A survey on attribute grammars.Part I main results on attribute grammars

Pierre Deransart, Martin Jourdan, Bernard Lorho

INRIA
CENTRE DE ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France
Tél. : (1) 39 63 55 11

Rapports de Recherche

N° 485

A SURVEY ON ATTRIBUTE GRAMMARS

PART I
MAIN RESULTS ON ATTRIBUTE GRAMMARS

Pierre DERANSART
Martin JOURDAN
Bernard LORHO

Janvier 1986

Pierre DERANSART, Martin JOURDAN, Bernard LORHO [1]

INRIA-ROCQUENCOURT
BP 105
78153 LE CHESNAY Cedex France

A SURVEY ON ATTRIBUTE GRAMMARS

Part I : Main Results on Attribute Grammars

Abstract: This paper is the first part of a Survey on Attribute Grammars consisting of three parts :

- Main results on Attribute Grammars
- Review of Existing Systems
- Classified Bibliography.

Key Words: Attribute Grammars, Evaluation, Semantics

Résumé: Cet article est la première partie d'une étude sur les Grammaires Attribuées qui comporte trois parties:

- Principaux Résultats sur les Grammaires Attribuées
- Revue des Systèmes Existants
- Bibliographie Classée

Mots-clé: Grammaires Attribuées, Evaluation, Sémantique

[1] Also at Université d'Orléans, Laboratoire d'Informatique, 45046 ORLEANS Cedex France

0. INTRODUCTION

This report gives an introduction to attribute grammars and a synthesis in this topic.

We shall show that the application domains of attribute grammars are growing. But the question of automatically constructing efficient evaluators remains the most important one, especially if we consider that attribute grammars may not only be a theoretical but also a practical model for programming. This is why we shall devote special interest to the complexity analysis of evaluators for special subclasses of attribute grammars.

We shall use a characterization introduced by Engelfriet [Eng84] and Barbar [Bar82] based on partial orders between attributes. The classification of attribute grammars obtained in this way seems to be the most valuable one and it relies on simple characterizations. This simplicity widely disappears when decomposing attribute grammars evaluable in phases: we shall then use Engelfriet and Filé approach [EF80a], though a characterization by orders is still possible [RS77]. The advantage of such a representation is that the results do not depend on a special application, i.e. if they were designed for the construction of efficient attribute evaluators, their extension to other applications is direct within this formalism.

Sections 1 to 4 present a classification of attribute grammars and the main results about the complexity of membership test and of evaluators built up for each category.

Section 5 deals with the question of the space needed during the evaluation process: we first give standards to be able to compare different systems, and then the most important results.

Section 6 is an introduction to the characterization of attribute grammars from a more theoretical point of view. Two approaches are presented using tree transductions and program schemes. These characterizations lead to results about complexity of attribute computation, about programs equivalence, and comparison of expression power.

Such a study allows to situate the attribute grammars approach among the numerous tools involved in the programming activity, but also helps in evaluating the ability of existing systems to solve some applications. The characterization in terms of program schemes is essential as long as the theoretical results obtained in that field may be *de facto* applicable to schemes defining attribute grammars and conversely.

1. NOTATIONS

We introduce here the usual definition of attribute grammars [Knu68] and present their semantics in an informal manner.

Let G = {N,T,P,Z} be a context free grammar. Each production rule p in P can be written:

$$p : X_0 \rightarrow X_1 X_2 \ldots X_{n_p}$$

with $X_i \in N$ if we *forget* about terminal symbols.

An attribute grammar (AG) consists of an *attribute system* associated to the underlying context free grammar G. For each non-terminal symbol $X \in N$, we associate two finite sets *Inh* (X) and *Syn* (X) of symbols, the Inherited and Synthesized attributes.

These sets verify:

for all X,Y $\in$ N, *Inh* (X) $\cap$ *Syn* (Y) = ϕ

We note *Attr* (X) = *Inh* (X) $\cup$ *Syn* (X), *Inh* = $\cup_{X \in N}$ *Inh* (X), *Syn* = $\cup_{X \in N}$ *Syn* (X), *Attr* = *Inh* $\cup$ *Syn*.

An attribute a associated in position i with the symbol Y is an *attribute occurrence* and is noted a(i)[2].

For each production p we give a set of semantic rules defining the computation of the elements of *Syn* (X_0) and *Inh* (X_j) for $1 \leq j \leq n_p$ in terms of the elements of *Attr* (X_i) for $0 \leq i \leq n_p$:

$$a(i) = f_{p,a,i}(a_1(i_1), \ldots, a_k(i_k))$$

for i = 0 and a $\in$ *Syn* (X_0), $1 \leq i \leq n_p$ and a $\in$ *Inh* (X_i)

We note W(p) the set of all the *attribute occurrences* in production p :

$$W(p) = \{ a(i) \mid a \in Attr(X_i) , 0 \leq i \leq n_p \}$$

The semantic rules induce on W(p) an order of computation called *relation of local dependencies* D(p) so defined:

b(j) D(p) a(i) <==> b(j) appears in a(i) definition

The *graph of local dependencies* in production p is the graph of relation D(p). When no confusion is possible between the notions of relation and graph, we shall use the same notation for both.

We note R^+ the transitive closure of relation R (or graph R). If the graph R is cycle free, R is a partial order.

---

[2] We note $a^Y i$ instead of a(i) in the examples in order to emphasize the non-terminal symbol Y associated to a, and to make reading easier.

An attribute grammar is in *Normal Form* or *normalized* if and only if the semantic rules use

$$Inh\ (X_0) \text{ and } Syn\ (X_j) \text{ for } 1 \leq j \leq n_p$$

instead of

$$Attr\ (X_i) \text{ for } 0 \leq i \leq n_p$$

i.e. they use only attributes defined *outside* production p.

Every attribute grammar can be put in Normal Form by means of a simple transformation of the semantic definitions if the relations D(p) are cycle free.

In the sequel, we shall consider only normalized attribute grammars even if this is not required for all the definitions and results except when especially stated. But the normalization makes examples and proofs easier. Some characterizations are more difficult to define without this hypothesis. In fact, if the attribute grammar is normalized, D(p) is cycle free and any path has a length of 1.

Let t be a derivation tree of G. We note $R_d(t)$ the *graph of compound dependencies* obtained by pasting together the graphs D(p) corresponding to each rule belonging to the tree. This graph may contain cycles: in that case, it will be called *circular*.

If we consider a tree t with root X_0, the relation $R_d^+(t)$ induces on $Attr\ (X_0)$ a relation noted $sd_t(X_0)$ (for synthesized dependencies induced by the tree t derived from X_0). This relation is a partial order and its arcs come from $Inh\ (X_0)$ to $Syn\ (X_0)$ if the attribute grammar is in Normal Form.

If u is a node of t, we call *sort* (u) the non-terminal symbol root of the subtree hanging from u and *label* (u) the production rule p applied to the root of that subtree. We shall generalize here the notation taken for a rule by putting together in X_u the name of the non-terminal (X_u) and its position (u) in the tree t. Nodes can be numbered "à la Dewey" in a way such that if a production p

$$p : X_0 \rightarrow X_1\ X_2 \ldots X_{n_p}$$

is instantiated at node u (*label* (u) = p), X_0 takes the position u (X_u) and its sons positions ui $(X_{ui}, 1 \leq i \leq n_p)$.

If we consider a node u the sort of which is X ($u \in$ *node* (t), the set of nodes of t), and if we remove the entire subtree hanging from u (except u itself), the relation $R_d(t)$ induces by transitive closure a partial order on *Attr* (X), noted $id_t(X_u)$ (for inherited dependencies induced by the context of X_u).

Giving a rule p instantiated at the node u of a tree t, the relation :

$$id_t(X_u) \cup sd_t(X_{u1}) \cup ... \cup sd_t(X_{un}) \cup D_u(p)$$

defines the dependency order[3] to be obeyed during computation of the attributes W(p), where $D_u(p)$ and $W_u(p)$ denotes the instances of D(p) and W(p) at node u for rule p.

It is worth noticing that, given a tree t, the relations *sd* can be computed during one bottom-up visit of t, but the computation of the relations *id* in one top-down visit requires the knowledge of *sd*[4].

We note IO(X) (Input/Output relation) (resp. OI(X) (Output/Input relation)) the union of all the relations sd(X) (resp. id(X)) on any tree derived from X (for any context of X).

Let :

$IO(X) = \cup_t sd_t(X)$, for t the root of which is X

$OI(X) = \cup_t id_t(X)$, for t including a subtree the root of which is X.

All the trees we consider here are *terminal* trees the leaves of which are terminal symbols.

We note W(t) the set of all attribute occurrences a(u) in the tree t. W(t) may be considered as the set of the *variables* to which the evaluation process must assign a value.

In the sequel, many relations noted R(X) will be defined on *Attr* (X). They will be partial orders. If an explicit total order is required, it will be noted T(X). In such a case, the dependency relation built up in t can be widened to a total order noted T(t). It is obvious that R(t) (or T(t)) figures out the order to be followed for computing the values of the variables in W(t).

1.1. Example

The following example exhibits all previous definitions through the conversion of a bit string into its decimal value (the original example [Knu68])

---

[3] That order is not necessarily a partial order as cycles may occur.

[4] This remark shows that the notions of *inherited* and *synthesized* are not exactly symmetrical as the relation id depends on sd but not the reverse.

G = {N,T,P,Z}

N = {Z, BIN, BIT}
T = {0, 1, .}

P = {Z ::= BIN . BIN
BIN ::= BIN BIT
BIN ::= empty
BIT ::= 0
BIT ::= 1}

Attr = {r,l,v} *Inh* = {r} *Syn* = {l,v}

where

v^X is the decimal value of the string derived from X,

l^X is the length of the string derived from X,

r^X is the rank of the rightmost bit of the string derived from X.

Attr (Z) = {v} *Attr* (BIN) = {r,l,v} *Attr* (BIT) = {r,v}

Attribute definitions

(The decimal value of a bit string is the sum of powers of 2 corresponding to rank of bits 1)

p_1: Z ::= BIN . BIN

$v^Z = v^{BIN}1 + v^{BIN}2$

$r^{BIN}1 = 0$

$r^{BIN}2 = -\ l^{BIN}2$

p_2: BIN ::= BIN BIT

$v^{BIN}0 = v^{BIN}1 + v^{BIT}$

$l^{BIN}0 = l^{BIN}1 + 1$

$r^{BIT} = r^{BIN}0$

$r^{BIN}1 = r^{BIN}0 + 1$

p_3: BIN ::= empty

$v^{BIN} = 0$

$l^{BIN} = 0$

p_4: BIT ::= 1

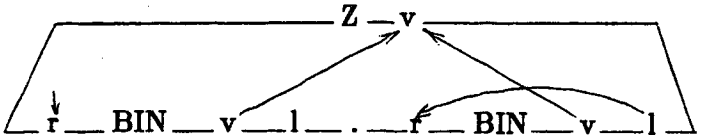
$v^{BIT} = 2^{r^{BIT}}$

p_5: BIT ::= 0

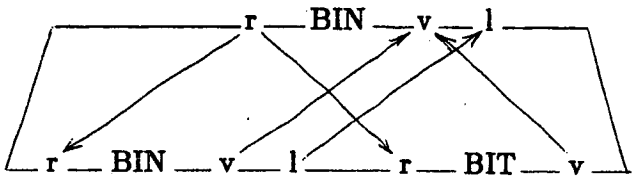
$v^{BIT} = 0$

Local dependency graphs:

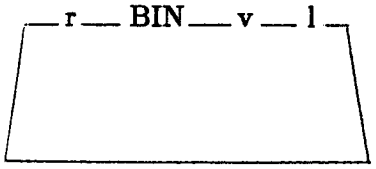
p_1:



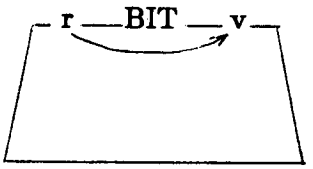
p_2:



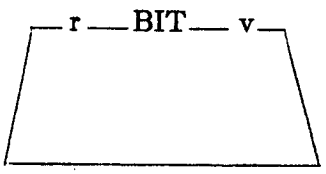
p_3:



