Acta Informatica 13, 229–256

Ordered Attributed Grammars

Uwe Kastens

Institut für Informatik II, Universität Karlsruhe, Postfach 6380,
D-7500 Karlsruhe 1, Germany (Fed. Rep.)

Summary. Ordered attributed grammars are defined as a large subclass of semantically well-defined attributed grammars proposed by Knuth. 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 definition does not refer to a predefined strategy for attribute evaluation, e.g. several passes from left to right. For each attributed grammar evaluable by any predefined evaluation strategy such an order exists. 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 ordered attributed grammar. They describe the control flow of an algorithm for attribute evaluation which can be part of an automatically generated compiler.

Contents

1. Introduction

Attributed grammars (AGs) are a well suited means for a complete definition of a programming language, including all statically determinable properties of the language. In [9, 10] Knuth established a condition for well-defined attributed

0001-5903/80/0013/0229/$05.60

grammars (WAGs): The semantic rules of an AG are well-defined if and only if there is no sentence of the language with circularly dependent attributes. In [4] it was proven that the decision whether an AG is well-defined is an intrinsical exponential problem.

In this paper we introduce "ordered attributed grammars" (OAGs) as a subclass of WAGs. Grammars of this class are characterized by the following condition: For each symbol of the grammar a partial order over the associated attributes can be defined, such that in any context of the symbol the attributes are evaluable in that order. Such grammars have several desirable properties with respect to the definition of programming languages and to automatic generation of compilers:

- An OAG is well-defined in the sense of [9].
- The class of OAGs is defined without any assumption on the strategy for attribute evaluation, e.g. one or more passes over the program from left to right as for example in [1]. Furthermore for all AGs of a class based on predefined evaluation strategies such an order exists.
- For an OAG one can automatically construct compiler algorithms evaluating the attributes of any sentence of the language.
- The class of OAGs is sufficiently large for programming language definition. The context dependencies of programming languages can usually be defined by OAGs. The dependencies between the properties (attributes) of a language element define a partial order in each possible context. In general the superposition of these partial orders result in a new partial or linear order. In this sense the condition for the subclass is natural.
- It is decidable whether an AG is ordered. The time needed for that decision and for the computation of the order depends polynomially on the input grammar. The analysis of OAGs for usual programming languages can be done in reasonably short time.

Compiler generating systems based on different subclasses of AGs already exist ([2, 11, 12, 14]). In order to generate complete compilers the AGs are extended such that they describe the code to be generated for a target machine. (Different approaches are discussed in [6, 16, 17].) A generator based on OAGs (GAG) is currently being implemented by the author. A rather complete bibliography on AGs is found in [15].

In Sect. 2 the notation of AGs and an example is given, which is referred to in further sections. Section 3 defines the condition for OAGs and how to check it. The position of OAGs within the hierarchy of classes of AGs is shown in Sect. 4. The ordering property of an OAG is used in Sect. 5 for computing tree walk rules (called "visit-sequences") for attribute evaluation. They can be implemented in automatically generated compilers as shown in Sect. 6. In Sect. 7 we give an abstract algorithm for checking whether an AG is ordered and computing the visit-sequences and discuss its complexity.

2. Attributed Grammars

In this section we introduce our notation for AGs. It differs to some degree from other notations used in literature in order to achieve completeness and readability.

As an example for further reference we consider the definition of a small expression language.

An AG is based on a context-free grammar which is augmented by attributes, functions defining values of attributes, and conditions over attributes. Each attribute describes a property of a language element, which is defined by a function in a context dependent manner. The conditions restrict the combinations of these properties according to static semantics.

An AG is defined by a 5-tupel

$$AG=(G, A, VAL, SF, SC).$$

$G=(N, T, S, P)$ is a reduced context-free grammar, where N is the set of nonterminal symbols, T is the set of terminal symbols, $V=N \vee V$ is the vocabulary of the grammar, $S \in N$ is the start symbol, and P is the set of syntactic rules. Each syntactic rule $p \in P$ has the form

$$p=X_0: X_1 \ldots X_{np}, \quad np \geqq 0$$

X_i denotes an *occurrence* of a symbol of N for $i=0$ and of V for $i>0$. In the following X_i will always denote an occurrence of a symbol X in such a rule p. The qualification of the index $0 \leqq i \leqq np$ will be implied.

A is a set of attributes. Each attribute is associated to exactly one symbol $X \in V$. A_X is the set of attributes associated to X. The elements of A_X are denoted $X \cdot a, X \cdot b, \ldots$. We have

$$A=\bigcup_{X \in V} A_X \quad \text{and} \quad A_X \wedge A_Y \neq \emptyset \text{ implies } X=Y.$$

A is partitioned into two disjoint subsets AI and AS, the *inherited* and the *synthesized* attributes:

$$A=AI \vee AS \quad \text{and} \quad AS \wedge AI=\emptyset.$$

Hence each A_X is partitioned into two subsets AI_X and AS_X. For each occurrence X_i of a symbol X in a rule p there is an *attribute occurrence* $X_i \cdot a$ for all attributes $a \in A_X$.

$$A_p=\bigcup_{i=0}^{np} \bigcup_{a \in A_X} X_i \cdot a$$

is the set of all attribute occurrences in p.

$$SF=\bigcup_{p \in P} SF_p$$

is a set of *semantic functions* associated to rules $p \in P$. Each semantic function defines the value of an attribute occurrence in p depending on $k \geqq 0$ attribute occurrences in p:

$$f \in SF_p$$
$$f \subseteq \bigtimes_{j=0}^{k} \mathrm{DOM}(b_j \in A_p) \rightarrow \mathrm{DOM}(a \in A_p), \quad \text{for a } k \geqq 0$$
e.g. $X_i \cdot a := g(X_m \cdot c, \ldots, X_n \cdot d, \ldots)$.

The values defined for the attribute occurrences are taken from a set of possible values, the domain of the attributes: $\mathrm{DOM}(X_i \cdot a) = \mathrm{DOM}(X \cdot a)$, for $X = X_i$.

VAL is the set of the domains of all attribute values:

$$VAL = \{\mathrm{DOM}(a) \mid a \in A\}.$$

For each $X_i \cdot a$ of the set of *defining occurrences*

$$AF_p = \{X_i \cdot a \mid (i = 0 \text{ and } a \in AS) \text{ or } (i > 0 \text{ and } a \in AI)\}$$

there is exactly one function in SF_p defining the value of $X_i \cdot a$. Thus it is ensured that the value of each attribute is determined uniquely in any context. The set of *applied occurrences* (not defined by SF_p) is

$$AC_p = A_p \setminus AF_p.$$

The attribute values are *statically* defined by the semantic functions, comparable to the objects and functions of the lambda calculus. The functions must not be looked upon as algorithms on variable attributes!

Let s be a sentence of $L(G)$ which is derived by

$$S \Rightarrow u\,Yy \rightarrow_q u\,v\,X\,x\,y \rightarrow_r u\,v\,w\,x\,y \Rightarrow s.$$

A node K representing the symbol X in the structure tree for s is called an *instance* of X denoted by K_X. For each attribute $X \cdot a$ an *attribute instance* $K_X \cdot a$ is associated to K_X. The values of the inherited attribute instances of K_X are defined by functions in SF_q, the values of the synthesized ones by functions in SF_r. A structure tree augmented by the attribute instances is called an *attributed structure tree*.

SC is a set of semantic conditions, one associated to each rule p:

$$SC_p \in \bigtimes_{j=1}^{k} \mathrm{DOM}(a_j \in A_p) \rightarrow \{\text{true, false}\}, \quad \text{for a } k > 0.$$

A sentence $s \in L(G)$ is a sentence of the language $L(AG)$ iff for each application of a rule p in the derivation of s the values of the corresponding attribute instances meet the condition SC_p. The following discussions are restricted to attribute dependencies determining the evaluation order. As semantic conditions and the domains of attribute values do not influence the attribute dependencies they are not considered here. In an actual language definition or compiler generation domains correspond to data types of modern high level programming languages. (For more details see [7].)

As an example for further reference we consider the definition of a simple expression language. The example covers four of the most important context sensitive properties of languages: scope rules, mode checking, coercion and operator identification. In addition an optimizing method (constant folding) is incorporated in a very simple way. We use an informal notation which can be converted into a systematic description language [7].

The following attributes are used:

description a pair (identifier, mode) describing a defined object;

access the set of descriptions of declared objects visible from the syntactic unit in question;

primode the mode (int or 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);

evaluable indicates whether the value of the syntactic unit can be computed statically;

value the value of the syntactic unit if it is computable statically;

id a unique representation of an identifier denotation.

