Institut für
Technische Informatik und
Kommunikationsnetze

R. Marti, T. Murer

Extensible Attribute Grammars

TIK-Report
Nr. 92-6 (Dezember 1992)

ETH Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich
Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo

R. Marti (marti@tik.ethz.ch), T. Murer (murer@tik.ethz.ch):
Extensible Attribute Grammars
December, 1992
Version 1
TIK-Report Nr. 92-6

Laboratory of Computer Engineering and Networks,
Swiss Federal Institute of Technology (ETH) Zurich

Institut für Technische Informatik und Kommunikationsnetze,
Eidgenössische Technische Hochschule Zürich

Gloriastrasse 35, ETH-Zentrum, CH-8092 Zürich, Switzerland

Abstract

This report introduces a new idea to make attribute grammars (AG) extensible. Both the context-free grammar and the attribution system of an AG may be extended. This concept is a valuable structuring technique when defining language-based programming environments or compilers. For instance, it allows passes of a multipass compiler to be decomposed into different grammar levels, which renders the definition much clearer. Another application consists of defining an interface for an external tool (browser) on an independent grammar level neatly separated from the actual language definition. The concept of extensible attribute grammars is first introduced using a formal model, and thereafter practical examples demonstrate possible applications.

Contents

R. Marti (marti@tik.ethz.ch), T. Murer (murer@tik.ethz.ch):
Extensible Attribute Grammars
December, 1992
Version 1
TIK-Report Nr. 92-6

Laboratory of Computer Engineering and Networks,
Swiss Federal Institute of Technology (ETH) Zurich

Institut für Technische Informatik und Kommunikationsnetze,
Eidgenössische Technische Hochschule Zürich

Gloriastrasse 35, ETH-Zentrum, CH-8092 Zürich, Switzerland

1. Introduction

During the last few years some promising systems for the automatic generation of language-based programming environments have been developed. Possibly the best known of these are the "Synthesizer Generator" Project [RT89], the PSG-System [BS86] and the FNC-2 Project [JPJDL90]. All projects have in common that the generation of the programming environment is based on a formal programming language specification. The objective of these projects was primarily the efficiency of the generated environment, and to a lesser extent the simplification of the development process of the formal specification. For example, new algorithms for the incremental context analysis or for space-efficient representation of programs have been discussed [Reps83, Filé86, Sne86]. Therefore, far more progress has been made in the implementation concepts of the environments than in the specification languages. Attribute grammars [Kas82, RT89, JPJDL90, Mös90] or algebraic specification [Kli90] are used as a basis for most specification languages. Both formalisms allow specification of the syntax and the static semantics of a programming language.

Regarding examples of such language specifications, the complexity that is required to produce just the front end of a compiler for a programming language becomes evident. Table 1 [Ode89] shows some examples measured by the number of lines the corresponding language specification counts (without code generation). Obviously, the complexity of the defined language as well as the type of specification language influence the specification's size. The last three examples are used solely to define the corresponding language and are not suitable for generating an efficient compiler front-end.

Specification Language	Defined Language	Size in Lines	Complete?
ALADIN (AG)	Pascal	2997	yes
ALADIN (AG)	Ada	14922	yes
VDM	Modula-2	ca. 3300	no
ADL	Turing	ca. 1200	no
CADET	Oberon	699	yes

Table 1: Some examples of language specifications

The entry in the last column concerns the language specification's completeness. The listed Modula-2 specification [And89] was a pre-release version and was not yet complete. The

Turing definition appeared in [HMRC88]. Some simplifying assumptions have been made in the definition, e. g., predefined identifiers are not mentioned and opaque types are not treated in full. ALADIN is the specification language used in the GAG System [Kas82] and is based on attribute grammars. The definition of Pascal may be found in [KHZ82], the one of Ada in [UDP82] and the Oberon definition in [Ode89].

Compared to modern programming languages such as object-oriented or functional languages, the specification languages are far less advanced, specifically concerning the possibilities of abstraction, modularization, extensibility or reusability. Specifying a grammar in a modular way is limited, because the functionality of such modules often concerns the whole grammar. For example, if a grammar specification is decomposed according to the non-terminal declarations (each non-terminal with its productions and context-rules in a separate module), then the declaration- or type-compatibility rules would be spread over several modules. Thus, a better approach would allow to bring those parts of a specification together, e.g., declaration analysis and type compatibility in two separate modules for the whole grammar. Thus, most of current specifications are monolithic definitions of syntax and semantics with little possibility for structurization.

If the specification language is not only to be used for the generation of a compiler but for a whole programming environment (including a compiler and several editors), it must live up to the software engineering standards of a modern programming language. Therefore, this paper presents an extension-model of attribute grammars comparable to the extension concept of object-oriented languages [Meyer88]. Here, starting with a base grammar, a new grammar may be defined by extending the syntax and static semantics. This extension-model allows, e.g., different compiler passes of a multipass compiler to be defined independently on different grammar levels, something not possible in today's specification languages. Furthermore, grammar extensions may be used to define interfaces between external tools and the programming environment's kernel. For instance, for an additional tool such as a browser or a graphical editor, new information to represent the program externally may be introduced. This allows the decomposition of the actual language definition and other subsystems of a programming environment.

In the subsequent chapter the attribute grammars are briefly explained. Readers familiar with attribute grammars may skip this. It is followed by the chapter containing the definition of extensible attribute grammars. In the fourth chapter some examples depicting previously mentioned applications of grammar extensions are given which has been implemented with the GIPSY System [MM92]. Finally, some conclusions are given together with an outlook of our next research activites.

2. Attribute Grammars

Attribute grammars have been introduced by Knuth in 1968 [Knu68]. They are used to define grammars belonging to a more powerful class than the one of context-free grammars or to define a language's semantics. For instance, they are well suited to define the context rules of programming languages in addition to their context-free syntax. An example of this is the rule that every designator in a program must be declared (declaration rule) or that an expression's type in an assignment must be compatible with the assigned variable's type. These rules limit the set of valid sentences but cannot be generally expressed by a context-free grammar. Consequently, attribute grammars represent a device to additionally define the context-sensitive parts of a language. Hereby, one commences with a superset of valid sentences defined by a context-free grammar. Thereafter, the actual grammar is defined by applying restricting context rules to specify the language. In the following, the class of context-free grammars is defined in short [DW83].

