Multiple Inheritance
in Object-Oriented
Attribute Grammars

J. Grosch

GESELLSCHAFT FÜR MATHEMATIK
UND DATENVERARBEITUNG MBH

FORSCHUNGSSTELLE FÜR
PROGRAMMSTRUKTUREN
AN DER UNIVERSITÄT KARLSRUHE

Project

Compiler Generation

Multiple Inheritance in Object-Oriented Attribute Grammars

Josef Grosch

Feb. 25, 1992

Report No. 28

Gesellschaft für Mathematik und Datenverarbeitung mbH
Forschungsstelle an der Universität Karlsruhe
Vincenz-Prießnitz-Str. 1
D-7500 Karlsruhe

Multiple Inheritance in Object-Oriented Attribute Grammars

Josef Grosch
GMD Forschungsstelle an der Universität Karlsruhe
Vincenz-Prießnitz-Str. 1, D-7500 Karlsruhe, Germany
grosch@karlsruhe.gmd.de

Abstract Object-oriented attribute grammars are a promising notation for language specifications. They have similar benefits as object-orientation in the area of programming languages. They support a compact and flexible style for language specifications. Existing definitions can be easily reused as well as the associated default behaviour. New definitions can be derived from existing ones by specialization. While previous approaches have been restricted to single inheritance this paper defines object-oriented attribute grammars with multiple inheritance. A system has been developed that processes those attribute grammars. We describe an example that uses multiple inheritance and compare the terminology and concepts of related areas.

Keywords attribute grammar, object-orientation, multiple inheritance

1. Introduction

Object-oriented attribute grammars have been introduced by several authors (e. g. [Gro90a, Hed89]) as a promising notation for attribute grammars. An overview of the current state of the art in this area is given in [Kos91]. The benefits are comparable to those of object-oriented programming languages. It is a concise notation and flexible notation for language specifications. The reuse of existing definitions is supported by the possibility to specify new definitions as extensions or specializations of existing ones. The duplication of information is avoided because common parts can be "factored out".

While the main building blocks of object-oriented programming languages are classes, the nonterminals play this part in object-oriented attribute grammars. More precisely, the notions nonterminal and production rule are unified. This means that there is exactly one production rule for every nonterminal. Additionally, a relation between nonterminals is specified, for example using chain rules, which describes a subtype relation or class hierarchy among the nonterminals. This subtype relation serves for two purposes. First, it allows to derive several different strings from one nonterminal because a nonterminal may be replaced by a right-hand side corresponding to a nonterminal that is a subtype of the replaced one. Second, the subtype relation describes the path for inheritance among the nonterminals. The items that are subject to inheritance are right-hand side elements, attributes, and attribute computations. Inherited attribute computations may be overwritten in the subtype by giving different computations for the same attributes.

Before we proceed we have to clarify the terminology: Originally, context-free grammars as well as attribute grammars are derivation systems for strings. In this paper we are interested in the specification of semantic analysis which is based on an abstract syntax tree. Therefore we use grammars to describe the structure of trees instead of strings.

In order to avoid confusion between the terms class, nonterminal, terminal, and (sub)type we will use the term *node type* to cover all those meanings. The term node type is motivated through a realistic description of what is happening: The node types specify the structure of the nodes of the abstract syntax tree.

The attributes in attribute grammars are usually classified as *synthesized* and *inherited*. Following Hedin [Hed89] we use the term *ancestral attribute* instead of the standard *inherited attribute* since we use the term *inherited* in the object-oriented sense.

There is one problem that arises especially from the combination of ancestral attributes and inheritance. Let A, B, and C be node types and let B be a subtype of A ($B \subseteq A$) having one ancestral attribute x. If the right-hand side of C contains an A, we have to know whether to compute the attribute A.x or not. The static type is A, but the dynamic type can be any subtype, that is A or B. If it is B we have to compute A.x, if it is A we may not compute it. The notions node type, subtype, and right-hand side are defined in section 2.

