An Object-Oriented Notation for Attribute Grammars

GÖREL HEDIN

Department of Computer Science, Lund University
Box 118, S-221 00 Lund, Sweden

email: gorel@dna.lth.se

Abstract:
This paper presents an attribute grammar notation which is based on the object-oriented concepts of classification hierarchies, inheritance, and late binding. The notation allows compact and flexible language specification through the use of inheritance and equation overriding. Furthermore, demand attributes can be implemented efficiently by using a technique similar to the one used for virtual procedures in Simula. Such attributes are important especially in incremental language-based environments as they do not consume storage. The notation also makes it possible to define general attributes which can be accessed without knowledge of the particular language modelled by the grammar. This can be utilized for integration of grammar independent tools. The notation is based on a single-inheritance classification, and a discussion is given on the problems which would arise if the notation was augmented to multiple-inheritance.

Keywords:
object-oriented, attribute grammars, demand attributes, Simula, language-based environments

1 INTRODUCTION

In language-based environments, programs are usually represented as abstract syntax trees (ASTs). Various language-generic tools can be designed which analyze or manipulate the ASTs, e.g. structure-editors, parsers, unparsers, static semantic analyzers, data flow analyzers, code generators, etc. The language specific parts of such tools are specified in grammars.

The core in a language specification is the context free abstract grammar which describes the structure of ASTs. The abstract grammar is usually formalized as some variant on a typed AND/OR grammar. AND productions describe the structure of a node type (i.e. how many children it has and what their types are) and OR productions describe union node types. One such formalism is the operator/sorts formalism which was introduced in the Mentor project [DHKLL 75]. Derivatives of this formalism are used in many language-based environments, e.g. the Gandalf environment [M 82] and the Cornell Synthesizer Generator [RT 84].

The abstract grammar is augmented with information used by the various language-generic tools. This is usually done by associating attribute declarations, rules, actions, schema, etc. to the productions of the abstract grammar. Attribute grammars are especially attractive for this purpose because of their high-level declarative nature; equations define in a declarative manner relations between the values of attributes of neighbor nodes in an AST. Furthermore, incremental attribute evaluators can be constructed automatically from the attribute grammar [RTD 82], [Y 83].

In this paper, we present an object-oriented formalism for attribute grammars. We have developed and used this formalism as a specification language in the incremental Mjølner[1] programming environment [HM 88]. The formalism has primarily been used to drive an incremental static semantic checker [H 88], but also to provide services for other tools such as the code generator and interactive interrogation facilities. Grammars have been developed for Simula [DMN 68], for various Algol-like toy languages, and also for some parts of Beta [KMNN 87].

The abstract grammar in our formalism is based on classifying the productions into a typed single-inheritance hierarchy. This involves reformulating the traditional AND/OR productions as a specialization hierarchy. A similar object-oriented view on the abstract grammar is used also in e.g. [MN 88], [N 87], [CS 87], and [TTTI 88]. The novel contribution of this paper is to bring the attributes and equations of attribute grammars into an object-oriented framework.

Attributes and equations are defined in the productions and inherited along the classification hierarchy, similarly to variables and procedures in a class hierarchy. We show examples of how the advantages of object-oriented notations can be applied to this field: Language concepts can be modelled in a simple and natural way by specializations; default behavior (in the form of attribute equations) can be specified and overridden in specialized productions. This results in a more compact and more readable specification than in the traditional attribute grammar formalisms. The notation also includes demand attributes, i.e. attributes which are not stored but instead evaluated whenever they are needed. The object-oriented notation allows demand attributes to be implemented efficiently, using late binding techniques. We have found these attributes to be very useful in our incremental system.

Attributes declared in the most general production will appear in all nodes. Such attributes can be accessed by tools which have no knowledge of the grammar. The object-oriented notation plays an important role here, since it allows the attributes to be specified very easily. In a traditional notation, the grammar would have to be cluttered with trivial attribute declarations and equations, and one would simply not think of using attributes for these purposes.

The synthesis of attribute grammars with object orientation leads to an interesting aspect on multiple inheritance. It turns out that multiple inheritance in general leads to grammars which are not well formed. A discussion of these problems is given in section 5.

---

[1]The Mjølner project is a Nordic research project involving companies and universities in Denmark, Norway, and Sweden. *The goal of the Mjølner project is to develop interactive programming environments for object-oriented languages such as Simula [DMN 68], Beta [KMNN 87], and OSDL [MBD 87]. The incremental Mjølner system is a part of this project. A description of the project as a whole is given in [DLMM 87].*

The rest of this paper is structured as follows: Section 2 describes the grammar formalism with abstract syntax, attributes and equations. Section 3 discusses access of general attributes by grammar independent tools. Section 4 describes our implementation of access to attributes. Section 5 discusses well-formedness of attribute grammars in connection with inheritance, and section 6 concludes the paper.

2 GRAMMAR FORMALISM

2.1 Abstract Grammar

We regard the nodes in an AST as instances of productions in a grammar. The productions are similar to classes, and are arranged in a classification hierarchy. In analogy to subclasses and superclasses, we talk of subproductions and superproductions. Figure 1 shows the hierarchy of productions used in the examples in this paper. Each production corresponds to a language concept, and the classification hierarchy is the generalization/specialization hierarchy of these concepts. E.g. the top production models the concept of a general node in an AST. This concept is specialized into root nodes and descendant nodes. Expressions, statements, and declarations are examples of descendant nodes. The while and assignment statements are examples of specialized statements.