The attribute sets are:

$AI = \{X.access | X \in \{expression, primary, assignment, declaration\}\}$
$\vee \{X.postmode | X \in \{expression, primary, assignment\}\}$

$AS = \{X.primode | X \in \{expression, primary, assignment\}\}$
$\vee \{X.evaluable | X \in \{expression, primary\}\}$
$\vee \{X.value | X \in \{expression, primary, intconstant, realconstant\}\}$
$\vee \{declaration.description\}$
$\vee \{identifier.id\}$

The language is defined by:

p_1: **rule** *program: primary*
semantic
primary.access := $\emptyset$;
primary.postmode := *primary.primode*
end

p_2: **rule** *primary*: '('*declaration*';' *assignment*')'
semantic
declaration.access := *primary.access*;
assignment.access :=
include (*primary.access*, *declaration.description*);
primary.primode := *assignment.primode*;
assignment.postmode := *primary.postmode*;
primary.evaluable := *false*;
primary.value := *undefined*
end

p_3: **rule** *primary: identifier*
semantic
primary.primode :=
identify (*identifier.id*, *primary.access*);
primary.evaluable := *false*;
primary.value := *undefined*;
condition
isdefined(*identifier.id*, *primary.access*)
end

p_4: **rule** *primary : intconstant*
semantic
primary . primode := int ;
primary . evaluable := true ;
primary . value :=
if *primary . postmode = real*
then *widen(intconstant . value)*
else *intconstant . value* **fi**
end

p_5: **rule** *primary : realconstant*
semantic
primary . primode := real ;
primary . evaluable := true ;
primary . value := realconstant . value
end

p_6: **rule** *assignment : identifier* ':=' *expression*
semantic
expression . access := assignment . access ;
assignment . primode :=
identify (identifier . id, assignment . access) ;
expression . postmode := assignment . primode ;
condition
isdefined(identifier . id, assignment . access) **and**
not *(expression . primode = real* **and**
expression . postmode = int)
end

p_7: **rule** $expression_1$: $expression_2$ '+' *primary*
semantic
$expression_2$ *. access* := $expression_1$ *. access ;*
primary . access := $expression_1$ *. access ;*
$expression_1$ *. primode :=*
if $expression_2$ *. primode = int*
and *primary . primode = int*
then *int* **else** *real* **fi** ;
$expression_2$ *. postmode* := $expression_1$ *. primode ;*
primary . postmode := $expression_1$ *. primode ;*
$expression_1$ *. evaluable :=*
$expression_2$ *. evaluable* **and** *primary . evaluable ;*
$expression_1$ *. value :=*
if $expression_1$ *. evaluable*
then *add(* $expression_2$ *. value, primary . value)*
else *undefined* **fi**
end

p_8: **rule** *expression : primary*
semantic transfer
end

p_9: **rule** *declaration*: 'new' *identifier*' := 'expression
semantic
expression.access := *declaration.access*;
declaration.description :=
(*identifier.id*, *expression.primode*);
expression.postmode := *expression.primode*
end

The meaning of the functions *include*, *isdefined*, *identify*, *add*, and *widen* are given informally:

include(a, d)	a is a set of descriptions, d is a description; the result is $a \vee \{d\}$, if a does not contain a description d' for the same identifier described by d, $(a \setminus \{d'\}) \vee \{d\}$ otherwise.
isdefined(id, a)	a is a set of descriptions, *id* is an identifier; the result is *true* if a contains a definition for *id*, *false* otherwise.
identify(id, a)	a is a set of descriptions, *id* is an identifier; the result is the mode defined for the identifier by a description in a. If no such description exists, the result is undefined.
add(v)	has the usual meaning for arithmetic values.
widen(v)	converts an integer value to a real value.

Rule p_2 describes the block structure of the language. For simplification only one declaration and one statement are allowed in each block. The scope rules described by the attribute *access* are defined as usual for block structured languages. The first semantic function of p_2 says that the declared identifier must not be applied in the declaration part. In order to enlarge the complexity of attribute dependencies the mode of a declared identifier is not given explicitly in rule p_9. It is determined by the mode of the initialization expression.

There are no semantic functions defining the attributes *identifier.id*, *constant.value*, and *realconstant.value*. Such attributes of terminal symbols are implicitly defined by the symbol text. Semantic conditions are headed by **condition**. **transfer** in p_8 is a shorthand notation for semantic functions which define corresponding attributes of *expression* and *primary* to have the same value.

3. Deciding Whether Attributed Grammars are Ordered

In this section we introduce a method for deciding whether a given AG is ordered. The method is based on the graph representation of dependency relations between attributes which was introduced by Knuth in [9]. The essential problem is to reduce a condition stated for the dependencies in the (infinite) set of sentences of a grammar to another condition stated for the dependencies between the finite set of attributes. The problem is solved by "projection" of the attribute dependencies in sentences into dependency relations associated to syntactic rules and symbols.

The basic idea of OAGs is the following: For each symbol $X \in V$ of a given AG a partial order DS_X over the attributes A_X is constructed. (*DS* abbreviates *dependencies* between attributes of *symbols*.) It determines an evaluation order for

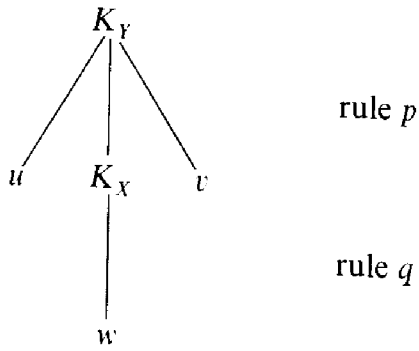
the attributes, applicable in any context X occurrs in. $(X \cdot a, X \cdot b) \in DS_X$ indicates that an attribute instance $K_X \cdot a$ must be evaluated before $K_X \cdot b$ (of the same K_X) for any symbol instance K_X in any structure tree. If neither $(X \cdot a, X \cdot b)$ nor $(X \cdot b, X \cdot a)$ is in DS_X, the evaluation order of corresponding attribute instances can be chosen arbitrarily for any K_X.

The existence of such an order is a sufficient but not neccessary condition for the well-definedness of the AG (see Sect. 4). The relation DS_X must comprise all direct and indirect dependencies, which may be derived from any possible context of X. Furthermore for different symbol occurrences in the same context the relations must be compatible with the dependencies between the attribute occurrences.

The evaluation order DS_X 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, for details see Sect. 5. 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 a node K_Y some attributes of AF_p are evaluated according to semantic functions of SF_p, 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 instructions of three types: move up to the ancestor, move down to a certain descendant, and evaluate a certain attribute.

The relations DS_X act as "interfaces" between visit-sequences, assuring that the visit-sequences for a tree node and for its descendants fit together in the following sense: A move down from K_Y to K_X is made in order to evaluate the attributes of a certain subset of AS_X. Their values can be used for further evaluations of functions in SF_p after the traversal has returned to K_Y. The existence of such a relation DS_X assures that the subset is the same for all rules q deriving X. Correspondingly after a move to K_X the subset of AI_X of additionally evaluated attributes is the same for each occurrence of X on the righthand side of any rule p. Therefore each DS_X must define a linear order over subsets of A_X, which contain alternatingly inherited and synthesized attributes only. In general DS_X is partially ordered, since the evaluation order within each subset is not relevant for the interface.

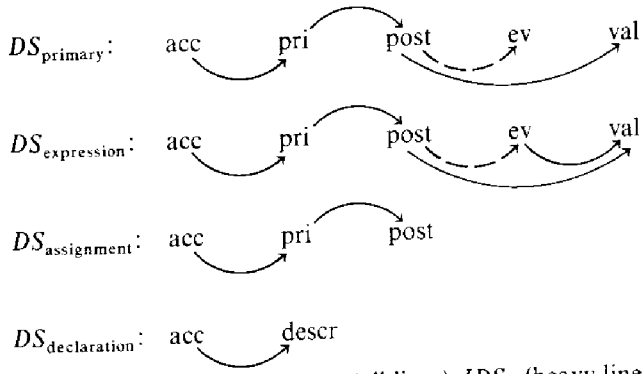


Fig. 1. Dependency graphs DS_X (all lines), IDS_X (heavy lines)

The AG is an OAG, if a dependency relation DS with the properties discussed above can be constructed according to the following definitions. Figure 1 shows the graph DS for our example.

Definition 1.

$$DP = \bigcup_{p \in P} DP_p, \quad DP_p \subseteq A_p \times A_p$$

is the relation of direct *dependencies* between attribute occurrences associated to *production rules*, where

$$DP_p = \{(X_i \cdot a, X_j \cdot b) | \text{there is a semantic function in } SF_p \text{ defining } X_j \cdot b \text{ depending on } X_i \cdot a\}.$$

The relations DP_p for our example are given in Fig. 2 by the heavy lines only. A dependency graph over the attribute instances in a structure tree for a sentence can be constructed by "pasting together" graphs DP_p according to the applications of rules p in the derivation of the sentence. Figure 3 shows such a graph for a sentence of our example.

In the next step we construct a dependency relation IDP over attribute occurrences. IDP is defined recursively: Starting from DP a direct or indirect dependency between attributes of one symbol occurrence induces a dependency between corresponding attributes of all occurrences of that symbol.

In the following we use the notation D^+ for the non reflexive transitive closure of D. In the graph representation an arc (a, b) is in D^+ iff an oriented path from a to b is in D.