There are several solutions to this problem. First, one can restrict the definition of ancestral attributes to top level node types, only. This makes the reuse of existing node types very hard in particular in combination with multiple inheritance.

Second, one could use a dynamic dispatch technique which inspects the dynamic type of the right-hand side child and decides during runtime whether to compute A.x or not. This solution is rather inefficient because of its runtime overhead.

Most existing systems therefore allow single inheritance, only, with the additional restriction that ancestral attributes have to be defined at top level node types. The last restriction is not severe because it somehow coincides in a natural way with the style of usual attribute grammars. Hedin [Hed89] follows this argumentation and calls object-oriented attribute grammars having the above problem not *well formed*.

This paper introduces a third solution to the above mentioned problem. It allows for a restricted form of multiple inheritance and still retains the capability to decide at generation time which ancestral attributes have to be computed. Attribute evaluators can still be implemented efficiently as dynamic dispatch is avoided.

In section 2 we formally define object-oriented attribute grammars with single inheritance. Section 3 contains two simple examples using single inheritance. Section 4 extends the definition of object-oriented attribute grammars to multiple inheritance. Section 5 presents an elaborate example with multiple inheritance. In section 6 we compare our approach with pure attribute grammars and with object-oriented programming in order to reveal the common properties as well as the differences. Section 7 summarizes the results.

2. Single Inheritance

This section formally defines the principles of object-oriented attribute grammars with single inheritance. As starting point we shortly recall the traditional definition of attribute grammars [Knu68, Knu71].

An attribute grammar is an extension of a context-free grammar. A context-free grammar is denoted by $G = (N, T, P, Z)$ where N is the set of nonterminals, T is the set of terminals, P is the set of productions, and $Z \in N$ is the *start* symbol, which cannot appear on the right-hand side of any production in P. The set $V = N \cup T$ is called the vocabulary. Each production $p \in P$ has the form $p: X \rightarrow \alpha$ where $X \in N$ and $\alpha \in V^*$. The relation $\Rightarrow$ (directly derives) is defined over strings in V^* as follows: if $p: X \rightarrow \alpha$, $p \in P$, $\nu X \omega \in V^*$, $\nu \alpha \omega \in V^*$ then $\nu X \omega \Rightarrow \nu \alpha \omega$. The relation $\Rightarrow^*$ is the transitive and reflexive closure of $\Rightarrow$. The language L(G) is defined as $L(G) = \{ w \mid Z \Rightarrow^* w \}$.

An attribute grammar augments a context-free grammar by attributes and attribute computations. A set of attributes is associated with each symbol in V. Attribute computations are added to every production describing how to compute attribute values in the local context of a production. This simple view of attribute grammars shall suffice for the scope of this paper.

In general there can be several productions having the same nonterminal on the left-hand side. This allows for different derivations starting from one nonterminal. In object-oriented attribute grammars, one production is permitted for one left-hand side symbol, only. This way the notions production and nonterminal (vocabulary respectively) are unified and are termed *node type* as already mentioned. Several different derivations are made possible through the newly introduced subtype relation.

An object-oriented attribute grammar is formally denoted by G = (N, T, A, C, Z) where N is the set of nonterminals, T is the set of terminals, A is the set of attributes, C is the set of attribute computations, and Z is the start symbol ($Z \in N$). The set $NT = N \cup T$ is called the set of *node types*. Each element $n \in NT$ is associated with a tuple n: (R, B, D, S) where $R \in NT^*$ is the right-hand side, $B \in A^*$ is the set of attributes, $D \in C^*$ is the set of attribute computations, and $S \in NT$ is the base type.

The elements of NT induce a relation $\subseteq$ (subtype) over NT as follows: if $n: (\alpha, \beta, \delta, m) \in NT$ then $n \subseteq m$. m is called *base* or *super* type, n is called *derived* type or *subtype*. The relation $\subseteq$ is transitive: if $n \subseteq m$ and $m \subseteq o$ then $n \subseteq o$.