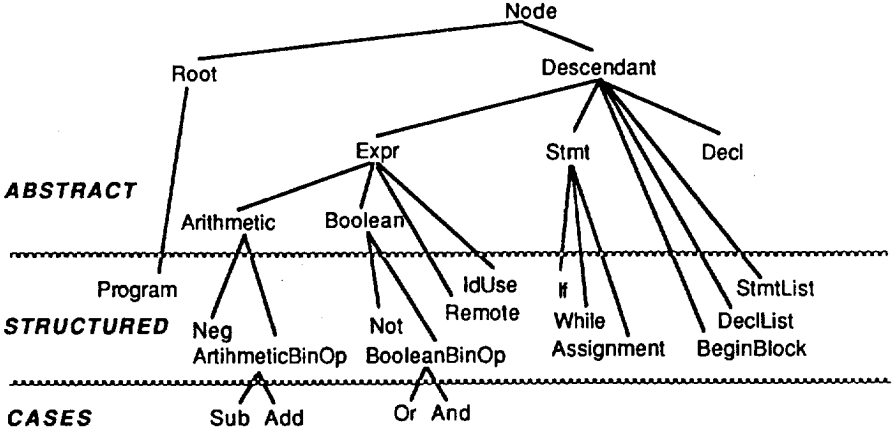

Figure 1. A production classification hierarchy.

The productions come in three kinds: abstract, structured, and cases.

- The *abstract* productions are used to model concepts for which the structure is not known. E.g. different statements have different structure.
- The *structured* productions are used to model language concepts with a particular structure. E.g. the while-statement has a certain structure, namely an expression node representing the predicate, and a statement list node representing the do-part.
- The *case* productions are used to model specializations of structured productions. The case productions inherit common structure and behavior from their structured superproduction. This is useful when modelling binary operators with similar behavior.

In comparison with the traditional AND/OR grammars, the abstract productions correspond to OR-rules and the structured productions to AND-rules. The classification hierarchy corresponds to the hierarchy of OR-rules.

Structured productions come in three kinds: constructions, lists, and lexemes. This is similar to e.g. the GRAMPS formalism [CI 84].

- A *construction* has a fixed number of son nodes with given names and productions.
- A *list* has a variable number of son nodes of a given production.
- *Lexemes* are used to model lexical information such as identifiers, numerical constants, and text constants. Each lexeme node has an intrinsic attribute "string" which contains this information.

The following notation is used to declare productions of the different kinds. Here P is the name of the production, S its superproduction (optional), C the production of a son node, and t the name of a son node. Boldfaced words are keywords in the formalism and italic words are comments.

```
<P> : <S> ::= Abstract                                  -- abstract --
<P> : <S> ::= <C>*                                      -- list --
<P> : <S> ::= {<t1:C1> & <t2:C2> & ... <tn:Cn>}         -- construction --
<P> : <S> ::= Lexeme                                    -- lexeme --
<P> : <S> ::= Case                                      -- case --
```

The following example shows some of the productions from figure 1 and contrasts the object-oriented notation with the traditional AND/OR notation:

Object-oriented notation:

```
<Stmt>:<Descendant> ::= Abstract
<Expr>:<Descendant> ::= Abstract
<StmtList>:<Descendant> ::= <Stmt>*
<While>:<Stmt> ::=
      {<predicate:Expr> & <doPart:StmtList>}
<IdUse>:<Expr> ::= Lexeme
<Arithmetic>:<Expr> ::= Abstract
<ArithmeticBinOp>:<Arithmetic> ::=
      {<left:Expr> & <right:Expr>}
<Add>:<ArithmeticBinOp> ::= Case
```

AND/OR notation:

```
<Stmt> ::= <While> | ...
<Expr> ::= <IdUse> | <Arithmetic> | ...
<StmtList> ::= <Stmt>*
<While> ::=
      {<predicate:Expr> & <doPart:StmtList>}
<IdUse> ::= some lexical definition
<Arithmetic> ::= <ArithmeticBinOp> | ...
<ArithmeticBinOp> ::= <Add> | ...

<Add> ::= {<left:Expr> & <right:Expr>}
```

The *Stmt* and *Expr* productions are abstract subproductions of Descendant. *StmtList* is a list production with son nodes of production Stmt. *While* is a construction production. It has two sons, one Expr son named "predicate" and one StmtList son named "doPart". *IdUse* is a lexeme production which models an identifier reference. *ArithmeticBinOp* is a construction production with two Expr sons, one named "left" and one named "right". The *Add* production is a case production. It inherits its structure from ArithmeticBinOp.

The classification hierarchy constitutes a type system where subproductions are subtypes of their superproductions. If a construction production declares a son node to be of production P, this means that the son node is *at least* of production P and possibly of a more specialized production. This type system is exactly analogous to the one in Simula. The type system governs which node combinations are syntactically legal. E.g. the son node named *"predicate" in the While statement is declared to be of production Expr. It is thus syntactically*

legal for this node to be an instance of Expr, or an instance of any of the specializations of Expr, e.g. IdUse or Add.

2.2 Attribute Grammars

This section gives a short background to attribute grammars and introduces some terminology used later on in the paper.

