Higher order attribute grammars

H.H. Vogt, S.D. Swierstra and M.F. Kuiper

RUU-CS-89-4
March 1989

Rijksuniversiteit Utrecht

Vakgroep informatica

Padualaan 14 3584 CH Utrecht
Corr. adres: Postbus 80.089, 3508 TB Utrecht
Telefoon 030-531454
The Netherlands

Higher order attribute grammars

H.H. Vogt, S.D. Swierstra and M.F. Kuiper

RUU-CS-89-4
March 1989

Rijksuniversiteit Utrecht

Vakgroep informatica

Padualaan 14 3584 CH Utrecht
Corr. adres: Postbus 80.089, 3508 TB Utrecht
Telefoon 030-531454
The Netherlands

Higher order attribute grammars

H.H. Vogt, S.D. Swierstra and M.F. Kuiper

Technical Report RUU-CS-89-4
March 1989

Department of Computer Science
University of Utrecht
P.O.Box 80.089, 3508 TB Utrecht
The Netherlands

Higher Order Attribute Grammars

H.H. Vogt, S.D. Swierstra and M.F. Kuiper

Department of Computer Science, University of Utrecht
P.O.Box 80.089, 3508 TB Utrecht, The Netherlands

Abstract

A new kind of attribute grammars, called higher order attribute grammars, is defined. In higher order attribute grammars the structure tree can be expanded as a result of attribute computation. A structure tree may be stored in an attribute. The term higher order is used because of the analogy with higher order functions, where a function can be the result or parameter of another function. A relatively simple method, using OAGs, is described to derive an evaluation order on the defining attribute occurrences which comprises all possible direct and indirect attribute dependencies. As in OAGs, visit-sequences are computed from which an efficient algorithm for attribute evaluation can be derived.

1 Introduction

For quite some time now attribute grammars (AGs) are used in the field of compiler construction. The GAG-system, described in [Kastens, Hutt and Zimmerman 81], and the Synthesizer Generator, described in [Reps 1982] and [Reps, Teitelbaum and Demers 1983], are typical examples. The term *compilation* is mostly used to denote the conversion of an algorithm expressed in a human-oriented *source language* to an equivalent algorithm expressed in a hardware-oriented *target language*. A compilation is usually implemented as a sequence of transformations (SL, L_1), $(L_1, L_2), \ldots, (L_k, TL)$, where SL is the source language, TL the target language and all L_i are called intermediate languages. In attribute grammars SL is parsed, a structure tree corresponding with SL is build and finally attribute evaluation takes place, the TL is obtained as the value of an attribute. So an attribute grammar implements the direct transformation (SL, TL) and no special intermediate languages are used. The concept of an intermediate language does not occur naturally in the attribute grammar formalism. Using attributes to emulate intermediate languages is difficult to do and hard to understand. Higher order attribute grammars (HAGs) provide an elegant and powerful solution for this weakness, as attribute values can be used to define the expansion of the structure tree during attribute evaluation.

This formalism is completely new and distinguishes our work from normal attribute grammars and attribute coupled grammars (see [Ganzinger and Giegerich 84]).

To derive an efficient algorithm for attribute evaluation of AGs, so called OAGs were defined by [Kastens 80]. We use the same framework to derive an efficient attribute evaluation algorithm for HAGs.

In section 2 an informal definition and some examples of higher order attribute grammars (HAGs) are given. Section 3 contains the definition of higher order attribute grammars. The reduction of a HAG to an AG and a mapping of an evaluation order of the reduced AG to an evaluation order of the HAG together with correctness proofs are given in section 4. Also visit-sequences for a HAG derived by those of the corresponding reduced AG are given. Section 5 contains the conclusion and final remarks.

This is a copy of the manuscript presented at the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation.

2 Informal definition and examples

2.1 Attribute grammars

Complete definition of a programming language (syntactically as well as semantically) can be done using attribute grammars (AGs). The context free grammar $G = (T, N, P, Z)$ of a language corresponds to structure trees. Figure 1 shows an example context free grammar for simple expressions.

In attribute grammars attributes are "associated" with the non-terminals and terminals of the underlying context free grammar. For example, the terminal *ident* in Figure 1 may have an attribute *symbol*, being a unique reference to the denoted identifier. Attribute values are computed by semantic functions in attribute rules. An attribute rule is always

$G = (T = \{ ident, int, real, *, + \}$
$N = \{ Z, E, F, OP \}$
Z
$P = \{ Z \rightarrow E$
$E \rightarrow F\ OP\ E$
$F \rightarrow ident$
$F \rightarrow int$
$F \rightarrow real$
$E \rightarrow F$
$OP \rightarrow *, + \})$

Figure 1: A context free grammar

"associated" with a production of the grammar. We distinguish three kinds of attributes: inherited, synthesized and local attributes. Local attributes are not associated with a non-terminal or terminal, but with productions and can be only used by attribute rules of that production.

2.2 An example multi-pass compiler

As an example used throughout this section we consider a multi-pass compiler for simple expressions in an Algol-68 like language where priorities of the *operators* will be defined within the program and not by the language. The *operands* of the simple expressions consist of int and real constants and **REF int** and **REF real** variables. We assume that two operators + and * are defined for integer- as well as real-arithmetic. Because the priorities of the operators are defined within the program, the conventional context free parsing based on fixed operator priorities cannot be based on them. There are two coercions (type conversions), dereferencing (**REF int** → **int** and **REF real** → **real**) and widening (**int** → **real**). The compilation of an expression now consists of the following four steps:

1) Parsing the operator declarations and the expression

2) Converting expressions based on the declared operator precedence

3) Type checking and insertion of coercions

4) Code generation

In the rest of this section we will consider each step, except parsing the operator declarations, in detail. The type checking and coercion algorithm was strongly influenced by the one used in [Waite and Goos 84].

2.3 An example attribute grammar

As an example (Figure 2) we consider the grammar describing the type checking and coercion computation of expressions. Each production of the grammar is marked by the keyword **rule** and written using EBNF notation (restricted to express only productions). The attribute rules which belong to a production follow the keyword **attribution**. We use a conventional expression-oriented programming language notation for the semantic functions. Particular instances of an attribute are distinguished by numbering multiple occurrences of symbols in the production (e.g. expression[1], expression[2]) from left to right. Although conditions are part of the example grammar we will not mention them any further.

In order to check the consistency of the assignment and to further identify the + and * operator (step 3 of the expression compiler), operand types must be taken into account. For this purpose serve two attributes, *primode* (a priori type) and *postmode* (a posteriori type), for the symbols EXPRESSION and NAME, and one attribute, *mode* for the symbol OPERATOR. *Primode* describes the type determined directly from the node and its descendants; *postmode* describes the type expected when the result is used as an operand by other nodes. Any difference between between *primode* and *postmode* must be resolved by coercions. The boolean function *coercible*(t_1, t_2) tests whether type t_1 can be coerced to t_2. The direct dependencies between attributes (as shown in Figure 3) are defined by the attribute rules in the productions. A dependency graph over the attribute instances on a structure tree for a sentence can be constructed by "pasting together" the direct dependencies according to the applications of productions in the derivation of the sentence. An essential property of a well-formed AG is that this does not give rise to circularities.

Figure 4 shows the analysis of $x := y * 2 + 3.5$ according to the grammar of Figure 2. (*assignment.env* would normally be computed from the declarations of x and y).

The following attributes are used:

env	the set of pairs (symbol, mode) visible from the syntactic unit in question.
primode	the mode (**int**, **real**, **REF int** or **REF real**) of a syntactic unit before applying coercion.
postmode	the mode of a syntactic unit, determined by the outer context (coercion will yield this mode).
operation	determines integer or real arithmetic of an operator.
symbol	a unique representation of an identifier denotation.

