Another Kind of Modular Attribute Grammars

Beate Baum

Rostock University, Computer Science Department
Albert-Einstein-Str.21, O-2500 Rostock, Germany
bbaum@informatik.uni-rostock.dbp.de

Abstract. The decomposition of attribute grammars into modules is investigated. In our approach the alternative rules of a nonterminal may be separated into different modules. The aim of our concept is the generation of special grammars in respect to the design decisions of a compiler writer. Therefore, a module represents a concrete syntactic or semantic design decision. The import- and export-interface of a module contains not only attributes of its nonterminals, but also semantic functions and constructions of the syntax tree. In the module body the syntactic as well as the semantic rules are arranged.
The set of already implemented modules can be reused for new applications. A short explanation of these ideas within the system FLR is given.

1 Introduction

For the specification of large software systems a suitable structure concept is necessary. Many existing systems like libraries, data bases and retrieval systems support this aspect. They consist of a lot of components with delimited tasks. Not only the structure and the management of these systems, but also the reuse of already existing components is important. In the notion of modules as used in programming languages these and other properties were associated. A module utilizes following aspects:

- Structuring aspect:
  A software part is decomposed into subparts with delimited tasks supporting its clearness and readability. In imperative languages modules (Modula-2), units (Pascal) or packages (Ada) represent these subparts.
- Abstraction aspect:
  Each module abstracts from implementation details by dividing into two parts:
  * interface part, which declares public data;
  * implementation part, which defines private data and realizes the implementation of all data.
- Prevention aspect:
  The principle of "information hiding" is realized by the import-/export-behaviour of data differently implemented in the languages. Their common property is the access to the data only about a well-defined interface, e.g. only public data can be used for an application. Private data are hidden from unauthorized users.
- Reuse aspect:
  Already existing modules can be reused for new applications because of the possibility of their separate compilation (for example in Modula-2 and Ada).

These modularization aspects are suitable to describe the behaviour of specifications written as attribute grammars. Attribute grammars in the sense of Knuth [Knu 68] have naturally no modularization facilities.

Therefore, many authors have been concerned with modular decomposition of attribute grammars in different ways.

In [DuC 90] "modular attribute grammars" were defined as consisting of patterns and templates. In this sense, a module contains the patterns and templates for the computation of a special attribute.
For each pattern describing a set of syntactic rules of context-free grammars a set of corresponding templates is given. It is tried to compose the syntactic rules of a context-free grammar by giving patterns. If a pattern matches with a context-free rule a production of an attribute grammar is provided. The corresponding semantic functions are generated by the templates. In this way a complete attribute grammar is generated.
In [GGV 86] the principle of abstract data types is used in attribute grammars by defining an "attribute coupled grammar". A grammar can be seen as a semantic signature consisting of syntactic and semantic sorts as well as syntactic functions. For the grammars G_1 and G_2 an attribute coupling is defined as a mapping between the term algebra of G_1 into the term algebra of G_2. Such a mapping is characterized by an association of the semantic rules with G_1. The background of this approach is that each compilation phase can be considered as a translation of one program into another over the same semantic signature.
Dziolloß [Dzi 87] developed a data-flow language for describing the structure of programs which is more or less hierarchical. The underlying grammar is a so called "decomposition grammar" which can be seen as an attribute grammar with an empty set of terminals and an attachment of input and output attributes to the start symbol. Therefore, a program can be decomposed into modules, where each of them contains a subgrammar of the decomposition grammar.
The aim of describing the decomposition of hierarchical functional programs structurally by means of attribute grammars is found in [Sim 86]. In this approach the structure of modules is strictly hierarchical. All alternative rules of a nonterminal are collected into a module. An application of a rule is controlled by means of decomposition conditions.
The object orientation is also a possibility to modularize attribute grammars. One approach is found in the system TOOLS [Kos 91] using the correspondence of Paakki [Paa 91] ´nonterminal = class´. Also the correspondence ´production = class´ [Paa 91] is possible to use in attribute grammars.