Definition 2.1: **Context-Free Grammars (CFG)**
A context-free grammar $G = (T, N, Z, P)$ consists of a set of terminal symbols T, a set of non-terminal symbols N, a set of context-free productions P and a start non-terminal $Z \in N$. A context-free production is an expression $X \to w$, where $X \in N$ and $w \in (T \cup N)^*$. The union $T \cup N$ is called the alphabet of the grammar. If p stands for the production $X \to u$ and $u, v \in (T \cup N)^*$, then $u \underset{p}{\Rightarrow} v$ means that there are words $x, y \in (T \cup N)^*$ such that $u = xXy$ and $v = xwy$. The set of productions defining a non-terminal X is denoted by $P(X)$, and $L(G)$ is called the *language generated* by G and is defined by $L(G) = \{u \in T^* \mid Z \Rightarrow^* u\}$

As an example, a CFG defining the syntax of arithmetical equations is presented:

$G_E = (T, N, Equation, P)$

$T = \{ int, +, -, *, /, (,), = \}$

$N = \{ Equation, Sum, Product, Factor, AddOp, MulOp \}$

$P = \{$
p_0: *Equation* $\to$ *Sum* = *Sum*,
p_1: *Sum* $\to$ *Product*,
p_2: *Sum* $\to$ *Product AddOp* Sum,
p_3: *Product* $\to$ *Factor*,
p_4: *Product* $\to$ *Factor MulOp Product*,
p_5: *Factor* $\to$ *int*,
p_6: *Factor* $\to$ *(Sum)*,
p_7: *AddOp* $\to$ +,
p_8: *AddOp* $\to$ -,
p_9: *MulOp* $\to$ *,
p_{10}: *MulOp* $\to$ / $\}$

Fig. 2.1: Example of a context-free grammar

The grammar G_E defined above also permits equations that are mathematically false: 3 + 5 = 9. Now, to allow only correct equations to belong to the defined language, the grammar needs to be further restricted. To enable this, attribute grammars associate attributes with the non-terminal symbols of the underlying context-free grammars. Suppose an attribute *value* is associated with the non-terminal *Sum* and its value corresponds to the sum's arithmetical value. In this case a simple comparison of the left and right sums' *value* attributes indicates whether the sentence is correct.

Attribute values are defined by the corresponding attribution rules. An attribution rule is always associated with a production. This leads to the following definition of attribute grammars [DJL88]:

Definition 2.2: **Attribute Grammars (AG)**
An attribute grammar $AG = (G, A, R)$ consists of a context-free grammar G, a set of attributes A and a set of attribution rules R. With each non-terminal symbol $X \in N$, there are two associated finite sets $Inh(X)$ and $Syn(X)$ attributes, the *inherited* and *synthesized* attributes. These sets verify: for all $X, Y \in N$, $Inh(X) \cap Syn(Y) = \varnothing$.
We note:

$A(X) = Inh(X) \cup Syn(X)$ the set of attributes associated with non-terminal X
$Inh = \bigcup_{X \in N} Inh(X)$
$Syn = \bigcup_{X \in N} Syn(X)$
$A = Inh \cup Syn$

In a given production, an attribute *a* associated with the i-th occurrence of a symbol Y is an *attribute occurrence* and is noted $Y_i.a$. If *i=0* and there is no other symbol Y in the production then the index i may be omitted resulting in $Y.a$.

For each production $p: X_0 \to X_1 X_2 \ldots X_n \in P$ there is a set of *attribution rules* defining the computation of the elements of $Syn(X_0)$ and $Inh(X_i)$ for $1 \le i \le n$ in terms of the elements of $A(X_i)$ for $0 \le i \le n$:

$$X_j.a := f(X_k.b, \ldots, X_m.z).$$

In this attribution rule, f is the name of a function with attributes as actual parameters. $R(p)$ denotes the set of attribution rules associated with production p.

Definition 2.3: **Set of Defining Occurrences of Attributes**
For each production $p: X_0 \to X_1 \ldots X_n \in P$ of an attribute grammar the set of defining occurrences of attributes is given by:

$$AD(p) = \{X_i.a \mid X_i.a := f(\ldots) \in R(p)\}$$

An attribute $X.a$ is called *synthesized* if there exists a production $p: X \to u$ with $X.a \in AD(p)$. It is called *inherited* if there exists a production $q: Y \to vXw$ and $X.a \in AD(q)$.

Thus, the grammar AG_E of valid arithmetical equations may be defined as follows:

$AG_E = (G_E, A, R)$

$A = Inh \cup Syn$

$Inh = Inh(AddOp) \cup Inh(MulOp)$

$Inh(AddOp) = \{lOp, rOp\}$

$Inh(MulOp) = \{lOp, rOp\}$

$Syn = Syn(Equation) \cup Syn(Sum) \cup Syn(Product) \cup Syn(Factor) \cup Syn(AddOp) \cup Syn(MulOp)$

$Syn(Equation) = \{equal\}$

$Syn(Sum) = \{value\}$

$Syn(Product) = \{value\}$

$Syn(Factor) = \{value\}$

$Syn(AddOp) = \{sum\}$

$Syn(MulOp) = \{prod\}$

$R = R(p_0) \cup R(p_1) \cup R(p_2) \cup R(p_3) \cup R(p_4) \cup R(p_5) \cup R(p_6) \cup R(p_7) \cup R(p_8) \cup R(p_9) \cup R(p_{10})$

$R(p_0) = \{Equation.equal := Sum_0.value = Sum_1.value, TEST(Equation.equal)\}$

$R(p_1) = \{Sum.value := Product.value\}$

$R(p_2) = \{Sum_0.value := AddOp.sum, AddOp.lOp := Product.value, AddOp.rOp := Sum_1.value\}$

$R(p_3) = \{Product.value := Factor.value\}$

$R(p_4) = \{Product_0.value := MulOp.prod, MulOp.lOp := Factor.value, MulOp.rOp := Product_1.value\}$

$R(p_5) = \{Factor.value := IntValue(int)\}$

$R(p_6) = \{Factor.value := Sum.value\}$

$R(p_7) = \{AddOp.sum := AddOp.lOp + AddOp.rOp\}$

$R(p_8) = \{AddOp.sum := AddOp.lOp - AddOp.rOp\}$

$R(p_9) = \{MulOp.prod := MulOp.lOp * AddOp.rOp\}$

$R(p_{10}) = \{MulOp.prod := MulOp.lOp / MulOp.rOp\}$

Fig 2.2: Example of an attribute grammar

The attribution rules of a production imply a partial order of calculation. The order may be expressed by a relation $DL(p)$ (local dependency relation):

$$DL(p) = \{(X_i.a, X_j.b) \mid X_j.b := f(\ldots, X_i.a, \ldots) \in R(p)\}.$$

The relation may be illustrated by a so-called *dependency graph*. In the graph, the attributes represent the nodes and the attribute pairs of the relation $DL(p)$ represent the edges, i. e. if $(X.a, Y.b) \in DL(p)$ then an edge exists from node $X.a$ to node $Y.b$.

