this underlying syntax tree.

The abstract syntax is specified as a collection of phyla and operators, a formulation of context-free grammars more appropriate for defining abstract syntax and better suited for defining structure editors [Donzeau-Gouge et al. 1975, Medina-Mora & Notkin 1981]. An *operator* is a uniquely-named (possibly 0-ary) Cartesian product of phyla. A *phylum* is a non-empty set of operators. A rule of the form

phy_0: $op(phy_1\ phy_2\ \cdots\ phy_k)$;

declares membership in phy_0 of a k-ary operator *op* with *arguments* phy_1, phy_2, $\cdots$ phy_k, and is analogous to the context free production

$phy_0 \rightarrow phy_1\ phy_2\ \cdots\ phy_k$

with the differences that (1) the operator name *op* differentiates this production from all other structurally identical alternatives of phy_0, and (2) the given operator *op* may be a member of other phyla, necessarily with the same arity and arguments.

For defining structure editors, the phylum/operator formalism offers several advantages over the nonterminal/production notion: (1) during editing, a pruned subtree is identified as an instance of a given operator, not as an instance of a given production, and accordingly may be grafted in the tree as an instance of any of the other phyla containing that operator; (2) the possibility of intersecting phyla eliminates from the grammar (and therefore from abstract-syntax tree) the need for nonterminals introduced only for the purpose of factoring.

As will be described later in the paper, the concept of an operator plays a key role in the notation used in the Synthesizer Generator's attribute grammar definitions. (1) As discussed in Section 4, operator names are used to express construction and selection operations on structures. (2) As discussed in Section 5, operator names are used to support modularity in editor specifications.

As the phyla/operator formalism coincides with the nonterminal/production formalism when all phyla are disjoint, and as the nonterminal/production vocabulary is more widely known, we shall refer to phyla synonymously as *nonterminals*, and refer to an occurrence of a given operator in a phylum as a *production*.

Example. To illustrate the Synthesizer Generator's specification language, we present, as a running example, the definition of a simple, full-screen desk calculator that allows creation and modification of an integer expression, during which time the expression's current value is incrementally computed and displayed. The abstract syntax of the desk calculator's arithmetic expressions is defined by the following rules:

```
exp: NullExp()
   | Sum(exp exp)
   | Diff(exp exp)
   | Prod(exp exp)
   | Quot(exp exp)
   | Const(INT)
   ;
```

In these rules, the definitions of the different operators of phylum exp are separated by vertical bars, and INT refers to a primitive, predefined phylum containing the nullary operators 0, -1, 1, -2, 2, etc. Other predefined phyla include CHAR, STR, FLOAT, DOUBLE, and BOOL.

The first declared operator of a phylum, such as NullExp in the example above, is termed the *completing operator*, and is used, by default, at unexpanded occurrences of that phylum in the derivation tree. Thus, a tree that the user considers to be a partial derivation tree, such as Sum(exp,exp), is really a complete derivation tree from the system's point of view, e.g. the tree Sum(NullExp(),NullExp()).

3. Attributes and semantic equations

A declarative specification of context-dependent computations on the set of abstract-syntax trees of a language is conveniently provided by the semantic rules of an attribute grammar, a context-free grammar extended by attaching attributes to the nonterminals of the grammar [Knuth 1968].

Associated with each production is a set of *semantic equations*, each of which defines an attribute of one of the production's nonterminals as the value of a *semantic function* applied to other attributes of nonterminals in the production. Attributes are divided into two disjoint classes: *synthesized* and *inherited*; each semantic equation defines a value for a synthesized attribute of the left-side nonterminal or an inherited attribute of a right-side nonterminal, attributes termed the *output attributes* of the production. The semantic equations of a specification must obey two constraints: there must be equations for *all* output attributes of each production, and it must not be possible to build a derivation tree in which attributes are defined circularly.

An attribute is attached to a nonterminal by specifying the name of the nonterminal, the type of the attribute, and whether the attribute is synthesized or inherited. An attribute's type can either be one of the predefined phyla (see above) or a user-defined phylum (see below). For example, the declaration:

```
exp { synthesized INT v; };
```

associates a synthesized INT-valued attribute v with the nonterminal exp. Attribute a of the ith occurrence of nonterminal n in a given production is referred to as n$i.a, where $i is optional if there is only one occurrence of n in the production.

