An efficient recursive evaluator for strongly non-circular attribute grammars

Martin Jourdan

IRIA

CENTRE DE ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France
Tel. (3) 954 90 20

Rapports de Recherche

N° 235

AN EFFICIENT RECURSIVE EVALUATOR FOR STRONGLY NON-CIRCULAR ATTRIBUTE GRAMMARS

Martin JOURDAN

Octobre 1983

AN EFFICIENT RECURSIVE EVALUATOR FOR STRONGLY NON-CIRCULAR ATTRIBUTE GRAMMARS

Martin JOURDAN

INRIA
Domaine de Voluceau - Rocquencourt
BP 105
78153 Le Chesnay Cedex
FRANCE

This work was supported by a grant from Ecole Polytechnique.

Résumé :

Nous présentons quatre algorithmes qui permettent de construire des évaluateurs efficaces pour les grammaires attribuées fortement non-circulaires, qui forment une classe très large. A chaque attribut synthétisé, on associe une fonction qui prend pour arguments un arbre syntaxique et les valeurs des attributs hérités à la racine de cet arbre qui sont nécessaires pour le calcul du premier attribut. Ces fonctions sont mutuellement récursives selon la structure de l'arbre syntaxique. L'évaluateur produit réalise un "appel par nécessité" dynamique.

Abstract :

We present four algorithms to build efficient evaluators for strongly non-circular attribute grammars, which form a very large class. To each synthesized attribute, a function is associated, which takes as arguments a parse tree and the values of the inherited attributes of the root of this tree which are necessary for the computation of the former attribute. These functions are mutually recursive according to the structure of the parse tree. The produced evaluator implements a dynamic "call by need".

I - INTRODUCTION

Attribute grammars were introduced by D. Knuth [Knu68] to formalize the semantics of programming languages syntactically described by as context-free grammar and, in fact, to describe and execute any syntax-directed computation.

Since then, much research was done to get real efficiency in using this method. Its advantages in description (simplicity, declarative (non-procedural) method, local method) imply indeed many problems in execution : a priori, attributes evaluation is non-deterministic. Further more, the number of attribute instances grows rapidly high as the source text becomes somewhat long, and then a memory usage problem arises.

Many authors (see the bibliographies in [Rai80] and [MN82]) have saved computing time and memory usage at the price of a sometimes important restriction on the class of usable attribute grammars. We present here an evaluation method, very efficient relatively to computing time, applicable to a very large class of attribute grammars, called strongly non-circular. They were introduced by B. Courcelle and P. Franchi-Zannettacci in [CF82] .

Their work is briefly recalled in section 2. In section 3 we present four different algorithms to implement this method, with increasing efficiency of the evaluators produced. In section 4 we present a practical system using this method. At last we state some concluding remarks. A short example is presented extensively in appendix.

II - STRONGLY NON-CIRCULAR ATTRIBUTE GRAMMARS

In this section we briefly recall the works by Bruno Courcelle and Paul Franchi-Zannettacci [CF82, F82].

1) Notations and definitions

The theory developped in [CF82] is set up in the algebraic frame defined in [ADJ77] , [Gue81] and many other works.

An attribute grammar is a triple <G,T,D> composed by :

- a context-free grammar G = <T,N,Z,P> where :
  - T is the set of terminal symbols (ignored in the sequel) ;
  - N is the set of non-terminal symbols ;
  - $Z \in N$ is the start symbol or axiom ;
  - P is the set of productions, considered as a signature on N [ADJ77];

- an attribute system Γ (see below) of type <P,F> for some signature F on $\underline{A}$, where $\underline{A}$ is the set of attribute types ;

- an interpretation D, i-e an F-algebra.

Let N and $\underline{A}$ be two disjoint sets of symbols : they will represent respectively the non-terminals and the types of the attributes. Let P and F be two signatures on N and $\underline{A}$ respectively : they will represent the productions of the underlying grammar and the quators used in the semantic rules. An attribute system Γ of type <P,F> is composed by :

- a finite set A of symbols called attributes, disjoint union of $A^{(s)}$ (synthesized) and $A^{(h)}$ (inherited). Each attribute $b \in A$ has a type $\underline{b}$ in $\underline{A}$. For each $a \in A$, we give a subset $N_a \subset N$, the non-terminals to which a is attached. For each $S \in N$, we denote $A_S^{(s)} = \{a \in A^{(s)} / S \in N_a\}$, and the same for $A_S^{(h)}$ and A_S.