For a given structure tree *t* the so-called *compound dependency relation DC(t)* is obtained by assembling the local dependency relations of the concerned productions according to their syntactical structures. The newly-formed relation may also be described by a *compound dependency graph*.

The following figures show examples of local dependency graphs originating from the AG described above as well as the compound dependency graph for the sentence "3*5 = 10+5" in the same AG.

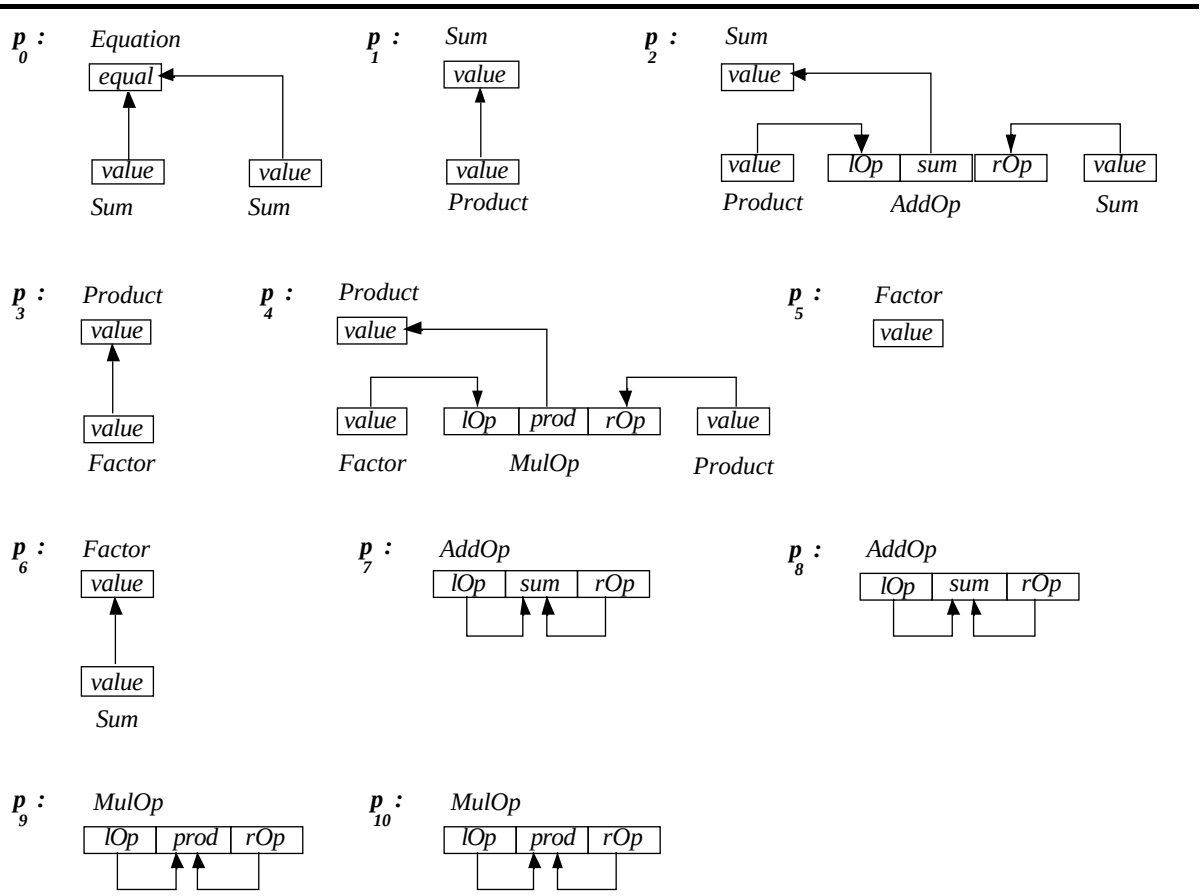

Fig. 2.3: local dependency graphs of AG_E

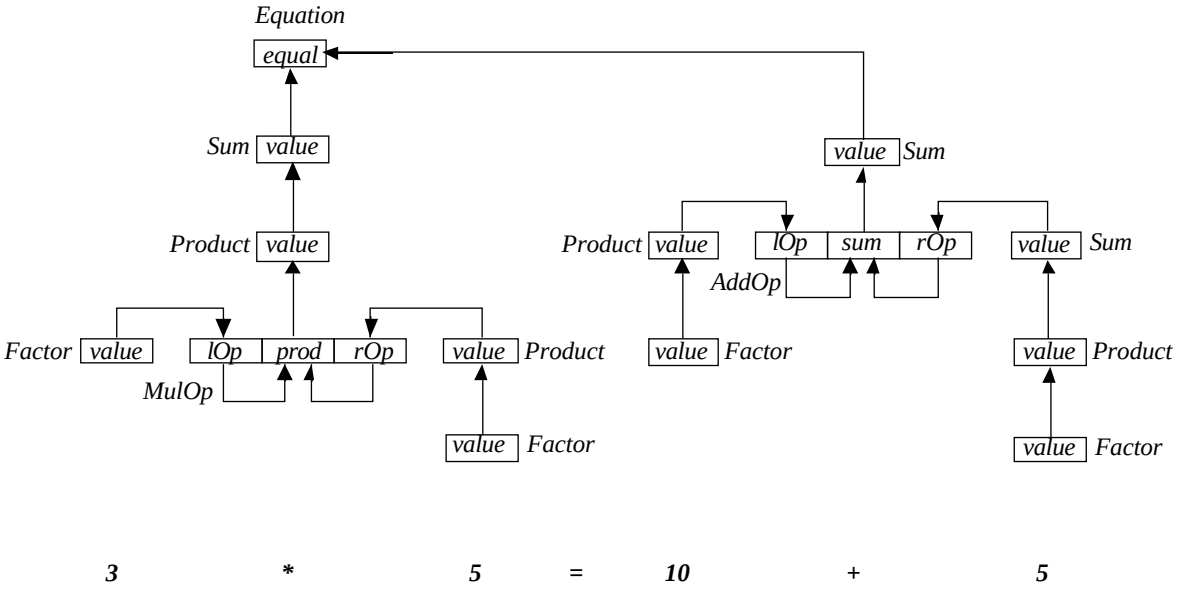

Fig. 2.4: compound dependency graph for the sentence "3*5 = 10+5"