The relation $\Rightarrow$ (directly derives) is defined here only for the context-free or syntactic part of an object-oriented attribute grammar. There are two possibilities for derivations which are defined over strings in NT^* as follows:

$$
\begin{aligned}
\text{if } \nu n_i \omega \in NT^* \text{ and } & n_1: (\alpha_1, \beta_1, \delta_1, n_0) \in NT, \\
& n_2: (\alpha_2, \beta_2, \delta_2, n_1) \in NT, \\
& \quad \cdots \\
& n_i: (\alpha_i, \beta_i, \delta_i, n_{i-1}) \in NT \text{ then } \nu n_i \omega \Rightarrow \nu \alpha_1 \alpha_2 \cdots \alpha_i \omega.
\end{aligned}
$$

$$\text{if } \nu n \omega \in NT^* \text{ and } m \subseteq n \text{ then } \nu n \omega \Rightarrow \nu m \omega.$$

We assume the existence of a predefined node type n_0: $(\emptyset, \emptyset, \emptyset, -)$ with empty components. In a direct derivation step, a node type can be replaced by its right-hand side $(\alpha_1 \cdots \alpha_i)$ or by one of its subtypes (m). All replacing right-hand sides are the union of right-hand sides according to the subtype hierarchy. The relation $\Rightarrow^*$ is the transitive and reflexive closure of $\Rightarrow$. The language L(G) is defined as $L(G) = \{ w \mid Z \Rightarrow^* w \}$.

The subtype relation has the following properties: a derived node type inherits the right-hand side, the attributes, and the attribute computations from its base type. As consequence of the transitive nature of this relation, a derived type inherits all the components from all base types according to the subtype hierarchy. It may extend the set of inherited items by defining additional right-hand side elements, attributes, or attribute computations. All accumulated right-hand side elements and attributes must be distinct because they are united. An attribute computation for an attribute may overwrite an inherited one.

3. Example

We implemented an attribute grammar system called *ag* based on object-oriented attribute grammars which is part of the Karlsruhe Toolbox for Compiler Construction [GrE90]. It supports the kinds of single and multiple inheritance described in this paper. The following examples of object-oriented attribute grammars with single inheritance are written in the specification language of *ag*. The language tries to adhere to the conventional style of context-free grammars as far as possible. It offers far more features for practical usage than can be explained here. The interested reader is referred to the user's manual [Gro89].

An attribute grammar is given in the form of nested node type definitions. The nesting expresses the subtype hierarchy or the subtype relation. A node type definition consists of properties of the node type followed by a list of subtype definitions enclosed in angle brackets < >. The properties include the structural or syntactic definition (right-hand side), attribute definitions, and attribute computations.

Example 1:

```
Expr        = [Value: INTEGER]    { Value := 0; } <
   Add      = Lop: Expr Rop: Expr { Value := Lop:Value + Rop:Value; } .
   Sub      = Lop: Expr Rop: Expr { Value := Lop:Value - Rop:Value; } .
   Const    = Integer             { Value := Integer:Value; } .
> .
Integer     = [Value: INTEGER] .
```

The example describes the evaluation of primitive expressions. Attribute definitions are given in brackets [ ]. The attribute *Value* is associated with all subtypes of *Expr* with a default computation "Value := 0;". The attribute computations are written in curly brackets { }. The computations for the node types *Add*, *Sub*, and *Const* overwrite the computation given in the base type *Expr*.

The structural or syntactic definition is given as a sequence of node type names, possibly prefixed by a selector *(Lop*, *Rop*) allowing unambiguous access to the component structures.