Attribute grammars were introduced by Knuth [K 68] as a means to specify the semantics of context-free languages. The idea is to represent context-sensitive information as *attributes* of the nodes in an AST. An attribute value is defined by a function applied to neighbor node attributes. Such definitions, or *equations*, are associated with the productions and have the following form:

$$a_1 := f(a_2, a_3, \ldots a_k)$$

The attribute on the left hand side (a_1) is said to be *defined* by the equation, whereas the attributes on the right hand side ($a_2, a_3, \ldots a_k$) are arguments to the function f and are said to be *accessed* by the equation. Furthermore, a_1 is said to *depend* on ($a_2, a_3, \ldots a_k$) since their values are needed in order to evaluate a_1. Although the equation syntax is similar to an assignment statement, the order between equations is irrelevant. The order of evaluation is calculated automatically from the attribute dependencies. Numerous attribute evaluation algorithms exist in the literature, see e.g. [E 84].

An attribute is classified as synthesized, inherited, or local, depending on where it is defined and accessed. For the purpose of this paper, however, we use the term "ancestral attribute" instead of the standard "inherited attribute" since we use the term "inherited" in the object-oriented sense.

- A *synthesized* attribute is defined by an equation in the node, but it is accessed by equations in the ancestor node.
- An *ancestral* attribute is defined by an equation in the ancestor node, but it is accessed by equations in the node itself.
- A *local* attribute is both defined and accessed by equations in the node itself.

2.3 Attributes and Equations

This section introduces the notation used for attributes and equations in our object-oriented formalism. The attributes and equations are declared in the productions and are inherited by subproductions. An equation which defines a particular attribute will override equations in any superproduction defining the same attribute. The equations are in this way analogous to late bound procedures of object-oriented languages. Attributes are analogous to instance variables.

The following notation is used to declare attributes. Here a is the name of an attribute and T is its type:

```
Anc  a : T      -- declaration of ancestral attribute --
Syn  a : T      -- declaration of synthesized attribute --
Loc  a : T      -- declaration of local attribute --
```

In addition to the attributes declared in the grammar, there are a number of intrinsic attributes. All lexeme nodes have an attribute `string` which holds the lexical string. All list nodes have `first` and `last` attributes which hold the indices of the first and last son nodes. Furthermore, all nodes have an attribute `myIndex` which is the index of the node relative to its ancestor node. The intrinsic attributes are treated as automatically defined local attributes.

Attributes declared in a production (or in any of its superproductions) are referred to directly by their name. Attributes declared in productions (or superproductions) of son nodes are referred to by dot notation "`t.a`" where `t` is the name of the son node and `a` the name of the attribute. Equation (1) below thus defines a P-node's local attribute `a` to be a function of its son node `t1`'s synthesized attribute `b`:

```
<P> ::= {<t1:Q> & <t2:Q>}
      Loc  a : aType;
      a := f(t1.b);              -- (1) --
<Q> ::= Abstract
      Syn  b : bType;
```

An equation is either simple or multiple. A simple equation defines the value of one attribute. A multiple equation defines the value of an attribute of all son nodes of a particular production. The multiple equation below defines the attribute `a1` of all son nodes of production P. The `x` is the index of the son node to which the equation applies:

```
a1 := f(a2, a3, ... ak);                    -- simple equation --
for all sons(x) in P                        -- multiple equation --
      son(x).a1 := f(a2, a3, ... ak, x);
```

The multiple equations are typically used in list productions to define ancestral attributes of the son nodes. They are also used in abstract productions to give default equations for ancestral attributes of son nodes of a particular production. Examples of this will be given later on.

The language used to define attribute types and functions is not part of the basic notation and is not important to the ideas presented in this paper. Specific types and functions will therefore be introduced in the examples as needed.

2.4 An Example

A typical application of attribute grammars is to define the static semantics of a block structured language. The usual approach is to represent the symbol environment as an ancestral attribute, which is propagated down in the AST and used in identifier reference nodes to look up types and other information. The type information about variables and expressions is represented as synthesized attributes which are propagated up in the AST. Static semantic errors, e.g. illegal types in expressions, are represented by local attributes in the appropriate nodes. This section illustrates how to specify such a language in the object-oriented notation.

The environment information has to be spread throughout practically the whole AST in order to reach all identifier references. In a traditional attribute grammar, an environment attribute has to be explicitly declared and propagated in all productions. Our object-oriented notation allows this to be specified in a more compact way by using a default equation for the normal propagation. For those productions which change the environment (e.g. blocks introducing

new scopes), the default equation is overridden. The following productions show the approach:

```
-------------------------------------------------------------
<Node> ::= Abstract
-------------------------------------------------------------
<Root>:<Node> ::= Abstract
      Loc  rootEnv : Environment;              -- (1) --
      rootEnv := EmptyEnvironment;             -- (2) --
      for all sons(x) in Descendant            -- (3) --
            son(x).env := rootEnv;
-------------------------------------------------------------
<Descendant>:<Node> ::= Abstract
      Anc  env : Environment;                  -- (4) --
      for all sons(x) in Descendant            -- (5) --
            son(x).env := env;
-------------------------------------------------------------
<Program>:<Root> ::= {<mainblock:BeginBlock>}
-------------------------------------------------------------
<BeginBlock>:<Descendant> ::=
      {<declPart:DeclList> & <stmtPart:StmtList>}
      Loc  localEnv : Environment;             -- (6) --
      localEnv := declPart.assembledEnv;       -- (7) --
      stmtPart.env := localEnv;                -- (8) --
-------------------------------------------------------------
```

- The *Node* production models general nodes in the whole AST. It is specialized into *Root* and *Descendant* representing root nodes of ASTs and their descendants respectively. We will assume that all productions in this grammar will be specializations of either Root or Descendant. The specializations of Root will be used as start productions.
- All *Descendant* nodes have an ancestral attribute `env` (4). Among descendant nodes, the default equation is to propagate the same `env` value to all son nodes. This is expressed by the multiple equation (5).
- The *Root* production has a local attribute `rootEnv` (1) which is an empty environment (2). This attribute is propagated to the son nodes (3).
- *Program* is the start production for an AST representing a whole program. It inherits its behavior from the Root production.
- The *BeginBlock* production models an Algol-style block with a declaration part and a statement part. The BeginBlock has an equation defining the `env` attribute of the statement part (8), overriding the default equation given in the Descendant production. Note however that the default equation still applies to the declaration part. The local environment passed down to the statement part is stored in a local attribute `localEnv` (6). The `localEnv` equals the `assembledEnv` attribute of the declaration part (7). The `assembledEnv` is computed by adding the local declarations to the environment of the BeginBlock (this is not shown in the example).