Definition 2.4: **Completeness of an Attribute Grammar**

An attribute grammar is complete if the following statements hold for all $X \in N$:

(1) For all $p: X \to u \in P, Syn(X) \subseteq AD(p)$

(2) For all $q: Y \to vXw \in P, Inh(X) \subseteq AD(q)$

(3) $Syn(X) \cap Inh(X) = \varnothing$.

Definition 2.4: **Well-Defined Attribute Grammars**

An attribute grammar is well-defined (WAG) if, for each structure tree corresponding to a sentence of *L(AG)*, all attributes are effectively computable.

The same definition may be given as a corollary: An attribute grammar is well-defined iff it is complete and the compound dependency graph for each sentence of *L(AG)* is cycle-free [WG84].

3. Extensible Attribute Grammars

The extensibility of attribute grammars is achieved by making the syntax and static semantics of a given language extensible. Thereby the extensibility may be used for different purposes, analogous to the methods of object-oriented programming. On the one hand, it provides an elegant method of structurization for complex grammars, e. g. if the various passes of a compiler are defined independently on different grammar levels. On the other hand, extensions may be added later by the user of a generated programming environment, e. g. to integrate new

tools into the environment or to further restrict the language by adding new context rules. Some specific applications will be depicted in chapter 4.

The definition of an extension of an attribute grammar will be presented in steps. As has been previously mentioned, both the syntax and the static semantics should be extensible. The extension is to be understood as a *structural extension* of a language. Therefore, primarily the language's grammar is extended and not the language itself. It is not the point to define a language extension by adding new sentences to the language's set of valid sentences. Rather, the CFG and the attribution system of an AG should be extensible. As will be seen, in general for an extension of a base grammar G_0 by a grammar G_1 it is true that $L(G_0) \not\subseteq L(G_1)$. First, the extension of a context-free grammar is considered.

3.1. Extensibility of Context-Free Grammars

A context-free grammar may be extended by enhancing the alphabet and/or the productions. The extension of the alphabet is achieved by adding new terminal or non-terminal symbols. The extension of the productions is slightly more complex and is detailed below. In any case it is essential that an extension of a context-free grammar again produces a context-free grammar.

Definition 3.1: **Extensions of Productions**
Let $G_0 = (T_0, N_0, P_0, Z_0)$, $G_1 = (T_1, N_1, P_1, Z_1)$ be two context-free grammars and $E \subseteq N_0 \times N_1$ a relation, then production $p_1 : Y \to v \in P_1$ is called an *extension* of $p_0 : X \to u \in P_0$ *with respect to* E, written $p_1 \succ_E p_0$, iff the following holds:

(1) $(X, Y) \in E$

(2) $v = u$ or v originates from u by applying the following modifications:
- *insertion* of symbols of the set: $A_0 \cup A_1 - \{s | s \text{ is in } u\} - \{s' | (s, s') \in E^*$ and s is in $u\}$ with $A_0 = T_0 \cup N_0$, $A_1 = T_1 \cup N_1$ and E^* be the transitive closure of E.
- *replacement* of a non-terminal symbol s by a non-terminal s' where $(s, s') \in E^*$.

For the insertion of symbols, the valid set of insertable symbols needs to be restricted to avoid any ambiguity about which symbols have been inserted, and to enable the decision about whether a symbol has been introduced by replacement or insertion. Therefore, from the alphabet $A_0 \cup A_1$ the set of symbols already occurring in the word $\{s | s \text{ is in } u\}$ or which could be introduced by replacement $\{s' | (s, s') \in E^*$ and s is in $u\}$ needs to be subtracted. The meaning of the relation E will be clarified in the following definition of extensions of a context-free grammar.

Definition 3.2: **Extensions of a Context-Free Grammar**
Let $G_0 = (T_0, N_0, P_0, Z_0)$ and $G_1 = (T_1, N_1, P_1, Z_1)$ be two context-free grammars, then G_1 is called an *extension* of G_0, written $G_1 \succ G_0$, iff there exists a relation $E \subseteq N_0 \times N_1$ with the following properties:

(1) For each non-terminal $X \in N_0$, a non-terminal $X' \in N_1$ exists such that $(X, X') \in E$

(2) For each pair $(X, X') \in E$ it is true that for each production $p \in P(X)$, a production $p' \in P(X')$ exists such that $p' \underset{E}{\succ} p$.

A relation $E \subseteq N_0 \times N_1$ fulfilling properties (1)-(2) is called an *extension relation*. If for two grammars $G_1 \succ G_0$ holds, then G_0 is called the *context-free base grammar* of G_1. Note: Since the predicate '$\succ$' defines only a structural extension of a grammar, $L(G_0) \subseteq L(G_1)$ does not hold in general! The notation $G_1 \underset{E}{\succ} G_0$ indicates that $G_1 \succ G_0$ holds for a given relation E. Let an extension of the context-free grammar G_E from chapter 2 serve as an example. Thereby, constant declarations and real numbers are added.