Definition 2.

$$IDP = \bigcup_{p \in P} IDP_p, \quad IDP_p \subseteq A_p \times A_p$$

is the relation of *induced dependencies* between attribute occurrences, where

$$IDP_p = DP_p \cup \{(X_i \cdot a, X_i \cdot b) | X_i \text{ occurrs in rule } p,\ Y_j \text{ occurrs in rule } q,\ X_i = Y_j,\ 0 \leqq j \leqq ng,\ (Y_j \cdot a, Y_j \cdot b) \in IDP_q^+\}.$$

The graphs IDP_p for our example are shown in Fig. 2.

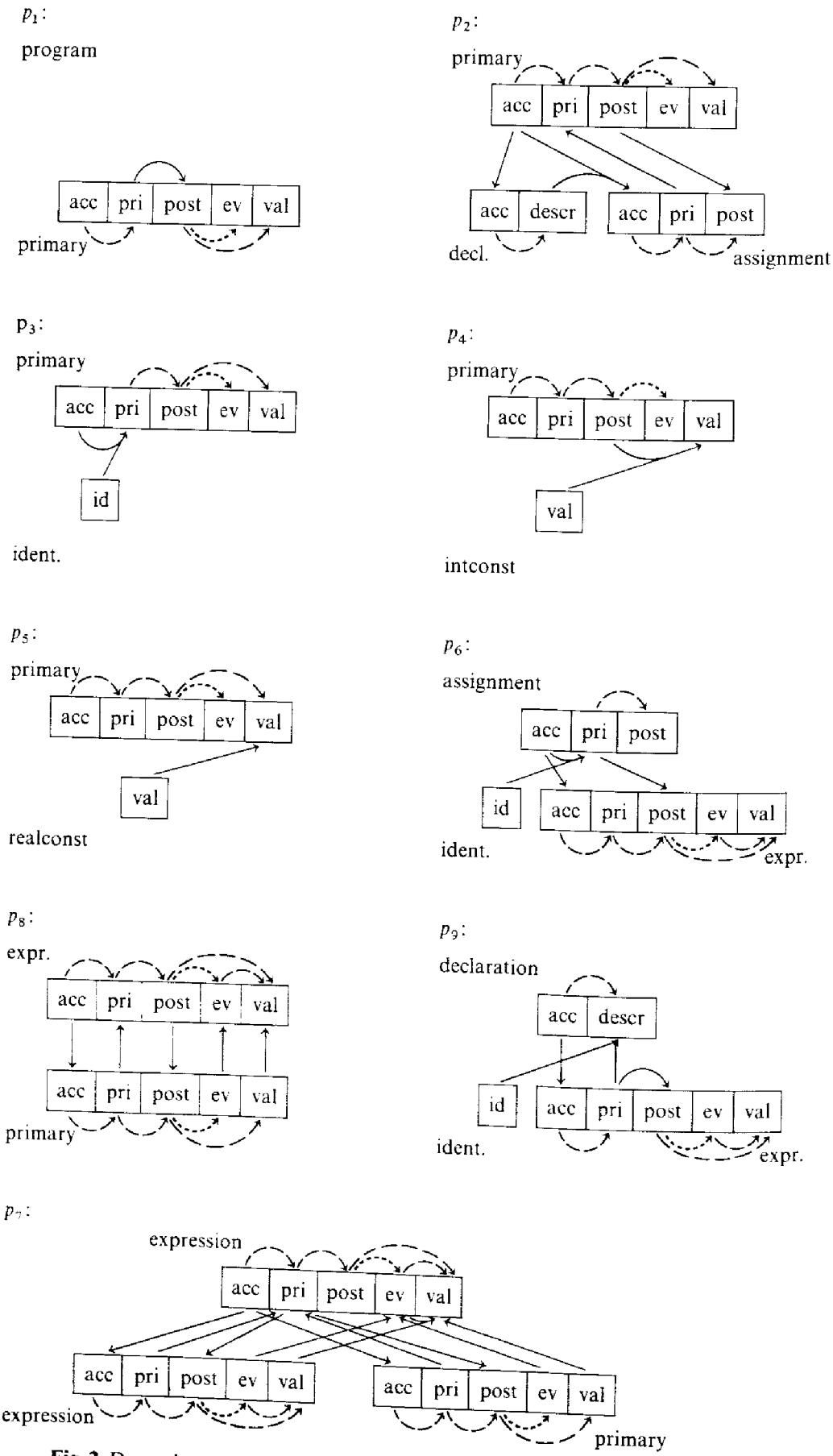


Fig. 2. Dependency graphs DP_p (heavy lines), IDP_p (heavy and dashed lines), EDP_p (all lines)

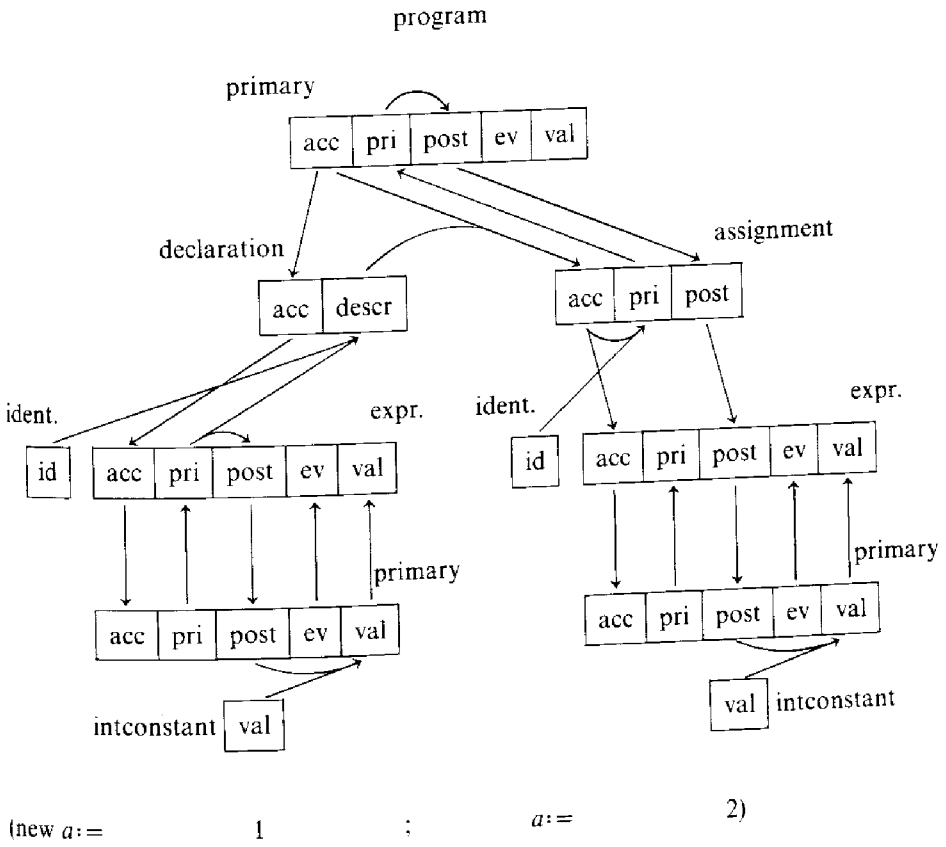


Fig. 3. Dependencies in an attributed structure tree

The construction of IDP "projects" indirect dependencies into dependencies between attribute occurrences. These dependencies are extracted in the next step.

Definition 3.

$$IDS = \bigcup_{X \in V} IDS_X \subseteq A \times A$$

is the relation of *induced dependencies* between attributes of *symbols*, where

$$IDS_X = \{(X \cdot a, X \cdot b) | \text{there is an } X_i = X \text{ in a rule } p \text{ and } (X_i \cdot a, X_i \cdot b) \in IDP_p\}.$$

Note. $(X \cdot a, X \cdot b) \in IDS$ implies that $(X_i \cdot a, X_i \cdot b) \in IDP_p$ for each occurrence X_i of X in each rule p. If there is a dependency between attribute instances $K_X \cdot a$ and $K_X \cdot b$ in the structure tree for any sentence of the language, $(X \cdot a, X \cdot b)$ is in IDS. On the other hand $(X \cdot a, X \cdot b) \in IDS$ does not imply that a dependency exists in any structure tree between corresponding attribute instances. In that sense IDS is "pessimistic". The graphs IDS_X for our example are given in Fig. 2.

In the next steps IDS will be completed to the relation DS such that for each two attributes $X \cdot a \in AI$, $X \cdot b \in AS$ either $(X \cdot a, X \cdot b) \in DS$ or $(X \cdot b, X \cdot a) \in DS$ holds. Then each DS_X defines a linear order over disjoint subsets $A_{X,k}$ of A_X. Each $A_{X,k}$ consists of those synthesized or inherited attributes whose values are additionally available after a move up or down in the tree. The evaluation order

$X=$	m_X	$A_{X,4}$	$A_{X,3}$	$A_{X,2}$	$A_{X,1}$
primary expression	4	acc	pri	post	val, ev
assignment	4	acc	pri	post	–
declaration	2			acc	descr

Fig. 4. Disjoint partitions $A_{X,k}$

corresponds to *decreasing* order of the index k. Hence the subsets are defined such that $A_{X,k}$ contains attributes which contribute directly to the computation of attributes in $A_{X,k-1}$.