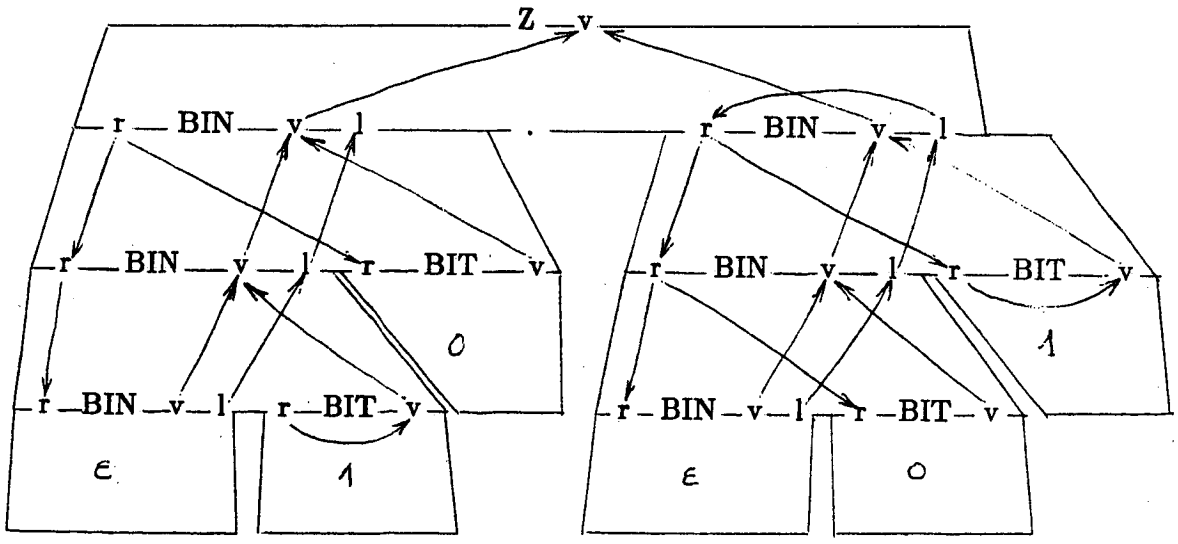
p_4:



p_5:



Compound dependency graph for the string 10.01



Relations IO(X), OI(X) for X ∈ { Z, BIN, BIT }

IO(Z) = v OI(Z) = v

IO(BIN) = r y l OI(BIN) = r v l

IO(BIT) = r y OI(BIT) = r v

2. ATTRIBUTE EVALUATION

Now, we are going to define the problem of attribute computation and, more precisely, the criteria for estimating the time efficiency of an evaluator.

The main problem to be solved as efficiently as possible is to compute the values of unknowns W(t) for a given tree t. This operation is called *tree decoration* in [Lor74] and corresponds to what is commonly called *attribute evaluation.* A first step towards the solution is reached when an *evaluation order* is found, i.e. a total order T(t) of unknowns W(t) compatible with R(t), i.e. $R(t) \subseteq T(t)$.

The first class of attribute grammars introduced in the literature by Knuth [Knu68] concerns AGs for which this order always exists for a given tree:

2.1. Definition:

An attribute grammar is *well-formed* or *non circular* if and only if, for every tree t, R(t) is cycle free.

From this definition comes the following property:

2.2. Property:

For every tree t, there exists an evaluation order if and only if the attribute grammar is well-formed.

The *well-formedness* property can be statically tested for any AG. It was proved [JOR75b, Jon80, Jaz81] that this test is intrinsically exponential in time and space. Fortunately, at least in the meta-compilation framework, this test can be done with an acceptable efficiency through numerous optimizations, without changing the worst case complexity [RS82a, DJL84]. In the sequel of this chapter, we shall consider only well-formed attribute grammars.

If the result of the test guarantees that an evaluation order can always be found, this order must be determined dynamically. Computation of semantic functions must be done according to this order.

In practice, the two steps are often done in parallel, but it is better to differentiate them:

> 1: *Plan building:* corresponds to the construction of an evaluation order T(t)[5]

[5] The notion of *plan* is only given here to help in fixing ideas. One can talk of *plan* only when the graph associated with R(t) relation is *explicitly* built, and this is usually

2: *Attribute computation:* corresponds to a call to the previously built evaluator in order to assign values to the variables in W(t).

An immediate consequence stems from the distinction we introduced: there are two ways of following a plan T(t): starting from minimal elements or from maximal ones in the relation. So we note:

$W_{in}(t)$ the minimal elements

$W_{out}(t)$ the maximal ones.

We immediately obtain two different classes of evaluators:

1. Evaluators that start by assigning values to variables in $W_{in}(t)$: we call them *bottom-up* or *data driven* evaluators.
2. Evaluators that first consider variables in $W_{out}(t)$ (or, at least, a subset of them): we call them *top-down* or *demand driven* evaluators.

An important advantage of top-down evaluators comes from the fact that it is possible to choose a subset of $W_{out}(t)$, let $W_{target}(t) \subseteq W_{out}(t)$, which is, in general, the set of all the synthesized attributes of the axiom.

In such an approach, it is possible to distinguish, among the variables, the *useful* ones, i.e. the variables actually involved in the computation of
$W_{target}(t)$ from variables in *dead-ends*. We shall note:

UsefulR $(t) \subseteq W(t)$ the set of variables on which variables in $W_{target}(t)$ depend in R(t).

Dead-endsR $(t) = W(t)$ - *UsefulR* (t).

We can say that an evaluator is *time-optimal* iff it computes only once *UsefulR* $_d(t)$ where $R_d(t)$ is the partial order relation obtained through local dependencies.

It is worth noticing that this criterion is extremely severe, but it induces the advantage to clearly fix ideas and helps in comparing existing systems. In fact, many authors use as a criterion, either the computation without any repetition of W(t) or of *UsefulR* (t) for a relation R larger than R_d, or a criterion of linearity in W(t) or *UsefulR* (t) but relaxing the non repetition constraint. We must be very careful in the criterion actually used.

---

not the case.

With the help of our criterion, we immediately observe that only top-down evaluators using relation $R_d(t)$ can be optimal in time, but we shall see that, in practice, for reasons connected to the exponential nature of the non-circularity test, except in an incremental context, $R_d(t)$ cannot be used. It turns out that, in practice, the advantage of the top-down approach is not essential over the bottom-up one, according to this criterion [Kav84]. If not essential, this advantage remains in the fact that a top-down evaluator can introduce dynamic optimizations by avoiding, for example, computation of attributes in useless branches of conditional statements. We call them *semantic dead-ends* as opposed to static dead-ends which are Dead-endsR(t); the former are determined by the control flow of semantic rules.

To have an idea of our way of understanding the time efficiency of an evaluator, we must keep in mind that the evaluator is part of a compiler and that its *practical efficiency* is fundamental for actual applications. If it is possible to know its theoretical complexity (for instance, linear in W(t) size), we may be not satisfied with that. If we consider that semantic functions are time consuming, it is necessary to know the linearity factor. If we consider (and it will be the case in the sequel) that each semantic function runs in a duration equal to one time unit, this factor is to be one.

We must not forget that an evaluation is composed of two parts (we considered until now only the second one, the evaluation phase), and that the first one is built, in fact, of two sub-steps:

1) building a plan,

2) optimizing the plan size.

If both sub-steps are done during evaluation, (they can be combined in some cases), their duration is to be added to the evaluation time. First subtask cannot be avoided. The second one is devoted to optimize the global size of the evaluation process, and this reduction plays practically an important role on the evaluation time. Some semantic functions, for instance definitions restricted to *identities* between variables, can be eliminated in the second substep, and decrease in an important manner the evaluation time.

Despite that, we shall examine the problem of plan size optimization and its consequences over evaluation in section 5. We conclude this part by a remark on plan construction time.

The most interesting solution consists in doing nothing during evaluation! Those systems that explicitly build $R_d(t)$ during evaluation are not efficient in practice (see for instance the system DELTA [Lor74]). The first idea that comes to mind is to avoid such a construction by statically finding (during evaluator construction) a total

order T over attribute occurrences, able to cover any relation R(t). If this is possible, we say that the attribute grammar is *l-ordered* and *this fact necessitates to introduce new dependencies and may oblige to compute UsefulT* (t) whose cardinal is larger than that of *UsefulR* $_d$(t).

This strategy illustrates the way new attribute grammars classes were introduced in order to improve the practical efficiency of evaluators. Theoretical and practical studies lead to define numerous categories (see the following section). Those categories are based on the plan structure, i.e. on some properties of order relations, and this fact induces the kind of evaluator that can be built for each of them. All known evaluators belong to one of these classes described in sections 3 and 4. Some categories do not correspond to any known evaluator and can be considered as the starting point of new ideas in the conception of yet more efficient evaluators.

2.3. Example (example 1.1 continued)

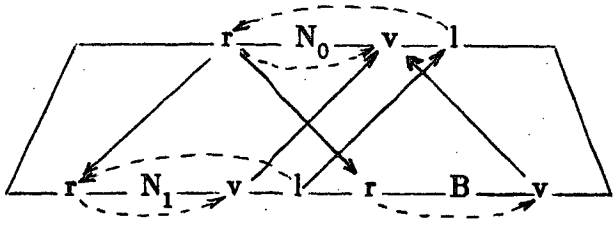
The attributes of a given non-terminal symbol can always be evaluated in the same order, for any tree :

$T(N)=IO(N) \cup OI(N)=$ 

$T(B)=IO(B) \cup OI(B)=$ 

For instance, in rule p_2, the following order can be obeyed for any context where this rule can be instantiated:

p_2:



Continuous arrows : D(p)

Dotted arrows : dependencies induced by the context = T(X)

A total order is the following one:

$$_1N_{1\,l}N_{0\,r}N_{0\,r}B_{\,v}B_{\,r}N_{1\,v}N_{1\,v}N_0$$

if we are interested in evaluating all the attributes in any tree t. Note that, for the string 10.01 given as an example, the value of l^N in the left subtree is not needed for computing v^Z, the only interesting value. There are also many other dead-ends.

We conclude this section by giving in the following figure useful criteria for estimating the practical efficiency (and, maybe, the theoretical efficiency) of an evaluator.

2.4. Criteria

Criteria for time efficiency of an evaluator (each semantic action is supposed to be done in one time unit, the attribute grammar being normalized) :

- difference from $UsefulR_{\,d}(t)$ (number of calls to semantic functions), i.e. number of attribute instances evaluated.
- time to build the plan.
- time to optimize the plan size (see section 5).

- effective global time and space of attribute evaluation process.

3. GENERAL CLASSIFICATION OF ATTRIBUTE GRAMMARS

In this section, we study different possible plans, the complexity of problems connected to their construction and the time and space efficiency of evaluators that can be built for some attribute grammars classes. But it is out of the scope of this paper to study in every detail each evaluator and the implementations they lead to. A more complete discussion about this question can be found in [Eng84].

3.1. Purely synthesized AGs

The second class of attribute grammar introduced in Knuth's original paper is that of purely *synthesized AGs*, or, in short, *pure-S-AGs*.

3.1.1. Definitions

An AG is *pure-S* iff $Inh = \phi$. In the same manner, we say that an AG is *purely inherited (pure-I)* iff $Syn = \phi$.

If symmetric definitions seem to characterize pure-S and pure-I attribute grammars, only the pure-S ones play a theoretical and practical role. In fact, pure-S have the same power as Turing machines (this result will be recalled in section 6), and the evaluation order is intrinsic to the AG: it does not need to be dynamically constructed. In fact, a time optimal system can be built that combines syntactic and semantic analysis (suboptimal bottom-up evaluator (linear in W(t)) or optimal top-down one that can be combined with a bottom-up or a top-down parsing method). We immediately see that such an evaluator can be optimized in space and time.

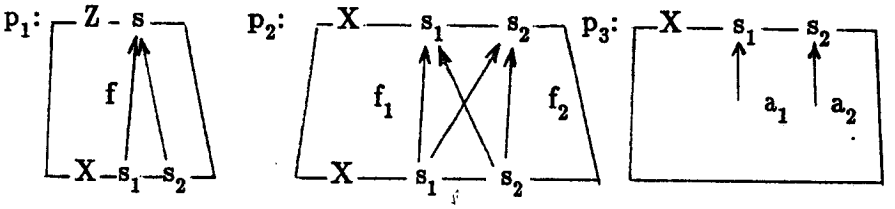
It is useful noticing that in a pure-S AG, every (synthesized) attribute can be considered as a recursive function defined over the tree t (they are primitive recursive schemes, see section 6). Therefore, it is easy to produce, from a pure-S AG, a top-down evaluator defined by recursive functions with only one argument (the tree t), whose result is precisely the value of the attribute(s) of its root.