$G_E' = (T', N', Equation, P')$

$T' = (\ int, \mathbf{real}, +, -, *, /, (,), =, \mathbf{ident})$

$N' = (Equation, \mathbf{Decl}, Sum, Product, \mathbf{RIFactor}, AddOp, MulOp)$

$P' = (p_0: Equation \rightarrow \mathbf{Decl}\ Sum = Sum,$
$\mathbf{p_0'}: \mathbf{Decl} \rightarrow \mathbf{ident = int\ Decl},$
$\mathbf{p_0''}: \mathbf{Decl} \rightarrow \mathbf{ident = real\ Decl},$
$\mathbf{p_0'''}: \mathbf{Decl} \rightarrow ,$
$p_1: Sum \rightarrow Product,$
$p_2: Sum \rightarrow Product\ AddOp\ Sum,$
$p_3: Product \rightarrow \mathbf{RIFactor},$
$p_4: Product \rightarrow \mathbf{RIFactor}\ MulOp\ Product,$
$\mathbf{p_5}: \mathbf{RIFactor} \rightarrow \mathbf{int},$
$\mathbf{p_5'}: \mathbf{RIFactor} \rightarrow \mathbf{real},$
$\mathbf{p_5''}: \mathbf{RIFactor} \rightarrow \mathbf{ident},$
$p_6: \mathbf{RIFactor} \rightarrow (Sum),$
$p_7: AddOp \rightarrow +,$
$p_8: AddOp \rightarrow -,$
$p_9: MulOp \rightarrow *,$
$p_{10}: MulOp \rightarrow / \}$

Fig. 3.1: A context-free grammar extending G_E

The following relation E may be used to show that G_E' is indeed an extension of grammar G_E according to definition 3.2:

$E = \{(Equation, Equation), (Sum, Sum), (Product, Product), (Factor, RIFactor), (AddOp, AddOp), (MulOp, MulOp)\}$

3.2. Extensibility of Attribute Grammars (EAG)

With the possibility of extending context-free grammars a basis for extensible AGs has been established. The extensibility of the attribution system remains to be discussed.

Definition 3.1 guarantees that no non-terminal symbols "disappear" on the right hand side of productions upon extension of a CFG. At most they are replaced by "extended" non-terminals

in the new grammar. To allow the continued usage of the existing attribution rules in the extended grammar, in the attribution system the "extended" non-terminals are required to contain all the attributes of their respective "base" non-terminals $(X, X') \in E \Rightarrow A(X) \subseteq A(X')$.

As may be inferred from definition 3.1, new symbols (terminals or non-terminals) may be added on the right side of the productions (insertion) upon extension of a CFG. In the case of non-terminals, this introduces new syntactical structures which may themselves be attributed. In order to allow the new parts of the attribution system to be joined with the existing ones, it is necessary that attribution rules of the base grammar are extensible.

Definition 3.3: **Extensions of Attribution Rules**
Let $AG_0 = (G_0, A_0, R_0)$ and $AG_1 = (G_1, A_1, R_1)$ be two attribute grammars and $E \subseteq N_0 \times N_1$ a relation, then an attribution rule $r' : X'_j.a := f'(X'_k.a, \ldots, X'_m.a, Y_n.a, \ldots, Y_q.a) \in R_1(p')$ is called *an extension* of the rule $r : X_j.a := f(X_k.a, \ldots, X_m.a) \in R_0(p)$ *with respect* to E, written $r' \succ_E r$, iff the following holds:

(1) $p' \succ_E p$

(2) For the functions f and f' holds one of the following statements:

(2.1) $Arity(f') = Arity(f)$ and f' is equivalent to f: $f' \equiv f$

(2.2) $Arity(f') > Arity(f)$ and there are values $v_1, \ldots, v_k$ for which the following holds:
$f'(x_1, \ldots, x_m, v_1, \ldots, v_k) = f(x_1, \ldots, x_m)$ for any x_i with $1 \le i \le m$, $m = Arity(f)$,
$k = Arity(f') - Arity(f)$

(3) For each attribute occurrence $X_i.a$ in r and the corresponding occurrence $X'_i.a$ in r' the statement $(X_i, X'_i) \in E$ holds.

Thus, an attribution rule may be extended by replacing attributes, associated with non-terminals of the base grammar, with the corresponding attributes, associated with extended non-terminals, or by enhancing the arity of the function. In both cases, the "meaning" of the function has to be retained (cf. 2.1 and 2.2).

This leads to the definition of valid extensions of an attribute grammar:

Definition 3.4: **Extensions of an Attribute Grammar**
Let $AG_0 = (G_0, A_0, R_0)$ and $AG_1 = (G_1, A_1, R_1)$ be two attribute grammars, then AG_1 is called *an extension* of AG_0, written $AG_1 \succ AG_0$, iff there exists a relation $E \subseteq N_0 \times N_1$ with the following properties:

(1) $G_1 \succ_E G_0$

For each pair $(X, X') \in E$ the following holds:

(2) $A(X') \supseteq A(X)$

(3) To each attribution rule r, associated with a production p defining X, there exists an attribution rule r', associated with a production p' defining X', such that $r' \succ_E r$ holds.

Property (1) guarantees AG_1 to be a valid extension of AG_0 only if the CFG of AG_0 is a context-free base grammar of the CFG of AG_1; (2) requires an extended non-terminal to contain all the attributes of its respective base non-terminal, whereas (3) ensures all attribution rules of the base grammar to occur as extensions in the extended grammar too. If for two grammars $AG_1 \succ AG_0$ holds, then AG_0 is called the *attribute base grammar* of AG_1.

An extension of the attribute grammar AG_E from chapter 2 serves as an example. In the new AG, the extended CFG G_E' in chapter 3 has been taken into consideration. The parts printed in bold indicate the added attributes and their attribution rules.

$AG_E' = (G_E', A, R)$

$A = Inh \cup Syn$

$Inh = \mathbf{Inh(Sum)} \cup \mathbf{Inh(Product)} \cup \mathbf{Inh(RIFactor)} \cup Inh(AddOp) \cup Inh(MulOp)$

$\mathbf{Inh(Sum)} = \mathbf{\{consts\}}$

$\mathbf{Inh(Product)} = \mathbf{\{consts\}}$

$\mathbf{Inh(RIFactor)} = \mathbf{\{consts\}}$

$Inh(AddOp) = \{ lOp, rOp\}$

$Inh(MulOp) = \{ lOp, rOp\}$

$Syn = Syn(Equation) \cup Syn(Sum) \cup Syn(Product) \cup Syn(Factor) \cup Syn(AddOp) \cup Syn(MulOp)$

$Syn(Equation) = \{ equal, consts\}$

$Syn(Sum) = \{ value \}$

$Syn(Product) = \{ value \}$

$Syn(Factor) = \{ value \}$

$Syn(AddOp) = \{ sum\}$

$Syn(MulOp) = \{ prod \}$

$R = R(p_0) \cup R(p_1) \cup R(p_2) \cup R(p_3) \cup R(p_4) \cup R(p_5) \cup R(p_6) \cup R(p_7) \cup R(p_8)$