The default equation in production Descendant is appropriate for almost all productions in the grammar, e.g. for statements such as While, If, Assignment, etc. and for expressions such as And, Or, Add, Subtract, etc. These productions need neither to declare the `env` attribute nor to propagate the value to their sons, since they inherit this behavior from the Descendant production. The default equation is overridden when needed, as in the BeginBlock production above and the Remote production in the expression part of the grammar below.

We now turn to type checking. Type checking is needed to check e.g. that the predicate in a while-statement is boolean, and that the operands of an addition are arithmetic. We do this by introducing a synthesized attribute `type` in all expressions and local `error` attributes in places where a type checking error can occur:

```
-------------------------------------------------------------
<StmtList>:<Descendant> ::= <Stmt>*
-------------------------------------------------------------
<Stmt>:<Descendant> ::= Abstract
-------------------------------------------------------------
<Expr>:<Descendant> ::= Abstract
      Syn  type : TypeDescriptor;                    -- (1) --
      type := UnknownType;                           -- (2) --
-------------------------------------------------------------
<While>:<Stmt> ::= {<predicate:Expr> & <doPart:StmtList>}
      Loc  error : boolean;                          -- (3) --
      error := predicate.type <> BooleanType and     -- (4) --
               predicate.type <> UnknownType;
-------------------------------------------------------------
<IdUse>:<Expr> ::= Lexeme
      Loc  error : boolean;                          -- (5) --
      Loc  descr : DeclarationDescriptor;            -- (6) --
      descr := Lookup(env, string);                  -- (7) --
      type := TypeOf(descr);                         -- (8) --
      error := descr = UndeclaredDescriptor;         -- (9) --
-------------------------------------------------------------
<Remote>:<Expr> ::= {<object:Expr> & <accessor:IdUse>}
      Loc  accEnv : Environment;                     -- (10) --
      accEnv := f(env, object.type);                 -- (11) --
      accessor.env := accEnv;                        -- (12) --
      type := accessor.type;                         -- (13) --
-------------------------------------------------------------
```

- *StmtList* models a list of statements and *Stmt* models a general statement.
- The *Expr* production models a general expression. It declares a synthesized attribute `type` (1) which is used for type checking. An equation defines `type` to have the value UnknownType (2) since nothing is known about the type of a general expression. UnknownType is considered to be compatible with all other types.
- The *While* production declares a local attribute `error` (3). The While production checks the type of the predicate (which should be either boolean or unknown) (4).
- The *IdUse* production models an identifier reference. It inherits the synthesized attribute `type` from the Expr production. It uses its `env` attribute and the intrinsic attribute `string` to look up the appropriate declaration description (7). The declaration description is used to determine the type of the identifier (8) in an equation which overrides the one defined in the Expr production. A local attribute `error` is *true* if the identifier is undeclared (9).
- The *Remote* production models remote access to a part (e.g. variable or procedure) of an object. The environment of the accessed part depends on the type of the object and a function f calculates an environment for the accessor (11). The equation (12) defines the `env` attribute of the accessor, overriding the default equation given in the Descendant production. Note however that the default `env` equation still applies to the object. The resulting type of the Remote expression equals the type of the accessor (13), overriding the equation in the Expr production.

The grammar is by no means complete, but it gives a flavor of how the static semantics of a language can be specified in the object-oriented notation.

2.5 Demand Attributes

The high consumption of storage is a well known problem for systems based on attribute grammars. The problem is especially severe for incremental systems where attribute values are saved in order to allow incremental update. The incremental Mjølner system relies heavily on *demand attributes* to reduce the storage consumption. Demand attributes are not stored in the AST; instead the defining function is evaluated each time a demand attribute is accessed. Ordinary attributes, which are kept up to date by the incremental system, are referred to as *stored attributes*.

To use demand or stored attributes is a non-linear time/space trade-off. On one extreme, demand attributes may degrade performance if they are accessed frequently. On the other extreme, demand attributes may save both time and storage. A node may e.g. have a synthesized attribute but be in a syntactic context which does not access this attribute. Or an ancestral attribute declared in a production may be unused by some subproductions.

In Mjølner, stored attributes are used for information which is expensive to compute, in particular for the environments constructed at each block and for the declaration descriptors looked up at the identifiers. Demand attributes are used for information which is cheap to compute, such as types and the propagated environment attributes.

Demand attributes are particularly attractive for information which is accessed from external tools and which does therefore not need to be updated for each modification to the AST. E.g., the code generation in Mjølner is incremental on a block basis and is triggered more seldom than the attribute evaluator for static semantics. The code generator accesses static semantic information however, and this information is described as demand attributes.