Example 2:

```
Stats       = <
   NoStat   = .
   Stat     = Next: Stats [Pos: tPosition] <
      If    = Expr Then: Stats Else: Stats .
      While = Expr Stats .
      Call  = Actuals [Ident: tIdent] .
   > .
> .
```

Example 2 describes a possibility for the specification of the abstract syntax of statement sequences. The example uses the node type *Stats* to describe a sequence and the node type *Stat* to describe various statements. The node types are related as subtypes showing a non-trivial subtype relation of nesting depth two. The subtype relation is: $NoStat \subseteq Stats$, $Stat \subseteq Stats$, $If \subseteq Stat$, $While \subseteq Stat$, $Call \subseteq Stat$. In Example 2 the node types *If*, *While*, and *Call* inherit the child *Next* of type *Stats* and the attribute *Pos* from the base type *Stat*. They add their own children and attributes.

4. Multiple Inheritance

The problem with multiple inheritance mentioned in the introduction can be solved if we distinguish two kinds of types: *node types* and *abstract types*. An abstract syntax tree is constructed only out of nodes whose type is a node type - there are no nodes whose type is an abstract one. While the node types describe production rules or tree nodes the abstract types describe concepts.

An object-oriented attribute grammar with multiple inheritance is formally denoted by G = (N, T, K, A, C, Z). N, T, A, C, Z represent the same entities as in the single inheritance case. In particular the set $NT = N \cup T$ represents the set of *node types*. K is the set of *abstract types*. Every element $n \in NT$ is now associated with a quintuple n: (R, B, D, S, L) where R, B, D, and S are as before. Every element $k \in K$ is associated with a quintuple k: (R, B, D, U, L) where $U \in K$ is the base type for the single inheritance mechanism and $L \in K^*$ is the set of base types for the multiple inheritance mechanism. We have two inheritance mechanisms which operate simultaneously. Multiple inheritance behaves similar as single inheritance: A subtype inherits all properties (right-hand side, attributes, attribute computations) from all its base types.

Why do we need two mechanisms for inheritance? We retained single inheritance for two reasons: First, it is good to be compatible with existing attribute grammars written in the single inheritance style. Second, the single inheritance notation allows to adhere largely to the conventional style of writing context-free grammars.

The above definition for object-oriented grammars with multiple inheritance distinguishes two levels (see Fig. 1). The set of abstract types represents the abstract or conceptual level. These types model concepts and properties which are common to several node types or even to different programming languages. Abstract types are not used for nodes in the syntax tree. The set of node types represents production rules of a context-free grammar. The node types describe the constructs of a programming language. Node types are used to classify the nodes in a syntax tree.

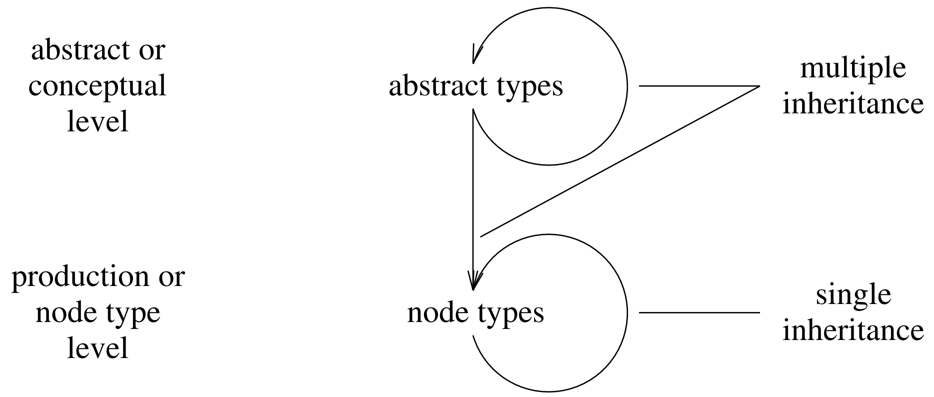

Fig. 1: Inheritance among abstract types and node types

The above definition of multiple inheritance allows multiple inheritance among abstract types and from abstract types to node types. Among node types, single inheritance is available, only. Ancestral attributes may be defined for all abstract types and for top level node types. With this restriction it is statically known for all children of all nodes whether ancestral attributes have to be computed or not.