$R(p_0) = \{Equation.equal := Sum_0.value = Sum_1.value,$ $\mathbf{Equation.consts := Decl.consts,}$ $\mathbf{Sum_0.consts := Equation.consts, Sum_1.consts := Equation.consts}, TEST(Equation.equal)\}$

$\mathbf{R(p_0') = \{Decl_0.consts := InsertConst(Decl_1.consts, ident, IntValue(int))\}}$

$\mathbf{R(p_0'') = \{Decl_0.consts := InsertConst(Decl_1.consts, ident, RealValue(real)\}}$

$\mathbf{R(p_0''') = \{Decl_0.consts := NewConstList()\}}$

$R(p_1) = \{ Sum.value := Product.value,$ $\mathbf{Product.consts := Sum.consts}\}$

$R(p_2) = \{ Sum_0.value := AddOp.sum , AddOp.lOp := Product.value, AddOp.rOp := Sum_1.value,$ $\mathbf{Product.consts := Sum_0.consts, Sum_1.consts := Sum_0.consts}\}$

$R(p_3) = \{ Product.value := RIFactor.value,$ $\mathbf{RIFactor.consts := Product.consts}\}$

$R(p_4) = \{ Product_0.value := MulOp.prod, MulOp.lOp := RIFactor.value,$ $MulOp.rOp := Product_1.value,$ $\mathbf{RIFactor.consts := Product_0.consts},$ $\mathbf{Product_1.consts := Product_0.sonsts}\}$

$R(p_5)$ = { ***RIFactor***.*value* := *IntValue(int)* }

$\mathbf{R(p_5')}$ **= {*RIFactor.value*:= *RealValue(real)*}**

$\mathbf{R(p_5'')}$ **= {*RIFactor.value*:= *ConstValue(RIFactor.consts, ident)*}**

$R(p_6)$ = { ***RIFactor***.*value*:= *Sum.value*, ***Sum.consts*** **:=** ***RIFactor.consts*** }

$R(p_7)$ = { *AddOp.sum* := *AddOp.lOp* + *AddOp.rOp* }

$R(p_8)$ = { *AddOp.sum* := *AddOp.lOp* - *AddOp.rOp* }

$R(p_9)$ = { *MulOp.prod* := *MulOp.lOp* * *AddOp.rOp* }

$R(p_{10})$ = { *MulOp.prod* := *MulOp.lOp* / *MulOp.rOp* }

Fig. 3..2: An attribute grammar extending AG_E

As in chapter 3.1, the extension relation E may be used to show that AG_E' represents an extension of AG_E.

4. Applications

In order to use EAGs in practice, only the actual enhancements of a grammar with respect to the base grammar need be defined. Therefore, only newly-added parts (alphabet, productions, attributes, attribution rules) require definition. Furthermore, it is convenient to define the extension relation explicitly, as this greatly simplifies the decision whether an extension is correct. For practical applications, it makes sense to restrict the set of valid extension relations. For instance, one might require that it is acyclic and that every non-terminal of the extended grammar to be associated with only one non-terminal of the base grammar (single inheritance).

The following sections describe some applications we have implemented using GIPSY (Generator for Integrated Programming Systems) [Mar91, MM92], whose specification language is based on EAGs. As a special feature, in GIPSY so-called *external* attributes may be associated with a non-terminal besides the *synthesized* and *inherited* attributes. These attributes are defined by external tools (editors) and they are read-only in the attribution rules. They serve as annotations of syntactical structures, e.g., comments or compiler hints, or as display-information for graphical editors, e.g., coordinates of syntactical objects.

4.1. A Program-Verification Tool

Using GIPSY, we developed a program verification tool as part of an Oberon [Ode89] programming environment. The tool uses the "weakest precondition transformation" method [Gri81]. In this method, pre- and postconditions are associated with the procedures as their specification. The pre- and postconditions are predicates on variables such as "x+y>0 AND x≠y". The precondition represents the predicate that needs to be true before calling the corresponding procedure, while the postcondition describes all possible variable and parameter values after execution of the procedure if the precondition was satisfied. A so-called predicate transformer is associated with every statement in the programming language (assignment, IF-statement, …). The transformer maps a predicate that needs to be true after execution of the respective statement to a "weakest precondition" of that statement. During verification of a procedure, the postcondition is transformed step by step into a weakest precondition, beginning with the last statement in a procedure, until the weakest precondition of the whole procedure has been calculated. Whether a procedure's implementation satisfies its specification is decided by the following implication:

$$\text{precondition(proc)} \Rightarrow \text{weakest precondition(proc)}$$

An attribute grammar AG_{Oberon} was already available when we begun to implement this tool. In order to get an nwe grammar AG_{WPT} we extended the grammar AG_{Oberon} by implementing the "weakest precondition transformation" with an attribution system. Hereby, the attributes *postcond* and *wprecond* have been associated with the non-terminals concerned (*ProcDecl, ProcBlock, StmtSequence, Statement, IFStatement, AssOrCall, WhileStatement*). A further external attribute *invariant* was associated with the*WhileStatement*. The attribution rules implement the respective production's predicate transformer. In addition, a simple interface needed to be defined for the predicate editor, which is used to enter the external attributes (*postcond* and *precond* of the *ProcDecl* and *invariant* of the *WhileStatement*).

With the extended grammar $AG_{WPT} \succ AG_{Oberon}$, we have generated an Oberon environment, which additionally contains a simple program verification tool.