This remark allows us to introduce the notion of *static* and *dynamic* size of the evaluator. By static size, we have in mind the *size of the evaluator as it is constructed* , i.e. the size of code and data of the constructed program. By dynamic size, we intend the *size of information added to the tree t and to the set of variables W(t)*. Of course the size is the sum of the static and dynamic sizes.

In order to illustrate these notions, let us note that a recursive evaluator for a pure-S AG has a static size proportional to the AG size

and a dynamic size corresponding to t plus its recursive calls stack whose height is that of the tree t. There is no need to keep W(t), but, in such a case, the evaluation has an exponential complexity, as shown in the following example:

3.1.2.
Example
(tree t is given as a parenthesized expression)



The recursive functions are the following ones:

$value_s(p_1(t))=f(value_{s_1}(t),value_{s_2}(t))$

$value_{s_1}(p_2(t))=f_1(value_{s_1}(t),value_{s_2}(t))$

$value_{s_2}(p_2(t))=f_2(value_{s_1}(t),value_{s_2}(t))$

$value_{s_1}(p3)=a_1$

$value_{s_2}(p3)=a_2$

If no result is stored, the time complexity of $value_s(t)$ computation is exponential in tree height ($O(4^n)$ if n is this height, $O(2^m)$ if $m = card(W(t))$) : we are far from a time optimal evaluator!

By opposition, if intermediate results are kept, or if functions are organized to compute all the synthesized attributes in one call, we find again the time optimality in the case where all the attributes are useful.

For instance, with the evaluator :

$value_s(p_1(t))=f(value_{s_1s_2}(t))$

$value_{s_1s_2}(p_2(t))=g(value_{s_1s_2}(t))$

$$value_{s_1 s_2}(p_3(t)) = \langle a_1, a_2 \rangle$$

where $value_{s_1 s_2}$ is a function defined on a pair of values with

$$g(\langle x,y \rangle) = \langle f_1(x,y), f_2(x,y) \rangle$$

the evaluator becomes time optimal and the dynamic size is limited to t plus the execution stack size.

This example was described in some details in order to emphasize size efficiency criteria for an evaluator; they are summarized in the following figure :

3.1.3. Criteria

Size efficiency criteria for an evaluator

- static size (size of code and data)
- dynamic size (size of manipulated data in term of t and W(t)).

To be complete, it would be necessary to take into account the *complexity of evaluator construction.* But we can observe that it is the same, in time, as the membership test to a class, and that, in space, it is of the same order as the static size of the evaluator or that of the membership test[6].

3.2. l-ordered AGs

Going back to the analysis of attribute grammars, we see that the pure-S AGs class is too restrictive in practice: it does not allow to transport information over the tree in any order. But we must keep in mind that this class of AGs avoids the dynamic building of a plan and we are going to characterize the largest possible class that obeys that condition, the l-ordered class.

---

[6] That comes from the fact that the membership test is a constructive method and remains the most complex part of the construction process which is done before any use.

3.2.1. Definition

An attribute grammar is *l-ordered* iff there exists a family of total orders $\{T(X)\}$ for $X \in N$ such that

$$\text{For any } p \in P,\ T(X_0) \cup T(X_1) \cup \ldots \cup T(X_{n_p}) \cup D(p) = T(p)$$

is cycle free.

If an AG is l-ordered, for every tree t, T(t) obtained by combining the orders T(p) is an evaluation order.

This property follows trivially from others stated in [Eng84]. It turns out that the attribute evaluation order (in a bottom-up way) is known at evaluator construction time. One can program the evaluator as sequences of computations including calls to visit subtrees of nodes in right parts, or calls to visit the context. Such sequences are called *visiting-sequences* in [Kas80, Eng84] and are associated to each production.

3.2.2. Example (continuation)

The visiting-sequence associated to rule p_2 is the same as that given in §2.3 augmented by the calls to visit the sons and the father:

- visit(i,X) is a call to execute the i-th visit of the subtree rooted X
- finish(i,X) is a call to go on the context visit
- compute(a,X) is a call to compute a^X.

The sequence

$$l^N1\ l^N0\ r^N0\ r^B\ v^B\ r^N1\ v^N1\ v^N0 \quad \text{(see §2.3.)}$$

is reduced to 2 phases of visits and computations : (for rule p_2)

visit(1,N_0):visit(1,N_1);compute(l,N_0);finish(1,N_0).

visit(2,N_0):compute(r,B);visit(1,B);compute(r,N_1);
visit(2,N_1);compute(v,N_0);finish(2,N_0)

The visiting-sequence does not describe the computations done outside of rule p_2, but as soon as a visit is finished (for instance, when visit(2,N_1) is complete) all the values needed for the immediately following computation (here compute(v,N_0)) are known.

It is clear that one can build efficient evaluators for such a category : a finite automaton directed by the visiting-sequences and a push-down stack for recursive calls to visits (whose height is at most that of tree t) are appropriate tools for the computation of variables in W(t).

The evaluator is bottom-up (not time optimal but linear in W(t) size, without computation repetition) and it looses no time in elaborating a plan. Its static size is obviously linear in the AG size (visiting-sequences size); its dynamic size, by opposition, uses t, all variables in W(t) which are kept in the tree t and a stack as stated before.

The evaluator construction must carry out the membership test. Definition 3.2.1 shows that it is easy to check, by means of transitive closure of T(p) in each production, that a AG is l-ordered if the family {T(X)} is known. However, one must find such a family, i.e. one partition of *Attr* (X) such that all the attributes in the same subpart can be computed during the same visit in any tree t. This problem was shown to be NP-complete [EF80c].

Faced with this complexity result (impractical even if done at construction time), Kastens [Kas80] discovered a subclass that can be tested in polynomial time (in the AG size) called *Ordered AGs* (in short *OAGs*). Barbar [Bar82] showed that there exists an infinity of such families, incomparable to each other. For the sake of simplicity, we shall not define them here, but we shall call them, following Barbar, OAG(i), $i \geq 0$. The family described by Kastens is OAG(0). The OAG(i) membership test is based upon the way used to totalize a partial order, which can be done in polynomial time.

3.2.3. Definition

An attribute grammar is CPO *(Closed Partial Order)* iff there exists a family of partial orders {R(X)} for $X \in N$ such that the three following conditions hold:

(i) *Compatibility* : for $X \in N$, $OI(X) \subseteq R(X)$

(ii) *Non-Circularity* : for $p \in P$, $R(X_1) \cup ... \cup R(X_{n_p}) \cup D(p)$ is cycle free

(iii) *Closure* : for $p \in P$, $[R(X_1) \cup ... \cup R(X_{n_p}) \cup D(p)]^+_{X_0} \subseteq R(X_0)$

BY R_X^+ we note the restriction of R^+ to the elements in *Attr* (X).

Such a family has the following properties :

- Categories OAG(i), l-ordered, CPO, well-formed are strictly included in an increasing order of generality [Bar82]; see figure 3.5.3;
- for $X \in N$, R(X) is cycle free;
- for $X \in N$, $IO(X) \subseteq R(X)$ (this property comes from (iii))

From this last property, it follows that R(X) contains the union of IO(X) and OI(X), so R(X) contains all the possible dependencies induced by any relation $R^+(t)$ restricted to *Attr* (u) for any complete tree t derived from the axiom.

The CPO category is especially interesting because there exists a polynomial algorithm computing the family of minimal R(X) [Bar82]. Starting from this point, it is easy to define the OAG(i) categories considering the way the R(X) family is completed to give a total order.

We use R(X) to realize a partition[7] of *Attr* (X) by putting in a first class the attributes of *Inh* (X) that depend on no other ones and those of *Syn* (X) that depend on them or depend on no other ones. In each step, we *forget* the attributes already put in a partition and the algorithm is continued until all the attributes have been processed.

In this way, we obtain in each class, except the first one, synthesized attributes that depend at least on an inherited attribute of the same class. The order can now be completed to make all the synthesized attributes of a class depend on all the inherited ones of the same class. An arbitrary order can be given between the inherited and synthesized attributes inside each class. After this construction of a family of total orders T(X), the l-ordered condition (definition 3.2.1) has to be tested. This can be done in polynomial time (transitive closures). If this condition holds, the attribute grammar is OAG(0) and the construction is done in polynomial time as the test is constructive.

To our knowledge, no system has been built for the categories CPO or l-ordered. Only the OAG(0) category is implemented in some systems, the most famous of them being the GAG system [KHZ82] whose properties correspond to those of l-ordered AGs evaluators.

The l-ordered category is particularly important as any non-circular AG can be transformed into an equivalent l-ordered AG (with the same semantic definitions) but of an exponential size [Fil83b]. This fact leads to assume that it would not be interesting to build evaluators for larger categories (between l-ordered and well-formed). But this would be wrong as it can be deduced from the existence of evaluators based on AGs called *strongly non circular AGs (SNC)* [CF80a] or *absolutely non*

[7] We give here the OAG(0) algorithm

circular (ANC) [KW76].

3.3. Strongly non circular AGs

3.3.1. Definition

An AG is SNC iff there exists a family of partial orders {R(X) X ∈ N} such that the conditions of ***non-circularity*** (ii of CPO) and ***closure*** (iii of CPO) hold.

This definition was first given by Courcelle and Franchi-Zannettacci[CF80a]. This category is equivalent to that defined by Kennedy and Warren [KW76]. See also Knuth's first algorithm [Knu68], and [LP75], [Jou82]. It is a super-category of CPO as its definition is that of CPO with the first condition (i) forgotten.

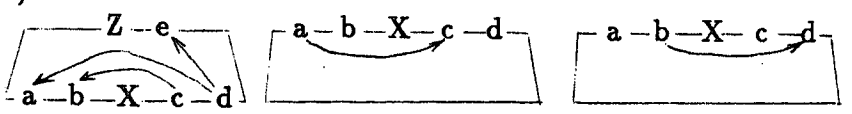
It satisfies the following properties:

- Categories CPO, SNC and well-formed are strictly included in each other [Bar82]. See figure 3.5.3.
- for X ∈ N, IO(X) ⊆ R(X).
- There exists a polynomial algorithm that computes the minimal R(X) family if the AG is SNC [CF80a].

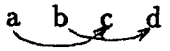
3.3.2. Example

In order to illustrate the strict inclusion of l-ordered, SNC and well-formed AGs, we give two school examples.

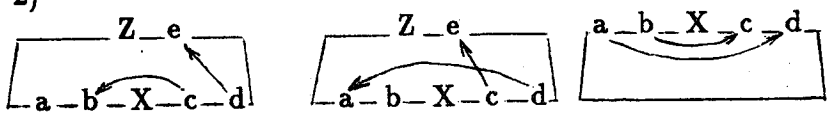
1)



This AG is well-formed as in each of the two possible trees R(t) is cycle free.

But it is not SNC: taking R(X)=IO(X)=  there is a circularity in the first rule.

2)



This AG is SNC: taking R(X)=IO(X)= a b c d there is no circularity in any rule. The closure condition is trivially satisfied.

By opposition, it is not CPO as OI(X)= a b c d and with R(X) at least equal to a b c d , there is a trivial circularity.

It is not l-ordered.

This fact could have been noticed by considering that in each of the two possible trees, no total order can cover the two partial orders.

Let us call *Use* (X,a) = {b | b ∈ *Inh* (X) and b R^+(X) a} the set of inherited attributes of X on which a depends.

The most interesting characteristic of an AG belonging to this category is that actual dependencies between inherited and synthesized attributes induced by R(t) in a tree t derived from X are covered by R(X) (that property comes from IO(X) ⊆ R(X)). It follows that any synthesized attribute a associated with a non-terminal X can be considered as a function of the tree derived from X and of the attributes in *Use* (X,a). The entire AG is equivalent to a set of functions recursively defined over the structure of the tree t, as shown by Courcelle and Franchi-Zannettacci [CF80a]. Kennedy and Warren [KW76] gave an iterative version of such an evaluator.

More generally, it was shown that any well-formed AG can be given as a system of recursive functions with fixed point [CM78] or without fixed point [May81, Fra82].