In [Paa 91] modular attribute grammars are classified by decomposing the set of productions with regard to the set of nonterminals or attributes. The classification of Paakki is the basis for our investigations described in chapter 2. Another approach decomposing attribute grammars is introduced informally.

2 Modularization of Attribute Grammars

2.1 Known Modularization Concepts

A modularization of attribute grammars can be reached by decomposing the set of grammar rules. In [Paa 91] two forms of this decomposition are classified:

/1/ nonterminal = module or
/2/ attribute = module.

The principle of the first modularization form is the arrangement of all alternative rules of one nonterminal in one module. This correspondence ´nonterminal = module´ produces a lot of small modules building a hierarchy. This form of modularization decomposes the grammar by syntactic aspects because the rules of a nonterminal derive a concrete structure of the language. In the export-interface of each module the attributes of the corresponding nonterminal are noted. The module body contains the syntactic and semantic rules of the corresponding nonterminal.
The second form of modularization ´attribute = module´ uses the semantic aspect of grammars represented by their attributes. This means, all production rules containing nonterminals with the same attribute on the left-hand side of the semantic rules are composed together to a module. The number of the attributes of the nonterminal determines the occurrence of the alternative rules of the nonterminals in the modules. The semantic rules of the nonterminals are arranged to the modules corresponding to the evaluation of the attributes. Therefore, only few but more compact modules are created. No module hierarchy is built. The export-interface of each module contains the attribute with its corresponding nonterminals. In the module body the syntactic and semantic rules of the nonterminals are settled.
Both modularization forms utilize the structuring aspect of modules by their delimited syntactic or semantic tasks and the abstraction aspect by their separation into an interface and an implementation part. The principle of "information hiding" is solved by their import-/export-behaviour. A module provides certain attributes of their nonterminals used by other modules. Here only global nonterminals exist, local nonterminals are not allowed in both forms.

2.2 Chacterization of a New Modularization Concept

For our purpose, the use of modularized attribute grammars for compiler writing, the previous modularization forms are not suitable. There exist following disadvantages:
- the 1:1-correspondence from /1/, e.g. one module represents the alternative rules of one nonterminal,
- the existence of the same rules in more than one module from /2/,
- the non-existence of local data from /1/ and /2/.

The property of generating the same language by the "pure" attribute grammar as well as by the modularized attribute grammar is not sufficient to describe the behaviour of attribute grammars in compiler writing. Our aim is the generation of special languages belonging to a language family, which depends on the design decisions of a compiler writer. Each language is produced by an attribute grammar composed by modules. Each module represents a concrete syntactic or semantic design decision of the compiler writer. That´s why the alternative rules of a nonterminal can be located to different modules. The correspondence /1/ is extended by local nonterminals, whereas /2/ is delimited in that way, that only one module contains one certain semantic design decision in form of semantic functions.

The interface of a module contains not only attributes of their nonterminals, but also semantic functions and constructions of the syntax tree. The interface is divided into a defining part (export-interface) and an using part (import-interface). The export and import behaviour can be characterized "unique" if only two modules are involved in the exchange of information. In the module body the syntactic as well as the semantic rules are arranged.

In the following our approach of modularization is illustrated by the example of translation simple arithmetic expressions into abstract representations. Look at the non-modularized rules of the attribute grammar AG_1:

X -> Y, OP, Y.
X.code = op(OP.code, Y_1.code, Y_2.code)
typeTest(Y_1.type, Y_2.type, X.type)

OP -> '+'.
OP.code = add

OP -> '*'.
OP.code = mul

Y -> CONST.
Y.code = const(CONST.val)
Y.type = int

CONST -> ...
CONST.val = ...

Y -> ID.
Y.code = id(ID.val)
symtab(ID.val, Y.type)

ID -> ...
ID.val = ...