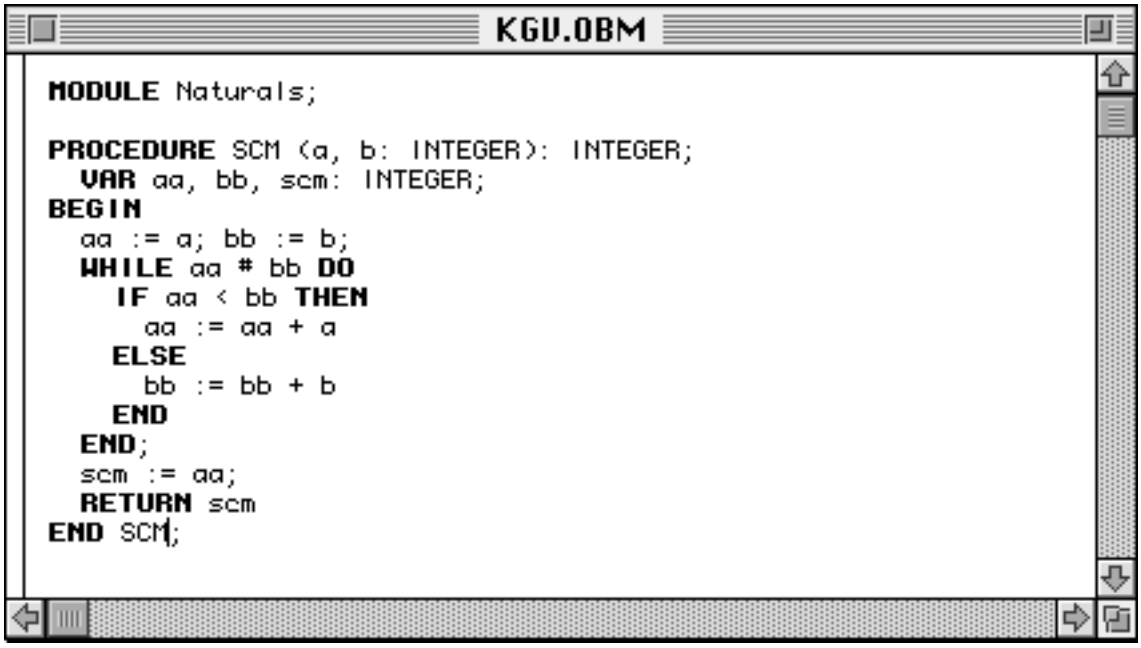

Fig. 4.1: Interface of the Oberon-Editor generated by AG_{Oberon}

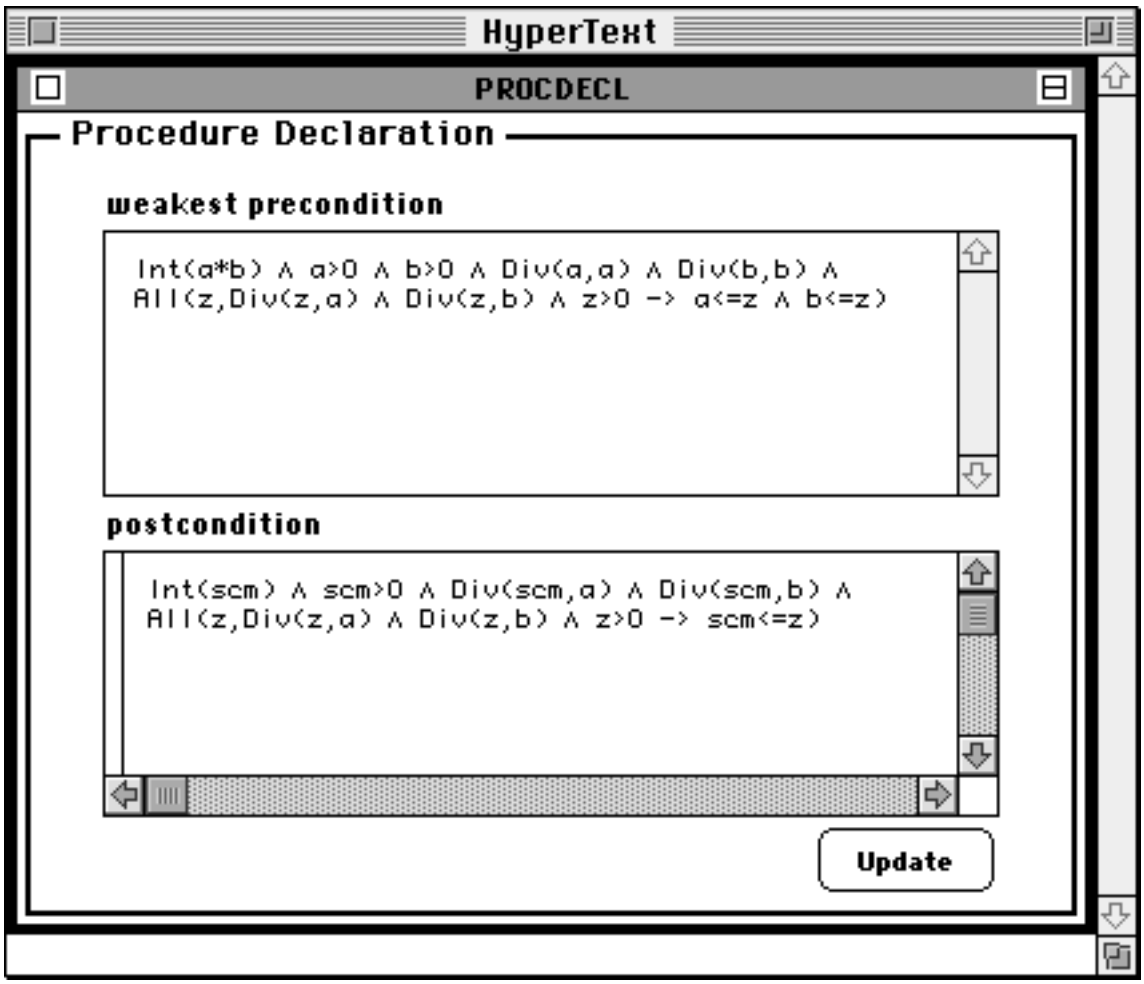

Fig. 4.1: Interface of the program verification tool generated by AG_{WPT}

4.2. Environment-Construction Tool-Kit

A further elegant application of EAGs is the creation of a compiler tool-kit or an environment construction tool-kit. Such a tool-kit makes language-independent components of a compiler or of a programming environment available as a library for implementing a specific environment. This idea is also found in object-oriented programming where such tool-kits are used to construct user interfaces or whole applications. The concept of extensibility plays an essential role in these tool-kits. Starting with a root class new classes are created by extending the functionality of one or several basic classes. The same principle may be used for environment construction tool-kits. Starting with a root grammar defining the basic properties of a class of programming languages, e.g., block-structured languages, new extended grammars may be defined. These extended grammars describe a specific language or they define an interface for a tool (graphical editor, module management tool, debugger) for an abstract or concrete language.