Definition 4. Let IDS be acyclic. For each $X \in V$ we define successively

$$A_{X,1} = \{X \cdot a \in AS \mid \text{there is no } X \cdot b \text{ such that } (X \cdot a, X \cdot b) \in IDS^+\},$$

$$A_{X,2n} = \{X \cdot a \in AI \mid \text{for all } X \cdot b \in A_X : (X \cdot a, X \cdot b) \in IDS^+ \text{ implies } X \cdot b \in A_{X,m},\ m \leqq 2n\} \setminus \bigcup_{k=1}^{2n-1} A_{X,k},$$

$$A_{X,2n+1} = \{X \cdot a \in AS \mid \text{for all } X \cdot b \in A_X : (X \cdot a, X \cdot b) \in IDS^+ \text{ implies } X \cdot b \in A_{X,m},\ m \leqq 2n+1\} \setminus \bigcup_{k=1}^{2n} A_{X,k}$$

until each $X \cdot a \in A_X$ is in an $A_{X,k}$. The sets $A_{X,k}$ form a disjoint partition of A_X:

$$A_X = \bigcup_{k=1}^{m_X} A_{X,k} \quad \text{for } m_X \geqq 1,$$

$$A_{X,k} \wedge A_{X,j} \neq \emptyset \text{ implies } k = j.$$

Note. The disjoint partition could have been as well defined starting with the attributes evaluable first. In that case we had to decide whether the first subset contains inherited or synthesized attributes, i.e. whether attribute evaluation starts at the root or at the leafs of the tree. This assumption is not needed in Definition 4.

The disjoint partitions for our example are shown Fig. 4.

Definition 5. Let IDS be acyclic. The dependency relation DS is defined as a completion of IDS:

$$DS = \bigcup_{X \in V} DS_X \subseteq A \times A$$

$$DS_X = IDS_X \vee \{(X \cdot a, X \cdot b) \mid X \cdot a \in A_{X,k},\ X \cdot b \in A_{X,k-1},\ 2 \leqq k \leqq m_X\}.$$

In the last step we complete the dependency relation IDP according to the completion of IDS, in order to check that the completion does not cause cycles.

Definition 6. The *extended dependency* relation over attribute occurrences EDP is defined as a completion of IDP:

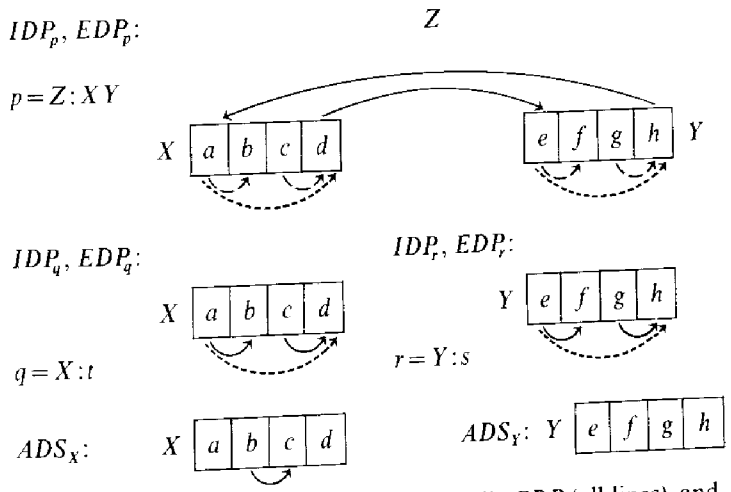


Fig. 5. Acyclic *IDP* (heavy and dashed lines), cyclic *EDP* (all lines), and *ADS*

$$EDP = \bigcup_{p \in P} EDP_p$$

$$EDP_p = DP_p \vee \{(X_i \cdot a, X_i \cdot b) | (X \cdot a, X \cdot b) \in DS, X_i = X\}.$$

DS is said to be *compatible* with the attribute dependencies if *EDP* is acyclic.

Definition 7. A given AG is an OAG iff the dependency relation *DS* exists and is compatible with the attribute dependencies.

There exist AGs (usually not occurring in practical applications) where *DS* is not compatible with the attribute dependencies. This situation is caused by the completion of the relations IDS_X. (Figure 5 gives an example.) If *IDP* is acyclic one can show that each cycle in an EDP_p contains at least two arcs, which are introduced by arcs $(X \cdot a, X \cdot b), (Y \cdot c, Y \cdot d) \in DS \setminus IDS$, where *X* and *Y* correspond to different symbol occurrences in *p*. Furthermore $X \cdot a$, $X \cdot b$ and $Y \cdot c$, $Y \cdot d$ are independent in IDS^+. The arcs are in *DS* because the attributes in question are associated to different subsets of A_X and A_Y according to Definition 4.

If there are attributes independent in IDS^+ one can enforce a different disjoint partition of *A* by adding dependencies which do not correspond to semantic functions. An arbitrary set of dependencies $ADS \subseteq A \times A$ is called *augmenting dependencies* between attributes of *symbols*.

Definition 8. Let *DP* be the direct dependencies of an AG and *ADS* a set of augmenting dependencies. Let *DS'* be computed on the base of *DP'*, where

$$DP'_p = DP_p \vee \{(X_i \cdot a, X_i \cdot b) | (X \cdot a, X \cdot b) \in ADS\}.$$

An AG is *arranged orderly* by *ADS* if the AG together with *DS'* is an OAG.

An AG is arranged orderly, if *ADS* has the following property: It contains arcs $(X \cdot b, X \cdot a)$ such that in each cycle of the original *EDP* at least one arc $(X_i \cdot a, X_i \cdot b)$ is replaced by $(X_i \cdot b, X_i \cdot a)$, and no new cycles are introduced. In general the computation of an *ADS* with that property is a combinatorial problem of high complexity. An algorithmic solution shall not be discussed here because the problem has little practical relevance for AGs defining programming languages. For some restricted classes of AGs *ADS* can be given easily (see next section).

4. Hierarchial Classification of Attributed Grammars

In this section we compare OAGs with other classes of AGs with respect to the complexity of the expressible attribute dependencies. The expressive power of OAGs is larger than that of any class of AGs defined by a fixed evaluation strategy. Compared with well-defined AGs the restriction for OAGs does not exclude practical cases. The hierarchy of classes of AGs is listed below in descending order of expressive power:

AG attributed grammars
WAG well-defined attributed grammars [9]
ANCAG absolutely noncircular attributed grammars [8]
OAG ordered attributed grammars (defined in this paper)

---

m-APAG attributed grammars evaluable in m alternating passes [5]
n-PAG attributed grammars evaluable in n passes from left to right [1]
1-PAG = L-AG
attributed grammars evaluable in 1 pass from left to right [1], L-L-attributed grammars [13]
S-AG S-attributed grammars [13]

The definitions of the classes below the line are based on an apriori defined evaluation strategy, whereas for those above the line evaluation strategies must be computed from the attribute dependencies. In the following we shall discuss the relations between these classes. By simple examples it will be shown that the inclusions are strict.

The definition of absolutely noncircular AGs [8] can be derived from Definition 2 for OAGs:

An AG is an ANCAG iff the following dependency relation is acyclic:

$$IDP\text{-}ANCAG = \bigcup_{p \in P} IDP_p\text{-}ANCAG$$

$$IDP_p\text{-}ANCAG = DP_p \vee \{(X_i \cdot a, X_i \cdot b) \mid i > 0 \text{ and there is a } q \in P: q = Y_0 : w, \quad Y_0 = X_i, \text{ and } (Y_0 \cdot a, Y_0 \cdot b) \in IDP_q\text{-}ANCAG^+\}.$$

Obviously $IDP\text{-}ANCAG \subseteq IDP$, and for some AGs this inclusion is strict. Thus OAG $\subseteq$ ANCAG is a strict inclusion. Figure 6 gives an example for an ANCAG which is not an OAG.

For each AG which is not well-defined there exists an attributed structure tree containing a cycle. Therefore $IDP\text{-}ANCAG$ is cyclic and the AG is not absolutely noncircular. Figure 7 shows the dependencies of an AG, which is well-defined but not absolutely noncircular. Thus ANCAG $\subseteq$ WAG is a strict inclusion.