One of the advantages of the SNC evaluator is that it is a top-down evaluator. However as often IO(X) ⊂ R(X), it will compute too many inherited attributes and thus compute a set *UsefulR* (t) larger than *UsefulR* $_d$(t). In addition to this, its time complexity is exponential in the size of W(t) if no information about the assigned variables is maintained (it is the same case as pure-S AGs). Of course, this is not acceptable and Jourdan's implementation [Jou84a] keeps the values of synthesized values, but recomputes inherited attribute values. In that case the SNC evaluator cannot be time-optimal even if all attributes are computed because the proportionality constant is greater than 1. Some

experiences studying SNC and l-ordered approaches [Kav84] show that in practice they are comparable[8].

The dynamic size of the SNC evaluator is interesting as one keeps only t and the synthesized attributes of W(t) and a recursion push-down stack whose maximal height is given by the tree height but containing in each element as many information as inherited attributes in the corresponding set *Use* (X,a).

The static size of the SNC evaluator depends on the way information is coded. But it can be considered as linear in the size of the graph

$$R(X_1) \cup ... \cup R(X_{n_p}) \cup D(p)$$

hence polynomial in the AG size.

3.4. Evaluators for general attribute grammars

The search for universal *time-optimal* evaluators led some people to show that SNC construction could be applied to well-formed AGs. Katayama [Kat80] showed that every well-formed AG can be transformed into a SNC AG. This idea can be used to build top-down time-optimal evaluators for well-formed AGs. The evaluator is made up of two phases:

1) Pure-S phase : bottom-up computation of relations $sd(X_u)$ in each node u of tree t.

2) Top-down computation (time-optimal if all the values of W(t) are stored according to the SNC method). It is sufficient to select during the previous phase the functions to use .

A second method consists in building the total order T(t) during two preliminary phases and then in applying the l-ordered method. The evaluator proceeds in three phases [Eng84] :

1) Pure-S phase : bottom-up computation of relations $sd(X_u)$ at each node u of tree t.

2) Pure-I phase : top-down computation of relations $id(X_u)$ at each node u of tree t and transformation of the order into a total one.

3) Bottom-up time-optimal (if all the attributes are useful) computation using a l-ordered method.

[8] Recomputed inherited attributes are almost always identities.

This approach is based upon the property that any well-formed AG can be transformed into a l-ordered one.

Both approaches can be efficient if all the possible relations sd and id are pre-computed during the evaluator construction (otherwise the plan construction time could be important). But in each case the transformed AG size is exponential and this would be the same for the evaluator's static size. This is a good reason to understand why those approaches have never been implemented although they were proposed by many authors [Kat80, CH79, Eng84]. In fact, they do not seem to be applicable unless optimizations corresponding to those proposed in [DJL84] are applied. But in that case, it is not sure at all that the benefit gained in generality balances the increased complexity of evaluators that could not, in any case, be time-optimal because of the optimizations.

The only case, to our knowledge, where the second approach was applied is in the Reps and Teitelbaum's incremental evaluator [RTD83]. In this system, relations sd and id are updated during cursor moves and this cost seems tolerable because of the practical slowness of moves in an interactive editing framework.

Other evaluation methods, top-down and time-optimal, applicable to well-formed AGs have been proposed by Heeg [HV80], Jallili and Gallier [JG83] and Jourdan [Jou84b]. The main idea is to consider each defined attribute as a function of the tree and the attributes it depends on. The evaluator static size is linear in the AG size, its dynamic size can be considerable because one needs to keep in memory t and W(t) in order to avoid useless recomputations and the stack size corresponds no longer to the tree height but to that of the cycle free graph R(t). Some experiments about the ERN method [Jou84b] show that if an ERN evaluator construction is quick, the evaluator dynamic size disallows to compile large examples. This kind of approach seems to be more convenient for language design than for the realization of production compilers.

3.5. Summary

Going on in our survey of AGs categories, we must mention two categories introduced in the literature but not yet implemented. We mention them because they seem to be interesting, at least from a theoretical point of view : benign AGs and partially-ordered AGs.

3.5.1. Benign AGs definition:

An AG is *benign* iff for all $p \in P$

$$IO(X_1) \cup ... \cup IO(X_{n_p}) \cup D(p)$$

is cycle free.

Introduced by Mayoh [May78], this category is placed strictly between SNC and well-formed in the categories inclusion order. Franchi-Zannettacci has shown that it is the largest category for which a recursive functions system can be built without using an undefined element [Fra82]. It is also the largest class for which a top-down evaluator based on the SNC method can be built, but, in such a case, a call-by-need language is necessary. The well known inefficiency of this kind of call is probably the reason why no practical evaluator was built for this category.

3.5.2. Ordered AGs definition:

An AG is partially-ordered or PO (resp. totally-ordered or TO) iff there exists a family of partial (resp. total) orders $\{R(X)\}$ for $X \in N$ such that:

i) *compatibility* : for $X \in N$, $IO(X) \cup OI(X) \subseteq R(X)$

ii) *non circularity* : $R(X)$ is cycle free.

Barbar has shown that this category is between CPO, whose definition implies the PO conditions, and well-formed, but is not comparable with benign and SNC [Bar82]. He also showed that TO and PO are equivalent categories decidable in an exponential time.

Figure 3.5.3 summarizes the inclusion relations between all the categories. Continuous arrows denote a strict inclusion and dotted lines connect not comparable categories (the intersection of which may be not empty, for instance CPO = SNC $\cap$ OP).

Before concluding this section, we must examine the *incremental evaluators* with a specific care. In fact they are attribute evaluators and re-evaluators that take place in systems for tree transformations defined by means of AGs (see, for example, the Cornell Program Synthesizer [RT84b]), or OPTRAN [MWW84]. In such a case, the system handles consistently decorated trees (i.e. correctly decorated), and each transformation locally changes the tree which becomes locally inconsistent. So there is a need to recompute the attributes that have changed and only them.

The notion of time-optimality must be redefined in such a framework: an incremental evaluator (or *re-evaluator*) is time-optimal iff it recomputes only once the attributes in $Useful_R(t)$ *whose value has changed.* If we consider that all the attributes are useful, that means that t and W(t) must be kept. This optimality criterion raises many questions:

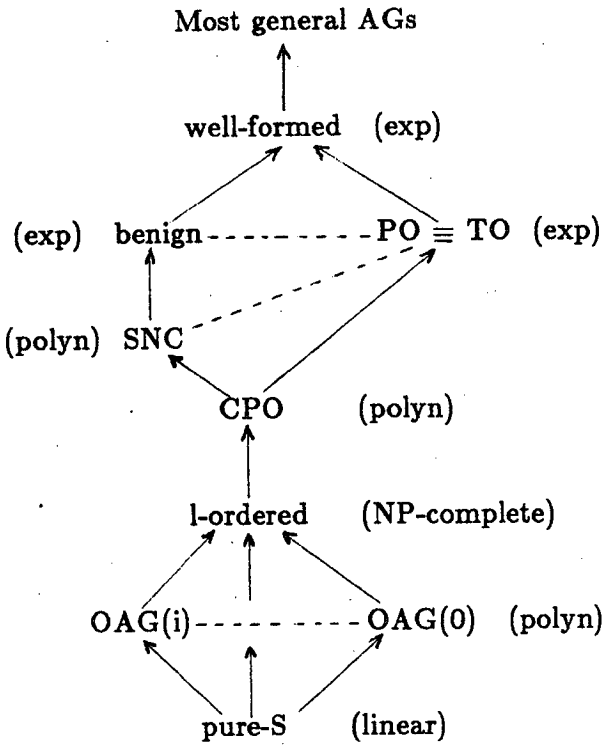


Figure 3.5.3 Categories and test complexity

- one may hope not to visit the whole evaluation plan (even if the number of semantic functions calls is time-optimal) [Eng84];

- to test in an efficient way wether a value has changed, closely depends on attribute values representation [Rep82b].

Few experiments exist in this area and looking for the maximal efficiency (in time) does not automaticaly lead to a good practical efficiency.

To our knowledge, two approaches for re-evaluators construction were proposed: Reps and Teitelbaum [RT84a] implement a bottom-up re-evaluator, very close to time-optimality. The second one, by Jallili and Gallier[JG83], is top-down; it avoids to compute semantic dead-ends. Reps [Rep82b] uses both approaches, putting apart attributes to be systematically computed again (in a bottom-up way) after each tree modification and attributes to be re-evaluated only when needed.

In conclusion, our study shows that there is no *best* AGs category, leading to an evaluator (or a re-evaluator) that, at the same time, is time-optimal and has a static and dynamic size allowing to use it

practically. One may point out a general rule which is somewhat amazing:

"The largest the category is, the closest possible time-optimality but also the largest static size of the evaluator can be reached".

The sequel of this study will show that time and space are related in a still more complex way.

4. EVALUATION BY TREE WALK

Using Engelfriet and Filé notations [EF81c], we study here the sub-categories of l-ordered AGs. Up to now these classes have given rise to the largest number of implementations. In addition to that, some sub-categories seem to play an interesting role from a theoretical point of view. If one takes into account time and space efficiency, we shall see that the evaluators built up in these categories (except Pure-S AGs) cannot contain notable improvements over l-ordered AGs, at least theoretically speaking, whereas their practical expression power is largely reduced (see §6.2).

The classification introduced here is based on a non-deterministic evaluator called PURE-VISIT whose non-determinism may be tuned in order to get different useful categories. Evaluators associated to these new categories are based on specific ways of visiting the syntactic parse tree, visits that are predefined in the tree t and during which values are assigned to the variables W(t). This is why we call these methods *tree-walk evaluators.*

4.1. Definition

We give first the general Pure-visit algorithm as introduced by Filé [Fil83c page 157]:

4.1.1. Algorithm

1- *procedure* visit-and-evaluate(u)
{ u is the current node of tree t, *sort* (u)=X, *label* (u)=p }
2- *begin* C(u) := C(u) + 1
{ C is a counter associated to each node u, initialized to 0 }
3- compute *some* inherited attributes of X_u
4- *guess* a sequence v=<v(1), v(2),..., v(m)>
with m $\geq$ 0 and v(j) $\in$ [1..n_p] for 1 $\leq$ j $\leq$ m
5- *for* j:=1 *to* m *do* visit-and-evaluate(uv(j))

6- compute *some* synthesized attributes of X_u
end visit-and-evaluate

7- *procedure* evaluate(t)
{ t is the tree to decorate }
8- *begin* initialize all the counters to 0
9- *guess* k0
{ k0≥1 }
10- *while* C(root(t))≤k0 *do* visit-and-evaluate(root(t))
end evaluate

It is important to note in which sense the procedure *evaluate* is non-deterministic. It describes a way of visiting a tree t and recursively calls the procedure *visit-and-evaluate* at nodes of the tree. Each time one can visit a node, the counter associated to this node is incremented (line 2) and some inherited attributes (maybe none) are guessed and evaluated (line 3). The choice is not defined here and this is the reason for the non-deterministic nature of the algorithm. Then a sequence, empty or finite but unbounded, of sons of the node u is guessed (the same node may appear more than once) (line 4). Such a sequence is called a *visit-sequence*. Procedure *visit-and-evaluate* is then recursively called for each node in the visit-sequence following the order specified by the sequence (line 5). When back, some synthesized attributes (maybe none) of node u are chosen and evaluated (line 6). The main procedure *evaluate* is easy to understand. After successful termination of *evaluate* (t), the *visit-counter* C(u) of every node u of t contains the number of calls to *visit-and-evaluate* (u) that have been executed during the computation. Each such call is in fact the beginning of a visit of the subtree issued from u. Therefore, we say that the call to evaluate(t) is *k-visit* ($k \geq 1$) if, after the computation, the visit-counter of each node is at most k (with $k0 \leq k$).

A computation of evaluate(t) may fail before evaluating all attribute instances if one chooses to evaluate some attribute that depends on not yet evaluated attributes. If this failure does not happen and the computation stops after evaluating all elements in W(t), it is said to be *complete* (it is *successful* in the sense of Filé [Fil83c]).

Now we are ready to introduce eight classes of attribute grammars based upon the non-determinism tuning inside the *Pure-visit* procedure. As pointed out in [EF81c], basically there are two kinds of non determinisms:

Non-determinism of type a : at each visit to a node u, attributes to be evaluated are chosen arbitrarily (line 3 and 6).