The attribute sets are:

Inherited attributes:

$\{ X.env \mid X \in \{name, expression\} \}$
$\cup \{ X.postmode \mid X \in \{name, expression\} \}$
$\cup \{ operator.mode \}$

Synthesized attributes:

$\{ X.primode \mid X \in \{name, expression\} \}$
$\cup \{ operator.operation \}$
$\cup \{ identifier.symbol \}$

The assignment is defined by:

rule *assignment ::= name ':=' expression*
attribution
name.env := assignment.env ;
expression.env := assignment.env ;
name.postmode := name.primode ;
expression.postmode := deref(name.primode) ;
condition *must_be_ref(name.primode)*

rule *expression ::= expression operator expression*
attribution
local *prim ;*
prim :=
if *coercible(expression[1].primode, int)* **and**
coercible(expression[2].primode, int)
then *int* **else** *real* **fi** ;
expression[0].primode := prim
operator.mode := prim ;
expression[1].postmode := prim ;
expression[2].postmode := prim ;
expression[1].env := expression[0].env ;
expression[2].env := expression[0].env ;
condition *coercible(prim, expression[0].postmode)*

rule *operator ::= '+'*
attribution
operator.operation :=
if *operator.mode = int*
then *int_addition* **else** *real_addition* **fi** ;

rule *operator ::= '*'*
attribution
operator.operation :=
if *operator.mode = int*
then *int_multiplication*
else *real_multiplication* **fi** ;

rule *name ::= identifier*
attribution
name.primode :=
defined_type(identifier.symbol, name.env) ;

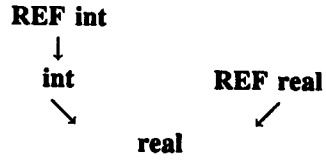
rule *expression ::= identifier*
attribution
expression.primode :=
defined_type(identifier.symbol, expression.env) ;
condition *coercible(expression.primode,*
expression.postmode)

rule *expression ::= intconstant*
attribution
expression.primode := int ;
condition *coercible(expression.primode,*
expression.postmode)

rule *expression ::= realconstant*
attribution
expression.primode := real ;
condition *coercible(expression.primode,*
expression.postmode)

The meaning of the functions deref, must_be_ref, coercible *and* defined_type *are given informally:*

deref(t)	t is a mode; returns the dereferenced mode of t.
must_be_ref(t)	t is a mode; returns `true` if t is a **REF int** or **REF real** variable.
coercible(t_1, t_2)	t_1 and t_2 are modes; the result is `true` if t_1 is coercible to t_2. t_1 is coercible to t_2 if there exists a path from t_1 to t_2 in the directed graph below.



defined_type(s, e)	s is a symbol, e is an environment; returns the mode of s if $s \in e$, otherwise `undefined`.

Figure 2: An attribute grammar

2.4 Higher order attribute grammars

There are two non-interchangeable concepts in a conventional AG: the structure tree and the attributes. It is not possible to define part of the structure tree by means of an attribute value and vice-versa. In a higher order attribute grammar (HAG), however, we allow both.

2.4.1 Part of the structure tree := attribute value

As an example we consider the first three steps of our example compiler: parsing, restructuring based on the previously declared operator precedence and type checking and

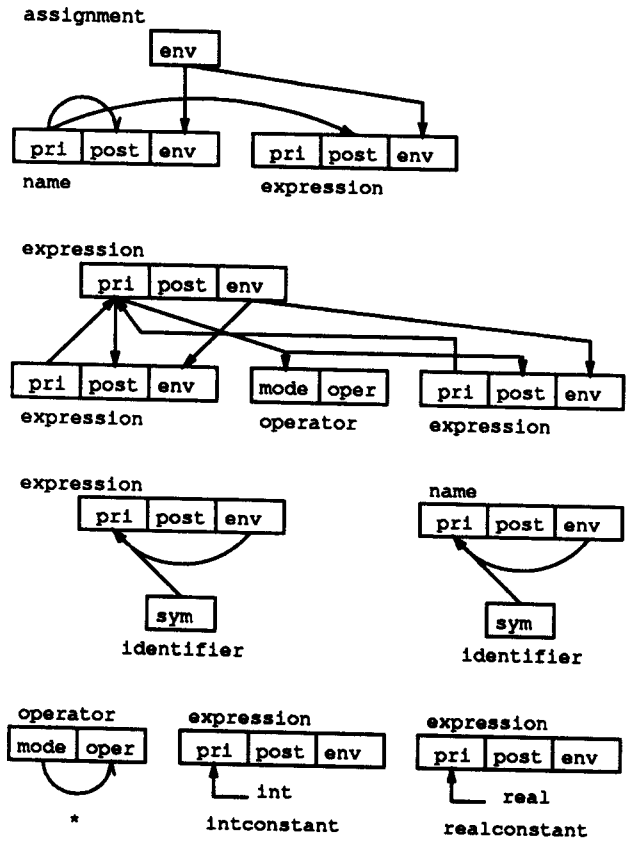


Figure 3: The direct dependency graphs

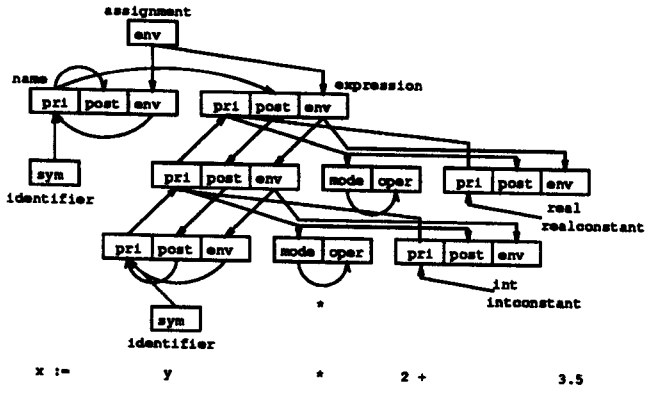


Figure 4: Dependencies in an attributed structure tree

coercion. Only after processing operator declarations the priorities of the operators are known and therefore it is not possible to define a parser based on the context free grammar reflecting the declared operator precedence. Figure 5 shows an expression and three sample trees which are syntactically correct according to the grammar of Figure 2. If we know that * has greater priority than + then only Figure 5.c reflects the tree with the correct operator precedence (we assume left to right expression evaluation). To be able to use the type checking and coercion defined by the attribute grammar of Figure 2 we need a semantically correct structure tree.

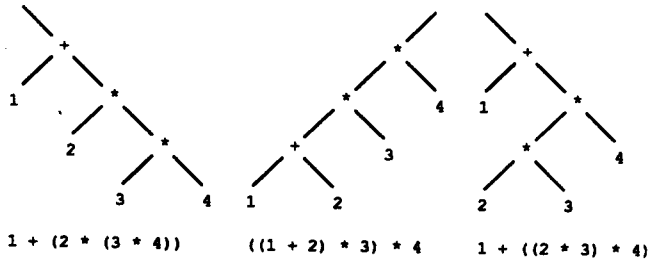


Figure 5: Three trees representing the expression 1 + 2 * 3 * 4, based on the ambiguous grammar of Figure 2

The first step, parsing the expression, is done according to the disambiguous context free grammar of Figure 1. Furthermore, as a result of attribute evaluation a structure tree reflecting the declared precedences of the operators is computed in the attribute $R_1.op\_tree$ (see Figure 6).

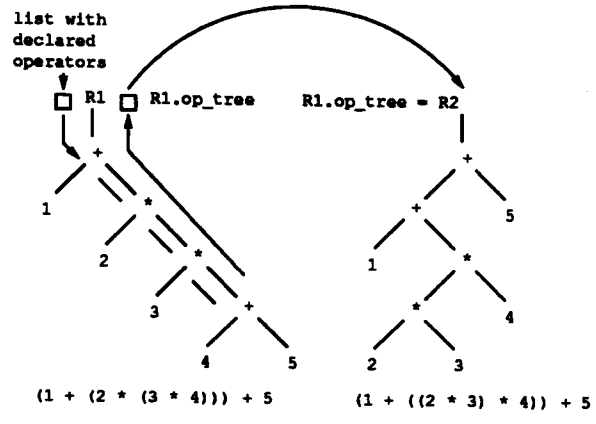


Figure 6: Computation of the operator precedence tree

We now extend the conventional AG-formalism with a construction which enables us to use the structure tree com-

puted as an attribute value in $R_1.op\_tree$ as the structure tree for the type checking and coercion attribute grammar of Figure 2. This is done by combining the grammars of Figure 1 and a slightly changed version of Figure 2 and adding a new production. The resulting grammar is shown in Figure 7. The computation of the tree reflecting the declared precedence of operators will be discussed later on.