Several classes of AGs are defined by apriori fixed strategies for attribute evaluation, such that for each sentence of $L(G)$ all attributes can be evaluated in

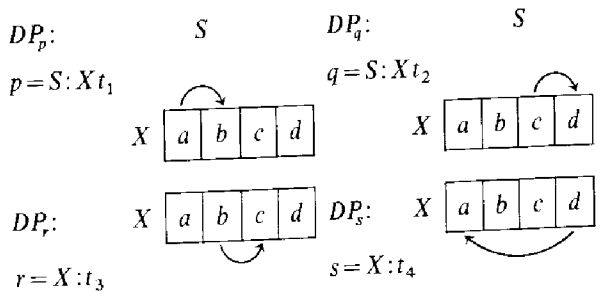


Fig. 6. ANCAG, but not OAG

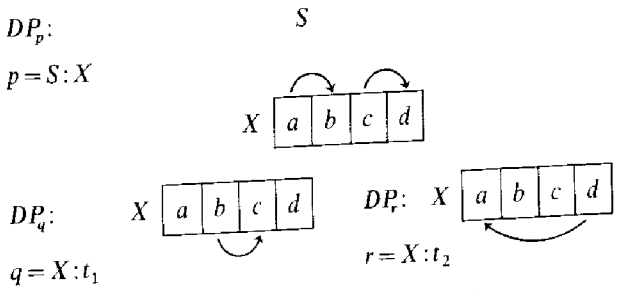


Fig. 7. WAG, but not ANCAG

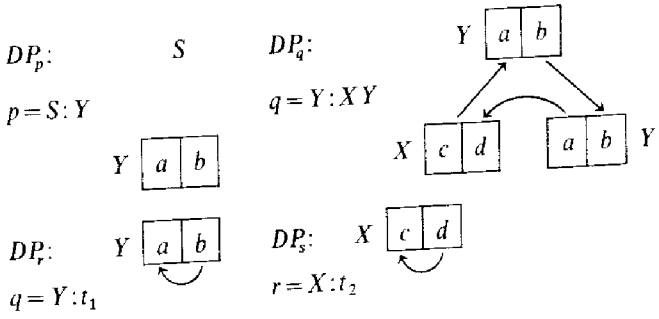


Fig. 8. 2-APAG, but not n-PAG

a certain number of passes over the structure tree. The most restrictive class contains the S-AGs defined in [13]. They only allow for synthesized attributes, which are evaluable in a single bottom-up pass.

Obviously they are included in 1-PAGs (called L-AGs in [13]), which are evaluable in a single top-down left to right pass. In [1] 1-PAGs are generalized to n-PAGs: For each $a \in A$ there is a number $1 \leqq k_a \leqq n$ such that any instance of a in any structure tree is evaluated in the k_a-th top-down left to right pass. The number of passes needed is the maximum number k_a.

A further generalization leads to m-APAGs, which are evaluable in m alternating top-down passes: the i-th pass proceeds from left to right (right to left) if i is odd (even). For each n-PAG there is an $m \leqq 2n-1$ such that it is an m-APAG, too. Figure 8 shows a situation for which the m-APAG condition holds but not the n-PAG condition for any n.

An m-APAG is an OAG or can be turned into an OAG by augmenting dependencies. For any m-APAG IDP and IDS are acyclic. If the completion of IDS leads to cyclic EDP, the AG can always be arranged orderly:

Let k_a be the number of the pass which evaluates the attribute a according to the APAG strategy. Since the passes proceed top-down any instance $K_X \cdot a$ is

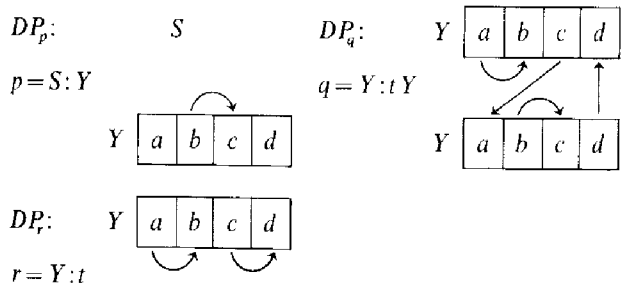


Fig. 9. OAG, but not *m*-APAG

evaluated before $K_X \cdot b$ if $k_a < k_b$ or $k_a = k_b$ and $a \in AI$ and $b \in AS$. Thus the *m*-APAG condition defines a dependency relation $IDS\text{-}APAG \subseteq A \times A$:

$$IDS\text{-}APAG = \{(X \cdot a, X \cdot b) | k_a < k_b \text{ or } (k_a = k_b, a \in AI, b \in AS)\}.$$

From Definition 3 follows that $IDS \subseteq IDS\text{-}APAG$ for all *m-APAGs*. Hence in the OAG check neither *IDP* nor *IDS* can be cyclic. In a situation as shown in Fig. 5 the completion of *IDS* causes *EDP* to be cyclic. In any such case the AG can be arranged orderly by $ADS = IDS\text{-}APAG$. So the class of *m*-APAGs is included in the class of OAGs in the sense that each *m*-APAG can be arranged orderly.

The inclusion is strict because there are OAGs which can not be evaluated in *m* alternating passes for any *m*. Such AGs contain recursive syntactic rules with attribute dependencies such that the number of evaluation passes is determined by the unlimited recursion depth (as shown in Fig. 9).

5. Visit-Sequences for Ordered Attributed Grammars

OAGs (and the larger classes as well) do not imply a predefined strategy for the walk through the structure tree during attribute evaluation. For each grammar of this class a special evaluation algorithm can be constructed based on the attribute dependencies. Such an algorithm implements the semantic analysis of a compiler for the defined language. It is independent of the compilation of any particular sentence: therefore it can be constructed at time of compiler generation. The situation is similar to the construction of a parsing algorithm from syntactic definitions.

The construction is based on the following idea: For each rule $p \in P$ a visit-sequence VS_p is computed. A visit-sequence is a local tree walk rule applied at each node of the structure tree, which is derived by *p*. It describes the order of visits to surrounding nodes and of evaluations of semantic functions between those visits. (It is assumed that the semantic functions are translated conveniently into the implementation language of the compiler. Only the applicability of the functions according to the attribute dependencies is considered here.)

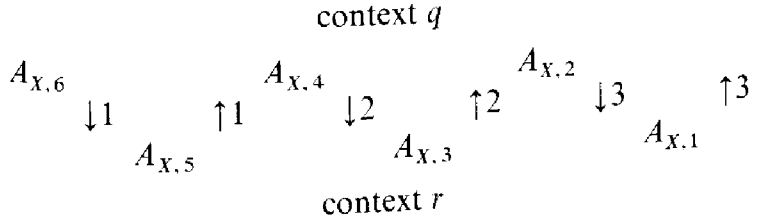
The ordering property of an attributed grammar yields both a rather simple construction of the visit sequences, and a rather simple implementation of the attribute evaluation algorithm. The visit-sequences for OAGs are linear sequences of actions (node visits and attribute evaluations). The next action to be executed

at evaluation time is completely determined by its predecessor. The comparable evaluation rules for ANCAGs ([8] and [3]) are partially ordered graphs. The next executable action is determined by its predecessor and a context dependent set of attributes already evaluated. Thus the evaluation algorithm is simpler for *OAGs* than for ANCAGs (see Sect. 6).

The construction of the visit-sequences is based on the dependency relation *EDP* defined in Sect. 3. As attribute evaluation may be done interleaved with the construction of the structure tree, we shall distinguish three cases for the construction of the visit-sequences:

TC: Attribute evaluation starts, when the whole structure tree is completed.
BU: Attribute evaluation is done interleaved with bottom-up tree construction.
TD: Attribute evaluation is done interleaved with top-down tree construction.

For each rule q a visit-sequence VS_q will be constructed separately. For any pair of rules $q = Y: uXv$ and $r = X: w$ VS_q and VS_r are constructed such that they fit together in the sequence of attribute evaluation and the moves between context q and context r (ancestor and descendant visits). This interface is defined by the disjoint partition of A_X (Definition 4). Such an interface for the TC- and TD-case and $m_X = 6$ is given below:

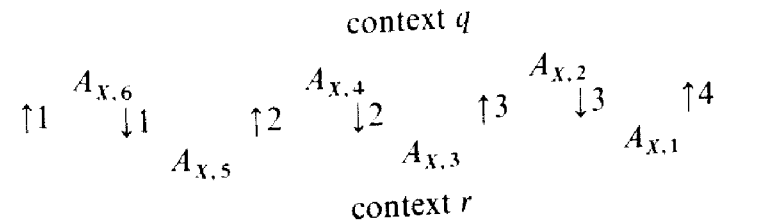


The inherited attributes ($A_{X,m}$ for even m) are evaluated in the context q, the synthesized ($A_{X,m}$ for odd m) in the context p. $\downarrow k$ denotes the k-th descendant visit from q, $\uparrow k$ denotes the k-th ancestor visit from r. The number of ancestor and descendant visits are both

$$nv_X = f_X \textbf{ div } 2,$$

where f_X is the smallest even number $f_X \geqq m_X$ (for the TD-case).

In the BU-case tree construction and attribute evaluation starts from the leafs of the tree. Thus the first visit in the interface moves in upward direction:



In the BU-case the number of ancestor and descendant visits is

$$nvup_X = (f_X + 1) \textbf{ div } 2, \quad nvdown_X = nvup_X - 1$$

where f_X is the smallest odd number $f_X \geqq m_X$.