5. Example

In this section we present a rather elaborate example for an object-oriented attribute grammar with multiple inheritance. The example is an excerpt from a specification of the demo language MiniLAX [Gro90b]. The attribute computations are written directly in the implementation language which is Modula-2.

The attribute grammar module in Example 3 describes an abstract symbol table. It is termed abstract because we deal with entities called objects which are not further specified. The symbol table handles declarations of objects, applications (uses) of objects, and scopes. It does not specify what kind of objects are to be declared, where those objects are used, and which constructs are associated with scopes. We use all upper-case names to denote abstract types.

The first three definitions describe lists of abstract declarations. A declaration *DECL* is characterized by an identifier and a reference to a succeeding declaration. The identifier is described by two attributes *Ident* and *Pos* holding an internal representation and a source position. The right-hand side child with the selector *Next* and the node type *Decls* refers to the succeeding declaration.

A list of declarations *(DECLS)* is either empty *(NODECLS)* or starts with one element of type *DECL*. The list has a threaded attribute called *Objects*. This threaded attribute actually stands for two attributes called *ObjectsIn* and *ObjectsOut*. The attribute computations given for *DECL* in curly brackets { } use this threaded attributes(s) to collect all declared objects in a list. They make use of functions from an external data type. *mNoObjects* creates an empty list,

Example 3:

```
MODULE SymbolTable

DECLS           := [Objects: tObjects THREAD OUT] <
   NODECLS      := .
   DECL         := Next: Decls IN [Ident: tIdent IN] [Pos: tPosition IN]
                   { Next:ObjectsIn     := mObject (ObjectsIn, Ident);
                     ObjectsOut         := Next:ObjectsOut;
                     CHECK NOT IsDeclared (Ident, ObjectsIn)
                     ==> Error ("identifier already declared", Pos);    } .
> .

ENV             := [Env: tEnv INH] .

USE   <- ENV    := [Ident: tIdent IN] [Object: tObjects SYN OUT]
                   { Object             := Identify (Ident, Env);
                     CHECK Object^.Kind # NoObject
                     ==> Error ("identifier not declared", Pos);        } .

SCOPE <- ENV    := [Objects: tObjects SYN] [NewEnv: tEnv SYN]
                   { Objects            := mNoObjects ();
                     NewEnv             := mEnv (Objects, Env);         } .

END SymbolTable
```

mObject adds a description of an object to a list, and *IsDeclared* checks for multiple declarations. The latter function is used in a condition (CHECK) that issues an error message in case of multiple declarations.

The abstract type *SCOPE* describes scopes such as blocks or procedures. The attribute *Env* (for environment) which is inherited from the abstract type *ENV* describes the set of objects that is visible at certain locations in a program. Multiple inheritance is expressed by writing an arrow <- and a list of abstract (base) types behind a type. A scope is supposed to reside in a surrounding environment described by the attribute *Env* and to introduce a new set of declarations represented by the attribute *Objects*. It computes a new environment attribute *NewEnv* valid inside this scope by calling the external function *mEnv*. The computation of the attribute *Objects* is a dummy to satisfy the completeness requirement of attribute grammars.

The abstract type *USE* describes the application or use of objects. A construct that uses an object has an attribute giving the identifier of the object *(Ident)*. This construct possesses an environment attribute *Env* that describes all objects visible at this construct. The attribute *Object* is used to refer to the symbol table entry of the used object. The external function *Identify* takes the attributes *Ident* and *Env* as arguments and computes the attribute *Object*. In case of the use of an undeclared identifier the CHECK statement will issue an appropriate error message.

The attribute definitions in Example 3 use a few keywords. These associate so-called properties with the attributes. IN characterizes input attributes that have already a value when attribute evaluation starts. The value is usually supplied during tree construction. OUT characterizes output attributes whose value is needed after attribute evaluation. Those attributes may not be removed from the tree nodes by an optimizer. SYN and INH classify the attributes as *synthesized* and *ancestral*.