$R \rightarrow R_1\ \overline{R_2}$ $\quad \overline{R_2} := R_1.op\_tree$

$R_1 \rightarrow E$ $\quad R_1.op\_tree :=$ *the correct operator precedence tree*

$E \rightarrow F\ OP\ E$
$F \rightarrow ident$
$F \rightarrow int$
$F \rightarrow real$
$E \rightarrow F$
$OP \rightarrow *, +$

$R_2 \rightarrow A$ $\quad A.env := R_2.env$; $A.post := R_2.pri$

$A \rightarrow A + A$ $\quad A[0].pri :=$ **if** *coercible(A[1].pri, int)* **and** *coercible(A[2].pri, int)* **then** *int* **else** *real* **fi**
$A[1].post := A[0].pri$
$A[2].post := A[0].pri$
$A[0].oper :=$ **if** *A[0].pri=int* **then** *int_add* **else** *real_add* **fi**
$A[1].env := A[0].env$
$A[2].env := A[0].env$
condition *coercible(A[0].pri, A[0].post)*

$A \rightarrow A * A$ $\quad$ *see above*

$A \rightarrow ident$ $\quad A.pri :=$ *defined_type(ident.symbol, A.env)*
condition *coercible(A.pri, A.post)*

$A \rightarrow int$ $\quad A.pri := int$
condition *coercible(A.pri, A.post)*

$A \rightarrow real$ $\quad A.pri := real$
condition *coercible(A.pri, A.post)*

Figure 7: Parsing the expression, computation of the operator precedence tree and type checking and coercion in one grammar

---

The overline in $R \rightarrow R_1\ R_2$ indicates that non-terminal $\overline{R_2}$ in the production is a *non-terminal attribute (NTA)*. Non-terminal attributes play a dual rôle. During the parsing of a sentence a non-terminal attribute X is considered as a non-terminal for which only the empty production ($X \rightarrow \perp$) exists. During attribute evaluation NTA X receives a value representing a non-attributed tree derivable from X. Next, the original parse tree is expanded with the non-attributed tree computed in NTA X, the non-attributed tree is attributed and attribute evaluation continues. Furthermore, a non-local attribute is a local attribute, being defined by a local attribution rule.

In the structure tree two kinds of non-terminal instances are distinguished, *virtual* non-terminals (NTAs which are not yet computed) and *instantiated* non-terminals (normal non-terminals and already computed NTAs). Leafs of a structure tree now can be terminals or virtual non-terminals.

A virtual non-terminal instance in an attributed structure tree will be indicated as a leaf of the form ○, an instantiated non-terminal as a leaf of the form ● (see Figure 8). After computation of the non-terminal attribute $\overline{R_2}$ in Figure 8 leaf ○ is replaced by a structure tree. So after expansion of the tree attribute evaluation continues and type checking and coercion takes place.

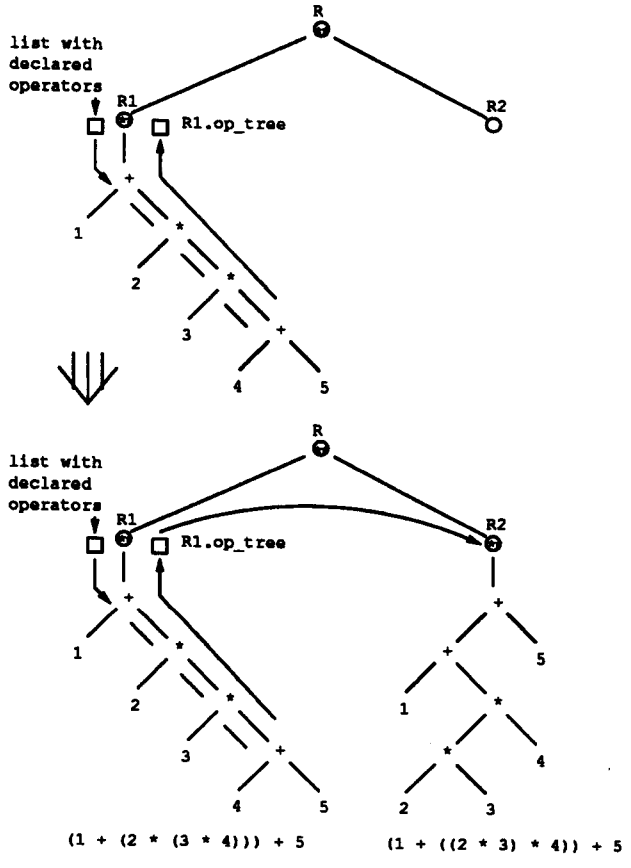


Figure 8: Expansion of the structure tree

In a normal AG the parser builds the complete structure tree, in a HAG the parser builds the structure tree of the grammar, where all non-terminal attributes are considered to

have empty productions. Furthermore, the attributed structure tree can be expanded as a result of attribute evaluation. After expansion the new part of the tree becomes attributed as well.

2.4.2 Attribute value := part of the structure tree

Besides expanding the structure using an attribute value, we introduce the possibility to store part of the structure tree in an attribute.

It is sufficient if R_1 contains a synthesized attribute describing the structure tree under R_1. The above described construction won't be used any further.

2.5 A linear notation for structure trees

In the previous section part of a compiler was described. We used an attribute *op_tree* in which the structure tree of the semantically correct expression was computed. Because the functions in attribution rules are defined by means of a linear notation we need a linear notation for the construction of structure trees.

We define a signature Σ_G associated with context free grammar G and a term-language generated by Σ_G. Terms in the term-language correspond to parse-trees for the grammar. Those familiar with signatures and algebras may skip the following two definitions and note the following: If Σ_G is the signature associated with grammar G then the objects of the Σ_G-term algebra, T_{Σ_G}, correspond to parse-trees for the grammar.

See Figure 9 for an example grammar G, the associated signature Σ_G, a parse tree and the corresponding term.

Definition 2.1 *A* signature $\Sigma = (S, OP)$ *consists of*

S : *the set of* sorts
OP : *the set of* operation symbols

With each operation symbol is associated an $(n+1)$ *tuple of sorts from* S, $n \geq 0$, *called its* arity. *If* F *is an operation symbol and* $(\sigma_1, \sigma_2, \ldots, \sigma_n, \sigma_{n+1})$ *its arity one writes*

$$F : \sigma_1 \times \sigma_2 \times \ldots \times \sigma_n \rightarrow \sigma_{n+1}$$

Definition 2.2 *Let* $\Sigma = (S, OP)$ *be a signature. A* Σ-term *and its* sort *are defined inductively by*

- *if* F *is an operation symbol with arity* $\rightarrow \sigma$ *then*

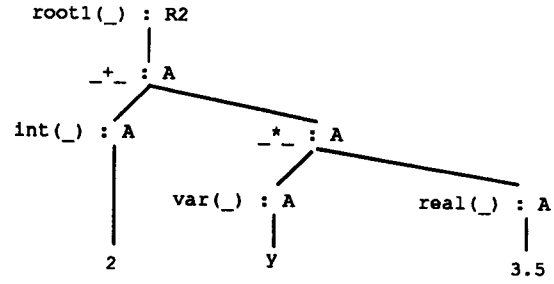
  F

  is a Σ*-term of sort* σ.

- *if* F *is an operation symbol with arity* $\sigma_1 \times \sigma_2 \times \ldots \times \sigma_n \rightarrow \sigma_{n+1}$, $n \geq 1$, *and if* $t_1, t_2, \ldots, t_n$ *are* Σ*-terms of sort* $\sigma_1, \sigma_2, \ldots, \sigma_n$ *respectively then*

  $F(t_1, t_2, \ldots, t_n)$

Grammar G:		*Associated signature* Σ_G*:*	
$root_1(\_)$:	$R_2 \rightarrow A$	$root_1(\_)$:	$A \rightarrow R_2$
$\_+\_$:	$A \rightarrow A\ A$	$\_+\_$:	$A \times A \rightarrow A$
$\_*\_$:	$A \rightarrow A\ A$	$\_*\_$:	$A \times A \rightarrow A$
$var(\_)$:	$A \rightarrow ident$	$var(\_)$:	$ident \rightarrow A$
$int(\_)$:	$A \rightarrow int$	$int(\_)$:	$int \rightarrow A$
$real(\_)$:	$A \rightarrow real$	$real(\_)$:	$real \rightarrow A$




```
root1(int(2) + (var(y) * real(3.5)))
```

Figure 9: The grammar G, the signature Σ_G and a structure tree together with the corresponding term

is a Σ*-term of sort* σ.

