Object-Orientation in the Cocktail Toolbox

J. Grosch

DR. JOSEF GROSCH

COCOLAB - DATENVERARBEITUNG

GERMANY

Cocktail

Toolbox for Compiler Construction

Object-Orientation in the Cocktail Toolbox

Josef Grosch

Sept. 11, 1994

Document No. 34

Dr. Josef Grosch
CoCoLab - Datenverarbeitung
Breslauer Str. 64c
76139 Karlsruhe
Germany

Phone: +49-721-91537544
Fax: +49-721-91537543
Email: grosch@cocolab.com

Object-Orientation in the Cocktail Toolbox

Josef Grosch
CoCoLab
Hagsfelder Allee 16
D-76131 Karlsruhe
Germany

Tel: +49-721-697061
Fax: +49-721-661966
EMail: grosch@cocolab.de

Abstract The Cocktail Toolbox for Compiler Construction supports the automatic generation of nearly all parts of a compiler. This article describes the aspects of object-orientation of some tools which appear primarily at the specification level. The areas discussed in the following are: Concrete grammars as input for parser generators, abstract syntax and symbol tables used as internal data structures, attribute grammars as specification for semantic analysis, and pattern-matching for the transformation or mapping of attributed trees into intermediate representations or target code.

1. Introduction

In the Cocktail Toolbox for Compiler Construction [GrE90] object-orientation appears in two areas: First, it is possible to generate scanner and parser classes in the languages C++ and Eiffel and thus several scanner and parser objects can be instantiated during run time. This area is not very inspiring and therefore it is not discussed in this paper. Second, there is a central specification language for the description of concrete syntax, abstract syntax, and attribute grammars. This language is based on context-free grammars and it offers many extensions with respect to attribute grammars and object-orientation. The following sections discuss the areas where object-orientation plays a role in this specification language by giving typical examples.

2. Concrete Syntax

The concrete syntax of a language is specified by a grammar which is used as input of a parser generator. The first example shows a few grammar rules. The names to the left of the character = are rule names. Rule names that are referenced on some right-hand side are treated as nonterminal symbols. This can be seen by the transformation of the rules to BNF as given in the second example. On the right-hand side the symbols can be augmented with selector names such as Then:, Else:, Lop:, or Rop: in order to allow unambiguous access to their attributes. Most notably is the nested structure of the grammar rules which is established by the brackets < and >. The grammar rules are also regarded as classes and the nesting structure describes the class hierarchy. For example the rules Assign, Calls, and If are subclasses of Stat and Call0 and Call are in turn subclasses of Calls. The subclass relation is interpreted in two ways: First, a subclass inherits the right-hand side of its superclass. Second, a superclass acts as nonterminal and all its subclasses describe associated right-hand sides. This interpretation can be seen best by looking at the result of the transformation of the rules to BNF.

The relationship to object-orientation is as follows: Classes correspond to nonterminals, terminals, and grammar rules, objects correspond to language constructs such as statements, and features correspond to right-hand side symbols such as terminals and nonterminals as well as to attributes

Example: concrete syntax

```
Stat            = <
   Assign       = Addr ':=' Expr .
   Calls        = Ident <
      Call0     = .
      Call      = '(' Actuals ')' .
   > .
   If           = IF Expr THEN Then: Stat <
      If2       = ELSE Else: Stat .
   > .
> .
Expr            = <
   Plus         = Lop: Expr '+' Rop: Expr .
   Times        = Lop: Expr '*' Rop: Expr .
   '()'         = '(' Expr ')' .
   Addr         = <
      Name      = Ident .
      Index     = Addr '[' Expr ']' .
   > .
> .
```

Example: transformation to BNF

```
Stat = Addr ':=' Expr .
Stat = Ident .
Stat = Ident '(' Actuals ')' .
Stat = IF Expr THEN Stat .
Stat = IF Expr THEN Stat ELSE Stat .

Expr = Expr '+' Expr .
Expr = Expr '*' Expr .
Expr = '(' Expr ')' .
Expr = Ident .
Expr = Addr '[' Expr ']' .

Addr = Ident .
Addr = Addr '[' Expr ']' .
```

and attribute computations (see sections 3 and 4). The benefit of object-orientation in the area of concrete syntax is not very high. First, the order of the right-hand side symbols is significant and therefore only common beginnings of right-hand sides can be factored out. Second, attribute computations during parsing mainly describe the construction of a syntax tree. These computations are in general different for every grammar rule and therefore there is hardly a chance to factor out common computations.