Example 4 shows the connection of the abstract symbol table with the abstract syntax of the language MiniLAX. The subset of the node type definitions relevant for the symbol table problem is given. Whereas abstract types are introduced with the symbol := the character = is used for node types.

A concrete list of declarations is described by the node type *Decls* which is a subtype of the abstract type *DECLS*. A single declaration is described by the node type *Decl* which inherits

Example 4:

```
MODULE AbstractSyntax

MiniLax           = Proc .
Decls <- DECLS    = <
   NoDecl         = .
   Decl <- DECL   = <
      Var         = Type .
      Proc        = Formals Decls Stats .
   > .
> .
Stats             = <
   NoStat         = .
   Stat           = Next: Stats <
      Assign      = Adr Expr            [Pos: tPosition] .
      Call <- USE = Actuals             [Pos: tPosition] .
      If          = Expr Then: Stats Else: Stats .
      While       = Expr Stats .
      Read        = Adr .
      Write       = Expr .
   > .
> .
Expr              =                     [Pos: tPosition] <
   Binary         = Lop: Expr Rop: Expr [Operator: SHORTCARD] .
   Unary          = Expr                [Operator: SHORTCARD] .
   IntConst       =                     [Value: INTEGER] .
   RealConst      =                     [Value: REAL   ] .
   BoolConst      =                     [Value: BOOLEAN] .
   Adr            = <
      Index       = Adr Expr .
      Ident <- USE = .
   > .
> .

END AbstractSyntax
```

from *DECL*. Two specializations are derived from the node type *Decl* describing two kinds of declarations: variables and procedures *(Var* and *Proc)*. Through inheritance from *DECL* every *Decl* specifies already an identifier (attributes *Ident* and *Pos*) and a reference to a succeeding declaration (attribute *Next*). Therefore the specializations have just to add the missing components which is a description of the type in case of a variable and the formal parameters, the local declarations, and the procedure body *(Formals*, *Decls*, *Stats*) in case of a procedure.

The language knows two locations where objects are used: at a procedure call and at a variable occurring in an expression. Therefore the node types *Call* and *Ident* are subtypes of the abstract type *USE*. The node type *Call* specializes the usage of an object by adding a list of actual parameters and an attribute *Pos* which is needed for an error message.

The attribute computations in the attribute grammar for MiniLAX are grouped into modules (see Example 5). We present excerpts from three modules that are involved in the symbol table problem. The part of the module *Decls* shown in Example 5 specializes the computation of the attribute *Next:ObjectsIn* for the concrete declarations of the language. This way the information in the symbol table is extended by the kind of the declared object, the type of variables, and the formal parameters of procedures.

The module named *Env* specifies all computations for the environment attribute. It is reproduced completely. All node types whose subtrees can contain applications of objects need an environment attribute *Env* and become therefore subtypes of the abstract type *ENV*: *Decls*,

Example 5:

```
MODULE Decls

Proc           = { Next:ObjectsIn := mProc (ObjectsIn, Ident, Formals); } .
Var            = { Next:ObjectsIn := mVar  (ObjectsIn, Ident, Type);    } .

END Decls

MODULE Env

Decls   <- ENV = .
Stats   <- ENV = .
Actuals <- ENV = .
Expr    <- ENV = .

MiniLax        = { Proc:Env  := NoEnv          ; } .
DECL          := { Next:Env  := NoEnv          ; } .
Decl           = { Next:Env  := Env            ; } .
Proc <- SCOPE  = { Objects   := Decls:ObjectsOut;
                   Stats:Env := NewEnv         ;
                   Decls:Env := NewEnv         ; } .

END Env

MODULE Conditions

Call           = { CHECK IsObjectKind (Object, Proc)
                   ==> Error ("only procedures can be called", Pos); } .
Ident          = { CHECK IsObjectKind (Object, Var)
                   ==> Error ("variable required"            , Pos); } .

END Conditions
```