Example. The following semantic equations define the v attribute of each exp in the syntax tree to be the value of the arithmetic subexpression with root exp. The equations below illustrate the use of synthesized attributes; the use of inherited attributes will be illustrated in Section 5.

```
exp: NullExp { exp.v = 0; }
  |Sum  { exp$1.v = exp$2.v + exp$3.v; }
  |Diff { exp$1.v = exp$2.v - exp$3.v; }
  |Prod { exp$1.v = exp$2.v * exp$3.v; }
  |Quot { exp$1.v = (exp$3.v == 0)
                  ? exp$2.v
                  : (exp$2.v / exp$3.v);
          local STR error;
          error   = (exp$3.v == 0)
                  ? "←Division By 0→"
                  : "";
        }
  |Const { exp.v = INT; }
  ;
```

The equation associated with the operator NullExp defines the value of an unexpanded expression to be 0. The equations associated with the operators Sum, Diff, and Prod define the value attribute as one would expect for such operators. Because the equations of the grammar must be total, the first equation associated with the operator Quot defines the value of a quotient with denominator 0 to be the value of the numerator. The notation

expression ? expression : expression

denotes a conditional expression.

The second equation for operator Quot illustrates the use of *local attributes*. Local attributes of a production permit defining a computation in one operator of a phylum without imposing the requirement that definitions be provided in every production of the phylum, as would be the case, for example, if the error attribute were a synthesized attribute of the phylum exp.

4. Defining semantic domains

As already illustrated above, primitive phyla (such as INT) serve both as atoms in abstract syntax and as primitive attribute types. One must also provide a way to define new attribute types from these primitive ones. In other attribute grammar systems, the language for defining new attribute types is distinct from the grammatical mechanism for defining syntax [Paulson 1981, Kastens et al. 1983]. In the Synthesizer Generator, precisely the same sort of rules are used to define new attribute types and abstract syntax; as in SIS [Mosses 1979] and as in a recent proposal by Ganzinger and Giegerich [Ganzinger & Giegerich 1984], there is a uniform treatment of syntactic and semantic domains. Thus, the abstract-syntax tree being edited and the attributes attached to its nonterminals are all just typed objects in a unified domain.

Example. Suppose our desk calculator were extended with *let expressions* of the form

let <name> = <exp> in <exp>

which evaluates the second expression with the name bound to the value of the first expression. The scoping of names is block-structured, so each exp must be evaluated in an environment of appropriate local name bindings provided by an (inherited) environment attribute. A representation for such environment attributes as a list of identifier-value pairs is defined by giving abstract-syntax rules to define phylum ENV:

```
ENV: NullEnv()
  |EnvConcat(BINDING ENV)
  ;
BINDING: Binding(ID INT);
ID: < [#] | [a-zA-Z][a-zA-Z]* >;
```

Paraphrased, an ENV is either the null list of bindings, or it is a BINDING concatenated with an ENV. A BINDING is an ID,INT pair. The definition of the phylum ID illustrates regular expressions, which permit defining subclasses of the phylum STR. An ID is either a hash mark, or it is an alphabetic string. (The hash mark will represent an unknown name).

As we have already seen, the expression language for semantic equations permits infix expressions computing primitive values from other primitive values. Now, following a style originally proposed in [Burstall 1969], the (user-defined) operator names are employed both as constructors and as discriminators:

(1) A k-ary operator can be applied to k argument expressions of the appropriate phyla to construct a new, composite object. For example, the operator Binding may be used to create an object such as Binding("#",0).

(2) A composite object of a given phylum can be analyzed by a multi-branch selection expression (in our terminology a *with expression*), where the different cases are labeled by operator names. Each alternative of a with expression is a local scope, and variable names appearing in argument positions of a case-selection operator are bound to the appropriate constituents within that scope. For example, a value of phylum ENV can be analyzed using a with expression of the form:

```
with (env) {
   NullEnv(): . . . ,
   EnvConcat(b, e): . . .
   )
```

The variables b and e would be bound to the first and second components, respectively, in the EnvConcat branch of such a with expression.