- for each $p \in P$, a set Γ_p of semantic rules : this is a set of equations satisfying the following conditions, where $p : S_0 \to S_1 S_2 \dots S_n$, $S_i \in N$:

  i) for each $a \in A_{S_0}^{(s)}$, there exists in Γ_p one and only one semantic rule defining a ($\in$) ; it is of the form :

$a(\varepsilon) = s\,[\ldots, z_i(\varepsilon), \ldots, b_j(\ell(j)), \ldots]$ where

- $\forall i,\ 1 \le i \le i_o,\ z_i \in A^{(h)}_{So}$;
- $\forall j,\ 1 \le j \le j_o,\ \ell(j) \in \{1,\ldots,n\}$ and $b_j \in A^{(s)}_{S_{\ell(j)}}$
- s is a term of M(F,X) such that the equation is well-formed.

ii) for each $k \in \{1\ldots n\}$ and for each $y \in A^{(h)}_{S_k}$, there exists in Γ_p one and only one semantic rule defining y(k), of the form :

$$y(k) = s\,[\ldots, z_i(\varepsilon), \ldots, b_j(\ell(j)), \ldots]$$

where s, z_i and b_j are like in i)

Expressions like $a(\varepsilon)$ and y(k) are called attribute occurences. Of course, the former denotes the attribute a of the left hand side non-terminal of production p, and the latter denotes the attribute y of the k-th non-terminal of the right hand side of production p.

Conditions i) and ii) imply that the attribute grammar be in Bochmann normal form. It is well-known that this is not a restriction.

Signature P expresses the compatibility between the context-free grammar G and the attribute system Γ. F expresses the compatibility between Γ and the interpretation D. In the sequel of this section, we shall focus on Γ, discarding G and D. All what we need is contained in P and F.

2) Computation of the values of the attributes

Let Γ be an attribute system of type <P,F> and t be an element of M(P) (a parse tree) such that $A^{(h)}_{root(t)} = \phi$. The semantic rules of Γ induce for this tree t a system of equations K(t), the unknowns of which are the values of the attribute instances at the different nodes of t. Evaluating the attributes is resolving this system of equations. As Knuth [Knu68], we define the semantic value of tree t to be the list of the values of the (synthesized) attributes of the root of t.

K(t) has one and only one solution if it is non-circular in the following sense : let w and w' two attribute instances (two unknowns) in K(t). The equations (in fact, the semantic rules of Γ) induce a dependency between w and w' in tree t (noted $w \xrightarrow[t]{} w'$) iff w' belongs to the right hand side term of the (unique) equation defining w. K(t) is then non-circular iff $w \overset{+}{\underset{t}{\not\to}} w$ for no attribute instance w.

The attribute system Γ is non-circular iff $K(t)$ is non-circular for each $t \in M(P)$. This property is decidable in an intrisically exponential time : see for instance [Jaz81].

The most natural method to solve a system $K(t)$ is to notice that the relation $\rightarrow_t$ is a partial order on the set of attribute instances. Thus we can compute them step by step, beginning with the greatest in the sense of $\rightarrow_t$. But this ordering depends on t, and this implies a dynamic sort of these attribute instances, what is an important cause of inefficiency : see for instance [L77].

3) Strongly non-circular attribute grammars.

Generally speaking, the order in which two attributes at a same node of a tree must be computed depends on that tree. So it would be interesting that this order be static, e.g depending only on the non-terminal to which these attributes are attached, or on the production in which they appear.

To achieve this, we approximate the relation $\overset{*}{\rightarrow}_t$ by a family of partial orders $\theta = \{\theta_S\}_{S \in N}$, where θ_S is a partial order on A_S : $a\ \theta_S\ b$ will then mean that $a(u)$ can be computed before $b(u)$ in any system $K(t)$ such that u is a node of t labelled by S. If such a family exists, satisfying some conditions, we say that Γ is strongly non-circular.

Let $p \in P$, $p : S_0 \rightarrow S_1, S_2 \ldots S_n$. Let $W_1(p) = \{a(i) \mid 1 \leq i \leq n, a \in A_{Si}\}$, $W_0(p) = \{a(\varepsilon) \mid a \in A_{S0}\}$ and $W_{(p)} = W_0(p) \cup W_1(p)$.

We define on $W_{(p)}$ the semantic dependency relation $\rightarrow_p$ by : $w \rightarrow_p w'$ iff w' belongs to the right hand side term of the semantic rule defining w.