Non-determinism of type b : at each node labeled p, the visit-sequence is chosen arbitrarily (line 4).

Removing non-determinism of type b only leads to four categories called *Pure-multi-Y* where Y $\in$ {visit, sweep, alternating pass, (L or R) pass} whose meaning is:

Y = visit : no restriction

Y = sweep : v is a permutation over $[1..n_p]$

Y = alt : v = $[1..n_p]$ (same order or L) or $[n_p..1]$ (reverse order or R)

Y = (L or R) pass : v is always $[1..n_p]$ (L) or always $[n_p..1]$ (R)

By removing non-determinism of type a, we get four new categories called *Simple-multi-Y*. In concrete terms, algorithm 4.1.1 is transformed in the following way: there exists a partition over *Attr* (X) for X $\in$ N, say

$$\{ A_l(X) \text{ for } 1 \leq l \leq k \}$$

such that lines 3 and 6 can be changed into

3- compute the inherited attributes of $A_{C(u)}(X_u)$.

6- compute the synthesized attributes of $A_{C(u)}(X_u)$.

Doing this, we avoid any non-determinism for Y = pass, but we must take care of modifying line 4 in the three other cases:

4- take the visit-sequence $v^p_{C(u)}$ associated to rule p.

This implies that, in addition to the partition on the attributes of each non terminal specifying which attributes are to be evaluated during the i-th visit, one can determine for each rule p the i-th corresponding visit-sequence.

For a given k, we shall denote by T-k-Y strategy the eight strategies previously defined with

T $\in$ {Pure, Simple}

Y $\in$ {visit, sweep, alt, pass}

k is the maximum number of visits to a node.

4.1.2. Definition

An attribute grammar is T-k-Y iff for any tree t, the computation of evaluate(t) with the strategy T-k-Y is complete.

4.1.3. Definition

An attribute grammar is T-multi-Y iff there exists some k such that it is T-k-Y.

In order to illustrate these definitions, we give two algorithms corresponding to Simple-k-sweep and Pure-k-alt.

4.1.4. Example, Simple-k-sweep

Each node is visited at most k times. There exists a partition over the attributes $A_i(X) \subseteq Attr\ (X)$ for $1 \leq i \leq k$, and in each rule $p \in P$, k permutations on the right hand side non-terminal symbols are defined for each visit, say v_i^P the i-th permutation for $1 \leq i \leq k$.

```
1- procedure Simple-sweep-evaluate(u)
{ u is the current node of tree t, sort (u) = X, label (u) = p
}
{ i is a global variable }
2- begin
3- compute the inherited attributes of A_i(X_u)
4-5- for j:=1 to m do Simple-sweep-evaluate(uv_i^P(j))
6- compute the synthesized attributes of A_i(X_u)
end Simple-sweep-evaluate

7- procedure evaluate(t)
{ t is the tree to decorate }
8- begin
9-10 for i:=1 to k do Simple-sweep-evaluate(root(t))
end evaluate;
```

This algorithm performs k walks over the tree in an order depending on the rule and the visit number.

In the case of a Simple-k-L-pass, there are k top-down left to right walks.

4.1.5. Example, Pure-k-alt

The only restriction is the following one: v_i^p tells whether the right hand side non-terminals of rule p are processed in a left to right order (L) or a right to left one (R).

```
1- procedure Pure-alt-evaluate(u)
{ u is the current node of tree t, sort (u)=X, label (u)=p }
{ i is a global variable }
2- begin
3- compute some inherited attributes of X_u
4-5- for j:=1 to m do Pure-alt-evaluate(u v_i^p(j))
6- compute some synthesized attributes of X_u
end Pure-alt-evaluate

7- procedure evaluate(t)
{ t is the tree to decorate }
8- begin
9-10 for i:=1 to k do Pure-alt-evaluate(root(t))
end evaluate;
```

It is clear that any well-formed AG is Pure-multi-visit, since by an appropriate choice of the attribute to be evaluated and of visit order of the nodes (following the evaluation order $R_d(t)$), there exists an integer k (equal to the maximum number of attributes associated to a non-terminal symbol) such that the Pure-k-visit is complete. In the sequel of this section, we shall restrict our attention to *Simple* classes for which there is no non-determinism, as categories *Pure* play essentially a theoretical role or correspond to categories already studied in the previous section.

4.2. Results (Simple categories)

The first results are concerned with the relative positions between the categories Simple-X-Y of which there exists an infinite number. Figure 4.2.1. summarizes the results obtained by Engelfriet and Filé [EF80c]; some categories among those defined in the previous section are also given again in order to help in giving the correspondence between both hierarchies.

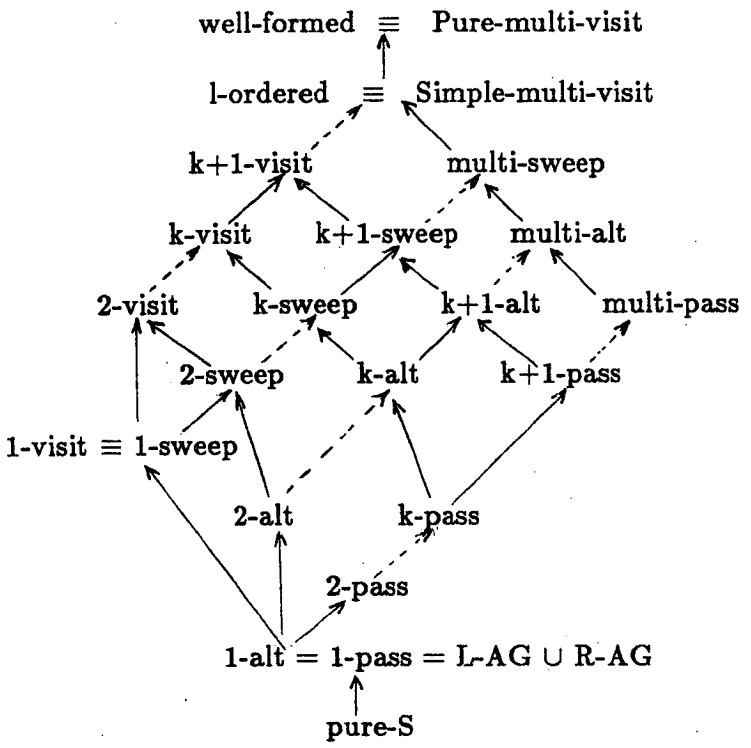


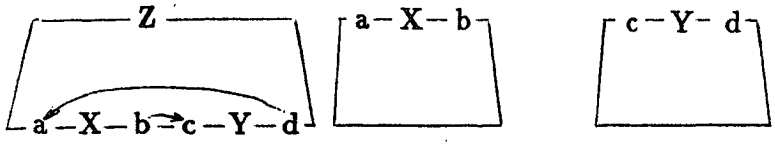
Figure 4.2.1. : Classification based on tree-walks

Arrows denote strict inclusion. No arrow indicates incomparability.

Note also another result: Simple and Pure categories are strictly included in each other (Simple in Pure) and OAG(i) are incomparable with multi-sweep and other subclasses [Eng83], with the exception of the 1-sweep category which trivially is a strict subclass of OAG(i). These results are illustrated in Figure 4.2.4 and we give now two examples in order to show the incomparability between OAG(0) and Simple-multi-sweep and the strict inclusion of OAG(0) in l-ordered.

4.2.1. Examples

This first AG is 2-sweep (in fact 2-alt) but is not OAG(0).

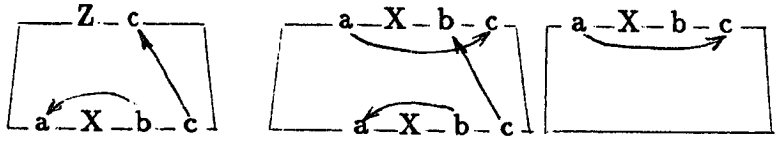


It is trivially 2-alt using sequences LR or RL. It is not OAG(0) since, by computing CPO relations, we obtain

$$R(X) = \quad a \quad b \qquad R(Y) = \quad c \quad d$$

Thus it is CPO but the totalization process, by putting all the attributes in the same partition class, introduces new dependencies from a to b and c to d, leading to a cycle in the first rule.

This second AG is OAG(0) but is not multi-sweep



It is in fact CPO with

$$R(X) = \quad a \quad b \quad c \quad = IO(X) \cup OI(X)$$

and OAG(0) with

$$A_1(X) = \{b\} \quad A_2(X) = \{a,c\} \quad \text{and} \quad R'(X) = R(X)$$

The total order in each rule is easily deduced.

It is not multi-sweep since in a tree of height 3, for example, it is not possible to compute all the instances of one of the attributes in a unique sweep, thus no attribute can be computed either in one (sweep) pass nor in more.

Both grammars are l-ordered.

Some among the categories introduced in Figure 4.2.1. are well known in the literature.

The L-AG class described in [Lor74, Boc76] corresponds to attribute grammars for which attribute evaluation can be done in parallel with LL parsing methods (i.e. top-down left to right tree-walk).

The Simple-1-sweep category seems to be particularly important because of its ability of modeling program schemes [CD84, DM84b]. It has a very simple relational characterization [Fil83c].

For each production rule

$$p: X_0 \rightarrow X_1 \ldots X_{n_p}$$

we can define the brother graph B(p) to be

$$B(p) = (\{1,2,\ldots,n_p\}, \rightarrow)$$

where $i \rightarrow j$ iff there exists a,b such that (a,i) D(p) (b,k)

4.2.2. Definition (AGs are supposed to be normalized)

An AG is Simple-1-sweep iff B(p) is acyclic for every p.

Multi-Y evaluators were the first ones studied and implemented. One can think that people tried to increase efficiency by some restrictions on AGs. They had probably in mind to model attribute evaluators as classical compilers by means of *passes (phases* in [EF81c]). Nevertheless, it appears now that theoretical results do not come up to their expectations.

On the one hand, the way an AG works allows precisely to free oneself from the notion of *pass* as the only thing we are concerned with is to describe local computations without worrying about the way the computations are done elsewhere. On the other hand, it is not obvious that one can get more efficient evaluators by restricting the class of attribute grammars used (unless when the L-AG category is reached).

The time complexity of a Simple-X-Y evaluator is the same as a l-ordered one. Its dynamic size is approximately the same as one must in general keep t, W(t) and a stack for tree walking whose size is bounded by its height. So it appears that the only saving is the evaluator static size which is largely decreased as it essentially consists of predefined tree-walks. But it is necessary to wonder if this improvement is not counterbalanced for by a new inefficiency factor : the plan may be processed many times as it is replaced by the tree in the control of the evaluator. So a new problem arises: the optimization of the number of visits in the tree. But we must note that even with this improvement, there is no guaranty that the plan will be processed in an optimal way.

These AGs categories raise two main problems to solve for the evaluator construction which have been extensively investigated [RS77, Alb83, EF81c, JP77a, JP77b, PJ78a, PJ78b, Poz79] :

> *Problem 1* (Characterization) : decide whether an AG is Pure/Simple-multi-Y.

Problem 2 (Optimization) : considering a Pure/Simple-multi-Y AG G, find the smallest k such that G is Pure/Simple-k-Y.

Figures 4.2.4. and 4.2.5. exhibit time complexity results for these problems in the *Simple* case (results taken from [EF81c]).

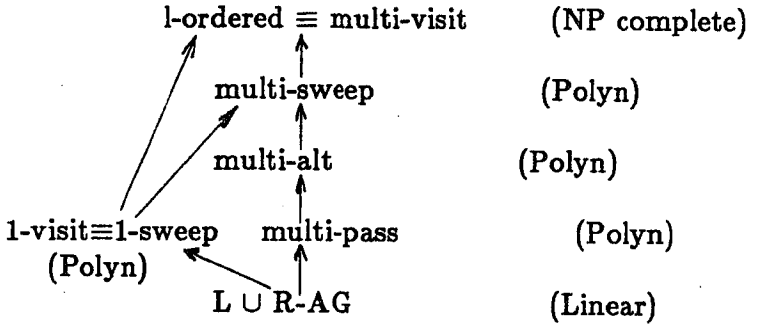


Figure 4.2.4. : Complexity of the characterization problem

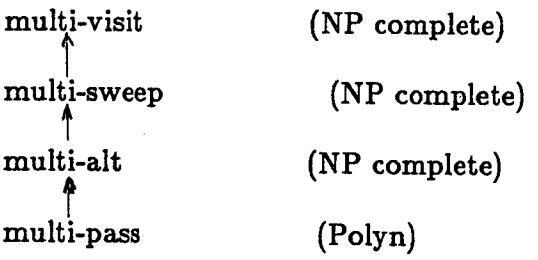


Figure 4.2.5. : Complexity of the optimization problem

The question of finding optimal evaluators (in the number of passes) for Simple-multi-alt evaluators has been studied and it has been shown that some constructions [PJ78a, PJ78b] requiring lower complexity lead to evaluators close to optimality [RS77].