We abbreviate Σ-term as *term*, if Σ is evident by the context. The set of all terms of sort σ is called the *term language of sort* σ. The family constituted by the term languages of sort σ, one for each $\sigma \in S$, is called the *term language (generated by* Σ*).*

Definition 2.3 *Let G=(T, N, Z, P) be a context free grammar. The* associated signature Σ_G*=(S, OP) is defined as follows:*

For each production $p \in P$, $p = N_0 \rightarrow T_0 N_1 T_1 \ldots N_n T_n$ *with* $T_i \in T$, $N_i \in N$ *for* $0 \leq i \leq n$, *let the arity of operator symbol* $p \in OP$ *be* $(N_1, \ldots, N_n, N_0)$ *or alternatively written*

$$p : N_1 \times N_2 \times \ldots \times N_n \rightarrow N_0$$

Now a term in the term language generated by the signature Σ_G associated with grammar G describes a parse tree for the grammar G. This provides in a powerful and elegant linear notation for structure trees.

2.6 The complete compiler example

2.6.1 Step 1 and 2, parsing and computation of the operator precedence tree

As we have seen compiling an expression consists of four steps:

1) Parsing the operator declarations and the expression

2) Converting expressions based on the declared operator precedence

3) Type checking and insertion of coercions

4) Code generation

Step 1 of this list is done by the context free parser. Figure 10 shows the higher order attribute grammar for step 1 and 2. The attribution rules defining the tree with the right operator precedence are printed in sans serif type style. In computing the intended operator precedences we use attributes representing an operator stack and an operand stack, by lists. A stack trace for the expression $1 * 2 + 3$ is:

optorstack	opndstack
[]	[]
[*]	[1]
[+]	[[*,1,2]]
[+]	[3,[*,1,2]]
[]	[+,[*,1,2],3]

2.6.2 Step 3, type-checking and coercion

Step 2 delivered a new structure tree reflecting the declared operator precedence, used by step 3, step 3 will deliver a new structure tree incorporating the coercions well suited for the code generation of step 4. Note that the first introductory example to attribute grammars (Figure 2), however, only checks whether an expression is correctly typed. Therefore we introduce new productions used by step 3 to construct a structure tree well suited for code generation. We introduce separate addition, multiply and dereference productions, like $\_+_i\_$, $\_+_r\_$, etc, for integers and reals. The resulting grammar defining steps 1,2 and 3 is shown in Figure 11.

2.6.3 Step 4, code generation

Step 3 generates a structure tree well suited for code generation. We will generate the usual code for a stack machine. Integers and reals may have different sized internal representations, this explains why we need two kinds of instructions for every basic operator. The instruction $push_{ia}(adr)$ pushes the integer located at address *adr* on the stack. The resulting higher order attribute grammar defining all steps of the example multi-pass compiler is shown in Figure 12.

Note that the computation of step i is based upon a structure tree of step i-1 and all the intermediate structures differ. We demonstrated here the use of a HAG as a powerful tool for constructing multi-pass compilers, a weakness of normal AGs.

Pr :	$R \rightarrow R_1\ \overline{R_2}$	$\overline{R_2}$:= R_1.*op_tree*
Ps :	$R_1 \rightarrow E$	*E.optor := [] ; E.opnd := []* *E.prioenv* := R_1.*prioenv* R_1.op_tree := E.op_tree
Pe :	$E \rightarrow F\ OP\ E$	*E[1].optor :=* π_1(*eval_top(E[0].optor,* *OP.op, [F.sym] ++ E[0].opnd,* *E[0].prioenv))* *E[1].opnd :=* π_2(*eval_top(E[0].optor,* *OP.op, [F.sym] ++ E[0].opnd,* *E[0].prioenv))* *E[1].prioenv := E[0].prioenv* E[0].op_tree := E[1].op_tree
Pid :	$F \rightarrow ident$	*F.sym := var(ident.id)*
Pint :	$F \rightarrow int$	*F.sym := int(int.val)*
Preal :	$F \rightarrow real$	*F.sym := real(real.val)*
Pend :	$E \rightarrow F$	E.op_tree := $root_1$(buildtree(π_2(eval_top(E.optor, $\perp_{op}$, [F.sym] ++ E[0].opnd, E[0].prioenv))))
Po :	$OP \rightarrow opname$	*OP.op := opname.op*

$root_1(\_)$:	$R_2 \rightarrow A$
$\_+\_$:	$A \rightarrow A\ A$
$\_*\_$:	$A \rightarrow A\ A$
var(_) :	$A \rightarrow ident$
int(_) :	$A \rightarrow int$
real(_) :	$A \rightarrow real$

The following functions are used:

prioenv(op)=
if *op*=$\perp_{op}$
then 0 **else** *the declared operator priority* (> 0) **fi**

$\pi_1(x,y)=x$, $\pi_2(x,y)=y$

eval_top([], op, opnd_stack, prioenv) =
([op], opnd_stack)

eval_top(top_op:rest_ops, op, o1:o2:rest, prioenv) =
let *p1=prioenv(top_op), p2=prioenv(op)*
in if *p1* ≥ *p2* → *(op:top_op:rest_ops, opnd_stack)*
□ *p1* < *p2* → *eval_top(rest_ops, op, [top_op,o1,o2]:rest, prioenv)*
ni

buildtree(singleton) = singleton,
buildtree([+,a,b]) = buildtree(a) + buildtree(b),
buildtree([,a,b]) = buildtree(a) * buildtree(b)*

Figure 10: Steps 1 and 2 of the multi-pass compiler

Pr :	$R \rightarrow R_1\ \overline{R_2}\ \overline{R_3}$	$\overline{R_2}$:= R_1.*op_tree* $\overline{R_3}$:= R_2.*coerce_tree*

See Figure 10 for the productions and rules defining R_1 *and* R_1.*op_tree.*

$root_2(\_)$:	$R_2 \rightarrow A$	*A.env* := R_2.*env* *A.post* := R_2.*pri* R_2.coerce_tree := A.coerce_tree
$\_+\_$:	$A \rightarrow A\ A$	*A[0].pri* := **if** *coercible(A[1].pri, int)* **and** *coercible(A[2].pri, int)* **then** *int* **else** *real* **fi** *A[1].post* := *A[0].pri* *A[2].post* := *A[0].pri* *A[0].oper* := **if** *A[0].pri=int* **then** *int_add* **else** *real_add* **fi** *A[1].env* := *A[0].env* *A[2].env* := *A[0].env* A[0].coerce_tree := findop(A[1].coerce_tree, +, A[2].coerce_tree, A[0].oper, A[0].post) **condition** *coercible(A[0].pri, A[0].post)*
$\_*\_$:	$A \rightarrow A\ A$	*see above*
var(_) :	$A \rightarrow$ *ident*	*A.pri* := *defined_type(ident.symbol, A.env)* A.coerce_tree := findderef(ident.sym, A.pri, A.post) **condition** *coercible(A.pri, A.post)*
int(_) :	$A \rightarrow$ *int*	*A.pri* := *int* A.coerce_tree := **if** A.post = real **then** widen(int(int.val) **else** int(int.val) **condition** *coercible(A.pri, A.post)*
real(_) :	$A \rightarrow$ *real*	*A.pri* := *real* A.coerce_tree := real(real.val) **condition** *coercible(A.pri, A.post)*
$root_3(\_)$:	$R_3 \rightarrow C$	
$\_+_i\_$:	$C \rightarrow C\ C$	
$\_+_r\_$:	$C \rightarrow C\ C$	
$\_*_i\_$:	$C \rightarrow C\ C$	
$\_*_r\_$:	$C \rightarrow C\ C$	
$deref_i(\_)$:	$C \rightarrow$ *ident*	
$deref_r(\_)$:	$C \rightarrow$ *ident*	
widen(_) :	$C \rightarrow C$	
int(_) :	$C \rightarrow$ *int*	
real(_) :	$C \rightarrow$ *real*	

The following functions are used:

findop determines int or real operation eventually followed by widening, findderef determines int or real dereferencing eventually followed by widening.