As pointed out by e.g. Engelfriet [E 84], a demand attribute can be seen as a function of the node. In our object-oriented formalism, the equations defining demand attributes can be seen as late bound functions, similar to Simula's virtual functions. Thus, a synthesized demand attribute is regarded as a virtual function in the production declaring the attribute, and an ancestral demand attribute is regarded as a virtual function in the ancestor production.

A synthesized or local attribute is turned into a demand attribute simply by adding the keyword "demand" to the declaration. E.g. the `type` attribute of section 2.4 is turned into a demand attribute by changing the Expr production as follows:

```
-----------------------------------------------------------------
<Expr>:<Descendant> ::= Abstract
      Syn  type : TypeDescriptor demand; -- demand attribute --
      type := UnknownType;
-----------------------------------------------------------------
```

Our implementation of demand attributes relies on that the node accessing the demand attribute has knowledge about the most general production containing a defining equation for the attribute. For local and synthesized attributes this is no problem, since a node knows its own production and the general productions of its son nodes. However, for an ancestral attribute, information is needed about the production of the ancestor node. A production containing an ancestral demand attribute is therefore required to declare the production of its ancestor node using a "son to" clause. Furthermore, we require the ancestor production to specify an "export" declaration of the attribute.

As an example, consider the ancestral attribute `env` declared in the Descendant production in section 2.4. `Env` is turned into a demand attribute by changing the Node and Descendant productions as follows:

```
------------------------------------------------------------
<Node> ::= Abstract
     Exp  Descendant.env;                          -- (1) --
------------------------------------------------------------
<Descendant>:<Node> son of Node ::= Abstract       -- (2) --
     Anc  env : Environment demand;                -- (3) --
     for all sons(x) in Descendant
          son(x).env := env;
------------------------------------------------------------
```

The `env` attribute is declared demand (3). The Descendant production declares that its ancestor is of production Node using a "son of" clause (2). This allows Descendant nodes to be descendants of all nodes in an AST, which is what we want. The Node production declares an "export" of Descendant's env attribute (1).

If a production P has declared itself to be "son of" production A, then subproductions of P may specify themselves to be "son of" subproductions of A.

To include information on the ancestor production is in principal a syntactic restriction. Such information is not usually included in a grammar. However, this restriction can be made arbitrarily weak by specifying the most general production (i.e. Node). In practice, the most general production will often be used. Suppose e.g. we need a demand ancestral attribute for production Expr. Since expressions can occur in many different syntactic constructs it may be hard to find a natural general ancestor production and we have to resort to the Node production. This may feel a little odd, but efficiency is not impaired in any way.

3 INTEGRATION OF GRAMMAR INDEPENDENT TOOLS

The object-oriented notation makes it easy for tools to access information in an AST without knowing anything about the grammar. This can be done by declaring attributes in the most general production ("Node" in our examples) which thus become accessible from all nodes in an AST. As an example, consider setting a breakpoint in a program. A structure-oriented editor could implement this by allowing the user to select an AST node in the unparsed screen representation and then issuing a "set breakpoint" command. Breakpoints are not allowed at all AST nodes, however, so a test is needed. The test can be implemented by checking the value of a boolean attribute `breakAllowed`, present in all nodes. We will thus require of all grammars that a local attribute breakAllowed is declared in production Node:

```
------------------------------------------------------------
<Node> ::= Abstract
     Loc  breakAllowed : boolean demand;
     ...
------------------------------------------------------------
```

Suppose we want to allow breakpoints on all statements. For the language used in the earlier examples, the specification of `breakAllowed` is done follows:

```
------------------------------------------------------------
<Node> ::= Abstract
      Loc  breakAllowed : boolean demand;
      breakAllowed := false;
------------------------------------------------------------
<Stmt>:<Descendant> ::= Abstract
      breakAllowed := true;
------------------------------------------------------------
```

Some languages allow expressions at all places where statements are allowed. Such a language would have a slightly different grammar, modelling expressions as specializations of statements and breakpoints would now be allowed at all sons to statement lists. This could be specified as follows:

```
------------------------------------------------------------
<Node> ::= Abstract
      Loc  breakAllowed : boolean demand;
      Exp  Descendant.inBreakpointPos;
      breakAllowed := false;
      for all sons(x) in Descendant
            son(x).inBreakpointPos := false;
------------------------------------------------------------
<Descendant>:<Node> son of Node ::= Abstract
      Anc  inBreakpointPos : boolean demand;
      breakAllowed := inBreakpointPos;
------------------------------------------------------------
<StmtList>:<Descendant> ::= <Stmt>*
      for all sons(x) in Stmt
            son(x).inBreakpointPos := true;
------------------------------------------------------------
<Stmt>:<Descendant> ::= Abstract
------------------------------------------------------------
<Expr>:<Stmt> ::= Abstract
------------------------------------------------------------
```

Here an ancestral attribute `inBreakpointPos` is used to convey the information that a statement or expression is in such a context that a breakpoint is allowed. Although this grammar is rather different from the previous one, the handler can access the `breakAllowed` attribute in exactly the same way, and is thus independent of which language the AST actually belongs to.

Attributes cannot be used in this way in a traditional attribute grammar system. Attributes and equations would have to be given explicitly for all productions in the whole grammar, which would clutter the grammar enormously. Another problem in a traditional setting would be in accessing the `breakAllowed` attribute without knowing the production for a particular node. In contrast, using the object-oriented notation, the attribute can be accessed in a generic way using late binding.