4.3. General classification

We summarize in Figure 4.3.1. the most significant categories from a theoretical and a practical point of view, especially for evaluators construction. Results about the characterization and optimization problems in the construction of multi-Y evaluators are mentioned again.

Now we reach the end of this section and the *rule* stated in §3.5. seems to be correct : except if we accept to go down to very restrictive categories such as L-AG, going down in the categories improve evalua-

tors sizes but bring no benefit to their efficiency in time and space; they even could be less efficient. And practical experiences, according to such systems implementors, show that the lowest categories have a very poor expression power. It remains to evaluate a factor we have not deeply studied till now: the plan optimization and its effect on the evaluator.

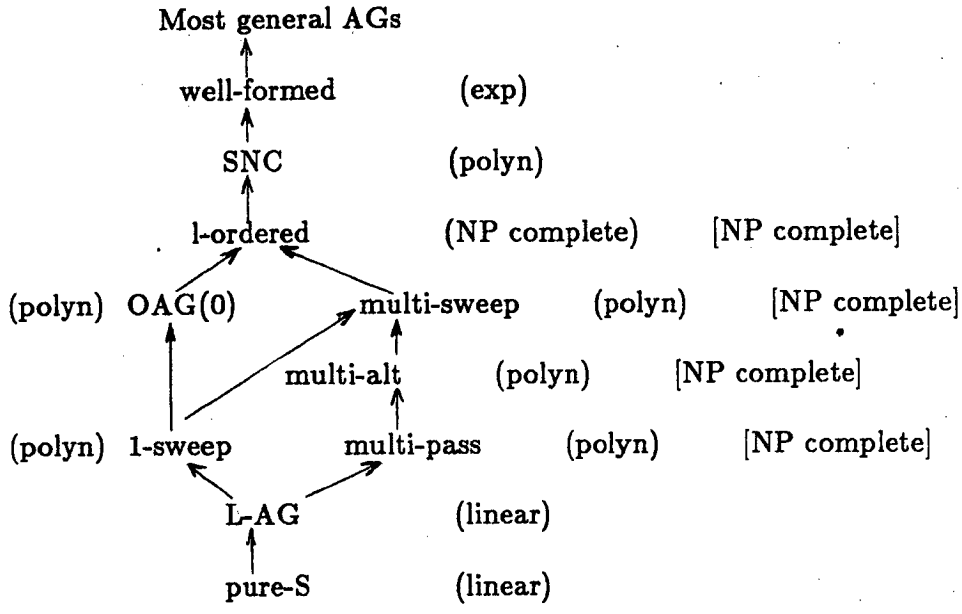


Figure 4.3.1 : General classification
(time complexity of the characterization test)
[time complexity of the multi-Y optimization]

5. REDUCTION OF THE DYNAMIC SIZE OF EVALUATORS

The most important problem the attribute grammars method must overcome, apart from the time efficiency of evaluators, is the amount of space needed for their implementation.

This space is composed of:

1) The derivation tree t.

2) The set of variables to be assigned values W(t).

3) The space for complex attribute values.

4) Additional data created by the evaluator (e.g. an evaluation stack).

The space for complex attribute values corresponds to the space needed to hold values too complex to fit into a memory cell. In fact, the variables in W(t) are associated with positions in the tree t and their values are placed into areas linked to each tree node. Most of the systems memorize the complex values in a sharable way and the place associated to the node is limited to a fixed size cell whose content is either a numeric value or a pointer to a shared area. The space needed to hold all attribute values can be divided into |W(t)| simple values and what we call *the size of space shared by complex attribute values.*

To decrease the dynamic size of the evaluator is to act on the four factors previously given in order to minimize them as much as possible. This way of doing induces necessarily a feed back on the evaluator time efficiency. Therefore a global approach to complexity in time and space is needed. Some *standardization* is also necessary in order to compare all the approachs. Finally, there is good reason to set apart the theoretical complexity in the worst case, from the practical efficiency which depends on a specific computer, a specific system or the class of examples taken into account. Most of the existing systems implementors supply little complexity analysis and still less study about improvement provided by some optimization. This question has yet received little investigation but seems necessarily to become the focus of attention in the future.

First we define some *standards* as a basis to build a complexity study in order to be able to make comparisons between evaluation systems. Then we tackle the global optimization problem and, in a third part, we get to the practical efficiency of some families of evaluators.

5.1. Criteria for a study of the dynamic size complexity

One of the four criteria previously stated seems little significant: the size of complex attributes. Most of the systems use a sharable data structure (GAG, FNC and many others). Some of them, but few, such as LINGUIST 86, work on the basis of alternated passes and copy of values, and, as a consequence, cannot share attribute values. Other systems, such as GAG or CSG deal with specific representation for data in order to optimize some processing for the most used structures [Asb79, Rep82b]. Because of the wide range of topics covered and of the appropriate solutions, it does not seem possible to include this point in a comparative study. According to [KHZ82], this storage corresponds to 20 to 25% of the whole space required.

It is therefore worth noticing that, in the case of a shared representation for the values, attributes method correctness can be proved only if the semantic functions work without side effects on the data (an applicative style language is needed: ALADIN in GAG, *pure LISP* in FNC), and this aspect leads to inefficiencies. These ones can be partially overcome if the user knows precisely the places where he can free himself from the applicative style, for instance if an attribute along with all the attributes it depends on can be considered as a unique global variable. Few works have been done in this area; only the GAG system deals with this approach [Kas84] by looking for attribute classes that can be implemented in the same global variable. It would be very interesting that the user could be freed from such a constraint or that the attribute definition language by itself could allow the system to detect in an automatic manner some inconsistencies in the data processing introduced by the user.

Considering now the additional data created by the evaluator (fourth point), they can come from the evaluator for its execution (recursive calls stack for SNC or l-ordered methods, tree-walk stack for Simple-X-Y methods, state storage at tree nodes for general iterative methods, or specific), or from the optimizations which need some space to be realized. In all the cases -and this fact can be noticed in practical implementations- additional data must have a complexity linear in the tree size to be acceptable. The general functional methods (ERN [Jou83], [JG83], LINGUA [HV80]) need a stack whose height is linear in the size of W(t) (in the worst case) and seem to be untractable for large examples even when used on very large computers. Is can be observed that additional data, when of an important size, must be measured in connection with the size of t and W(t). So we shall remind $n=|W(t)|$ and $m=|t|$ as complexity parameters.

Moreover it is useful to keep in mind that there is some relation between n and m, depending on the number of attributes associated with each node. Considering some restrictions, as suggested by Reps [Rep82b], we can formulate this relation. If we call *Maxnbattr* the largest number of attributes associated to each node and if we suppose that

for all $X \in N$, $Attr(X) \neq \phi$ this relation is $m \leq n \leq m \times Maxnbattr$ [9]

Finally, in order to facilitate comparisons between systems, we suppose that all AGs are in normal form (this helps the complexity formulation in some cases) and that we are only interested to compute the values of the synthesized attributes of the axiom. In the opposite case, no improvement in the size of t and W(t) could be done (sublinear evaluators). However in order to make comparisons between top-down and bottom-up methods easier, we suppose that there are no dead-ends because avoiding computation of attributes in dead-ends in a top-down method leads to time and space saving that must be added to improvements coming from other optimizations.

5.1.1. Criteria

The following criteria summarize the set of conditions to be followed in order to be able to measure theoretical and practical complexity of an attribute evaluator:

- AG in normal form
- for $X \in N$, $Attr(X) \neq \phi$
- $n = |W(T)|$, $m = |t|$, $m << n$
- The only interesting attributes are synthesized attributes of the axiom but all the attributes take part in their computation
- We do not consider the attribute values actual size, which is supposed to be one unit
- All the semantic functions have the same duration, including even identities.

---

[9] These inequalities are not symmetrically balanced: if n practically corresponds to actual memory cells, m is associated to nodes often implemented by numerous memory cells. This fact makes the left inequality uncertain, so the criterion used in the sequel will be $m << n$ in order to make comparisons at least at a theoretical level.

the latter case, the attribute values size can be increased[11] and the evaluator code augmented by the stack processing instructions. Result is positive for W(t) but negative for time (only for attributes represented as stacks). Following published results, the gain is about 60% [KHZ82 page 63] if one takes into account only variables changed into global ones and if one considers that attributes are regularly distributed among trees t.

In the evaluator FNC [Jou83], such a method cannot be applied as there is no total order (category SNC). On the other hand, inherited attributed are not kept in the tree. If time and space remain roughly linear in n, but with possible repetition of inherited attributes computation, there is an improvement on W(t).

All the results given in this section are put together in Table 5.3.1. where n+ and n- denote a linear complexity in n with an increased (n+) or decreased (n-) coefficient.

	HLP	GAG	FNC	DELTA
Observed W(t) reduction	none	60%	?	95%
Time	$n+$	$n-$	$n+$	n^2
Space	$n+$	$n-$	$n-$	n
Prevailing Optimization	Attributes Values Deallocation	Variables Globalisation	Allocation of Inh in a stack	Allocation outside tree t

Table 5.3.1 : Practical Efficiency Results

A study of this table leads to numerous observations:

1) Practically, we have little information about actual efficiency of evaluators. It would be desirable that any system provides measure elements allowing to quantitatively

[11] as noticed in [KHZ82 page 68] as a result of the additional space required for implementing global stacks.

evaluate the optimizations effects.

2) It appears that there is no objection to consider that a system which is located in a general category of AGs could be as efficient in time and space as another one placed in a lower category (in the inclusion sense).

3) Each system puts emphasis on a specific optimization when one can imagine to combine some of them. They are:

- Variables globalization (GAG) (allocated outside the tree).
- Lack of allocation for some variables (FNC).
- Deallocation of variables values (HLP).
- All allocations outside the tree (DELTA).

One could add:

- Deallocation of W(t) and t [JP77a].
- Optimization of the tree itself (FNC, GAG, HLP).

This last optimization already exists in most systems when *simple productions with trivial identities are eliminated,* with a decrease of m and n.

Finally, one can notice something else: criteria we have used do not exhibit all the optimizations that can be automatically done and put at hand in order to improve attribute definitions when performance measures are known. It is really noteworthy that experiments done inside the GAG system [KHZ82 page 67] show that, on a total benefit of 60% of $m+n$ (space tree), about 20% are obtained by manual changes and the 40% automatically obtained can be split into 10% for variables globalization and 30% for simple productions elimination. This factor is less interesting in systems dealing with abstract tree decoration. It would be attractive to be able to observe the same points in other systems.

And to conclude this study on optimization, we can say that, till now, too few studies have been done and especially too few measures. If, considering the GAG system experiment results, many information provided to the user allow him to change his definitions in order to improve the global size of compilers he is writing without diminishing their performances, much work remains to be done to automatically improve them. We think that such researches must impose a priori no restriction on AGs categories.

Practical experience makes these criteria acceptable, although they do not take into account the precise nature of data processed in the semantic definitions. For instance, it was measured with the GAG system running large examples of Pascal compilation that the storage taken up by the tree t and W(t) (variables are put at tree nodes in a cell containing either a numeric value or a pointer to a complex value) is about 2 to 4 times larger than space needed by attribute values [KHZ82]. This point was confirmed by comparisons between GAG and FNC [Kav84] and shows that n and m are the essential parameters in the formulation of time and space complexity for an evaluator. According to the relation between n and m, it appears that the main target is the optimization of n^{10}.

Comparisons given in the sequel consider that criteria 5.1.1. are obeyed.

5.2. Global complexity analysis

We are interested here in the question of minimizing W(t) and its influence on the evaluator time complexity. We also try to situate the methods described in the previous sections from a complexity point of view.