Figure 11: Steps 1,2 and 3 of the multi-pass compiler

Pr :	$R \rightarrow R_1\ \overline{R_2}\ \overline{R_3}\ \overline{R_4}$	$\overline{R_2}$:= R_1.*op_tree* $\overline{R_3}$:= R_2.*coerce_tree* $\overline{R_4}$:= R_3.*code_tree*

See Figure 10 for the productions and rules defining R_1 *and* R_1.*op_tree.*

See Figure 11 for the productions and rules defining R_2 *and* R_2.*coerce_tree.*

$root_3(\_)$:	$R_3 \rightarrow C$	R_3.code_tree := $root_4$(C.code)
$\_+_i\_$:	$C \rightarrow C\ C$	C[0].code := C[1].code ; C[2].code ; add_i
$\_+_r\_$:	$C \rightarrow C\ C$	C[0].code := C[1].code ; C[2].code ; add_r
$\_*_i\_$:	$C \rightarrow C\ C$	C[0].code := C[1].code ; C[2].code ; mul_i
$\_*_r\_$:	$C \rightarrow C\ C$	C[0].code := C[1].code ; C[2].code ; mul_r
$deref_i(\_)$:	$C \rightarrow$ *ident*	C.code := $push_{ia}$(getadr(ident.sym))
$deref_r(\_)$:	$C \rightarrow$ *ident*	C.code := $push_{ra}$(getadr(ident.sym))
widen(_) :	$C \rightarrow C$	C[0].code := C[1].code ; wide
int(_) :	$C \rightarrow$ *int*	C.code := $push_i$(int.val)
real(_) :	$C \rightarrow$ *real*	C.code := $push_r$(int.val)
$root_4(\_)$:	$R_4 \rightarrow I$	
$\_;\_$:	$I \rightarrow I\ I$	
$push_i(\_)$:	$I \rightarrow$ *int*	
$push_r(\_)$:	$I \rightarrow$ *real*	
$push_{ia}(\_)$:	$I \rightarrow$ *address*	
$push_{ra}(\_)$:	$I \rightarrow$ *address*	
mul_i :	$I \rightarrow *$	
mul_r :	$I \rightarrow *$	
add_i :	$I \rightarrow +$	
add_r :	$I \rightarrow +$	
wide :	$I \rightarrow w$	

The function getadr(ident) computes the address of the identifier.

Figure 12: Steps 1,2, 3 and 4 of the multi-pass compiler

3 Higher order AGs

In this section higher order attribute grammars (HAGs) are defined. In AGs there exists a strict boundary between attributes and the parse tree. HAGs remove this boundary. A new kind of attributes, so called non-terminal attributes (NTAs), will be defined. These are non-terminals of the grammar as well as attributes defined by a semantic function. During the parsing of a sentence a non-terminal attribute X is considered as a non-terminal for which only the empty production ($X \rightarrow \perp$) exists. During attribute evaluation NTA X receives a value denoting a non-attributed tree derivable from X. Next, the original parse tree is expanded with the non-attributed tree computed in NTA X, the non-attributed tree is attributed and attribute evaluation continues. A necessary condition for a HAG to be well-formed is that the dependency graph of every possible partial tree does not give rise to circularities.

First, a definition of normal attribute grammars (almost literally taken from [Waite and Goos 84]) including local attributes is given. Then higher order attribute grammars are defined.

3.1 Definition of attribute grammars

A context free grammar $G = (T, N, P, Z)$ consists of a set of terminal symbols T, a set of non-terminal symbols N, a set of productions P and a start symbol $Z \in N$.

To every node in a structure tree corresponds a production from G.

Definition 3.1 *An* attribute grammar *is a 3-tuple* $AG = (G, A, R)$. *$G = (T, N, P, Z)$ is a context free grammar.* $A = \bigcup_{X \in T \cup N} AIS(X) \cup \bigcup_{p \in P} AL(p)$ *is a finite set of attributes,* $R = \bigcup_{p \in P} R(p)$ *is a finite set of attribution rules.*
$AIS(X) \cap AIS(Y) \neq \emptyset$ implies $X = Y$. For each occurrence of non-terminal X in the structure tree corresponding to a sentence of $L(G)$, exactly one attribution rule is applicable for the computation of each attribute $a \in A$.

Elements of $R(p)$ have the form

$$\alpha := f(\ldots, \gamma, \ldots).$$

In this attribution rule, f is the name of a function, α and γ are attributes of the form $X.a$ or $p.b$. In the latter case $p.b \in AL(p)$. In the sequel we will use the notation b for $p.b$ whenever possible. We assume that the functions used in the attribution rules are strict in all arguments.

Definition 3.2 *For each $p : X_0 \rightarrow X_1 \ldots X_n \in P$ the set of* defining occurrences *of attributes is*

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

$$\cup \{p.b \mid p.b := f(\ldots) \in R(p)\}$$

An attribute $X.a$ is called synthesized *if there exists a production $p : X \rightarrow \chi$ and $X.a$ is in $AF(p)$; it is* inherited *if there exists a production $q : Y \rightarrow \mu X \nu$ and $X.a \in AF(q)$. An attribute b is called* local *if there exists a production p such that $p.b \in AF(p)$.*

AS(X) is the set of synthesized attributes of X. *AI(X)* is the set of inherited attributes of X. *AL(p)* is the set of local attributes of production p.

Definition 3.3 *An attribute grammar is* complete *if the following statements hold for all X in the vocabulary of G:*

- *For all $p : X \rightarrow \chi \in P, AS(X) \subseteq AF(p)$*
- *For all $q : Y \rightarrow \mu X \nu \in P, AI(X) \subseteq AF(q)$*
- *For all $p \in P, AL(P) \subseteq AF(p)$*
- $AS(X) \cup AI(X) = AIS(X)$
- $AS(X) \cap AI(X) = \emptyset$

Further, if Z is the root of the grammar then AI(Z) *is empty.*

Definition 3.4 *An attribute grammar is* well defined *(WAG) if, for each structure tree corresponding to a sentence of L(G), all attributes are effectively computable.*

Definition 3.5 *For each $p : X_0 \rightarrow X_1 \ldots X_n \in P$ the set of* strict attribute dependencies *is given by*

$$DDP(p) = \{(\beta, \alpha) \mid \alpha := f(\ldots \beta \ldots) \in R(p)\}$$

where α and β are of the form $X_i.a$ or b. The grammar is locally acyclic *if the graph of DDP(p) is acyclic for each $p \in P$.*

We often write $(\alpha, \beta) \in DDP(p)$ as $(\alpha \rightarrow \beta) \in DDP(p)$, and follow the same conventions for the relations defined below. If no misunderstanding can occur, we omit the specification of the relation. We obtain the complete dependency graph for a labeled structure tree by 'pasting together' the direct dependencies according to the syntactic structure of the tree.

Definition 3.6 *Let S be the attributed structure tree corresponding to a sentence in $L(G)$, and let $K_0 \ldots K_n$ be the nodes corresponding to an application of $p : X_0 \rightarrow X_1 \ldots X_n$ and γ, δ attributes of the form $K_i.a$ or b corresponding with the attributes α, β of the form $X_i.a$ or b. We write $(\gamma \rightarrow \delta)$ if $(\alpha \rightarrow \beta) \in DDP(p)$. The set $DT(S) = \{(\gamma \rightarrow \delta)\}$, where we consider all applications of productions in S, is called the* dependency relation over the tree *S.*

The following theorem gives another characterization of well-defined attribute grammars. A proof can be found in [Waite and Goos 84].

Theorem 3.1 *An attribute grammar is* well-defined *iff it is complete and the graph DT(S) is a-cyclic for each structure tree S corresponding to a sentence of L(G).*

3.2 Definition higher order AGs

We are now ready to formalize the intuition of the preceding sections.

3.2.1 Definitions

An higher order attribute grammar is an attribute grammar with the following extensions:

Definition 3.7 *For each $p : X_0 \rightarrow X_1 \ldots X_n \in P$ the set of* non-terminal attributes (NTAs) *is defined by*

$$NTA(p) = \{X_j \mid X_j := f(\ldots) \in R(p)\}$$

Because a non-terminal attribute is also an attribute, an actual tree may contain NTAs (not yet computed non-terminal attributes) as leafs. Therefore we change the notion of a tree. Two kinds of non-terminals are distinguished, virtual non-terminals (NTAs without a value) and instantiated non-terminals (NTAs with a value and normal non-terminals).