The construction of VS_p replaces in the graph EDP_p all attribute occurrences not evaluated in the context p by corresponding ancestor or descendant visits.

Definition 9. Each *visit-sequence* VS_p associated to a rule $p \in P$ is a linearly ordered relation over defining attribute occurrences and visits:

$$VS_p \subseteq AV_p \times AV_p, \quad AV_p = AF_p \vee V_p,$$

$$V_p = \{v_{k,i} | 0 \leqq i \leqq np,\ 1 \leqq k \leqq nv_X,\ X = X_i\}.$$

$v_{k,0}$ denotes the k-th ancestor visit, $v_{k,i}, i > 0$ denotes the k-th visit of the descendant X_i. The function $MAPVS$ maps nodes of EDP to nodes of VS:

$$MAPVS(X_i \cdot a) := \begin{cases} X_i \cdot a & \text{if } X_i \cdot a \in AF_p \\ v_{k,i} & \text{if } X_i = X, \quad X_i \cdot a \in (A_{X,m} \wedge AC_p), \\ & k = (f_X - m + 1) \textbf{ div } 2,\ k > 0 \\ \text{undefined} & \text{if } X_i = X, \quad X_i \cdot a \in (A_{X,m} \wedge AC_p), \\ & 0 = (f_X - m + 1) \textbf{ div } 2. \end{cases}$$

$$VS_p = \{(MAPVS(X_i \cdot a), MAPVS(X_j \cdot b)) | (X_i \cdot a, X_j \cdot b) \in EDP_p\} \vee$$
$$\{\text{arbitrary arcs such that } VS_p \text{ is linearly ordered and}$$
$$v_{k,0},\ k = nv_X,\ X = X_0 \text{ is the "largest" element}\}.$$

Note that attributes evaluated before the context p is entered the first time (the "smallest" inherited attributes of X_0 in the TC-case and the "smallest" synthesized attributes of all X_i, $i > 0$ in the BU-case) are not represented by a visit. Usually all attributes of terminal symbols are synthesized and implicitly defined by the symbol text. Thus terminal nodes need not be visited.

The TD-case is similar to the TC-case. Additionally for each VS_p it must be assured that the first visits of the descendants are ordered from left to right:

(a) $0 < i < j \leqq np$ implies $(v_{1,i}, v_{1,j}) \in VS_p^+$,

(b) $0 < i \leqq np$ implies $(v_{1,i}, v_{1,0}) \in VS_p^+$.

The visit-sequences are computed as described for the TC-case. The freedom of the arbitrary linearisation is used in order to achieve the above condition. In general that is not possible for all VS_p. Thus we consider all symbols which would be visited "too late" for the first time in some context p:

$$LATE = \{X | \text{there is a } p \text{ such that } X = X_i,\ i > 0, \text{ and } v_{1,i} \text{ occurrs}$$
$$\text{"too late" in } VS_p \text{ according to (a) or (b)}\}.$$

The set $LATE$ contains all symbols X, for which the interface, i.e. the disjoint partitions of the attributes, must be changed such that the visit-sequences X occurrs in fit to conditions (a) and (b). Thus we add an empty set $A_{X,k}$, $k = m_X + 1$, to the disjoint partitions of each $X \in LATE$, if m_X is even. (If m_X is odd, the first visit of X does not depend on attributes and can be placed at an appropriate position in any visit-sequence.) Then the visit-sequences are recomputed using the updated interfaces. The added sets $A_{X,k}$ will yield an additional visit of X for syntactic purpose only. It can be placed such that (a) and (b) hold, because it does not depend on any attributes.

In general some recomputed visit-sequences VS_q, for $q = Y:w$ and $Y \in LATE$ violate condition (b), because the interface for Y changed. Therefore we iterate the computation of a new set $LATE$ and new visit-sequences until the conditions hold for all VS_q. The iteration terminates because the sets $LATE$ are disjoint for all iteration steps, and V is finite.

The evaluation of the semantic condition SC_p can be inserted in VS_p at any place after the evaluation of the attributes it depends on.

The visit-sequences for our example (for the TC-case) can be computed as follows:

$VS_1 = primary.access,\ v_{1,primary},\ primary.postmode,\ v_{2,primary},\ v_{1,ancestor}$

$VS_2 = declaration.access,\ v_{1,declaration},\ assignment.access,\ v_{1,assignment},$
$primary.primode,\ v_{1,ancestor},\ primary.evaluable,\ primary.value,$
$assignment.postmode,\ v_{2,assignment},\ v_{2,ancestor}$

$VS_3 = condition,\ primary.primode,\ v_{1,ancestor},\ primary.evaluable,\ primary.value,$
$v_{2,ancestor}$

$VS_4 = VS_5 =$
$primary.primode,\ v_{1,ancestor},\ primary.evaluable,\ primary.value,\ v_{2,ancestor}$

$VS_6 = expression.access,\ v_{1,expression},\ assignment.primode,\ v_{1,ancestor},$
$expression.postmode,\ condition,\ v_{2,expression},\ v_{2,ancestor}$

$VS_7 = expression_2.access,\ v_{1,expression_2},\ primary.access,\ v_{1,primary},$
$expression_1.primode,\ expression_2.postmode,\ v_{2,expression_2},$
$primary.postmode,\ v_{2,primary},\ v_{1,ancestor},\ expression_1.evaluable,$
$expression_1.value,\ v_{2,ancestor}$

$VS_8 = primary.access,\ v_{1,primary},\ expression.primode,\ v_{1,ancestor},\ primary.postmode,$
$v_{2,primary},\ expression.evaluable,\ expression.value,\ v_{2,ancestor}$

$VS_9 = expression.access,\ v_{1,expression},\ expression.postmode,\ v_{2,expression},$
$declaration.descr,\ v_{1,ancestor}$

6. Implementation of Visit-Sequences

The compiler phase for semantic analysis traverses through the structure tree and computes the attribute values. The control flow of this algorithm is exactly given by the visit-sequences. They describe which tree nodes are to be visited and which semantic functions are to be called. The visit-sequences can be translated into recursive procedures or coroutines or into the transition table of an automaton. The semantic functions (translated into the implementation language of the compiler) complete the compiler phase. (The situation is comparable to LL-grammars, which can be parsed by recursive procedures or table driven parsers, both augmented by actions appended to production rules.) In general a table driven algorithm is the most efficient implementation. An implementation using recursive procedures or coroutines is well suited to extend a recursive descent parsing algorithm.

In this section we show the principles of four implementation techniques. We assume that the visit-sequences are constructed such that attribute evaluation starts when tree construction is completed. It will be obvious how the implemen-

tations must be modified if the attributes are evaluated interleaved with tree construction.

6.1. Implementation Using Coroutines

The implementation technique given here uses SIMULA-classes as coroutines. It can be easily transferred to other programming languages providing comparable control structures. We assume that each node of the structure tree is represented by an instance of a class. The type of the class is determined by the corresponding syntactic rule p. The tree structure is represented by references from a node to its descendants.

For each rule $p = X_0 : X_1 \ldots X_n$ a class definition is constructed. For rule p_2 of our example it reads as follows:

```
node class p−2;
begin ref (node) declaration, assignment;
      attributes access, primode, postmode, evaluable, value;
   (* syntactic part:
      the subtress for declaration and assignment are constructed *)
      detach;
   (* semantic part: *)
      declaration.access;
      call (declaration);
      assignment.access:=...;
      call (assignment);
      primode:=...;
      detach;
      evaluable:=...;
      value:=...;
      assignment.postmode:=...;
      call (assignment);
      detach
end
```

The semantic part is a straightforward implementation of the visit-sequence VS_p: An ancestor visit is translated into a **detach**-statement. A visit of descendant X_i is translated into a coroutine call **call**(X_i). Each attribute $X_i \cdot a$ in the visit-sequence is translated into a call of a semantic function in SF_p defining its value.

6.2. Implementation Using Recursive Procedures

We assume that the nodes of the structure tree are represented by data structures with components for the attributes, references to the descendants, and a component indicating the derivation rule applied to the node. For each rule $p = X_0 : X_1 \ldots X_n$ the visit-sequence VS_p is split into m parts each terminated by an ancestor visit:

$$VS_p = VS_{p,1}, VS_{p,2} \ldots VS_{p,m}.$$

For each $VS_{p,i}$ a procedure p^i is constructed. For rule p_2 of our example it reads as follows:

```
procedure p_2^1 (ref node primary)
begin
    primary.declaration.access:=...;
    p_9 (primary.declaration);
    primary.assignment.access:=...;
    p_6^1 (primary.assignment);
    primary.primode:=...;
end;

procedure p_2^2 (ref node primary)
begin
    primary.evaluable:=...;
    primary.value:=...;
    primary.assignment.postmode:=...;
    p_6^2 (primary.assignment);
end
```

The body of the procedure is a straightforward translation of the i-th part of the visit sequence $VS_{p,i}$: Each $X_i \cdot a$ is translated into an appropriate call of a semantic function, and each descendant visit $v_{k,i}$ into a call $q^k(X_i)$. If there is more than one rule q for the i-th descendant X_i the r-th visit $v_{r,i}$ is translated into a case statement, which determines the procedure to be called by inspecting the *rule_indicator* of X_i:

```
case X_i · rule_indicator of
    q_1: q_1^r (X_i);
      ⋮      ⋮
    q_n: q_n^r (X_i)
esac
```

The ancestor visit at the end of $VS_{p,i}$ is implicitly implemented by the return from the procedure at the end of the body.