We shall not use the partial orders families θ, but instead a function $\gamma = A^{(s)} \times N \rightarrow P(A^{(h)})$ such that $\gamma(a,S) \subset A^{(h)}_S$, if $a \in A^{(s)}_S$ else $\gamma(a,S) = \phi$. Such a function is called an argument selector. The argument selector associated to θ is defined by :

$$\gamma(a,S) = \{y \in A^{(h)}_S \mid y\ \theta_S\ a\} \text{ if } a \in A^{(s)}_S,$$
$$= \phi \text{ otherwise.}$$

To each argument selector γ, we associate a relation $\rightarrow_\gamma$ on $W_1(p)$ for each $p \in P$, such that :

$$w \rightarrow_\gamma w' \text{ iff } \exists\, i \in \{1, \ldots, n\},\quad a \in A^{(s)}_{S_i},\quad y \in A^{(h)}_{S_i},$$
$$w = a(i),\ w' = y(i) \text{ and } y \in \gamma(a,S_i)$$

We denote $\xrightarrow[p,\gamma]{}$ the relation $\xrightarrow[p]{} \cup \xrightarrow[\gamma]{}$ on $W_1(p)$.

An attribute system Γ is strongly non-circular if there exists an argument selector γ such that :

i) γ is closed : for each $p \in P$, $p : S_0 \longrightarrow S_1, S_2 \ldots S_n$, for each $y \in A_{S_0}^{(h)}$ and for each $a \in A_{S_0}^{(s)}$:

if $a(\varepsilon) \xrightarrow[p]{} y(\varepsilon)$

or if $\exists\, w,w' \in W_1(p)$,

$$a(\varepsilon) \xrightarrow[p]{} w \xrightarrow[p,\gamma]{*} w' \xrightarrow[p]{} y(\varepsilon)$$

then $y \in \gamma(a,S)$.

ii) γ is non-circular : for each $p \in P$ as in i), there exists no $w \in W_1(p)$ such that $w \xrightarrow[p,\gamma]{+} w$.

To get an intuitive grasp of what that means, we shall notice that, if γ is closed, then for each $t \in M(P)_S$, for each $y \in A_S^{(h)}$ and for each $a \in A_S^{(s)}$:

$$a(\varepsilon) \xrightarrow[t]{*} y(\varepsilon) \Longrightarrow y \in \gamma(a,S).$$

This means that γ contains, maybe in a pessimistic way, all the dependency relations that can arise in any parse tree between synthesized and inherited attributes of a same non-terminal.

Then, if an attribute system Γ is strongly non-circular, it is also (plainly) non-circular.

Futhermore, for each attribute system T, there exists a closed argument selector γ_0, minimal in the sense of set-theoretical functions. And Γ is strongly non-circular iff γ_0 is non-circular : see [CF82].

γ_0 is computed by the following algorithm :

Let us consider an argument selector γ as a set of triples :
$R = \{\langle y,a,S\rangle \;/\; y \in \gamma(a,S)\} \subset A^{(h)} \times A^{(s)} \times N$
The argument selector corresponding to such a set R will be denoted by $\gamma(R)$.

We define : $a(\varepsilon) \xrightarrow[p,\ \gamma(R)]{*} y(\varepsilon)$ iff

$$\begin{cases} a(\varepsilon) \xrightarrow[p]{} y(\varepsilon) \\ \text{or} \\ \exists\, w,w' \in W_1(p),\ a(\varepsilon) \xrightarrow[p]{} w \xrightarrow[p,\ \gamma(R)]{*} w' \xrightarrow[p]{} y(\varepsilon) \end{cases}$$

Then we have :

1. $R_0 \leftarrow \phi$; $i \leftarrow 0$;

2. repeat

$i \leftarrow i + 1$;

$R_i \leftarrow R_{i-1} \cup \{ \langle y,a,S \rangle$ / there exists $p \in P$, the left hand side of which is S and such that $y \in A_S^{(h)}$, $a \in A_S^{(s)}$ and $a(\varepsilon) \xrightarrow[p,\ \gamma(R_{i-1})]{*} y(\varepsilon) \}$

until $R_i = R_{i-1}$;

3. $\gamma_0 \leftarrow \gamma(R_i)$; stop.

The fourth step would be to check that γ_0 is non-circular. This algorithm, including the latter test is polynominal. It is equivalent [Jou 82] to algorithm A 21 by Lorho and Pair [LP 75], and can be improved by the methods presented in [DJL 83]. It not very different from the one by Kennedy and Warren [KW 76], where their "IO-graphs" play the role of our argument selector ; besides, the strongly non-circular is exactly their class of absolutely non-circular grammars.