Stats, *Actuals*, and *Expr*. The first attribute computation of *Proc* connects the "interfaces" of the abstract types *DECLS* and *SCOPES*. The attribute *Decls:ObjectsOut* is the collected list of locally declared objects. It is assigned to the attribute *Objects* which is required to hold this information from the point of view of the abstract type *SCOPE*. *SCOPE* computes an attribute *NewEnv* describing the objects visible inside the procedure. The value of this attribute is passed to the attributes *Stats:Env* and *Decls:Env* to make the environment available for the components of the procedure. The rules given in the module *Env* suffice to specify all computations necessary for the environment attribute. The many missing rules are inserted automatically by the tool *ag* as simple copy rules.

Finally, the module *Conditions* adds checks to the locations where objects are used. The abstract type *USE* already checks whether an object is declared or not. We still need to check if an object is of the right kind. This test is performed by the external procedure *IsObjectKind*.

6. Comparison

This section compares object-oriented attribute grammars as introduced in this paper with the well-known concepts of (attribute) grammars, tree and record types, type extensions, and object-oriented programming. The goal is to reveal the common properties as well as the differences among these concepts. These areas are related because of the following reasons: attribute grammars are usually based on context-free grammars. An attribute grammar specifies an evaluation of attributes of a tree defined by such a context-free grammar. Trees can be implemented using a set of record type declarations. Therefore context-free grammars, trees, and record types deal more or less with the same concept. Table 1 compares the most important notions from these areas. Additionally we included the notions from the area of object-oriented (oo) programming as described e. g. in [Bla89].

(attribute) grammars	trees	types	oo-programming
rule	node type	record type	class
attribute	field in a node type	record field	instance variable
nonterminal	set of node types	union of record types	-
terminal	distinct node type	record type without pointer fields	-
rule application	tree node	record variable	object, instance
attribute computation	-	procedure declaration	method
-	-	procedure call	message
-	-	base type	superclass
-	-	derived type	subclass
-	-	extension	inheritance

Table 1: Comparison of notions from the areas of grammars, trees, types, and oo-programming

Object-oriented attribute grammars are missing in Table 1. For them we used the notions from attribute grammars and added the notions node type, base type, subtype or derived type, and inheritance from the other areas.

6.1. Attribute Grammars

Conventional grammars in BNF allow several productions with the same nonterminal symbol on the left-hand side. A node type in object-oriented attribute grammars, which corresponds to a nonterminal as well as to a rule name, has exactly one right-hand side. The selector names can be regarded as syntactic sugar. To allow for several different derivations, a subtype relation between node types is added. During a derivation, a node type may be replaced by its right-hand side or by a subtype. Inheritance is a notation to factor out parts that are common to several node types such as right-hand sides, attributes, and attribute computations. Fortunately, attributes local to a rule (node type) are possible without any special construct.

Object-oriented attribute grammars are a notation to write BNF grammars in a short and concise way and where the underlying tree structure can be exactly described. With respect to attribute grammars the same notational advantages hold. Attribute grammars are a special case of object-oriented attribute grammars. They are characterized by a one level subtype hierarchy, right-hand sides and attribute computations are defined for subtypes only, and attributes are associated only with base types. In terms of attribute grammar classes or attribute grammar semantics object-oriented attribute grammars are equivalent to attribute grammars.

6.2. Trees and Records

When trees are stored in memory, they can be represented by linked records. Every node type corresponds to a record type. Object-oriented attribute grammars directly describe the structure of attributed syntax trees. The node types can be seen as record types. The right-hand side elements resemble pointer valued fields describing the tree structure and the attributes are additional fields for arbitrary information stored at tree nodes. The field name and field type needed for record types are also present in the node types of object-oriented attribute grammars.

6.3. Type Extensions

Type extensions have been introduced with the language Oberon by Wirth [Wir88a, Wir88b, Wir88c]. They allow the definition of a record type based on an existing record type by adding record fields. This extension mechanism induces a subtype relation between record types. The subtype and inheritance features are equivalent in object-oriented attribute grammars and type extensions with the difference that Wirth uses the word extension in place of inheritance and restricts it to single inheritance.