Example. The recursive function lookup, which returns the binding for a given identifier if it exists in the given environment and returns Binding("#", 0) if the identifier does not exist in the environment, is written as follows, using two with expressions:

```
BINDING lookup(id,env)
   ID id; ENV env;
   {
   return(
      with (env) (
         NullEnv(): Binding("#", 0),
         EnvConcat(b, e): with (b) (
               Binding(s, i): (id==s ? b : lookup(id, e)))
         )
      );
   };
```

5. Supporting modular specifications

The Synthesizer Generator's specification language incorporates notation that permits the specifications of separate aspects of a language to be placed in separate portions of a specification. Such modularity enhances the comprehensibility of editor specifications. It also facilitates the generation of a collection of related editors that offer different degrees of static-semantic analysis for different language dialects. Three concepts permit specifications to be factored in this way: (1) the specification language allows us to add a new attribute to an existing phylum, (2) it allows us to add new semantic equations to existing operators, and (3) it allows us to add new operators and their semantics to an existing phylum.

Example. The definitions given in the previous section for the phyla ENV, BINDING, ID, and the function lookup, together with the rules given below, extend the desk calculator with the abstract syntax and evaluation semantics for let expressions and the use of bound names. These definitions may be factored into a module separate from the rules of Section 2 and 3, which define expression evaluation.

First, we add to the previously defined nonterminal exp, an additional declaration for an inherited environment attribute env of type ENV. A new nonterminal for the <name> in a let expression is also required.

```
exp  { inherited ENV env; };
name { synthesized ID  id; };
```

Second, the semantics of the existing arithmetic operators is extended by adding semantic equations to pass the inherited environment to both the left and right operands.

```
exp: Sum, Diff, Prod, Quot {
      exp$2.env = exp$1.env;
      exp$3.env = exp$1.env;
      }
  ;
```

Third, new operators are added to the existing phyla exp for both the let expression and the use of a bound name.

```
exp: Let(name exp exp) {
      exp$2.env = exp$1.env;
      exp$3.env = EnvConcat(Binding(name.id, exp$2.v),
                            exp$1.env);
      exp$1.v   = exp$3.v;
      }
   |Use(ID) {
      local STR error;
      error = with(lookup(ID, exp.env))(
               Binding(s, i): s==="#"
                              ? "←Undefined"
                              : ""
               );
      exp.v = with(lookup(ID, exp.env))(Binding(s, i): i);
      }
   ;
name: NullDef() { name.id = "#"; }
   | Def(ID)  { name.id = ID;  }
   ;
```

The specification of the desk calculator's underlying (attributed) abstract syntax is completed by identifying the root symbol and giving its semantics:

```
root calc;
calc: Top(exp) { exp.env = NullEnv(); };
```

6. Defining user interfaces

Thus far, we have defined only an attributed abstract syntax for edited objects; we turn now to specification of their external representations.

6.1. Unparsing schemes

The display of an object is defined by an unparsing scheme given for each production consisting of a sequence of strings, names of attribute occurrences, and names of right-side nonterminals. The display is generated by a left-to-right traversal of the tree that interprets these unparsing schemes. Formatting is defined by control characters that can be included in the strings of an unparsing scheme. For example, the character \n means "line-feed carriage-return to the current left-margin".

Example. The following unparsing scheme specifies that an expression of the desk calculator is to be displayed, fully parenthesized, together with the value computed for the expression, the attribute exp.v of the operator Top.

```
calc: Top [ exp "\nVALUE = " exp.v ]
  ;
exp: NullExp [ "<exp>" ]
  |Sum [ "(" exp$2 "+ " exp$3 ")" ]
  |Diff [ "(" exp$2 "-" exp$3 ")" ]
  |Prod [ "(" exp$2 "*" exp$3 ")" ]
  |Quot [ "(" exp$2 "/" error exp$3 ")" ]
  |Let [ "(let " name " = " exp$2 " in " exp$3 ")" ]
  |Use [ ID error ]
  |Const [ INT ]
  ;
name: NullDef [ "<name>" ]
  |Def [ ID ]
  ;
```

The display is also annotated with error messages at the locations of undefined names and quotients with zero-valued denominators. This is specified above by incorporating the "error" attributes in the unparsing schemes for Quot and Use. With this unparsing scheme, the abstract tree Top(Let(x, Quot(1, 0), Sum(x, y))) would be unparsed as:

```
(let x = (1/←Division By 0→0) in (x+ y←Undefined))
VALUE = 1
```

The incremental attribute evaluation implicit in every editing modification guarantees that each unparsing reflects appropriate error messages (null strings when no error exists) and the correct value of the expression.