6.3. *Implementation by a Stack Automaton*

This implementation technique can easily be deduced from the previous one using recursive procedures: The structure tree is represented as in 6.2. All parts of all visit-sequences $VS_{p,r}$ as defined above are collected into a transition table. It is comparable to the program text of the procedure bodies in 6.2. A stack is maintained containing pairs of a reference to a treenode and the next position in the table to be encountered. It is an explicit implementation of the runtime stack needed for the procedures of 6.2, which contains a return address and the node reference for each procedure incarnation. An additional function *MAP-DOWN* is provided mapping the visit number r of a descendant visit $v_{r,i}$ and the rule indicator p of the visited node into the first table entry of $VS_{p,r}$ (comparable to the case statements of 6.2). The control loop of the automaton then has the following structure:

```
push (root, MAPDOWN(1,root.rule_indicator))
repeat
   case stack_top.table_entry of
   X_i · a : call semantic function
             for evaluation of X_i · a;
             increment(stack_top.table_entry);
   v_r,i   : (* descendant visit *)
   i>0       increment(stack_top.table_entry);
             push (stack_top.X_i,
                   MAPDOWN(r,stack_top.X_i.rule_indicator));
   v_r,0   : (* ancestor visit *)
             pop
   esac
until stack_is_empty
```

6.4. Implementation by a Finite Automaton

This implementation technique avoids the (implicit or explicit) stack of the previously described techniques. We assume that the tree is linked both upwards and downwards. Each node contains an indicator of the applied rule and the number s, when the node is the s-th descendant of its ancestor. Two functions *MAPDOWN* and *MAPUP* are provided. *MAPDOWN* is defined as in 6.3. $MAPUP(r, s, p)$ maps the number r of an ancestor visit, the descendant number s of the currently visited node and the rule indicator p of its ancestor into that element of VS_p, which is preceeded by $v_{r,s}$. The control loop of the automaton then has the following structure:

```
current_node:=root;
state:=MAPDOWN(1,root.rule_indicator);
repeat
   case state of
   X_i · a    : begin
                  call semantic function for
                    evaluation of X_i · a;
                  increment(state);
                end;
   v_r,i      : begin (* descendant visit *)
   i>0             state:=MAPDOWN(r,X_i.rule_indicator);
                   current_node:=X_i
                end;
   v_r,0      : begin (* ancestor visit *)
                   state:=
                     MAPUP(r,current_node.s,
                           current_node.ancestor.rule_indicator);
                   current_node:=current_node.ancestor
                end
   esac;
until current_node=root and state=v_1,0
```

7. An Algorithm for Computing Visit-Sequences

This section discusses an abstract algorithm which checks whether a given AG is ordered, and computes the visit sequences for it. We show that in the worst case the complexity of the algorithm is bound by the product of the 4th power of the maximum number of attributes associated to a symbol, the 3rd power of the maximum length of a syntactic rule, and the length of the underlying context-free grammar. This is an important result compared to the intrinsic exponential complexity of well-definedness of AGs [4]. For most programming languages the number of attributes is rather small. Thus the complexity of the algorithm is suitable small for practical applications. The complexity of the algorithm is expressed by the size of the input grammar. The parameters used are:

$|P|$ the number of syntactic rules;

$|R|$ the maximum number of symbols in a single syntactic rule;

$|X|$ the maximum number of attributes of a single symbol;

$|V|$ the number of symbols, bound by $|V| \leqq |G|$.

These parameters are used to compute the following expressions:

$|G| = |P|\,|R|$ the length of the grammar;

$|D| = |R|\,|X|$ the maximum number of nodes of a dependency graph for a rule;

We shall describe the algorithm in form of a program for an abstract machine, which is defined by the following data structures and functions.

A dependency relation D over n attributes is considered as a graph with n nodes. The arcs $(a, b) \in D$ are represented by a vector of n sets over n nodes. D is initially empty. Primitive operations on D are $(a, b) \in D$, $(a, b) \notin D$ and *addarc* $(D, (a, b))$ which adds the arc (a, b) to D. The costs of these operations are O(1).

Each rule $p = X_0 : X_1 \ldots X_{n_p}$ is represented by a vector X of symbol occurrences, a dependency relations TDP_p over the attribute occurrences $X_i \cdot a$, and a graph VS_p representing the visit-sequence.

For each symbol $X \in V$ there is a linked list of its occurrences, and a dependency graph TDS_X over the attributes A_X. The arcs of this graph are marked if they are already induced at all occurrences of X. A vector $PART$ maps the attributes $X \cdot a$ to the index k of $A_{X,k}$.

The following functions are used for updating dependency relations.

addarctrans$(D, (a, b))$:
It is assumed that D is a transitively closed, non-reflexive relation. The arc (a, b) is added to D. All arcs needed to close D transitively again are added to D, too. The costs of this operation are $O(n^2)$, if D is a graph over n nodes.

addarcinduce$(D, (a, b))$:
This function is applied to relations TDP_p associated to rules p. It updates the transitive closure D as *addarctrans* does. Additionally, for each new arc $(X_i \cdot a, X_i \cdot b)$ an arc $(X \cdot a, X \cdot b)$, which is not marked, is added to TDS_X, if $(X \cdot a, X \cdot b) \notin TDS_X$, $X = X_i$. The costs of this function are the same as the costs of *addarctrans*, because the additional action is only applied to a subset of the totally inspected arcs.

The main idea of the algorithm is the following: Instead of the dependency relations DP_p, IDP_p, EDP_p. their transitive closures are computed. The recursive

definitions of Sect. 3 are transformed into iterative algorithms. The algorithm consists of 5 steps: Steps 1 and 2 compute the relations IDP_p^+ and DS_X^+. Step 3 computes the disjoint partitions. In step 4 EDP^+ is computed and it is decided whether the AG is ordered. Step 5 computes the visit-sequences.

Step 1. Computation of DP^+

Input: The semantic functions SF.

Output: Dependency graphs $TDP_p = DP_p^+$, and updated graphs TDS_X.

Method: For each dependency established by a semantic function $f \in SF_p$ an arc is added to the transitively closed TDP_p by calling *addarcinduce*.

for each $f \in SF_p$ defining $X_j \cdot b$
loop for each argument $X_i \cdot a$ of f
loop if $(X_i \cdot a, X_j \cdot b) \notin TDP_p$
then *addarcinduce* $(TDP_p, (X_i \cdot a, X_j \cdot b))$ **fi**
repeat
repeat

The graphs TDS_X contain the arcs to be induced due to direct dependencies; they are not marked.

Complexity: $O(|P|\,|D|^4)$

The upper bound for the number of arcs in TDP_p is $|D|^2$; it is multiplied with $|D|^2$, the complexity of *addarcinduce*.

Step 2. Computation of IDP

Input: Acyclic dependency graphs $TDP_p = DP_p^+$, graphs TDS_X containing arcs to be induced due to direct dependencies.

Output: Dependency graphs $TDP_p = IDP_p^+$ containing all induced arcs according to Definition 2, and the graphs $TDS_X = IDS_X^+$.

Method: Each arc in TDS which is not marked is induced at each occurrence of the symbol. If new arcs to be induced are found they are added to TDS by *addarcinduce*.

while there is an arc $(X \cdot a, X \cdot b)$ in IDS which is not marked
loop *mark* $(X \cdot a, X \cdot b)$;
for each occurrence X_i of X in any rule p
loop if $(X_i \cdot a, X_i \cdot b) \notin TDP_p$
then *addarcinduce* $(TDP_p, (X_i \cdot a, X_i \cdot b))$
fi
repeat
repeat

The invariant condition for the **while**-loop is: If a TDP_p contains an arc $(X_i \cdot a, X_i \cdot b)$, and if $(X \cdot a, X \cdot b)$ is marked, then $(X \cdot a, X \cdot b)$ is already induced in all TDP_q. Thus $TDP_p = IDP_p^+$ when finally all arcs in TDS are marked, and $TDS_X = IDS_X^+$.

Complexity: $O(|G|\,|X|^2\,|D|^2)$

There are at most $|X|^2$ arcs in each TDS_X. The **for**-loop is executed once for each arc and each occurrence of the symbol. There are $|G|$ symbol occurrences where arcs may be induced. The marking of arcs in *TDS* ensures that no arc is induced more than once.

Step 3. Computation of the disjoint partitions of A_X

Input: Acyclic dependency graph $TDS = IDS^+$.