6.4. Object-Oriented Programming

The concepts of subtype and inheritance in object-oriented attribute grammars and object-oriented programming have many similarities and this explains the name object-oriented attribute grammars. The notions class, instance variable, object, superclass, and subclass have direct counter parts (see Table 1). There are also some differences. Object-oriented programming allows an arbitrary number of named methods which are activated by explicitly sending messages. In object-oriented attribute grammars there is exactly one attribute computation for an attribute. This computation corresponds to an unnamed method. There is nothing like messages: The attribute computation for an attribute is activated implicitly and exactly once.

7. Summary

We presented object-oriented attribute grammars with single and multiple inheritance. The distinction between abstract types and node types allows for a restricted form of multiple inheritance that can still be implemented efficiently. The nonterminals or node types are the entities constituting the inheritance hierarchy. The items that are subject to inheritance are right-hand side elements, attributes, and attribute computations.

Object-oriented attribute grammars are a compact and flexible notation for language specifications. The repetition of information is avoided as common parts can be factored out. The reuse of definitions is supported - new definitions can be derived from existing ones by specializations.

We extended the attribute evaluator generator *ag* to process object-oriented attribute grammars with multiple inheritance. It turned out that it is possible to generate efficient attribute evaluators for this kind of attribute grammars.

While we are very satisfied with the advantages of single inheritance we have just started to explore the feasibility of multiple inheritance. Our current goal is to build a collection of attribute grammar modules containing abstract types that model concepts of programming languages such as the discussed symbol table. Together with abstract data types these will result a library of reusable parts oriented towards semantic analysis of programming languages. If possible those parts will be designed to specify aspects of semantic analysis independent of concrete languages.

References

[Bla89] G. Blaschek, Implementation of Objects in Modula-2, *Structured Programming 10*, 3 (1989), 147-155.

[Gro89] J. Grosch, Ag - An Attribute Evaluator Generator, Compiler Generation Report No. 16, GMD Forschungsstelle an der Universität Karlsruhe, Aug. 1989.

[Gro90a] J. Grosch, Object-Oriented Attribute Grammars, in *Proceedings of the Fifth International Symposium on Computer and Information Sciences (ISCIS V)*, A. E. Harmanci and E. Gelenbe (ed.), Cappadocia, Nevsehir, Turkey, Oct. 1990, 807-816.

[Gro90b] J. Grosch, Specification of a Minilax Interpreter, Compiler Generation Report No. 22, GMD Forschungsstelle an der Universität Karlsruhe, Mar. 1990.

[GrE90] J. Grosch and H. Emmelmann, A Tool Box for Compiler Construction, *LNCS 477*, (Oct. 1990), 106-116, Springer Verlag.

[Hed89] G. Hedin, An Object-Oriented Notation for Attribute Grammars, *Proc. of the European Conference on Object-Oriented Programming (ECOOP '89)*, Nottingham, 1989, 329-345.

[Knu68] D. E. Knuth, Semantics of Context-Free Languages, *Mathematical Systems Theory 2*, 2 (June 1968), 127-146.

[Knu71] D. E. Knuth, Semantics of Context-free Languages: Correction, *Mathematical Systems Theory 5*, (Mar. 1971), 95-96.

[Kos91] K. Koskimies, Object-Orientation in Attribute Grammars, *LNCS 545*, (1991), 297-329, Springer Verlag.

[Wir88a] N. Wirth, Type Extensions, *ACM Trans. Prog. Lang. and Systems 10*, 2 (Apr. 1988), 204-214.

[Wir88b] N. Wirth, From Modula to Oberon, *Software—Practice & Experience 18*, 7 (July 1988), 661-670.

[Wir88c] N. Wirth, The Programming Language Oberon, *Software—Practice & Experience 18*, 7 (July 1988), 671-690.

Contents