Definition 3.8 *A non-terminal instance X in a tree is called*

- *a* virtual *non-terminal if $X \in \bigcup_{p \in P} NTA(p)$ and the function defining X has not yet been evaluated*
- *an* instantiated *non-terminal if $X \notin \bigcup_{p \in P} NTA(p)$ or $X \in \bigcup_{p \in P} NTA(p)$ and the function defining X has been evaluated*

Definition 3.9 *A labeled tree is defined as follows*

- *the leafs of a labeled tree are labeled with terminal or virtual non-terminal symbols*
- *the nodes of a labeled tree are labeled with instantiated non-terminal symbols*

From now on, the terms 'structure tree' and 'labeled structure tree' are all used to refer to a labeled tree. In the text a non-terminal attribute X will be indicated as $\overline{X}$, in the labeled tree a leaf labeled with a virtual non-terminal will be displayed as $\circ$, an instantiated non-terminal node as $\bullet$.

Definition 3.10 *A semantic function f in a rule $\overline{X} := f(\ldots)$ is* correctly typed *if f returns a term representing a parse tree derivable from X (see Definition 2.3 for a definition for term).*

This definition will be used to ensure that a NTA $\overline{X}$ will be expanded with a labeled tree which is derivable from X. Note that a check whether a function is correctly typed can be done statically.

Definition 3.11 *An higher order attribute grammar is* complete *if the underlying AG is complete and the following holds for all productions $p : Y \rightarrow \mu \in P$:*

- *$NTA(p) \subseteq AL(p)$*

and for all $\overline{X} \in NTA(p)$:

- *$\overline{X} \in \mu$*
- *For all rules $\alpha := f(\gamma)$ in $R(p)$, $\overline{X} \notin \gamma$*
- *For all rules $\overline{X} := f(\ldots)$ in $R(p)$, f is correctly typed*

The above definition defines NTAs as local attributes which only occur as a non-terminal at the right-hand-side of a production and as an attribute at the left-hand-side of a semantic function.

3.2.2 Attribute evaluation

Evaluation of attribute instances, expansion of the labeled tree and adding new attribute instances is called *attribute evaluation* and might be thought to proceed as follows:

To analyze a string according to its higher order attribute grammar specification, we first construct the labeled tree derived from the root of the higher order attribute grammar. Then evaluate as many attribute instances as possible. As soon as virtual non-terminal instance $\overline{X}$ is computed, expand the labeled tree derived from the root at the corresponding leaf $\overline{X}$ with the labeled tree in $\overline{X}$ and add the attribute instances resulting from the expansion. The virtual non-terminal $\overline{X}$ has now become an instantiated non-terminal $\overline{X}$. Continue the evaluation until there are no more attribute instances to evaluate and all possible expansions are done.

The order in which attributes are evaluated is not defined yet, but subjected to the constraint that each semantic function be evaluated only when all of its argument attributes are available. When all the arguments of an unavailable attribute instance have come available, we say it is *ready for evaluation.*

Using the definition of attribute evaluation and the observation to maintain a work-list S of all attribute instances that are ready for evaluation we get, as is stated in [Knuth 1968, Knuth 1971] and [Reps 1982], the following Attribute Evaluation Algorithm (Figure 13).

The difference with the algorithm defined by [Reps 1982] is that the labeled tree T can be expanded during semantic analysis. This means that if we evaluate a NTA $\overline{X}$, we have to expand the tree at the corresponding leaf $\overline{X}$ with the tree computed in $\overline{X}$. Furthermore, the new attribute instances and their dependencies of the expansion (the set $DT(X)$) have to be added to the already existing attribute instances and their dependencies and the work-list S must

be expanded by all the attribute instances in $DT(X)$ that are ready for evaluation.

If we look at the Attribute Evaluation Algorithm, there are two potential problems:

- non-termination
- attribute instances may not receive a value

---

evaluate(T)
let T = *an unevaluated labeled tree*
D = *a dependency relation on attribute instances*
S = *a set of attribute instances that are ready for evaluation*
α, β = *attribute instances*
in
$D := DT(T)$ { *the dependency relation over the tree* T }
$S :=$ *the attribute instances in* D *ready for evaluation*
while $S \neq \emptyset$ *do*
select and remove an attribute instance α *from* S
evaluate α
if α *is a NTA of the form* $\overline{X}$
then expand T *at* $\overline{X}$ *with the unevaluated tree in* α
$D := D \cup DT(X)$
$S := S \cup$ *the attribute instances in* $DT(X)$ *ready for evaluation*
fi
for each β *that is a successor of* α *in* D *do*
if β *is ready for evaluation*
then insert β *in* S
fi
od

Figure 13: Attribute Evaluation Algorithm

The algorithm might not terminate if the labeled tree T grows indefinitely, in which case there will always be virtual non-terminal attribute instances which can be instantiated (Figure 14).

There are two reasons why an attribute might not receive a value:

- there is a cycle in the dependency relation D: attribute instances involved in the cycle will be never ready for evaluation, so they will never be evaluated.
- there is a non-terminal attribute instance, say $\overline{X}$, which depends on a synthesized attribute of X.

The second reason may deserve some explanation. Suppose we have a tree T containing rule p and $\overline{X}$ is a non-terminal attribute instance in T. Furthermore the dependency relation D of all the attribute instances in T contains no cycles (Figure 15).

$R \rightarrow X$
$pA: \quad X \rightarrow \overline{A}$
$\overline{A} := pX$
$pX: \quad A \rightarrow \overline{X}$
$\overline{X} := pA$

Figure 14: This grammar will expand indefinitely

If we take a closer look at node $\overline{X}$ in T, then if $\overline{X}$ doesn't depend on synthesized attributes of $\overline{X}$ it can be computed. But should X depend on synthesized attributes of $\overline{X}$, as in Figure 15 it can't be computed. This is because the synthesized attributes of $\overline{X}$ are computed after the tree is expanded. So a non-terminal attribute shouldn't depend on its own synthesized attributes. To prevent this we let every synthesized attribute of X depend on $\overline{X}$. Therefore the set of extended direct attribute dependencies is defined.

Definition 3.12 *For each* $p : X_0 \rightarrow X_1 \ldots X_n \in P$ *the set of* extended direct attribute dependencies *is given by*

$$EDDP(p) = \{(\alpha \rightarrow \beta) \mid \beta := f(\ldots\alpha\ldots) \in R(p)\}$$
$$\cup \{(\overline{X} \rightarrow \gamma) \mid \overline{X} \in NTA(p) \text{ and } \gamma \in AS(X)\}$$

Thus a non-terminal attribute can be computed if and only if the dependency relation D (using the *EDDPs*) contains no cycles. This result is stated in the following lemma.

Lemma 3.1 *Every virtual non-terminal attribute will be computed if and only if there will be no cycles in D (using the EDDP) during attribute evaluation.*

Proof The use of *EDDP(p)* prohibits a non-terminal attribute β to be defined in terms of attribute instances in the tree which will be computed in β. Suppose β, which is of the form X, depends on attributes in the tree which is constructed in β. The only way to achieve this is that β somehow depends on the synthesized attributes of X, but by definition of *EDDP(p)* all the synthesized attributes of X depend on β and we have a cycle.

□

$R \rightarrow \overline{X}$
$\overline{X} := f(X.s)$
$X \rightarrow one$
$X.s := 1$
$X \rightarrow two$
$X.s := 2$

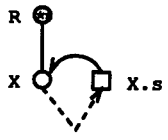


Figure 15: The non-terminal attribute can't be computed, a cycle occurs if the extra dependency is added (dashed arrow)

3.3 Well-definedness and bounded well-definedness

If D, the previously mentioned dependency relation over the attribute instances in the tree, contains never cycles during attribute evaluation all the (non-terminal) attributes can be computed.

Definition 3.13 *An higher order attribute grammar is* well-defined *if, for each labeled structure tree S corresponding to a sentence in L(G) where parts derived by non-terminal attributes are removed, all attributes are effectively computable and there are only finite expansions of the tree T and the relation D during attribute evaluation.*

It is clear that if D never contains a cycle during attribute evaluation, all the (non-terminal) attribute instances will be computed. It is generally undecidable whether a given *HAG* will have only finite expansions. For instance whether the following grammar (Figure 16) is well-defined is undecidable.