What is the use of all that ? We know [CF 82] that the value of a synthesized attribute at a node of a tree depends only on :

- the subtree issued from that node ;
- the values of the inherited attributes at that node .

Among these latter attributes, we can discard those which will surely not participate to the computation, i-e. those which an not in the corresponding argument selector.

So, for each $S \in N$ and $a \in A_S^{(s)}$, we introduce a function $\varphi_{a,S}$ which takes as arguments :

- a derivation tree, the root of which is labelled by S (more precisely, by a production, the left hand side of which is S) ;
- the values of the attributes $y \in \gamma(a,S)$;

and which returns the value of attribute a of S, for this tree and these inherited attributes.

These functions can be defined by a primitive recursive scheme with parameters , which were introduced by Courcelle and Franchi-Zannettacci (see [CF 82], or better [F 82]).

A primitive recursive scheme Σ of type $\langle P,F\rangle$ with parameters (where P and F are two signatures on N and $\underline{A}$ respectively) is composed by :

i) a finite signature Φ on $N \cup \underline{A}$, called the set of functions symbols ; each $\varphi \in \Phi$ has an arity $\alpha(\varphi)$ in NA^* and a type $\sigma(\varphi)$ in $\underline{A}$;

ii) a set Y of variables, typed on $\underline{A}$, called parameters ;

iii) for each $p \in P$ and each $\varphi \in \Phi$ such that the left hand side of p is $\beta(\varphi)$, a defining equation $\Sigma_{p,\varphi}$ of the form :

$$\varphi(p(x_1, \ldots, x_n), y_1, \ldots, y_m) = \Gamma$$

where Γ is a term in $M(F \cup \Phi, \{x_1, \ldots, x_n, y_1, \ldots, y_m\})_{\sigma(\varphi)}$, the x_i are distinct variables typed on N, the y_i are distinct variables in Y, and the equation is well-formed.

If we recall that the elements of M(P) are derivation trees, then the above equation expresses a recursion according to the structure of the parse tree, since the x_i are the subtrees of $p(x_1, \ldots, x_n)$.

These primitive recursive schemes have the following properties :

- in any interpretation, they have one and only one solution ;

- this solution can be computed first symbolically in the initial algebra, and then interpreted ;

- this computation in the initial algebra can be done by using the associated term rewriting system, which is noetherian and confluent ;

- they are simply related to our strongly non-circular attribute systems.

Let Γ be an strongly non-circular attribute system of type P,F . We can construct a primitive recursive scheme $\Sigma(\Gamma)$ of same type which defines the functions $\varphi_{a,S}$. The proof of this theorem can be found in [CF 82] or in [F 82]. However, we recall here the construction of $\Sigma(P)$:

i) take $Y = A^{(h)}$ as set of parameters .

ii) for each $a \in A^{(s)}$ and each $S \in N_a$ such that $\gamma(a,S) = y_1, \ldots, y_m$, introduce a function symbol $\varphi_{a,S}$ of arity $Sy_1 \ldots y_m$ and of type $\underline{a}$. Let Φ be the set of all these function symbols.

iii) for each $\varphi_{a,\,S} \in \Phi$ and each $p \in N$, $p : S \rightarrow S_1 \ldots S_n$, define an equation :

$$\varphi_{a,\,S}\,(p(x_1, \ldots, x_n), y_1, \ldots, y_m) = \tau$$

where $x_i, \ldots, x_n$ are distinct variables of type $S_1, \ldots, S_n$ respectively and τ belongs to $M(F \cup \Phi, \{x_1, \ldots, x_n, y_1, \ldots, y_m\})_{\underline{a}}$.

This forms our scheme $\Sigma(\Gamma)$, of type <P,F>. Now let us show how we build the term τ. Let us consider the semantic rule in Γ_p defining $a(\varepsilon)$. It is of the form :

$$a(\varepsilon) = s\,[\ldots, z(\varepsilon), \ldots, b(j), \ldots]$$

where $z \in A_S^{(h)}$, $j \in \{1, \ldots, n\}$, $b \in A_{S_j}^{(s)}$, and s is an F-term. We have :

i) $z \in \gamma(a,S) = \{y_1, \ldots, y_m\}$;

ii) b(j) can be defined by a term $\tau_{b,j}$.
Then we take $\tau = s[\ldots, z, \ldots, \tau_{b,j}, \ldots]$.