5. Conclusions

Good abstraction and structurization mechanisms are becoming increasingly important for environment specification languages as specification's complexity increases constantly. While a few years ago it was sufficient to generate a compiler from a language definition, today's goal is to automatically create entire programming environments. In order to specify such a programming environment, the language to be supported as well as the interfaces between tools, such as editors, debuggers, browsers, etc., and the kernel of the generated environment must be defined. With the *extensible attribute grammars* introduced in these paper, a model has been presented that may be used as an important structuring technique for environment specification. Starting with a base grammar, a new attribute grammar may be defined by extending the syntax and its static semantics. In contrast to modular grammar specification, extensible attribute grammars allow to keep together the parts of a specification concerning the whole grammar, e.g., declaration or type checking rules. In a modular specification these parts would be spread out over several modules resulting in a very fragmented specification.

The applications shown in chapter four demonstrate how individual components of a programming environment may be treated and specified separately. As has been shown, the idea to develop environment construction tool-kits is based on the extension model of attribute grammars. Thus, in the future we are going to investigate the design of such a framework in more detail.

6. Acknowledgments

The authors wish to thank all members of the ADAM research group, which in long, fruitful discussions helped to develop the ideas presented in this paper, especially Daniel Scherer and Philipp Färber for their valuable assistance in translating the paper in readable English and Prof. Albert Kündig for providing the research facilities.

The GIPSY-project reported here has been supported by grant no. 20-29093.90 of the “Schweizerischer Nationalfonds”.

7. References

[And89] Andrews, D.J.: "A Formal Definition of Modula-2", intermediate draft. document D91 of the ISO working-group TC97/SC22/WG13, 1989.

[BS86] Bahlke, R., Snelting, G.: "The PSG-Programming System Generator". Proceedings of the ACM-SIGPLAN'85 Symposium on Language Issues in Programming Environments, Seattle, June 1985. pp. 28-33.

[DJL88] Deransart, P., Jourdan, M., Lorho, B.: "Attribute Grammars; Definitions, Systems and Bibliography". Springer Verlag Berlin, Heidelberg. LNCS vol. 323. 1988.

[DW83] Davis, M.D., Weyuker, E.: "Computability, Complexity and Languages; Fundamentals of Theoretical Computer Science", Computer Science and Applied Mathematics, Academic Press, 1983.

[Filé86] Filé, G.: "Classical and Incremental Evaluators for Attribute Grammars". in Proceedings of the 11th Coll. on Trees in Algebra and Programming, Nice (France), March 1986. Springer Verlag, LNCS vol. 214, pp. 112-126.

[Gri81] Gries, D.: "The Science of Programming", Texts and Monographs in Computer Science, Springer Verlag, 1981.

[HMRC88] Holt, R.C., Matthews, P.A., Rosselet, A., Cordy, J.R.: "The Turing Programming Language - Design and Definition", Prentice Hall, 1988.

[JPJDL90] Jourdan, M., Parigot, D., Julié, C., Durin, O., Le Bellec, C.: "Design, Implementation and Evaluation of the FNC-2 Attribute Grammar System", Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation, White Plains, NY, 1990, pp. 209-222.

[KHZ82] Kastens, U., Hutt, B., Zimmermann, E.: "GAG: A Practical Compiler Generator", Springer Verlag, LNCS vol. 141, Berlin, 1982.

[Kli90] Klint, P.: "A meta-environment for generating programming environments". Centre for Mathematics and Computer Science, Amsterdam, Report CS-R9064, November 1990.

[Knu68] Knuth, D.E.: "Semantics of Context-Free Languages". Math. Systems Theory 2, 2, pp. 127-145, June 1968.

[Mar91] Marti, R.: "Design and Implementation of a Token-Editor". Proceedings of TOOLS'91, Technology of Object Oriented Languages and Systems, Paris, April 1991. pp. 349-359.

[Mey88] Meyer, B.: "Object-oriented Software Construction". Prentice Hall International, Series in Computer Science, 1988.

[MM92] Marti, R., Murer, T.: "GIPSY: A Generator for Incremental Programming Systems", TIK-Report, Computer Engineering Laboratory, ETH Zurich, October 1992.

[Mös90] Mössenböck, H.: "Coco/R: A Generator for Fast Compiler Front-Ends", Departement Informatik, Institut für Computersysteme, ETH Zürich, Februar 1990.

[Ode89] Odersky, M.: "A New Approach to Formal Language Definition and its Application to Oberon". Informatik-Dissertation Nr. 18, ETH Zürich. vdf. 1989.

[Reps83] Reps, Th.: "Generating Language-Based Environments", The MIT Press, Cambridge, Massachusetts, 1983.

[RT89] Reps, Th., Teitelbaum, T.: "The Synthesizer-Generator; A System for Constructing Language-Based Editors". Springer Verlag, New York, 1988.

[Sne86] Snelting, G.: "Inkrementelle Semantische Analyse in unvollständigen Programmfragmenten mit Kontextrelationen", Darmstädter Dissertation D 17, Fachbereich Informatik der Tech. Hochschule Darmstadt, 1986.

[UDP82] Uhl, J., Drossopoulou, S., Persch, G., Goos, G., Dausmann, M., Winterstein, G., Kirchgässner, W.: "An Attribute Grammar for the Semantic Analysis of Ada", Springer Verlag, LNCS vol. 139, 1982.

[Uhl86] Uhl, J.: "Spezifikation von Programmiersprachen und Übersetzern". GMD-Bericht Nr. 161. R Oldenbourg Verlag, 1986.

[VSK89] Vogt, H.H., Swierstra, S.D., Kuiper, M.F.: "Higher Order Attribute Grammars", in Proceedings of the ACM-SIGPLAN'89 Conference on Programming Language Design and Implementation, Portland Oregon, 1989. SIGPLAN NOTICES vol. 24. July 1989.

[WG84] Waite, W.M., Goos, G.: "Compiler Construction", Springer Verlag, 1984.

TIK-Reports

Nr. 90-1 Perrouchod, F., Lanz, C., Plattner, B.: "A Relational Database Design for an X.500 Directory System Agent", July 1990.

Nr. 90-2 Lubich, H.P.: "Model and Functionality Definition for the Collaborative Editing and Conferencing System MultimETH", July 1990.

Nr. 90-3 Müller, M.: "X.400 Security Capabilities: Evaluation and Constructive Criticism", August 1990.

Nr. 92-4 Schibli, P., Tadjan, M. "CPU Evaluation für ADAM", Januar 1992

Nr. 93-5 Lubich, H.P.: "Aspekte computergestützter Kooperation", Januar 1993