Whatever the method we consider, an evaluator must follow the evaluation order R(t). This problem is similar to the optimal allocation of registers which is well known to be NP-complete [Set75]. It turns out that minimizing W(t) for a given relation R(t) is a NP-complete problem. Ganzinger set up this result for the attributes method [Gan79b] and showed that the problem remains NP-complete even for the L-AG class. So it is hopeless to reduce the complexity by taking into account restricted classes of AGs, and, in general, the inefficiency in time of such an optimizer makes it unusable in practice.

The methods described in sections 3 and 4 (SNC, l-ordered, multi-Y without optimization) are O(n) in time and space in the worst case. As we described them, all these methods contain no specific optimization and, if in the top-down method, inherited attributes are not kept, they are pushed onto a stack whose height is less than n (height of the tree in the worst case with $m \leq n$).

[10] Considering a previously made remark related to the implementation of trees and of information at each node, optimization of m remains an important practical element. See for instance [KHZ82].

Reps [Rep82b] proposed two evaluation algorithms the space complexity of which is, respectively, $O(\sqrt{n})$ (for a time O(n) algorithm named REPS1) and O(log n) (for a quadratic time, algorithm REPS2). Furthermore, he proves that if space complexity is O(log n), time complexity cannot be linear. It is likely that one reaches here a lower bound in W(t) optimization if we keep in mind that a non-linear evaluator cannot be considered as acceptable. This fact seems to be confirmed in practice in DELTA where evaluation is quadratic in time [Lor74], but can be slowed down by experiment of the HLP system [Rai80a] where quadratic optimizations seem not to be too detrimental to the evaluator efficiency. We shall give more details in § 5.3.

Results are given in Table 5.2.1. It is worth noticing that for an increasing size (from top to bottom), time can be considered as decreasing since in REPS1 evaluation is linear in time but with repeated computations (other methods are supposed to be without repetition).

This kind of comparison shows its limits as there is no guaranty that for the same class of examples, all the methods behave correctly.

Evaluator	time	space
Optimized	NP-complete	optimal
REPS2	n^2	$\log n$
REPS1	n	$\sqrt{n}$
SNC,l-ordered, general	n	n

Table 5.2.1. : Global Complexity of an Evaluator

There are two remarks to be made about this table:

1) An important saving in space must be paid for by an important loss in time. Results in the last line are for non-optimized evaluators. We known little about the practical influence of optimizations in time and space.

2) Reps' results show that a global approach can be made and that a more systematic analysis of this problem could lead to more efficient evaluators both in space and time.

5.3. Practical efficiency

Qualitatively we analyse optimization results obtained in some systems and we deduce some evaluation criteria for optimizations in existing systems.

In the system DELTA [Lor74] which produces an equations system, optimization is done in the following steps:

- graph R(t) is built in a time O(n) (size O(n)).
- during the topological sort (time O(n)), a *dead-ends elimination* and a *variables allocation* are carried out in a quadratic algorithm in the worst case (additional space O(n), time $O(n^2)$).

As an effect of these optimizations, there is a saving of about 90% and the tree t is no longer kept. An advantage of this method is to allow a quicker processing of resulting equations. In practice, the saving does not compensate optimization time. The method remains O(n) in space and $O(n^2)$ in time in the worst case. It can be observed that we are very close to time linearity, the allocation algorithm, the only quadratic factor, behaving rather linearly.

Up to now, it seems that no efficient implementation was ever done for this method.

Jazayeri and Pozefsky [JP77a, JP79, JP80a] applied a similar approach in the multi-alt case. Starting from a preliminary tree visit (which may be done during parsing) they propose to build ordered lists of computations to be done in each pass. Time complexity remains O(n) but the tree t may be discarded or better never built. Some optimizations could be done, but no running system was realized on such ideas. Space complexity remains O(n).

In the system HLP [Rai80a] based upon multi-alt AGs, *deallocation* of attribute values is aimed at. Managing space with garbage collection allows to save space needed to keep attribute complex values. The salvage technique is strongly connected to this AGs category for which the (dynamic) order T(t) between variables of W(t) can be computed efficiently. The basic idea is that two attribute occurrences are ordered by the order of the passes they belong to (this order is statically defined), by passes direction (L or R) and according to their nature (synthesized or inherited) and if they are associated to node located on a same path or not.

Starting from this order, it is possible to dynamically compute the *last user* of some attributes equivalence class, defined as the set of all variables related by a chain of identities *(copy rules)*. In this system, all the attributes cells are kept in the tree, so the size is O(n). Identities are processed by means of copy of pointers. Therefore all the cells in an equivalence class point to the same shared value. The last user in a class is the highest element in T(t) which uses an element of the class in its computation. So, after the computation of the last user, the value shared by all the elements in the class may be discarded. The main result of this optimization consists in a decreasing of the complex attribute values size.

However, time complexity is larger than O(n) in the worst case. Testing is done in two steps. The first one computes during parsing in a pure-Ş way the equivalence classes and the last-user in each class. In practice, space needed is not changed as cells are used to implement the equivalence classes. Is added only a linear information, whose size is not significant. The complexity of this step can be considered as $O(m \times Maxnbattr^3)$ coming from transitive closures done in each class.

In the second step, as soon as the last-user of a class is computed, the complex value associated to this class can be deallocated. This phase is O(n) in time and, according to our hypothesis ($m \leq n \leq m \times Maxnbattr$), only the first phase may be considered as most expensive in time.

Despite of an execution time that seems to be quite long, the authors noticed no actual change in system behaviour, essentially because of a space saving of about 40% on attribute values. It turned out for them that this space saving improved the execution speed in a paged environment. Therefore this point remains highly dependent of the data representation method and of the operating system used. The important gain is related to the way of implementation of (not shared) values in the system HLP. From our criteria point of view, this method leads to time loss without substantial saving in space.

In the system GAG [KHZ82], another approach is proposed in l-ordered category framework. The main idea is the static detection of attribute occurrences that can be implemented in one *global variable*. Visit-sequences are then built using this variable. There is a decreasing in W(t) along with an execution time improvement as identities lead to no processing. Test is based upon life time analysis for attribute occurrences inside a visit-sequence. It is described in [Asb79, KHZ82, Kas84]. According to life time, an attribute occurrence can be implemented as a global variable or a stack, but in

6. FORMAL POWER OF ATTRIBUTE GRAMMARS

By formal power, we have in mind to define to which *class of transductions* corresponds a class of attribute grammars or, more generally, to which class of programs belongs an AG (and conversely). It is clear that the knowledge of the corresponding *class of programs* (or of program schemes) is useful to know the properties of an AG. This also enables to get theoretical results about complexity or decidability by establishing precise connections between AGs, tree transductions and program schemes.

We therefore present in an informal way both approaches (transductions and program schemes) and we give the most useful results for applications. We first describe a preliminary example showing, from a practical point of view, the limits of well formed AGs and the need to introduce more general classes.

6.1. Introduction to the formal power

We know from Knuth[Knu68][12] that *pure-S* AGs have the same power as Turing machines.

One could deduce that it is not necessary to introduce other classes of AGs. This fact is not, in general, sufficient to make a method easy to use, and, it does not take into account attribute values domain. When considering the transduction defined by an AG, we are not allowed to use the tree as a domain for values, in the way similar to denotational semantics where the syntactic domain plays a special role [SS71].

We are going to show on a partial example that, from an opposite point of view, the *well-formed* category does not allow to describe the (dynamic) semantics of programming languages if the syntactic domain is not used for the attributes.

Suppose we want to specify an AG that describes the semantics of a programming language according to the denotational approach. The basis in this approach consists in associating to a program structured through an (abstract) syntax a function to compute the memory states transformation obtained during its execution.

Let P denote the program (syntactic domain), F denotes a state function

$$F : P \rightarrow (S \rightarrow S)$$

[12] This point is worth noticing only in this framework as one can decide to make computations in the rule axiom.

where S is the memory state.

For specifying the semantics of statement composition, we shall write:

$$F[\![P_1;P_2]\!]\, i = F[\![P_2]\!](F[\![P_1]\!]\, i)$$

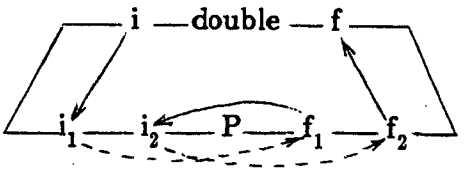
This definition can easily be interpreted by an attribute grammar with two attributes : *Inh* = { i } the initial state, and *Syn* = { f } the final state. One gets the production and the following definitions (all of them are identities), where each of P_1 and P_2 corresponds to the program derived from two occurrences of the non-terminal symbol P.

This translation comes directly from the correspondence described in [CF80a] between primitive recursive scheme (definition of F) and strongly non circular AGs (SNC).

Now suppose that we want to specify the semantics of a statement *double* that would be written in a denotational approach[13]:

$$F[\![\text{double } P]\!]\, i = F[\![P]\!](F[\![P]\!]\, i)$$

The corresponding transformation in term of an AG obliges us to make the following construction, with only one non-terminal symbol P:



where f_1 gives the result of $F[\![P]\!]\, i_1$ and s_2 that of $F[\![P]\!]\, i_2$, if the sub-grammar describing the trees rooted P is supposed to correspond to the function $F[\![P]\!]$. But such a construction is not allowed: $F[\![\text{double double } P]\!]$ cannot be represented in the AG because the non-terminal symbol double would have two attributes the first time and four attributes the second one.

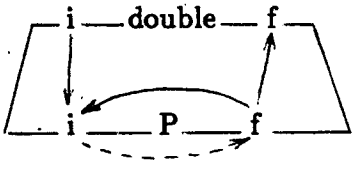
The only ways to solve this problem are to allow to use either the syntactic domain or a circular AG.

---

[13] Example taken from Engelfriet [Eng82b]

In the first solution, we consider that *double P* is in fact the repetition of *P;P*. But in such a case, we loose the notion of function defined over a syntactic tree (unique); in fact we are handling another object that describes the recursive function F itself. That kind of object has been described as dynamic AGs [Gan78b] and is studied more deeply in [CD84]. The syntax used does not strictly speaking correspond to the syntactic domain but take into account recursive functions calls.

In the second solution, we may introduce a finite recomputation of states by giving visit-sequences with different definitions of the same attribute. We shall write



with two definitions for i^P ($i^P=i^{double}$ and $i^P=f^P$) and visit-sequences:

visit(1,double) : $i^P := i^{double}$; visit(1,P) ;
$i^P := f^P$; visit(2,P) ; $f^{double} := f^P$;
finish(1,double).
visit(2,double) : $i^P := i^{double}$; visit(1,P) ;
$i^P := f^P$; visit(2,P) ; $f^{double} := f^P$;
finish(2,double).

As f^P depends on i^P in the sub-tree, the dependencies induce a circularity. This AG is not well-formed.

This example illustrates the need to introduce new classes of AGs, not well-formed, to process the area of denotational semantics. One could imagine specific evaluators for these classes. This point needs to be studied more deeply.

The two approaches we are going to define now show these limitations in a theoretical way: some tree transductions, more complex than those corresponding to well-formed AGs must be defined in order to reach the domain of denotational semantics [Eng82b]. The primitive recursive schemes have not enough power to describe denotational semantics. (For instance, the description of a *While* statement is out of their scope).

6.2. Attribute grammars characterization by means of tree transductions

This approach consists in defining automata which, taking a derivation tree as input, compute attribute values. These machines play for attribute grammars the role that push down stack automata play for context free grammars. Each machine defines by itself a class of transductions.

It is not possible to give here all the results obtained through this approach as the machines built for different classes of AGs are quite complex. Almost all definitions and results are stated in [LRS74, ERS80,Ful81, EF81a, EF82a, Kam82, Fil83c]. We give as an example the automaton corresponding to the pure-S AGs class.

A Tree Pushdown Transducer (TPT) is a transducer from a tree to a string, i.e. a machine that accepts a tree as input, uses as output a one-direction tape and a pushdown stack as memory. Figure 6.2.1. illustrates this definition and the way the machine is running: the stack and the path from the root to the visited point have the same length.

It has been shown in [ERS80,Fil83c] that TPTs correspond to pure-S AGs where attributes are strings.

Extensions of TPT with sets of registers associated to nodes of the input tree allow to define transductions from trees to any domain and to characterize larger classes of AGs, including circular AGs [Eng82b, Fil83c].