Let us come back to ii) above. Let us define a set DEF of all attribute occurences c(k) with $k \in \{1, \ldots, n\}$ and $c \in A_{Sk}$, which can be defined by a term $\tau_{c,k}$. DEF is the increasing union of the sequence $(DEF_i)_{i \geq 0}$, built as follows :

$DEF_o = \phi$;

DEF_{i+1} is DEF_i augmented with the set of the c(k) which satisfy one of the (excluding) following conditions :

i) c is synthetized, $\gamma(c,Sk) = \{z_1, \ldots, z_m,\}$ and each of the $z_1(k)$, $\ldots$, $z_m,(k)$ belongs to DEF_i. Then we define :

$$\tau_{c,k} = \varphi_{c,Sk}(x_k, \tau_{Z1,k}, \ldots, \tau_{zm',k})$$

ii) c is inherited and the semantic rule of τ_p defining c(k) is : $c(k) = s'\,[\ldots, z(\varepsilon), \ldots, d(j'), \ldots]$ where s' is a F term, $z \in \{y_1, \ldots, y_m\}$, $j' \in \{1, \ldots, n\}$, $d \in A_{Sj'}^{(s)}$ and $d(j') \in DEF_i$. Then we define :

$$\tau_{c,k} = s'\,[\ldots, z, \ldots, \tau_{d,j'}, \ldots]$$

In [CF 82], it is shown that all the attribute occurences b(j) which appear in the right hand sides of the semantic rules of T_p are in DEF. This ends the construction of $\Sigma(T)$.

To summarize, the definition of a function $\varphi_{a,S}$ for production p is the semantic rule of τ_p defining $a(\varepsilon)$ in which :

i) the synthesized attributes of the sons (right hand side non-terminals) are replaced by the call to the corresponding function, to which we pass the right arguments : the subtree and the inherited attributes in the argument selector, replaced (recursively) by their definition ;

ii) the inherited attributes of the left hand side are left in place since they are parameters.

Courcelle and Franchi-Zannettacci have shown that such a construction is correct, and provides an efficient way to evaluate the attributes, as the next sections will show.

In [F 82], this construction is extended to broader classes of attribute grammars. But this forces to introduce non-determinism and / or undefined terms, and this is inapplicable to any efficient implementation.

To compute the semantic value of a derivation tree, we have only to call successively the functions $\varphi_{a,Z}$ corresponding to the attributes of the axiom, passing to them as a parameter that tree. These functions have no other parameter, since the axiom has no inherited attribute

We must notice here that the class of strongly non-circular grammars is very broad, including all the practical cases of compilation and meta compilation.

Kennedy and Warren [KW 76] and Katamaya [Kat 80] also studied this class of grammars, and reache nearly the same results.

III - BUILDING AN EVALUATOR

1) First method

The evaluator which we build is strictly derived from the primitive recursive scheme associated to a strongly non-circular attribute grammar. It is thus a set of functions $\varphi_{a,S}$ ($S \in N$, $a \in A_S^{(s)}$), the arguments of which are :

- a syntactic tree, the root of which is labelled by a production p, the left hand side of which is S ;

- the values of attributes $y \in \gamma(a,S)$, computed in the upper context of the considered node.

For the practical implementation, we stay close to the theory, in the sense that we demand that the attributes be defined by unctions rather than by sequences of instructions. Using a functionnal language frees us from memory allocation problems, which is not the case of the implementation completed by Obaïd [O 82].

Our semantic rules are thus written in a Lisp-like language ; each of them is an expression, the result of which will be "assigned" to the attribute defined. These expressions contain generally attribute occurences.