Note that print representations of attribute values are also defined with unparsing schema, by giving schema associated with each phyla's abstract syntax.

6.2. Defining input interfaces

For the purpose of specifying flexible input interfaces, we have availed ourselves of the full power of the attribution mechanism. In particular, we allow *any* grammar rule (including syntactic type definitions) to be a valid attribute type definition, and we let syntactic values be constructed by attribute computations.

To specify the input interface, productions are given for a concrete input syntax, along with semantic equations that define a translation to abstract syntax. The *tilde rules* specify the correspondance between cursor positions in the abstract syntax tree and entry points within the concrete syntax; a tilde rule of the form "n ~ N.a;" says that when the cursor is positioned at nonterminal n in the abstract-syntax tree, input is to be parsed as an N, and attribute a is to be inserted in the abstract-syntax tree at the position of the editing cursor.

This mechanism for translating input text to an abstract-syntax tree provides an editor designer with the ability to define textual and structural interfaces in whatever balance is desired. As input languages need not be restricted to legal fragments of programs, a degree of input error tolerance can also be specified.

Example. Continuing our desk-calculator example, suppose we want a textual interface for the arithmetic expressions, and a structural interface for let expressions using the command "=". The following rules define a concrete syntax for expression input as an "Exp", and designate "Exp" as an entry point to the parser when the cursor is positioned at an "exp" nonterminal in the abstract syntax tree.

```
Exp { synthesized exp abs; };

exp ~ Exp.abs;

INTEGER: < [0-9]+ >;

left '+' '-';
left '*' '/';

Exp: Exp '+' Exp
       { Exp$1.abs = Sum(Exp$2.abs, Exp$3.abs); }
  | Exp '-' Exp
       { Exp$1.abs = Diff(Exp$2.abs, Exp$3.abs); }
  | Exp '*' Exp
       { Exp$1.abs = Prod(Exp$2.abs, Exp$3.abs); }
  | Exp '/' Exp
       { Exp$1.abs = Quot(Exp$2.abs, Exp$3.abs); }
  | INTEGER { Exp.abs = Const(STRtoINT(INTEGER)); }
  | ID { Exp.abs = Use(ID); }
  | '(' Exp ')' { Exp$1.abs = Exp$2.abs; }
  ;
```

In this example, the translation of concrete syntax to abstract syntax uses only synthesized attributes, but inherited attributes may be used as well. Note that concrete syntax can be specified with ambiguous productions and disambiguating precedence rules, as is done above for the tokens '+', '-', '*', and '/'.

To define the commands for template-style insertion that are legal when the cursor is positioned at an exp nonterminal, we give an second set of input rules:

```
ExpCommand { synthesized exp abs; };

exp ~ ExpCommand.abs;

ExpCommand: '=' {
  ExpCommand.abs = Let(NullDef(),NullExp(),NullExp());
  };
```

We also permit *context-sensitive translations* by allowing attributes at the cursor position in the abstract tree to be inherited into the parse tree. For example, the following additional rules define input of the form ".name" to mean "insert, at the current cursor position, the value of the given name in the scope containing the cursor":

```
ExpCommand { inherited ENV env; };

exp ~ ExpCommand.abs
  { ExpCommand.env = exp.env; };

ExpCommand: '.' ID {
  ExpCommand.abs =
        with(lookup(ID, ExpCommand.env)) {
              Binding(s, i): Const(i)
              };
  };
```

In the tilde rule above, exp.env refers to the environment attribute of the exp nonterminal at which the cursor is positioned when the command ".name" is typed. Note that the typed text is not a fragment of the concrete representation of our desk calculator language; rather, it is an editor command interpreted in the context of the current cursor position. In an editor for a language such as Pascal, templates that depend on user defined types and procedures can be defined in a similar fashion.

7. Syntactic references within semantics

It is often the case that a piece of the abstract-syntax tree is itself a sufficiently convenient representation of a semantic value needed for attribute computations. In a system such as GAG, where a syntax tree is a different sort of value from an attribute value, one must resort to replicating the syntactic tree in the semantic domain. However, as a result of defining attribute types with grammar rules, we can permit attribute values to refer to and to perform computations on syntactic components. In fact, scrutiny of the semantic equations in Section 3 will reveal that we have already been using this feature to define the exp.v attribute in the equations associated with the Const operator.