3. Abstract Syntax and Symbol Table

Data structures in compilers such as a syntax tree or a symbol table are supported by a Cocktail tool for abstract syntax trees [Gro91]. Usually trees are built out of nodes that are linked together. Symbol tables describe the entities declared in programs. One possible implementation uses linked lists for the representation of sets of entities. The elements of those lists can be treated like tree nodes.

An abstract syntax is specified by a context-free grammar. Every rule of this grammar defines a so-called node type. The right-hand side of a rule lists the children and the attributes. The set of possible children is described by an associated node type. Attributes are enclosed in brackets [ ] and

typed with types of the implementation language. The subclass relation and the inheritance mechanism work similarly as in the case of the concrete syntax. For example every node type describing an expression (Expr, Binary, Addr, Ident, Index) has among others two attributes called Pos and Type.

Example: abstract syntax

```
Stat            = <
   Assign       = Addr Expr .
   Call         = [Ident: tIdent] [Pos: tPosition] Actuals .
   If           = Expr Then: Stat Else: Stat .
> .
Expr            = [Pos: tPosition] [Type: tType] <
   Binary       = Lop: Expr Rop: Expr [Operator] .
   Addr         = <
      Ident     = [Ident: tIdent] .
      Index     = Addr Expr .
   > .
> .
```

From such a specification a tool generates an abstract data type that handles attributed trees and graphs. This generated program module offers numerous tree/graph operations such as node constructors using a procedure notation to store attribute values and to take care of storage management, list processing (loop construct, reverse order), ASCII and binary graph reader and writer routines, as well as interactive graph browsers.

The next example uses the specification method to describe a symbol table. The node type Entity represents an abstract entity which is described by a name, a source position, and a pointer to the next entity. The node types Var, Const, and Proc specialize the abstract entity. They inherit the attributes and children from Entity and add further node specific features. Nodes of the specified node types can be chained into lists where the elements may be of different types as long as their types are subtypes of Entity.

Example: symbol table

```
Entity          = [Ident: tIdent] [Pos: tPosition] Next: Entity <
   Var          = [Type: tType] .
   Const        = Expr [Value: tValue] .
   Proc         = Formals <
      Func      = [Type: tType] .
   > .
> .
```

Again, we can identify the following relationship to object-orientation: Classes correspond to node types, objects correspond to tree nodes or list elements, and features correspond to children, attributes and attribute computations.

4. Attribute Grammars

Attribute grammars are one possibility for the specification of semantic analysis. They are processed by a tool that generates attribute evaluators [Groa]. Attribute grammars extend grammars by attributes and attribute computations. Attribute computations are formulated in the implementation language and enclosed in curly brackets. They may contain so-called attribute designators of the form child_name:attribute_name for the access of attributes of right-hand side symbols.

Additionally, the CHECK statement can be used to test context conditions. The Cocktail tools handle attribute grammars with single and multiple inheritance [Gro90, Grob].

While the previous section implemented the symbol table as a separate data structure, the following attribute grammar integrates abstract syntax and symbol table. The tree nodes for declarations are treated as symbol table entries. The node type Decl describes an abstract entity in the same way as Entity in the previous section. The attribute grammar computes additional links between the symbol table entries using the threaded attribute Objects[In/Out]. The node type Env represents scopes. It refers to a set of local entities (Objects) and a surrounding scope (Env).