The functions $\varphi_{a,S}$ will be produced in the same Lisp-like language by the following algorithm, composed by a procedure "print-attr-def" and a body.

```
procedure print-attr-def (attribute, production, position) ;
{ "position" is the index of the non-terminal in the production considered :
0 for left hand side, 1, 2,..., n for right hand side. Each alternative
is a separate production }

if (position = 0 and attribut ∈ A(h)) then
   {inherited attribute of left hand side : a parameter}
        print ("attribute")
else if (position > 0 and attribute ∈ A(1)) then
   {synthetized attribute in right hand side : generate the corresponding call}
        NT = non-terminal (production, position) ;
        print ("(phi-attribut-NT
                    (subtree tree position)") ;
```

```
        {the function "subtree" returns the subtree of "tree"
               of index "position"}
        for each y ∈ γ_o (attribut, NT) do
               print-attr-def (y, production, position)
        end for ;
        print (")")
else {there is a semantic rule}
        print this semantic rule, replacing each attribute occurence b(i)
              by print-attr-def (b, production, i)

endif
endif
end print-attr-def ;

{body}
for each a ∈ A^(s) do ;
   for each S ∈ N_a do
        print ("(defun  phi-a-S (tree  ") ;
        for each y ∈ γ_o (a,S) do
              print ("y")
        end for ;
        print (")") ; {end of parameter list}
        print ("(cond ") ; {beginning of a switch driven by the production
                           which labels the root of "tree" ; its left
                           hand side is S}

        for each p ∈ P, LHS (p) = S do
             print ("((equal (root tree) p)") ;
             print-attr-def (a, p, 0) ;
             print (")")
        end for ;
        print ("))")  {end of the switch and the function}

   endfor ;
endfor ;
{define the function returning the semantic value of the tree}
print ("(defun semantic-value (tree)
              (list ") ;
for each a ∈ A_Z^(s) {Z = start symbol} do
        print ("(phi-a-Z tree)")
endfor ;
print ("))").
```

The recursive calls to "print-attr-def" implement automatically the construction of the set DEF. We must point out too that this algorithm accepts without modification grammars which are not in normal form : the rewriting is done by these recursive calls. Noticing too that the algorithm to compute γ_0 also accepts non-normal form grammars, this contraint is completely avoided.

The "compilation" of a source text proceeds then in two steps :

- scanning and parsing with construction of a parse tree ;
- call to the function "semantic-value", passing this tree as parameter.

The evaluator produced by this algorithm has the intrisic qualities and disadvantages of the theoretical method of Courcelle and Franchi-Zannettacci. Let us quote, among the advantages :

- very simple and natural method ;
- dynamic evaluation by need : let a semantic rule be of the (Lisp) form :

```
(cond (a(X) (...c(Y) ...))
      (t b(Z)))
```

To evaluate this, we start by evaluating a(X) ; if the value is "nil" (false), we compute b(Z) without computing c(Y) ; else we compute c(Y) without computing b(Z). This inclusion of the control flow of the semantic rules into the control flow of the evaluation is an important improvement with respect to evaluators which compute all the attribute occurences which appear in the right hand side of a semantic rule before computing it, like the one of Jalili and Gallier [JG 83] and many others ;

- no useless computation : we compute only the attributes (dynamically) necessary to compute those of the root of the tree ;
- no memory usage to store the attributes ;
- height of the evaluation stack proportionnal to the height of the parse tree ;

and among the disadvantages :

- too many useless computations : if the complete dependency (acyclic) graph for a given tree is not a tree, i-e a same attribute is used in two or more different semantic rules (and this is often the case),

that attribute and all the ones on which it depends are recomputed for each use : this leads to prohibitive computing times if the grammar is somewhat big [Jou 82].

2) Decorating the parse tree

To overcome this problem, a great one we must admit, the simplest way is to store the values of the attributes that we compute. The most suited place for this store is at the corresponding node of the tree : this is called a decoration of the tree.

The structure of the primitive recursive scheme is unchanged but each function phi-a-S is modified as follows :

```
function phi-a-S (tree y1 ... ym) :
    if the value of a is already attached to the root of "tree"
{already computed} then return (this value)
    else compute it according to the production labelling the root
         of "tree", as before ;
         store it at the root of "tree" ;
         return (it)
    endif ;
end phi-a-S ;
```

Thus, at each call of function phi-a-S, only the inherited attributes in γ_0 (a,S) are recomputed ; a itself and the attributes on which it depends in the subtree are computed only the first time.

Of course the memory size used is greater for this version of the algorithm than before. But in Lisp it is not a dramatic problem since, on one hand, an assignment is only a copy of pointers and, on the other hand, the list structures are often shared : for instance if we have :

a(X) = (cons 'foo b(Y)),

the room used to store a(X) and b(Y) together will be only one cell more than the one used to store b(Y) alone. We can draw a parallel between this usage of the memory and the one proposed by Räihä [Räi 79], without deallocation.

This new construction thus achieves, together with Lisp, a good compromise between computing time and memory usage.

We must point out the inherited attributes in $\gamma_0(a,S)$ are recomputed for each call of phi-a-S. In that sense our method is not optimal. But

- it is not a serious problem, since generally they depend "simply" on synthesized attributes which are themselves stored ;

- this is inherent to the method and cannot be easily improved.