$R \rightarrow X$
pA: $X \rightarrow \overline{A}$
$\overline{A} :=$ **if** $f(\ldots)$ **then** pX **else** pa **fi**
pX: $A \rightarrow \overline{X}$
$\overline{X} :=$ pA
pa: $A \rightarrow a$

Figure 16: Finite expansion is not guaranteed

The tree can grow indefinitely or stop after some expansions, which depends on the behavior of the function f.

This behavior is generally not computable. So we need to be sure that the number of expansions of the tree T and the relation D is finite. A sufficient, but not necessary, condition is given in the following lemma.

Lemma 3.2 *If on every path in every structure tree a particular non-terminal attribute occurs at most once, there will be a finite number of expansions of the labeled tree T and the dependency relation D during attribute evaluation.*

Proof The Attribute Evaluation Algorithm is activated starting with a finite labeled tree. Every expansion costs one non-terminal attribute. Suppose the starting finite labeled tree meets the requirements of the above theorem and there are infinite expansions of the labeled tree T and relation D.

Then it is necessary for a branch in the tree to become infinite. So there will be infinitely many nodes in that branch, but there are only a finite number of non-terminal attributes. This leads to the observation that there must be a double non-terminal attribute instance in that branch. This is in contradiction with our lemma.

□

The above lemma looks harmless, but an effect of this lemma is the following which we will not prove here. If a HAG satisfies the above lemma, then the underlying context free grammar is a special coupled sum of disjoint context free grammars. Every non-terminal attribute turns out to be the root symbol of a "non-recursive" separate context free grammar.

Furthermore, if a non-terminal occurs twice on a path in the structure tree, then the context free grammar is circular and this is a decidable problem. It can be solved in time polynomially depending on the size of the input grammar.

Theorem 3.2 *A higher order attribute grammar HAG is* well defined *if*

- *the HAG is complete*
- *for each labeled structure tree S corresponding to a sentence in L(G) where parts derived by non-terminal attributes are removed, no partial built tree contains cycles in D using EDDP.*
- *on every path in every structure tree a particular non-terminal attribute occurs at most once*

Proof It is clear that a well-defined HAG must be complete. The second and third items guarantee that every (non-terminal) attribute will be computed (see also Lemma 3.1) and that there will be only a finite number of expansions (see Lemma 3.2) of the labeled tree and the dependency relation of the attribute instances.

□

Definition 3.14 *Let* $p : X_0 \rightarrow X_1 \ldots \overline{X_i} \ldots X_n$ *be a* recursive production *in the sense that* $\overline{X_i}$ *is derivable from* X_i.

An higher order attribute grammar is bounded well-defined *if, for each labeled structure tree S corresponding to a sentence in L(G) where parts derived by non-terminal attributes are removed, all attributes are effectively computable and each recursive production* $p : X_0 \rightarrow X_1 \ldots \overline{X_i} \ldots X_n$ *generates finite sentences.*

A bounded well-defined higher order attribute grammar gives us the power to define and evaluate recursive functions. Finite expansion of the structure tree, however, is no longer guaranteed. The following example computes the faculty number.

$Pr : R \rightarrow F$ $\quad F.n := an\ integer > 0$
$P1 : F \rightarrow \overline{F}$ $\quad \overline{F}[1] :=$ **if** $F[0].n = 1 \rightarrow P2$
$\quad \square\ F[0].n \neq 1 \rightarrow P1$
$\quad$ **fi**
$\quad F[1].n := F[0].n\text{-}1$
$\quad F[0].r := F[1].r * F[0].n$
$P2 : F \rightarrow done$ $\quad F[0].r := 1$

We used the terms 'attribute evaluation' and 'Attribute Evaluation Algorithm' to define whether an AG is well-defined. Instead of using an algorithm we could have defined a relation on labeled trees, indicating whether a non-attributed labeled tree is well-defined. We used the algorithm because from that it is easy to derive conditions by which it can be checked whether a HAG is well-defined.

4 Ordered HAGs

In [Kastens 80] a condition is described for well-defined attribute grammars (WAGs): The semantic rules of an AG are well-defined if and only if there is no sentence in the language with circularly dependent attributes. In [Jazayeri 1975] it was proven that the deciding whether an AG is well-defined is an exponential problem. In [Kastens 80] ordered attribute grammars (OAGs) were defined, an attributed grammar is ordered if for each symbol a partial order over the associated attributes can be given, such that in any context of the symbol the attributes are evaluable in an order which includes that partial order. The ordering property can be checked by an algorithm, which depends polynomially in time on the size of the input grammar. "Visit-sequences" are computed from the attribute dependencies given by an OAG.

An ordered HAG is now characterized by the following condition: A partial order on the defining attribute occurrences in a production p can be defined. It determines a fixed sequence of computation for the defining attribute occurrences, applicable in any tree production p occurs in.

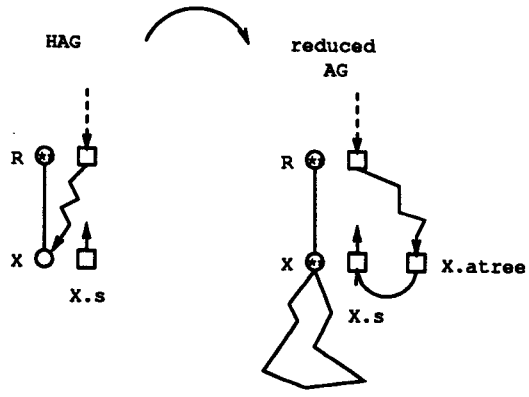


Figure 17: The same part of a structure tree in a HAG and the corresponding reduced AG

In this section a condition, using OAGs, is given in order to check whether a HAG is ordered.

4.1 Deriving partial orders from AGs

To decide whether a HAG is ordered the HAG is transformed into an AG and it is checked whether the AG is an OAG. The derived partial orders on defining attribute occurrences in the OAG can be easily transformed to partial orders on the defining occurrences of the HAG.

In the previous section (Lemma 3.1) it was shown that the EDDP ensured that every NTA could be computed. The reduced AG of a HAG is now defined as follows:

Definition 4.1 *Let H be a HAG. The* reduced AG H' *is the same as H except that all occurrences of NTA* $\overline{X}$ *in attribution rules are replaced by new inherited attributes* X.atree. *There were X is not a NTA the attribute X.atree gets a dummy value. Furthermore, all synthesized attributes of previously NTAs* $\overline{X}$ *now contain the attribute X.atree in the right-hand-side of their defining semantic function.*

The transformation is demonstrated in Figure 17.

This definition ensures that all synthesized attributes of NTA $\overline{X}$ (X.atree in the reduced AG) in the HAG can be only computed after NTA $\overline{X}$ (X.atree in the reduced AG) is computed.

Theorem 4.1 *A HAG is ordered if the corresponding reduced AG is an OAG.*

Proof Map the occurrences of X.atree in the partial orders of the reduced AG derived from a HAG to NTAs $\overline{X}$. The result are partial orders for the HAG in the sense that the HAG is ordered.

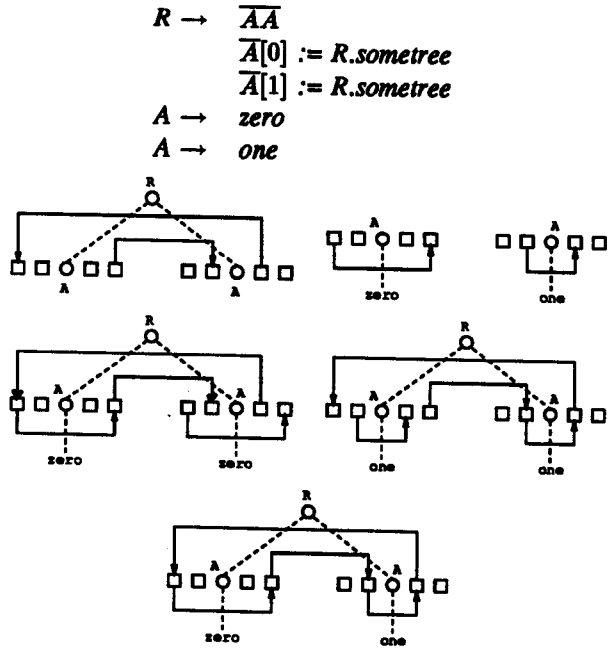


Figure 18: The lowest tree shows a cycle in the attribute dependencies as it is only possible in the reduced AG

□

The reduced AG of a HAG is somewhat pessimistic. Sometimes a HAG is ordered although the reduced AG is not an OAG, as is shown in Figure 18.