Output: A vector *PART* mapping $a \in A_X$ to k if $X \cdot a \in A_{X,k}$ according to Definition 4.

Method: Starting with $k=1$ for each attribute it is checked, whether it can be included into $A_{X,k}$. When no more attributes can be added to $A_{X,k}$, k is incremented.

```
for each X∈V
loop k:=1; not_assigned:=A_X;
    while not_assigned≠∅
    loop found_one:=false;
    n:  for each X·a∈(not_assigned ∧
                        if odd k
                        then AS_X else AI_X fi)
        loop condition_holds:=true;
        m:  for each X·b∈not_assigned
            loop if (X·a, X·b)∈TDS_X
                 then condition_holds:=false;
                      exitloop m
                 fi
            repeat
            if condition_holds
            then PART[X·a]:=k;
                 not_assigned:=not_assigned\{X·a};
                 found_one:=true;
                 exitloop n
            fi
        repeat;
        if not found_one and not_assigned≠∅ then k:=k+1 fi
    repeat;
    m_X:=k;
    f_X:=if odd k then k+1 else k fi
repeat
```

Complexity: $O(|V|\,|X|^3)$

In the worst case each $A_{X,k}$, $k>1$ contains one attribute, and $A_{X,1}$ is empty. Thus the **while**-loop is repeated at most $2|X|+1$ times. ($|X|$ steps succeed in assigning $X \cdot a$ to a subset, and $|X|+1$ steps fail.) In the loop n success or failure is determined after at most $|X|$ steps. The loop n checks the condition for not more than $|X|$ arcs.

Step 4. Computation of EDP

Input: Graphs TDP_p and the vector $PART$ describing the disjoint partitions of the A_X.

Output: Graphs $TDP_p = EDP_p^+$

Method: Arcs are added to TDP according to the relation given by $PART$.

```
for each p∈P
loop for i:=0 to np
    loop X:=X_i;
        for each X·a
        loop for each X·b
            loop if PART[X·a]>PART[X·b]
                then addarctrans(TDP_p,(X_i·a, X_i·b))
                fi
            repeat
        repeat
    repeat
repeat
```

Now $TDP_p = EDP_p^+$ holds for each $p \in P$. If each TDP_p is acyclic the AG is an OAG.

Complexity: $O(|P|\,|R|\,|X|^2\,|D|^2)$

The complexity is the product of the upper bounds for the number of steps each nested loop is executed.

Step 5. Construction of the visit-sequences

Input: Acyclic graphs TDP_p as computed in step 4, and the vector $PART$ as computed in step 3.

Output: A visit-sequence for each $p \in P$.

Method: Starting from an empty graph VS_p over AV_p arcs are added according to Definition 9.

```
for each p∈P
loop for each (X_i·a, X_j·b)∈TDP_p
    loop Xi:=X_i; Xj:=X_j;
        mi:=PART[Xi·a]; mj:=PART[Xj·b];
        ki:=(f_Xi − mi + 1) div 2;
        kj:=(f_Xj − mj + 1) div 2;
        if ki>0 and kj>0
        then addarctrans(VS_p,
                         (if X_i·a∈AF_p
                          then X_i·a else v_ki,i fi,
                          if X_j·b∈AF_p
                          then X_j·b else v_kj,j fi)
                         )
        fi
    repeat;
```

```
(* add arcs until VS_p is linearly ordered *)
  for each g∈AV_p
  loop for each h∈AV_p
       loop if (g, h), (h, g)∉VS_p
            then
                 if g=v_{k,0} and k=nv_X, X=X_0
                 then addarctrans(h, g)
                 else addarctrans(g, h)
                 fi
            fi
       repeat
  repeat
repeat
```

Complexity: $O(|P|\,|D|^4)$

For the two loops on the second level $|D|^2$ is an upper bound for the arcs added by *addarctrans*.
The complexity of the whole algorithm is given by the following sum:

Step 1: $O(|P|\,|D|^4)+$
Step 2: $O(|G|\,|X|^2\,|D|^2)+$
Step 3: $O(|V|\,|X|^3)+$
Step 4: $O(|P|\,|R|\,|X|^2\,|D|^2)+$
Step 5: $O(|P|\,|D|^4)$

Since $|G|\geqq|V|$ step 3 does not contribute to the total complexity. The items for step 2 and 4 are equal, because $|G|=|P|\,|R|$. Hence the formula reduces to

$$O(|P|\,|D|^4+|G|\,|X|^2\,|D|^2)=$$
$$O(|X|^4\,|G|(|R|^3+|R|^2))=$$
$$O(|X|^4\,|G|\,|R|^3)$$

The length of the context-free grammar contributes only linearly to the total complexity. The maximum length $|R|$ of the syntactic rules is usually bound by a small constant. So the most significant item is $|X|^4$ the maximum number of attributes associated to a single symbol. For AGs defining programming languages this number will be rather small, too. Thus the computation of transitive closures can be implemented by more powerful set operations, which will reduce the complexity to $O(|X|^3\,|G|\,|R|^2)$. An implementation of this algorithm on a SIEMENS 7760 needed about 60 seconds for the analysis of an AG for the rather large language PEARL ($|X|=25$, $|G|=849$, $|R|=8$).

8. Conclusion

In this paper a new class of attributed grammars is introduced: ordered attributed grammars. The expressive power of ordered attributed grammars is larger than that of any other subclass of attributed grammars based on a fixed evaluation strategy. It is sufficiently large for the definition of programming languages.

The class is well suited for both programming language definition and automatic compiler generation. The definition is based on the natural concept of linearly ordered dependencies between the attributes of syntactic units. Therefore a language designer can define the context dependent properties of the language statically, without considering any (predefined) evaluation strategy. The evaluation algorithm can be implemented in a simple and efficient way. It is parameterized by local evaluation rules, which are computed from the given attributed grammar. This principle is comparable to well known and widely used techniques for parser generation (e.g. LALR(1)-technique). In [7] it is shown how this method can be integrated in a compiler generating system based on attributed grammars.

Acknowledgements. I am indebted to G. Goos and my colleagues for many stimulating discussions, to R. Loos for his valuable remarks on the complexity analysis, and to W.M. Waite for his comments on the manuscript.

References

1. Bochmann, G.V.: Semantic evaluation from left to right. CACM **19**, 55–62 (1976)
2. Ganzinger, H., Ripken, K., Wilhelm, R.: MUG1 – an incremental compiler-compiler. In: Proc. of ACM 1976 Ann. Conf., pp. 535–540, 1976
3. Giegerich, R., Wilhelm, R.: Implementierbarkeit attributierter Grammatiken. In: Informatik Fachberichte 10, pp. 17–36, 1977
4. Jazayeri, M., Ogden, W.F., Rounds, W.C.: The intrinsically exponential complexity of the circularity problem for attributed grammars. CACM **18**, 679–706 (1975)
5. Jazayeri, M., Walter, K.G.: Alternating semantic evaluator. In: Proc. of ACM 1975 Ann. Conf., pp. 230–234, 1975
6. Kastens, U.: Ein Übersetzer-erzeugendes System auf der Basis attributierter Grammatiken. Fak. f. Informatik, Universität Karlsruhe, Interner Bericht 10, 1976
7. Kastens, U.: ALADIN – Eine Definitionssprache für attributierte Grammatiken. Fak. f. Informatik, Universität Karlsruhe, Interner Bericht 7, 1979
8. Kennedy, K., Warren, S.K.: Automatic generation of efficient evaluators for attribute grammars. Conference record of the 3rd ACM Symp. on Principles of programming languages, pp. 32–49, 1976
9. Knuth, D.E.: Semantics of context-free languages. Math. Syst. Theory **2**, 127–145 (1968)
10. Knuth, D.E.: Semantics of context-free languages: correction. Math. Syst. Theory **5**, 95–96 (1971)
11. Lecarme, O., Bochmann, G.: A (truly) usable and portable compiler writing system. In: Information Processing 74, 1974
12. Lewi, J., de Vlaminck, K., Huens, J., Huybrechts, M.: Project LILA: The ELL(1) generator of LILA, an introduction. In: International Comp. Symp. 1977, pp. 237–251, North-Holland Publ. 1977
13. Lewis, P.M., Rosenkrantz, D.J., Stearns, R.E.: Attributed translations. Journal of Computer and System Science 9, 279–307 (1974)
14. Lorho, B.: Semantic attributes processing in the system DELTA. In: Methods of Algorithmic language Implementation, pp. 21–40, Springer Verlag 1977
15. Räihä, K.-J.: On attribute grammars and their use in a compiler writing system. Rep. A-1977-4, University of Helsinki, Dep. of Comp. Sc. 1977
16. Schulz, W.A.: Semantic analysis and target language synthesis in a translator, University of Colorado, Dep. of Comp. Sc., PhD thesis 1976
17. Wilhelm, R.: Baumtransformatoren. Ein Vergleich mit Baumtransduktoren und Aspekte der Implementierung. TU München, Bericht, pp. 77–13, 1977

Received September 21, 1978 / July 20, 1979 / October 5, 1979