The procedure "print-attr-def" can be modified to take advantage of the decoration of the tree when the grammar is not on normal form : if in the definition of a synthesized attribute a(S), appears an attribute occurence b(S) where b is synthesized, we do not expand the definition of b(S) but rather generate a call to the corresponding function.
An other boolean parameter of "print-attr-def" will distinguish in the course of the construction between the definitions which must be expanded and those which may not be (see next section). That implementation is described with details in [Jou 82].

3) Compressing the evaluation functions

Let us draw our attention onto the structure of an evaluation function $\varphi_{a,S}$:

```
function phi-a-S (tree y1... ym) ;
    if the value of a is stored at the root of "tree" then
       return (it).
    else case production (tree) is
         p1 : a =     ;
         p2 : a =     ;
                :
         pn : a =     ;
         endcase ;
         store a  at the root of "tree" ;
         return (a)
    endif
end phi-a-S ;
```

If we compare several functions phi-a-S_1, phi-a-S_2, ..., phi-a-S_n with the same a, we notice that the different choices of the case are exclusive among these different functions, since they are the names of the productions, the left hand sides of which are S_1, S_2, ..., S_n, and they are of course exclusive. So it comes to the mind to group all these functions together, and this saves the room (in the code) corresponding to the common part of these functions, i.e the part edged with } above.

The difficulty which then arises is that all these functions have not necessarily the same parameter list, even with the same a. So we modify the header of the function :

function phi-a (tree parameter-list) ;

where "parameter-list" is the list of name-value couples (association list) of the inherited attributes $y_1,\ldots,y_m$. This list will be different according to the (old) function phi-a-S which would have been called. It is built up by the functions which call phi-a, and the different necessary attributes an extracted from this list in the different choices of the case.

Here is the new construction algorithm :

```
procedure print-attr-def (attribute, production, position, to-be-expanded) ;
    if (position = 0 and attribute ∈ A(h)) then
            print ("(cdr(assoc 'attribute parameter-list))")
            {extract the value of "attribute" from parameter list}
    else if (position > 0 and attribute ∈ A(s)) then
            print ("(phi-attribute (subtree tree position)
                                    (list  ") ;
            {build up assoc-list of inherited attributes}
            for each y ∈ γo(attribute, non-terminal (production)) do
                    print ("(cons 'y ") ;
                    print-attr-def (y, production, position, false) ;
                    print (")").
            endfor ;
            print ("))").
```

```
else if ((position = 0 and attribute ∈ A(s)) and not to-be-expanded) then
        print ("(phi-attribute tree parameter-list)")
      {not in normal form, but do not expand the definition. The called
       function will find the good parameters in "parameter-list" since
       a(ε) ⟶p b(ε) with a,b ∈ AS(s) imply γ(b,S) ⊂ γ(a,S)}

else {there is a definition}
     print this definition, replacing each attribute occurence b(i) by
print-attr-def (b,production,i,false)

endif
endif
endif
end print-attr-def ;

{body}
for each a ∈ A(s) do
    print ("(defun phi-a (tree parameter-list)
                  (cond ((already-computed 'a tree)
                         (extract 'a tree))
                        (t(let((a(cond ") ;

    for each S ∈ Na do
          for each p ∈ P, LHS(p) = S do
            print ("((equal (root tree) p)") ;
            print-attr-def (a,p,o,tree) ;
            print (")")
          endfor ;
    endfor ;


   print (")))
          (store a tree)
          a))))")

endfor ;
{semantic value}
print ("(defun semantic-value (tree)
              (list ") ;
for each a ∈ AZ(s) do
      print ("(phi-a tree nil)")  {empty parameter-list}
endfor ;
print ("))").
```

In this new version, we lose the check of the number and types of the parameters of each (old) function. But the initial construction assures there will be no problem, especially using an association list, where the attributes are named.

With respect to the previous version, we save in the average 30 % of code size, without increasing the computation time (unsignificant figures). In fact, we used a trick which allows us to recognize true identities (a(X) = b(Y)) and to transform almost every form : (cons 'X (cdr (assoc 'X parm-list))) into (assoc 'X parm-list), which is (of course) equivalent.

4) Eliminating simple productions

We meet often, in practical examples of attribute grammars, simple productions : they are of the form

$$S_0 \longrightarrow S_1$$

and such that no semantics be attached to them. More precisely, S_0 and S_1 have the same set of attributes and all the semantic rules are identities between the corresponding attributes of S_0 and S_1. These productions carry no semantics and have only a syntactical purpose.