4. IMPLEMENTATION

The order of attribute evaluation is not stated explicitly in the grammar, but derived from the attribute dependencies introduced by the equations. The traditional methods for calculating such orders can with trivial adaptions be applied to our object-oriented grammar formalism. However, in implementing access to attributes, we have used implementation techniques for object-oriented languages. In particular, demand attributes are accessed efficiently by using a mechanism similar to Simula's virtual procedure call.

We will now describe a "run-time organization" for access to stored and demand attributes. Each node in the AST has an array of stored attributes, *AttributeArray*, and a pointer to a "prototype" object which represents the production of the node. The prototype has an array, *DemandArray*, of functions for demand attributes. A prototype also has a pointer to its "superprototype" representing the superproduction.

DemandArray contains slots for local and synthesized demand attributes and for exported attributes (ancestral demand attributes of son nodes). The order of the corresponding attribute declarations is used to determine the order in the array, with the declarations of general productions preceding the ones of specialized productions. For an attribute which is defined by an inherited equation, the slot will refer to the same function as the corresponding slot in the superproduction. All functions defining a particular attribute will thus be located at the same index, and late binding is implemented efficiently by indirect access via DemandArray. This way of organizing the functions is analogous to the way virtual procedures are implemented in Simula [DM 73].

Each demand function has a parameter referring to the node defining the demand attribute. This is needed in order for the demand function to access other attributes. A demand function defining an ancestral attribute has two additional parameters: One for the prototype defining the function, and one whose value is the index of the son node whose attribute is to be calculated. These parameters are needed to take care of partially overridden equations in which case "super" is called for some son indices.

Given a node, N, access to attributes is coded in the following way (the k-indices are calculated at grammar compile time):

Stored attributes:

```
Local attribute:        N.AttributeArray[k]
Synth. attr of son[i]:  N.Son[i].AttributeArray[k]
Ancestral attribute:    N.AttributeArray[k]
```

Demand attributes:

```
Local attribute:        N.Prototype.DemandArray[k](N)
Synth. attr of son[i]:  N.Son[i].Prototype.DemandArray[k](N.Son(i))
Ancestral attribute:
      N.Ancestor.Prototype.DemandArray[k]
                  (N.Ancestor, N.Ancestor.Prototype, N.myIndex)
```

As an example, consider the grammar given in section 2.4, but with the env attribute declared demand as described in 2.5. Figure 2 shows a small syntax tree following this grammar:

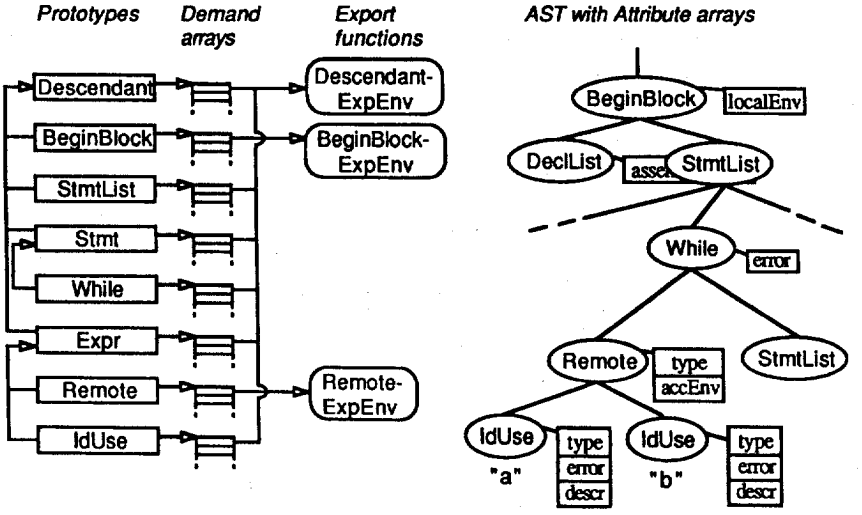

Figure 2. "Run-time" structures for demand attributes

We will examine what happens at accesses to the ancestral demand attribute `env`. Equations in Descendant, BeginBlock, and Remote for defining the `env` attribute of son nodes are compiled into functions as follows (#x indicate grammar compile time constants used as indices in AttributeArray, DemandArray, or as son node indices):

```
-----------------------------------------------------------------
function DescendantExpEnv
    (N: Node, P:Prototype, SonIndex:integer): return Environment
begin
    DescendantExpEnv :=
        N.Ancestor.Prototype.DemandArray[#expenv]
            (N.Ancestor, N.Ancestor.Prototype, N.myIndex)
end;
-----------------------------------------------------------------
function BeginBlockExpEnv
    (N:Node, P:Prototype, SonIndex:integer): return Environment
begin
    BeginBlockExpEnv :=
        if SonIndex=#stmtPart then N.AttributeArray[#localEnv]
        else    -- no overriding, call super --
            P.SuperPrototype.DemandArray[#expenv]
                (N, P.SuperPrototype, SonIndex)
end;
-----------------------------------------------------------------
function RemoteExpEnv
    (N:Node, P:Prototype, SonIndex:integer): return Environment
begin
    RemoteExpEnv :=
        if SonIndex=#accessor then N.AttributeArray[#accEnv]
        else     -- no overriding, call super --
            P.SuperPrototype.DemandArray[#expenv]
                (N, P.SuperPrototype, SonIndex)
end;
-----------------------------------------------------------------
```

As mentioned in section 2.4, the equations in BeginBlock and Remote do only partially override the equation in Descendant. This is taken care of by a call to "super" for cases when the overriding equation does not apply.