The class of OAGs is a sufficiently large class for defining programming languages, and it is expected that the above described way to derive evaluation orders for HAGs provides a large enough class of HAGs.

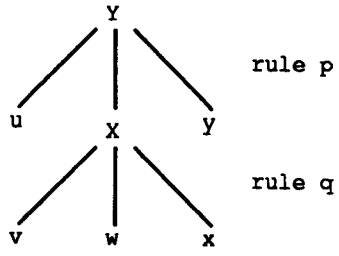
4.2 Visit-sequences for a HAG

The difference with visit-sequences as they are defined by [Kastens 80] for an OAG is that in a HAG the instruction set is expanded by an instruction to evaluate a non-terminal attribute and expand the labeled tree at the corresponding virtual non-terminal. The following introduction to visit-sequences for a HAG is almost literally taken from [Kastens 80].

The evaluation order is the base for the construction of a flexible and efficient attribute evaluation algorithm. It is closely adapted to the particular attribute dependencies of the AG. The principle is demonstrated here. Assume that an instance of X is derived by

$$S \Rightarrow uYy \rightarrow_p uvXxy \rightarrow_q uvwxy \Rightarrow s.$$

Then the corresponding part of the structure tree is



An attribute evaluation algorithm traverses the structure tree using the operations "move down to a descendant node" (e.g. from K_y to K_x) or "move up to the ancestor node" (e.g. from K_x to K_y). During a visit of node K_y some attributes of $AF(p)$ are evaluated according to semantic functions, if p is applied at K_y. In general several visits to each node are needed until all attributes are evaluated. A local tree walk rule is associated to each p. It is a sequence of four types of moves: move up to the ancestor, move down to a certain descendant, evaluate a certain attribute and evaluate followed by expansion of the labeled tree by the value of a certain non-terminal attribute. The last instruction is specific for a HAG.

Visit-sequences for a HAG can be easily derived from visit-sequences of the corresponding reduced AG. In an OAG the visit-sequences are derived from the evaluation order on the defining attribute occurrences. A description of a visit-sequence in an OAG is given, see [Kastens 80] for a precise definition. The visit-sequence of a production p in an AG will be denoted as VS(p) and in the HAG as HVS(p).

Definition 4.2 *Each* visit-sequence *VS(p) associated to a rule $p \in P$ in an AG is a linearly ordered relation over defining attribute occurrences and visits.*

$$VS(p) \subseteq AV(p) \times AV(p),\ AV(p) = AF(p) \cup V(p)$$

$$V(p) = \{v_{k,i} | 0 \leq i \leq np, 1 \leq K \leq nv_X, X = X_i\}$$

$v_{k,0}$ denotes the k-th ancestor visit, $v_{k,i}$, $i > 0$ denotes the k-th visit of the descendant X_i. For the definition of VS(p) see [Kastens 80]. We now define the HVS(p) in terms of the VS(p).

Definition 4.3 *Each* visit-sequence *HVS(p) associated to a rule $p \in P$ in a HAG is a linearly ordered relation over defining attribute occurrences, visits and expansions.*

$$HVS(p) \subseteq HAV(p) \times HAV(p),\ HAV(p) = AV(p) \cup VE(p)$$

$$VE(p) = \{e_i \mid 1 \leq i \leq np\}$$

where AV(p) is defined as in the previous definition.

$$HVS(p) = \{g(\gamma) \rightarrow g(\delta) \mid (\gamma \rightarrow \delta) \in VS(p)\}$$

with $g : AV(p) \rightarrow HAV(p)$ *defined as*

$$g(a) = \begin{cases} e_i & \text{if } a \text{ is of the form } X_i.atree \\ a & \text{otherwise} \end{cases}$$

e_i denotes the computation of the non-terminal attribute X_i and the expansion of the labeled tree at $\overline{X}_i$ with the tree computed in X_i.

Note that a descendant of a virtual non-terminal only can be visited *after* the virtual non-terminal is instantiated. The visit-sequences in the OAG are defined in such a way that during a visit to a node one or more synthesized attributes are computed. Because all synthesized attributes of a virtual non-terminal $\overline{X}$ depend by definition on the non-terminal attribute, the corresponding attribute *X.atree* in the OAG will be first computed; after that the first visit to a descendent of the instantiated non-terminal will be made.

In [Kastens 80] it is proved that the check and the computation of the visit-sequences $VS(p)$ for an OAG depends polynomially in time on the size of the input grammar. The mapping from the HAG to the reduced AG and the computation of the visit-sequences $HVS(p)$ depend also polynomially in time on the size of the input grammar. So the subclass of well-defined HAGs derived by computation of the reduced AG, analyzing whether the reduced AG is an OAG, computation of the visit-sequences for an HAG and checking whether there are no infinite expansions (see Lemma 3.2) can be checked and computed in time polynomially depending on the size of the input grammar. Furthermore an efficient and easy to implement algorithm, as for OAGs, based on visit-sequences can be used to evaluate the attributes in a HAG.

5 Conclusion and final remarks

Higher order attribute grammars were presented. It has been shown how HAGs can be used to describe multi-pass compilers. A method to derive evaluation orders was presented, the algorithm computing the method depending polynomially on the size of the input grammar. An efficient and easy to implement algorithm for attribute evaluation based on visit-sequences was derived by using OAGs.

Currently parallel evaluation of attributed trees is a topic of research at Utrecht University. Given the parallel evaluation of attributed structure trees and higher order attributed trees, parallel evaluation within semantic functions can be achieved too. Semantic functions can be defined by means of structured trees where the arguments of the semantic function are definitions for the inherited attributes of the corresponding attributed tree and the result of the function is defined as a synthesized attribute of the corresponding attributed tree. So attributed trees and the semantic functions occurring in them are merged into a single formalism and may thus both be evaluated in parallel. Semantic functions defined by attributed structure trees are defined in a somewhat different way by [Reps and Teitelbaum 1987, p. 47] and called *attribution expressions.*

Furthermore, the HAG-formalism will be used as a basis for the development of a program transformation system.

References

[Jazayeri 1975] M. Jazayeri, W.F. Ogden, W.C. Rounds. *The intrinsically exponential complexity of the circularity problem for attributed grammars.* In CACM 18, pages 679–706, 1975.

[Kastens, Hutt and Zimmerman 81] U. Kastens, B. Hutt, and E. Zimmerman. *GAG: A Practical Compiler Generator.* Springer, 1982.

[Ganzinger and Giegerich 84] H. Ganzinger and R. Giegerich. Attribute Coupled Grammars *Sigplan Notices Vol. 19, No. 6,* pages 157–170, 1984.

[Kastens 80] U. Kastens. Ordered Attributed Grammars. *Acta Informatica,* 13, pages 229–256, 1980.

[Knuth 1968] D.E. Knuth. Semantics of context-free languages. *Math. Syst. Theory,* 2(2):127–145, 1968.

[Knuth 1971] D.E. Knuth. Semantics of context-free languages (correction). *Math. Syst. Theory,* 5(1):95–96, 1971.

[Reps 1982] T. Reps. *Generating language based environments.* Tech. Rep. 82-514 and Ph.D dissertation, Dept. of Computer Science, Cornell Univ., Ithaca, N.Y., Aug. 1982.

[Reps, Teitelbaum and Demers 1983] T. Reps, T. Teitelbaum and A. Demers. *Incremental Context-Dependent Analysis for Language Based Editors.* In ACM Transactions on Progr. Lang. and Systems, Vol. 5, No. 3, pages 449-477, July 1983.

[Reps and Teitelbaum 1987] Reps, T. and Teitelbaum, T. *The Synthesizer Generator Reference Manual.* Cornell University, July, 1987.

[Waite and Goos 84] W.M. Waite and G.Goos. *Compiler Construction.* Springer, 1984.

Higher order attribute grammars

H.H. Vogt, S.D. Swierstra and M.F. Kuiper

RUU-CS-89-4
March 1989

Rijksuniversiteit Utrecht

Vakgroep informatica

Padualaan 14 3584 CH Utrecht
Corr. adres: Postbus 80.089, 3508 TB Utrecht
Telefoon 030-531454
The Netherlands