Example. Consider extending the desk calculator with a block-structured, editing-macro facility allowing a name to be bound, not to a value, as in the let expression, but to a symbolic expression. To represent such bindings in the environment attributes we must extend the phylum BINDING with an additional operator:

```
BINDING: MacroBinding(ID exp);
```

We then extend the phylum exp with a new operator for macro definitions:

```
exp: Macro(name exp exp) {
   exp$3.env = EnvConcat(MacroBinding(name.id, exp$2),
                         exp$1.env);
   exp$1.v = exp$3.v;
   };
```

Note that the macro binding created by the first equation uses a syntactic reference to exp$2. We also extend our previous input rule so that a macro body can be inserted by invoking the macro name:

```
ExpCommand: '.' ID {
  ExpCommand.abs =
        with(lookup(ID, ExpCommand.env)) (
                Binding(s, i): Const(i),
                MacroBinding(s, e): e
                );
  };
```

Finally, the lookup function needs to be modified to handle the additional binding operator (not shown).

Allowing syntactic references within semantic equations complicates the problem of incremental attribute updating, but not unduly. The value of an attribute defined with a syntactic reference to node N may become inconsistent whenever a modification is made inside the subtree rooted at N; consequently, a modification at node M may introduce inconsistencies in attributes at nodes along the "spine" of the tree from M to the tree's root, rather than just in the immediate neighborhood of M. However, an incremental updating algorithm can be applied as long as the region containing all initial inconsistent attributes is known [Reps 1984, Reps et al. 1983]. Note that the inconsistent region does not necessarily extend all the way to the tree's root, for only certain productions have equations with syntactic references.

Although in general concrete and abstract syntaxes are distinct, in practice, they are nearly isomorphic (especially when concrete syntax can be specified with ambiguous productions and disambiguating precedence rules). This leads to the possibility of uniting phyla for abstract and concrete syntax (e.g. exp and Exp), with self-referential syntactic references defining the abs attribute in productions where the parse tree and abstract tree are isomorphic. The result is a far more succinct specification.

8. Computing with hierarchies of attributable trees

In Section 4 we discussed how attribute types are defined with grammar rules, giving the specification language a uniform approach to defining structured data in the system. In Section 6, this approach allowed us to define translations from concrete to abstract syntax according to semantic equations of the concrete-syntax grammar. In doing so, we made the additional assumption that the grammar rules defining *syntactic types* were valid definitions of *attribute types*. The consequence of this step is that the denotable values in our specification language are attributable, tree-structured objects whose attributes are themselves attributable, tree-structured objects, *ad infinitum*.

However, with the language operations described thus far, the power of such attribute hierarchies cannot be fully exploited. The expression language permits infix expressions computing primitive values, construction expressions computing composite values, and selection expressions; missing from the language is a way of forcing the attribution of a (previously unattributed) structure. To fill this need, the specification language permits *attribution expressions* of the form "*expression*{*equations*}.*attribute*". The value of such an expression is computed as follows: (a) the *expression* is evaluated, yielding some attributable (but as yet unattributed) object S, (b) the inherited attributes of S are defined by the given *equations*, (c) the value of S.*attribute* is computed by demand and is returned.

Example. In Section 4, the lookup operation was written as a recursive procedure that received an environment as argument. Alternatively, it could be written as an attribute computation of environment objects themselves by letting each ENV object inherit the argument id to be looked up, and synthesize the appropriate found binding.

```
ENV { inherited ID target;
      synthesized BINDING result;
    };
```

```
ENV: NullEnv() { ENV.result = Binding("#", 0); }
  |EnvConcat(BINDING ENV) {
      ENV$1.result = with(BINDING) (
          Binding(s,v): s == ENV$1.target
              ? BINDING : ENV$2.result
          );
      ENV$2.target = ENV$1.target;
      }
  ;
```

Calls on lookup(id,exp.env) to find the binding of id in the environment env are then replaced by the attribution expression

```
env{env.target = id;}.result
```

As a second example of a computation using an attribute hierarchy, we return to the macro example of Section 6 and consider how macro uses are evaluated. The semantics of the Use operator is revised so that if it is a macro use, then e, the exp bound to the macro name, is extracted from the environment attribute and e *itself* is attributed in the environment of the use in order to find its value v:

```
exp: Use(ID) {
      exp.v = with(lookup(ID, exp.env))(
          Binding(s, i): i,
          MacroBinding(s, e): e{e.env = exp.env;}.v
          );
      };
```

We note that this approach allows us to support a SMALLTALK-like, object-oriented programming style in editor specifications, insofar as an attribute that is itself an attributed tree can be viewed as an object able to respond to messages that provide its inherited attributes and demand its synthesized attributes.