The nonterminals X, Y, OP, ID and CONST of the context-free basic grammar are augmented by *attributes*. Each synthesized attribute stands for a semantic meaning, *code* - for generated syntax trees, *type* - for the types of expressions and *val* - for the values of constants and identifiers. For evaluating *type* and *val* semantic actions are necessary: *symtab* - delivers the type of identifiers from the symbol table and *typeTest* - calculates the result type of expressions. The rules of constants and identifiers are omitted here.
Possible sentences of the generated language with their corresponding semantic meaning are:

1 + x ---> op(add, const(1), id(x))
y * z ---> op(mul, id(y), id(z))

The following figure represents a possible decomposition of the AG_1:

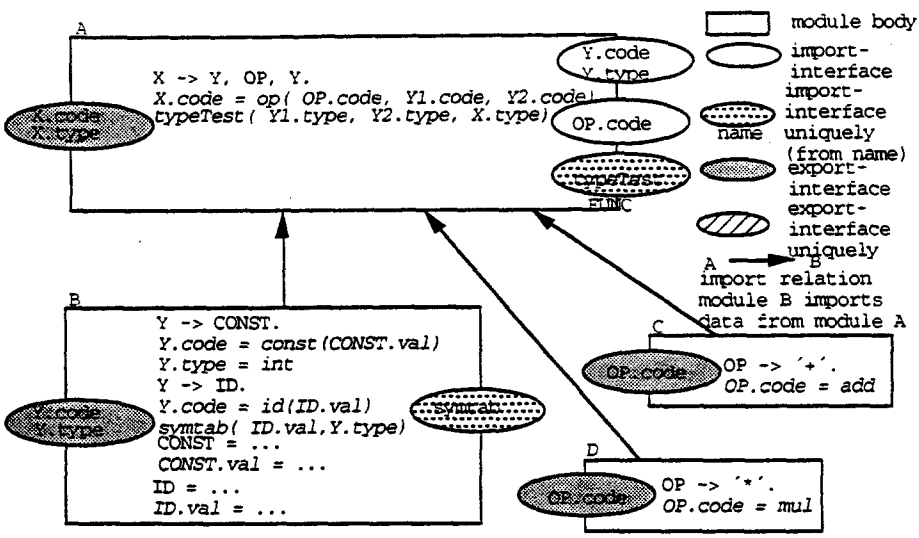

Fig. 1. First possible decomposition of the grammar AG_1

Figure 1 represents the decomposition of the grammar AG_1. Module A provides the attributes *code* and *type* of the nonterminal X to its environment and has to import the semantic function *typeTest* uniquely from module FUNC and the attributes *code* and *type* of the nonterminal Y as well as *code* of OP. Y is provided by module B, whereas OP can be provided by module C and/or D. The import of the OP-attributes depends on the decision of the compiler writer, whether he allows only the addition, only the multiplication or both operations.

The graphical description shows all possible import-/export-relations between the modules. All nonterminals which are not contained in the export-interface of a module are local. For our example in figure 1 the nonterminals ID and CONST are local in module B. The following notation expresses these aspects:

```
module A
 interface
  DefNt X(+,+);
  UsedNt Y(+,+), OP(+);
  UsedFu typeTest(-,-,+) unique FUNC;
  DefT op(+,+,+);
 body
  X -> Y, OP, Y.
       X.code = op( OP.code, Y1.code, Y2.code)
       typeTest( Y1.type, Y2.type, X.type)
end module;
```

```
module B
 interface
  DefNt Y(+,+);
  UsedFu symtab(-,+) unique;
  DefT const(+), id(+);
 body
  Y -> CONST.
       X.code=const(CONST.val)
       Y.type=int
  Y -> ID.
       code=id(ID.val)
       symtab( Id.val, Y.type)
  ID -> ...
       Id.val = ...
  CONST -> ...
       Const.val = ...
end module;
```

```
module C
 interface
  DefNt OP(+);
  DefT add;
 body
  OP -> '+'.
       OP.code= add
end module;
```

```
module D
 interface
  DefNt OP(+);
  DefT mul;
 body
  OP -> '*'.
       OP.code= mul
end module;
```