Suppose the `env` attribute of the IdUse *b* is accessed. This access is coded as follows:

```
b.Ancestor.Prototype.DemandArray[#expenv]
      (b.Ancestor, b.Ancestor.Prototype, b.myIndex)
```

b's ancestor is a Remote-node, so this will result in a call to RemoteExpEnv which returns the Remote-node's stored `accEnv` attribute since sonIndex = #accessor.

Now consider an access to the `env` attribute of IdUse *a*. This access will result in accessing the demand attributes of ancestor nodes successively up to the BeginBlock node. The access is coded as:

```
a.Ancestor.Prototype.DemandArray[#expenv]
      (a.Ancestor, a.Ancestor.Prototype, a.myIndex)
```

This will result in a call to RemoteExpEnv which will call "super" since sonIndex ≠ #accessor. "Super" is the DescendantExpEnv function which accesses the `env` attribute of the Remote-node itself. This results in calls to DescendantExpEnv of Assignment and StmtList, and then to BeginBlockExpEnv of the BeginBlock node. The stored attribute `localEnv` of the BeginBlock node becomes the final resulting value.

5. INHERITANCE AND WELL FORMED ATTRIBUTE GRAMMARS

An attribute grammar must be *well formed*, i.e. it must be guaranteed that for any syntactically legal AST, each attribute must have a defining equation. In this section we will discuss what implications this requirement has on our formalism. It turns out that well-formedness fits nicely with the single-inheritance notation, but is hard to combine with multiple inheritance.

In terms of the productions, an attribute grammar is well formed if

(1) each root production has no ancestral attributes,

(2) each production has defining equations for all synthesized and local attributes declared in the production, and

(3) each production has defining equations for all ancestral attributes declared in the productions of syntactically legal son nodes.

5.1 Single Inheritance

The two first criteria of well-formedness are straight forward to check. In checking the third criterion, full knowledge is needed about the ancestral attributes of syntactically legal son nodes. To be able to check this in a reasonable way, we introduce the following rule:

> *Rule 1*: If a production P exists which has a son declared to be of production C, then subproductions of C must not declare new ancestral attributes.

This means that it is sufficient to check that P has equations which define the ancestral attributes declared in C. Although nodes of subproductions to C are syntactically legal, they will according to rule 1 not have any additional ancestral attributes, so they cannot affect the well-formedness of the grammar.

This rule does not cause any practical problems when designing an attribute grammar. If one finds that an ancestral attribute `a` is needed for a subproduction of C, then one simply has to declare `a` in C instead of in the subproduction.

5.2 Multiple Inheritance

In this section we consider the consequences of augmenting the object-oriented notation to support multiple inheritance. It turns out that such grammars are in general not well formed.

An example which illuminates the problem is the following (see also figure 3). The notation is augmented by allowing a production P to have a set of superproductions instead of only one. Suppose we have a production P which has two superproductions S1 and S2 which each declare an ancestral attribute, a1 and a2, respectively. Suppose furthermore that a construction production A declares a son node of production S1. The production A must naturally include an equation defining the `a1` attribute of its S1 son. However, it is syntactically legal for an A-node to have a son of production P, since P is a subproduction of S1. In that case, the son node will have two ancestral attributes: `a1` and `a2`, but the A node has no equation for calculating the `a2` attribute. Thus the grammar is not well formed.

```
-------------------------------------------------------------
<N> ::= Abstract
-------------------------------------------------------------
<S1> : <N> ::= Abstract
     Anc  a1: boolean;
-------------------------------------------------------------
<S2> : <N> ::= Abstract
     Anc  a2: boolean;
-------------------------------------------------------------
<P> : (<S1>, <S2>) ::= {}
     Loc  l: boolean;
     l := a1 and a2;
-------------------------------------------------------------
<A> : <N> ::= {<son:S1>}
     son.a1 := true;
-------------------------------------------------------------
```

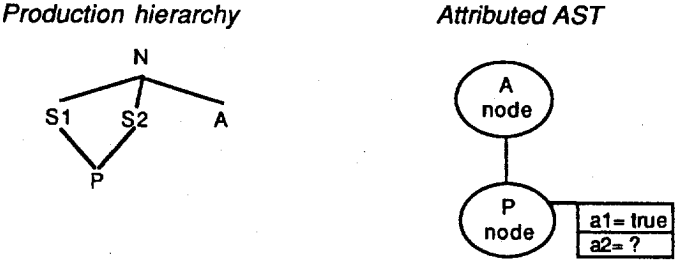

Figure 3. A multiple inheritance production hierarchy. The grammar is not well formed since the A node lacks an equation defining the P node's ancestral attribute `a2`.

In order to make this grammar well formed, one solution would be to move the declarations of `a1` and `a2` to the lowest common superproduction of S1 and S2, i.e. to production N. The A production would then naturally have to contain defining equations for both `a1` and `a2`. This would however have the effect that all subproductions of N would obtain the `a1` and `a2`

attributes, which was probably not intended. In particular, if a production used for root nodes is a subproduction of N, then this would violate the first criterion on well-formedness.

6. CONCLUSIONS

We have presented a new notation for attribute grammars which is based on object-oriented concepts. We see three main advantages of this notation compared to traditional attribute grammars.

First, organizing productions in an inheritance hierarchy allows compact specification in that attributes, equations and node structure is inherited to subproductions. The repeated specification of similar behavior which is necessary in traditional attribute grammars can thus be avoided.