One result of this approach is to allow the comparison of transductions classes defined by AGs categories [EF81c, Fil83c]. We use here Filé's definitions.

An attribute grammar is supposed to define for any tree t a value for the attributes in *Syn* (Z), where Z is the grammar axiom. An AG defined over a domain D characterizes a transduction as the relation defined on L(G) x Domains(*Syn* (Z)), where L(G) is the language generated by G. We note this relation T_G.

Let C be an AG category and D a semantic domain, thus
$T(C,D) = \{TG \mid G \in C$ and D is the semantic domain of G$\}$
is the class of *C-transductions on D*. Considering two AGs categories C_1 and C_2, we say that C_1 has less *power* than C_2 if for any semantic domain D,
$T(C_1,D) \subseteq T(C_2,D)$. In the same manner, C_1 has the same *power* as C_2 if for any domain their transduction classes are equal.

Comparison results coming from this approach are given in Figure 6.2.2.

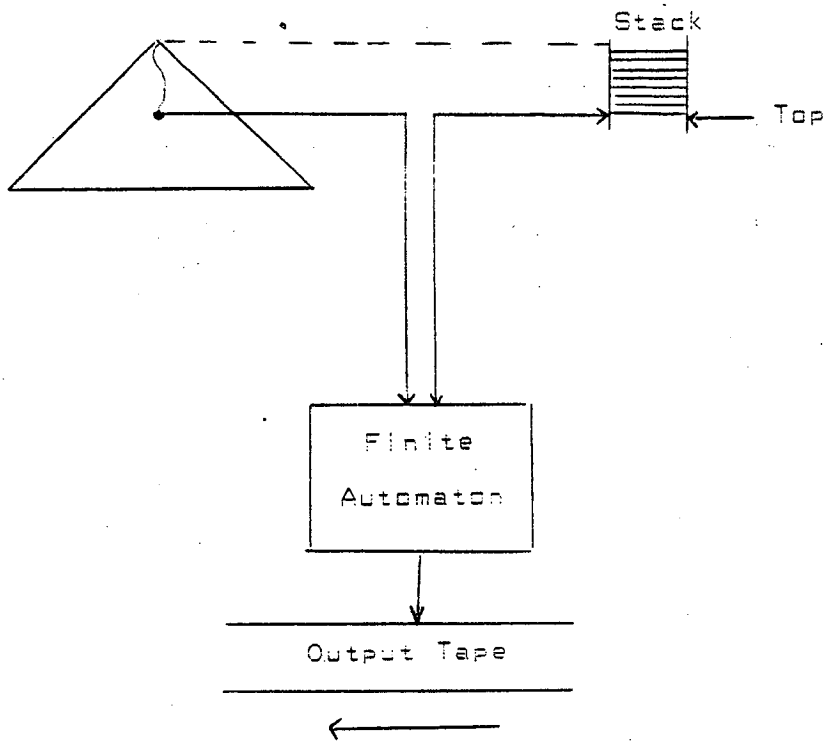


Figure 6.2.1. : Tree Pushdown Tranducer Automaton

One can observe first that Pure- and Simple- categories are of the same power. In particular, the fact that Pure-multi-visit (well formed) and Simple-multi-visit (l-ordered) are equivalent shows that any intermediate class does not increase the expressive power (in term of tree transductions). This result is not surprising as every well-formed AGs can be transformed into an l-ordered AGs.

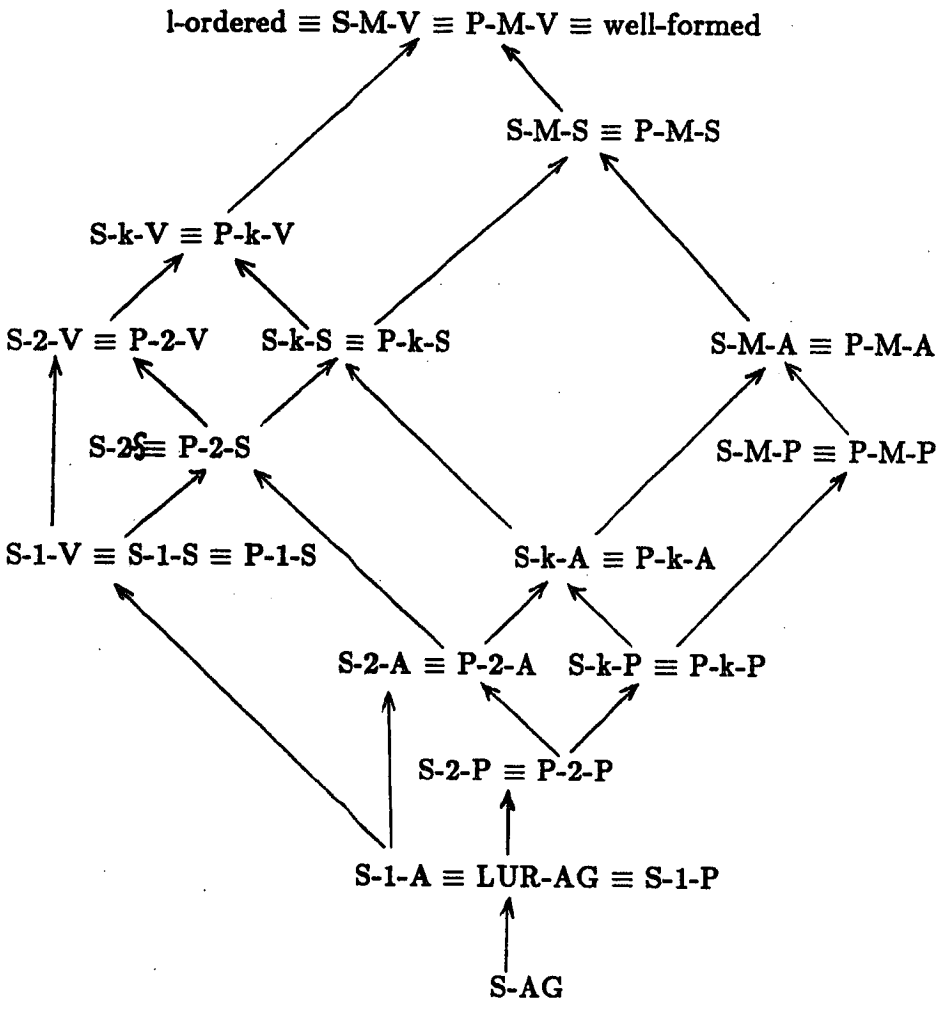


Figure 6.2.2.: Comparison between transductions power.

- arrows denote strict inclusion
- P-M-P : Pure-Multi-Pass
- S-M-S : Simple-Multi-Sweep

6.3. Attribute grammars characterization by means of program schemes

This method corresponds to an algebraic approach for attribute grammars. Most of the definitions and results are stated in [CM79, Kat80, Gan80a, Gan83d, May81, CF80a, Fra82, CD84, Der84].

Because of the extreme variety and complexity in definitions, we limit ourselves to make comments about Figure 6.3.1. which exhibits the

main results from the most general to the most particular ones.

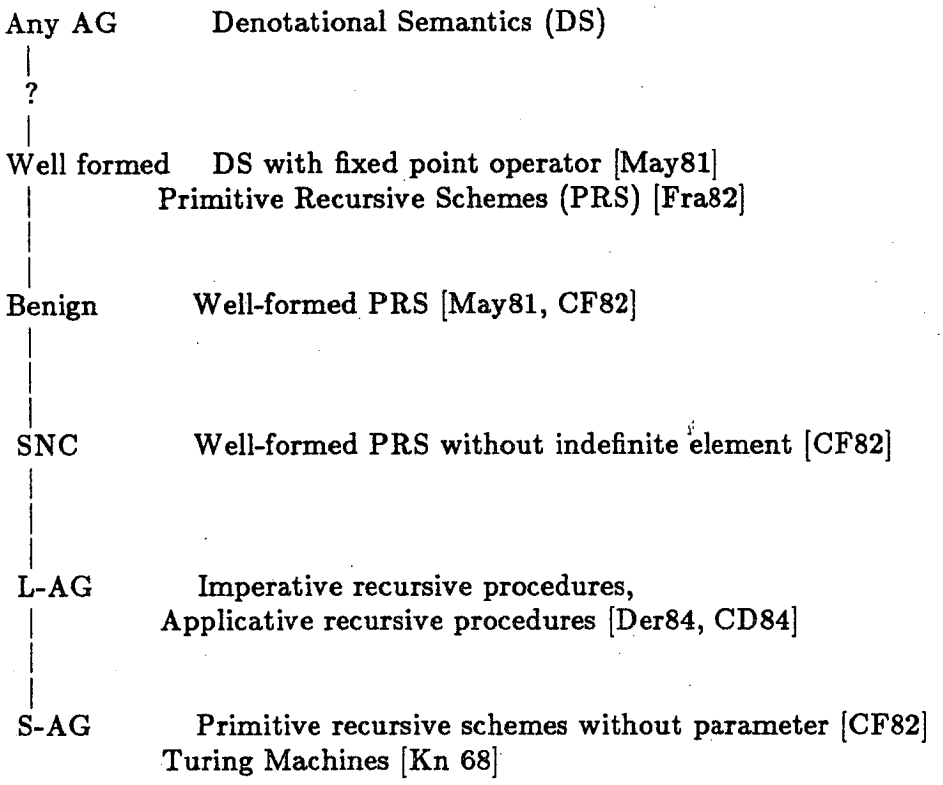


Figure 6.3.1. Classes of program schemes and classes of AGs

We do not give any AG sub-class that could modelize denotational semantics. Some points have been proposed [Eng82b, Fil83c] but remain a matter of research. Chirica and Martin [CM79] were the first to use the algebraic approach to formalize the semantics of attribute grammars and gave a proof method based on structural induction. The following studies that go deeper into this approach showed that, in the case of well-formed AGs, the formulation of algebraic semantics was simplified. Mayoh [May81] describes it by means of a system of equations without fixed point, showing that, by opposition to Chirica and Martin's formulation, it was possible in this case to get a denotational semantics without a fixed point operator. He also defines the *benign* AGs category, for which the mathematical semantics is drastically simplified. Franchi-Zannettacci [Fra82] showed that it is possible to associate to well-formed AGs specific program schemes called *primitive recursive schemes*, in the

sense that the recursion is defined over one parameter (which corresponds to what we have called the *syntactic domain*). He reached the same results as Mayoh and introduced with Courcelle [CF82] the category SNC characterized by *well formed* primitive program schemes. This approach leads to evaluators described in §3.3. The primitive schemes without parameter define a subclass where only the syntactic domain appears as a parameter. The authors use them to characterize the category of pure-S AGs (there is a class equivalence). This approach is complementary to the previous one but seems to be better adapted to study the problems of AGs transformations and equivalence. In particular, it is shown in [CF82] that pure-S AGs equivalence is a decidable problem. It is also decidable in the case of *quasi-pure-S* AGs where inherited attribute definitions (a, b) are only of the form (a, k) = (b, 0) with $k > 0$. This problem remains to study for other categories.

It is shown in [CD84] that imperative recursive procedures, i.e. recursive procedures without explicit loops or global variables but, maybe, local variables, can be transformed into L-AG whose grammar represents the possible call trees. This transformation applies also to applicative recursive procedures with call by value and call by name, under the condition to accept partially decorated trees.

These last results are not completely surprising as the L-AG category is the largest one for which attribute evaluation can be done in parallel with L-parsing. These transformations also show that if no limitation is imposed over domains, the L-AG category is large enough to describe the operational semantics of any applicative recursive function and, especially, the denotational semantics.

As a conclusion, we mention two algebraic approaches essentially devoted to compilers specification by means of semantic attributes : Ganzinger [Gan83] defines AGs by a language morphism described as an algebraic abstract types morphism. This allows to introduce a modular construction of a compiler specification in a framework where known correctness proofs techniques can be applied and allows to generate an attribute evaluator to produce the compiler. Other authors [JT83, CJ83, Pau82] automatically produce a compiler from a denotational semantics specification. In such an approach, an abstract object machine is first specified. In the system CGSG [Pau81], the attribute evaluator is mainly useful to build a source program representation in term of lambda-expressions which is thereafter reduced by appropriate modules. Only Ganzinger's approach can be viewed as an attempt to formulate AGs semantics with the precise target of specifying translators.

ISSN 0249-6399