The import-/export-behaviour of these modules is noted in the following way. Here, each exported symbol is defined and each imported symbol is used. Symbols may be nonterminals (DefNt and UsedNt resp.), names of semantic functions (DefFu and UsedFu resp.) or constructions of the syntax tree (DefT and UsedT resp.). For each symbol the number of attributes or parameters and the direction of their evaluation: + for bottom-up and - for top-down are noted.
The unique-statement specifies, that only one module has to import or export data. In association with a module name, unique determines exactly which module imports or exports these data, for example, by

UsedFu *typeTest(-,-,+)* unique *FUNC;*

the semantic function *typeTest* is uniquely provided by the module FUNC). Without a module name unique provides data only by one module which is not specified. In the above mentioned example the semantic function *symtab* is provided by the module SYMTAB. But it is also possible to define a new module exporting this function.

Because of the possible occurrence of semantic functions in the import-interface of a module there have to be modules providing these functions. In our example these are the semantic functions *symtab* and *typeTest*. The realization of these functions depends on a concrete implementation language. Therefore, only the skeletons of the modules exporting the functions are given.

```
module SYMTAB
 interface
   DefFu symtab(-,+) unique;
 body ...
end module;
```

```
module FUNC
 interface
   DefFu typeTest(-,-,+) unique;
 body ...
end module;
```

Another decomposition of the grammar AG_1 is possible:

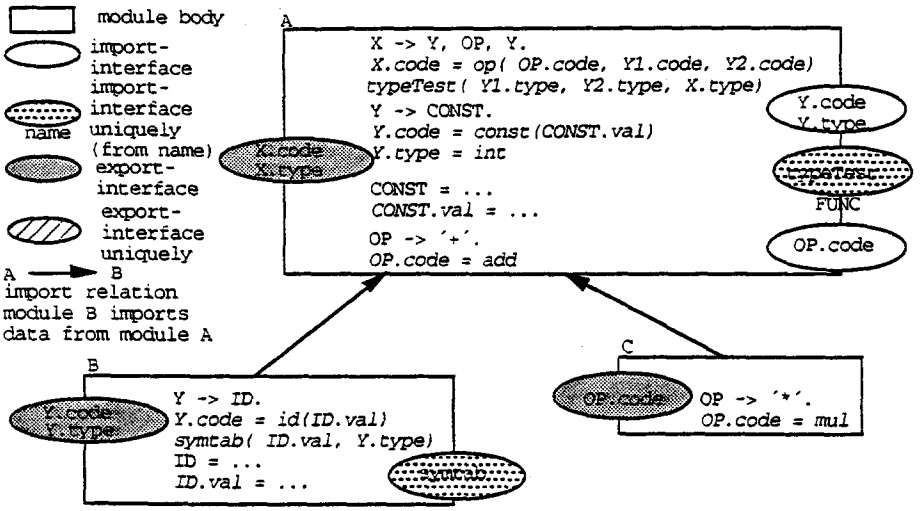

Fig.2. Second possible decomposition of the grammar AG_1

Here, a particular basic grammar representing a complete subgrammar is noted in module A. This basic grammar can be extended by additional syntactical constructions which are contained in the modules B and C. Therefore, although a rule for the nonterminal Y exists in the module A an import of the attributes of Y from B is allowed. In this example the nonterminals CONST and ID have local character. Look at the notation of these modules:

```
module A
 interface
   DefNt X(+,+);
   UsedNt Y(+,+), OP(+);
   UsedFu typeTest(-,-,+) unique FUNC;
   DefT op(+,+,+), const(+), add;
 body
   X -> Y, OP, Y.
        X.code = op( OP.code, Y1.code, Y2.code)
        typeTest( Y1.type, Y2.type, X.type)
   Y -> CONST.
        Y.code = const(CONST.val)
        Y.type = int
   CONST -> ...
        CONST.val = ...
   OP -> '+'.
        OP.code = add
end module;
```

```
module B
 interface
   DefNt Y(+,+);
   UsedFu symtab(-,+) unique;
   DefT id(+);
 body
   Y -> ID.
        Y.code = id(ID.val)
        symtab( ID.val, Y.type)
   ID -> ...
        ID.val = ...
end module;

module C
 interface
   DefNt OP(+);
   DefT mul;
 body
   OP -> '*'.
        OP.code = mul
end module;
```

3 Conclusions

In the previous chapters known modularization concepts were described. A new approach was introduced to support the design decisions for a compiler writer. Our modularization concept does not always include the fact from which concrete module data have to be imported. It is only determined that one module at least has to provide these data. Data are nonterminals with their attributes, semantic functions or constructions of the syntax tree. The reason for this property is the possible separation of alternative rules of a nonterminal into different modules.
It is our aim to introduce this modularization concept in the system FLR. FLR (Fast Laboratory for Recomposition) [For 91] developed at our department is a tool for generating text documents, programs in a programming language, formal specifications and grammars. FLR utilizes the generative aspect of attribute grammars. The knowledge of a problem domain is described by special attribute grammar rules. The solution of a special problem is generated by selecting special rules from the knowledge base.
In our approach FLR is used for generating attribute grammars for compiler construction. This means, the FLR-grammar rules contain grammar parts of compiler descriptions for imperative languages. So, the compiler writer has the possibility to select certain grammar parts for his application. The result of this generation is a concrete grammar.
With the help of modularization attribute grammars certain parts of the grammar should be reused. Therefore, mechanisms in FLR supporting the modularization are needed. For example, symbol tables collecting information about each symbol, like their number and direction of parameters, their internal representation, the names of their export modules, information about their using, are necessary. By means of these tables the import-/export-relations between the modules have to be checked, the interfaces of user-defined modules have to be controlled with regard to the use of already existing names of nonterminals or functions and local nonterminals have to be renamed. Also a type system can help to find conflicts.

References

[AlM 91] Alblas,H.; Melichar,B. (Eds.): Attribute Grammars, Applications and Systems, SAGA, Prague, June 1991, LNCS 545, Springer-Verlag Berlin.

[DuC 90] Dueck,G.D.P.; Cormack,G.V.: Modular Attribute Grammars, The Computer Journal, vol. 33, no. 2, 1990, pp. 164-172.

[Dzi 87] Dziolloß,A.: Datenflußorientierte Programmierung mit Attributgrammatiken, Dissertation A, TU Dresden, Informatikzentrum, 1987.

[For 91] Forbrig,P.: Using the generative aspect of attribute grammars in a knowledge based way, in: [AlM 91], pp. 438-459.

[GGV 86] Ganzinger,H.; Giegerich,R.; Vach,M.: MARVIN - A Tool for Applicative and Modular Compiler Specifications, Universität Dortmund, Abteilung Informatik, Forschungsbericht Nr. 220, 1986.

[Kas 91] Kastens,U.: Attribute Grammars as a Specification Method, in: [AlM 91], pp.16-47.

[Knu 68[Knuth,D.E.: Semantics of Context-Free Languages, Mathematical System Theory 2, 2, 1968, pp. 127-145.

[Kos 91] Koskimies, K.: Object-Orientation in Attribute Grammars, in: [AlM 91], pp. 297-329.

[Paa 91] Paakki,J.: Paradigms for Attribute-Grammar-Based Language Implementation, Department of Computer Science, University of Finland, Helsinki, Report A-1991-1.

[Sim 86] Simon,E.: A new programming methodology using attribute grammars, Acta Cybernetica, tom.7, fasc.4, 1986, pp. 425-436.