References

[Burstall 1969]
Burstall, R.M. Proving properties of programs by structural induction. *Comp. J. 12*, 1 (Feb. 1969), 41-48.

[Demers et al. 1981]
Demers, A., Reps, T., and Teitelbaum, T. Incremental evaluation for attribute grammars with application to syntax-directed editors. In Conference Record of the 8th ACM Symposium on Principles of Programming Languages, Williamsburg, Va., Jan. 26-28, 1981, pp. 105-116.

[Donzeau-Gouge et al. 1975]
Donzeau-Gouge, V., Huet, G., Kahn, G., Lang B., and Levy, J.J. A structure-oriented program editor. Rep. No. 114, IRIA-LABORIA, Rocquencourt, France, Apr. 1975.

[Ganzinger & Giegerich 1984]
Ganzinger, H. and Giegerich, R. Attribute coupled grammars. To appear in Proceedings of the SIGPLAN Symposium on Compiler Construction, Montreal, Can., June 20-22, 1984.

[Johnson 1983]
Johnson, G.F. An approach to incremental semantics. Ph.D. dissertation, Dept. of Computer Science, Univ. of Wisconsin, Madison, Wisc., 1983.

[Kastens et al. 1982]
Kastens, U., Hutt, B., and Zimmermann, E. *Lecture Notes in Computer Science*, vol. 141: *GAG: a Practical Compiler Generator.* Springer-Verlag, New York, 1982.

[Knuth 1968]
Knuth, D.E. Semantics of context-free languages. *Math. Sys. Theory 2*, 2 (June 1968), 127-145. Correction. *ibid. 5*, 1 (Mar. 1971), 95-96.

[Medina-Mora & Notkin 1981]
Medina-Mora, R. and Notkin, D.S. ALOE users' and implementors' guide. Tech. Rep. CMU-CS-81-145, Dept. of Computer Science, Carnegie-Mellon Univ., Pittsburgh, Pa., Nov. 1981.

[Mosses 1979]
Mosses, P. SIS -- Semantics Implementation System: Reference manual and user guide. Tech. Rep. DAIMI MD-30, Computer Science Dept., Aarhus Univ., Aarhus, Denmark, Aug. 1979.

[Paulson 1981]
Paulson, L. A compiler generator for semantic grammars. Ph.D. dissertation, Dept. of Computer Science, Stanford Univ., Stanford, Calif., Dec. 1981.

[Ramanathan & Soni 1980]
Ramanathan, J. and Soni, D. The model for program development and analysis used in TRIAD. Tech. Rep. TRIAD-TR1-80, Dept. of Computer and Information Science, Ohio State Univ., Columbus, Ohio, May 1980.

[Reps 1984]
Reps, T. *Generating language-based environments.* The M.I.T. Press, Cambridge, Mass., 1984.

[Reps 1983]
Reps, T. Static-semantic analysis in language-based editors. In Digest of Papers of the IEEE Spring CompCon 83, San Francisco, Calif., Mar. 1983, pp. 411-414.

[Reps & Alpern 1984]
Reps, T. and Alpern, B. Interactive proof checking. In Conference Record of the 11h ACM Symposium on Principles of Programming Languages, Salt Lake City, Utah, Jan. 15-18, 1984, pp. 36-45.

[Reps et al. 1983]
Reps, T., Teitelbaum, T., and Demers, A. Incremental context-dependent analysis for language-based editors. *ACM Trans. Program. Lang. Syst. 5*, 3 (July 1983), 449-477.

[Teitelbaum & Reps 1981]
The Cornell Program Synthesizer: A syntax-directed programming environment. *Commun. ACM 24*, 9 (Sept. 1981), 563-573.