Second, the object-oriented notation allows a simple and efficient implementation of demand attributes. Demand attributes do not consume any storage and are thus useful in situations where storage is scarce, e.g. in incremental systems. Access to a demand attribute is coded as an indirect function call using an index calculated at grammar compile time. Traditional grammars do not give immediate support for a similar implementation.

Third, the object-oriented notation makes it possible to define generically accessible attributes present in all nodes of an AST. This makes it possible for tools, which have no knowledge of the particular grammar, to access information in an AST. This possibility is not present in traditional attribute grammars.

We have implemented an incremental compiling system, Mjølner, where the static semantics is specified using our object-oriented grammar notation. Auxiliary attributes have also been used to specify information needed by other tools, such as the code-generator and interactive interrogation facilities. We have found the object-oriented notation very useful, in particular in combination with demand attributes.

A discussion has also been given in the paper about the problems in augmenting the notation to allow multiple inheritance. We have found that severe restrictions are needed on the attribute declarations in order to obtain a well formed multiple inheritance grammar.

Acknowledgements

This work has been carried out within the Mjølner project which is partially financed by Nordisk Industrifond (the Nordic Fund for Technology and Industrial Development) and STU (the Swedish national board for Technical Development). I would like to thank my colleagues in this project, especially Boris Magnusson, Claus Nørgaard, Mats Bengtsson and Sten Minör for helpful comments on this paper. Thanks also to the anonymous referees.

References

[CI 84] Cameron R.D., Ito M.R.: *Grammar-Based Definition of Metaprogramming Systems.* ACM TOPLAS, Vol 6, No 1, Jan 1984. 20-54.

[CS 87] Christ-Neumann M.-L., Schmidt H.-W.: *ASDL - An Object-Oriented Specification Language for Syntax-Directed Environments.* Proceedings of ESEC '87. LNCS 289, Springer-Verlag. 71-79.

[DHKLL 75] Donzeau-Gouge V., Huet G., Kahn G., Lang B., Levy J.J.: *A structure-oriented program editor.* Rep no 114, IRIA-LABORIA, Rocquencourt, France, Apr 1975.

[DMN 68] Dahl O.-J., Myhrhaug B., Nygaard K.: *SIMULA 67 Common Base Language.* Norwegian Computing Center, Oslo, 1968. Revised 1970, 1972, and 1984. Swedish standard 1987 in "Data Processing - Programming Languages - *SIMULA*. SS 63 61 14. SIS. Stockholm, Sweden, June 1987".

[DM 73] Dahl O.-J., Myhrhaug B.: *Simula 67 Implementation Guide.* Pub. 9., Norwegian Computing Center, Oslo, Norway, 1973.

[DLMM 87] Dahle, H.P., Löfgren, M., Madsen, O.L., Magnusson, B., *The MJØLNER Project*, Proceeding of the Conference held at Software Tools, Online Publications, London, 1987.

[E 84] Engelfriet J.: *Attribute Grammars: Attribute Evaluation Methods.* In Lorho B. (ed.): Methods and Tools for Compiler Construction. Cambridge University Press. 1984. 103-138.

[RTD 82] Reps T., Teitelbaum T., and Demers A.: *Incremental Context-Dependent Analysis for Language-Based Editors.* ACM TOPLAS Vol 5, No 3, July 1983, 449-477.

[HM 88] Hedin G., Magnusson B.: *The Mjølner Environment: Direct Interaction with Abstractions.* In S. Gjessing, K. Nygaard (eds.). Proceedings of the European Conference on Object-Oriented Programming (ECOOP'88), Oslo, Norway, August 1988. LNCS 322, Springer-Verlag.

[H 88] Hedin G.: *Incremental Attribute Evaluation with Side-Effects.* Proceedings of the workshop on Compiler Compiler and High Speed Compilation, Berlin, G.D.R., Oct. 1988.

[K 68] Knuth D. E.: *Semantics of context-free languages.* Math. Syst. Theory 2:2, June 1968, 127-145.

[KMNN 87] Kristenssen B.B., Madsen O.L., Møller-Pedersen B., Nygaard K.: The BETA Programming Language. In Shriver B.D. and Wegner P. (ed.) Research Directions in Object-Oriented Programming. MIT Press 1987. pp 7-68.

[M 82] Medina-Mora R.: *Syntax-Directed Editing: Towards Integrated Programming Environments.* Ph. D. Thesis, Dept of Computer Science, Carnegie-Mellon University, March 1982.

[MBD 87] Møller-Pedersen B. et. al.: *Rationale and Tutorial on OSDL: An Object-Oriented Extension of SDL.* Computer Networks. Vol 13, No 2, 1987. 97-117.

[MN 88] Madsen O.L., Nørgaard C.: *An Object-Oriented Metaprogramming System.* Proceedings of the 21 Annual Hawaii International Conference on System Sciences. January 1988. Vol II. 406-415.

[N 87] Nørmark K.: *Transformation and Abstract Presentations in a Language Development Environment.* Thesis. Computer Science Department, Aarhus University, Denmark, January 1987.

[RT 84] Reps T., Teitelbaum T.: *The Synthesizer Generator.* Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium of Practical Software Development Environments, 1984. SIGPLAN Notices, Vol 19, No 5, 42-48.

[TTTI 88] Tenma T., Tsubotani H., Tanaka M., and Ichikawa T.: *A System for Generating Language-Oriented Editors.* IEEE Trans. on Soft. Eng., Vol 14, No 8, August 1988. 1098-1109.

[Y 83] Yeh D.: *On Incremental Evaluation of Ordered Attribute Grammars.* BIT 23 (1983), 308-320.