Example: attribute grammar

```
Prog            = Proc .                                /* abstract syntax */
Decls           = <
   NoDecl       = .
   Decl         = [Ident: tIdent] [Pos: tPosition] Next: Decls <
      Var       = .
      Const     = .
      Proc      = Decls Stats <
         Func   = Type .
      > .
   > .
> .
Stats           = <
   NoStat       = .
   Stat         = Next: Stats <
      Use       = [Ident: tIdent] [Pos: tPosition] .
      Call      = [Ident: tIdent] [Pos: tPosition] .
   > .
> .

MODULE DefTab                 /* attribute computations for name analysis */

Decls           = [Objects: tTree THREAD] .     /* attribute declarations */
Decls Stats     = [Env: tTree] .
Use Call        = [Object: tTree] .
Env             = [Objects: tTree IN] Env IN .  /* node type for scopes   */

Prog    = { Proc:ObjectsIn      := mNoDecl ();
            Proc:Env            := mEnv (Proc:ObjectsOut, NoTree);    } .
Proc    = { Decls:ObjectsIn     := mNoDecl ();
            Stats:Env           := mEnv (Decls:ObjectsOut, Env);
            Decls:Env           := Stats:Env;                         } .
Decl    = { Next:ObjectsIn      := DEP (SELF, ObjectsIn);
            ObjectsOut          := Next:ObjectsOut;
            CHECK IdentifyObjects (Ident, ObjectsIn)->Kind == kNoDecl
            => Error (Pos, "identifier multiply declared", Ident);    } .
Use     = { Object := IdentifyWhole (Ident, Env);
            CHECK Object->Kind != kNoDecl
            => Error (Pos, "identifier not declared", Ident) AND_THEN
            CHECK Object->Kind == kVar || Object->Kind == kConst
            => Error (Pos, "variable or constant required", Ident);   } .
Call    = { Object := IdentifyWhole (Ident, Env);
            CHECK Object->Kind != kNoDecl
            => Error (Pos, "identifier not declared", Ident) AND_THEN
            CHECK Object->Kind == kProc
            => Error (Pos, "procedure required", Ident);              } .

END DefTab
```

The attribute grammar uses library routines such as IdentifyObjects and IdentifyWhole for lookup in the symbol table. Given an identifier both routines return a symbol table entry (Object). A special entry denotes undefined objects. While IdentifyObjects just searches in the current scope IdentifyWhole searches in all surrounding scopes as well.

In the example one rule for declarations (Decl) suffices to establish the links in the symbol table and to check for multiple declarations. Both computations are inherited by all node types describing declarations (Var, Const, Proc, Func). At the locations where identifiers are used (Use and Call) specific checks are necessary to insure that objects of the proper kind are used.

The following relationship to object-orientation can be identified: Classes correspond to node types, objects correspond to tree nodes, and features correspond to children, attributes, attribute computations, and checks of context conditions. Especially the inheritance of attribute computations is quite useful because it allows to factors out common computations and thus leads to short attribute grammars.

5. Pattern-Matching

The Cocktail Toolbox contains a tool for the transformation or the mapping of attributed trees to arbitrary output such as intermediate representations or target code [Gro92]. The tool is based on pattern-matching and pattern-matching has been extended to deal with the subtype relation of the node types in the input trees. From a very simplified point of view a pattern is either a variable or a node type followed by a list of patterns enclosed in parentheses. A pattern variable matches every node. A pattern of node type t matches if the type of the current node is a subtype of t and if the subpatterns match their corresponding subtrees.

Example: transformation routine

```
PROCEDURE Code (Decls)

Func (Ident, Pos, Next, Decls, Stats, Type) :-
   ...                  /* generate code for Func */
   Code (Next);
   .
Proc (Ident, Pos, Next, Decls, Stats) :-
   ...                  /* generate code for Proc */
   Code (Next);
   .
Decl (_, _, Next) :-    /* do not generate code for Var and Const */
   Code (Next);
   .
```

The patterns of the rules are checked in the given order. When a pattern matches then the associated statements are executed and afterwards the activation of the transformation procedure terminates. The patterns in the example have to be arranged exactly in the given order. A node of type Func would match all three patterns and a node of type Proc would match the last two patterns.

Taking advantage of the subclass relation among node types in pattern-matching allows to factor out common computations and thus leads to short problem solutions.

6. Summary

We surveyed the areas where object-orientation appears at the specification level in the Cocktail Toolbox for Compiler Construction: Concrete syntax, abstract syntax, symbol table, attribute grammars, and pattern-matching. The correspondence between the notions of the fields of object-oriented programming and compiler construction was described. The most benefit comes from the inheritance of attribute computations in attribute grammars and from the extension of pattern-matching to the subtype relation among node types, because this allows to factor out a significant amount of common computations.

References

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

[Gro90] 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.

[Gro91] J. Grosch, Tool Support for Data Structures, *Structured Programming 12*, (1991), 31-38.

[Gro92] J. Grosch, Transformation of Attributed Trees Using Pattern Matching, *LNCS 641*, (Oct. 1992), 1-15, Springer Verlag.

[Groa] J. Grosch, Ag - An Attribute Evaluator Generator, Cocktail Document No. 16, CoCoLab Germany.

[Grob] J. Grosch, Multiple Inheritance in Object-Oriented Attribute Grammars, Cocktail Document No. 28, CoCoLab Germany.

Contents