Here is a classical example :

```
1 : E = E " + " T ;
    val (E) = (+ val(E') val (T))
    symtab (E') = symtab (E)
    symtab (T)  = symtab (E)

2 : E = T
    val (E) = val (T)
    symtab (T) = symtab (E)

3 : T = T " * " P ;
    val (T) = (* val (T') val (P))
    symtab (T') = symtab (T)
    symtab (P)  = symtab (T)
```

```
4 : T = P
    val (T) = val (P)
    symtab (P) = symtab (T)

5 : P = id
    val (P) = (value id symtab (P))

6 : P = "(" E ")" ;
    val (P) = val (E)
    symtab (E) = symtab (P)
```

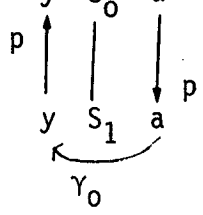
This is a (part of) attribute grammar describing arithmetic expressions, where operator precedence is treated by syntactical means.

Productions 2 and 4 are simple productions ; production 6 is not a simple production, because of the two parentheses.

Lemma : for a simple production $S_0 \longrightarrow S_1$ (recall : $A_{S_0} = A_{S_1}$), we have : $\forall a \in A^{(s)}_{S_0} = A^{(s)}_{S_1}$, $\gamma_0(a,S_1) \subset \gamma_0(a,S_0)$.

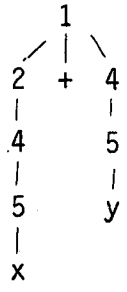
Proof : a small drawing is better than a long talk :

$y \in \gamma_0(a,S_1) \implies$



$\implies y \in \gamma_0(a,S_0)$

Let us come back to our example. Let the source text to analyze be "x + y", with "x" and "y" lexically recognized as "id" 's.
The corresponding parse tree is :



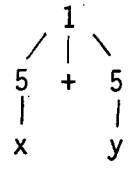
And the Lisp function phi-val could be, omitting the code to decorate the tree :

```
(defun phi-val (tree parm-list)
      (cond ((equal (root tree) 1)
             (+ (phi-val (subtree tree 1)
                         (list (cons 'symtab
                                     (cdr (assoc 'symtab
                                                 parm-list)))))
                (phi-val (subtree tree 3)
                         (list (cons 'symtab
                                     (cdr (assoc ' symtab
                                                 parm-list)))))))
            ((equal (root tree) 2) ; identity
             (phi-val (subtree tree 1)
                      (list (cons 'symtab
                                  (cdr (assoc 'symtab parm-list))))))


            ((equal (root tree) 5)
             (value (subtree tree 1) ; the "id"
                    (cdr (assoc 'symtab parm-list))))


                          ))
```

Applied to the above tree, phi-val will, of course, return the expected result. If now we assume that we can reduce the simple productions without constructing the corresponding nodes, we shall get the following tree :



and the same function phi-val will return the same result, but with less computations, and less memory use. The above lemma assures that all the needed parameters are present in "parm-list". There can be more, but it 's not bothering.

In that example, the code produced for production 1 assumed that (subtree tree 1) returned a tree labelled by E (1 or 2). Here we shall get a tree labelled by T (3 or 4) or P (5 or 6), but this causes no problem since function phi-val is compressed.

If we have a parser that can detect simple productions, like SYNTAX [Bou 80], then the produced parse trees will be valid - unless some syntactical error arises - for the compressed evaluation functions.

Note that Räihä [Räi 79] already used such a method.

Practical measures show that we can save up to 27 % of memory size used for the tree (decorated or not) and up to 15 % computation time (parser + evaluator).

This ends up the presentation of our algorithms.

IV PRACTICAL IMPLEMENTATION

We completed an implementation of that latter algorithm to construct an evaluator for a strongly non-circular attribute grammar, on the Multics system at INRIA.

1) Attribute grammar description format

An attribute grammar description is composed by :

- a header, which is empty or composed by a sequence of Lisp forms (auxiliary function definitions, variables settings, etc...) which will be copied unchanged in the file which will contain the evaluation functions ; that file will be loaded in the run-time system ;

- a list of attribute declarations : their name, their type and the list of non-terminals to which they will be attached ; these declarations may be incomplete, and then the system will try to generate the lacking ones : see next paragraph ;

- the list of productions, in the BNF-like format of SYNTAX, with the list of semantic rules attached to them ; these rules are of the form :

    a (S) = Lisp form which may contain attribute occurences
    or a (S) = b (T) (identities) ;

the system accepts non-normalized grammars ; some semantic rules may be ommitted : the system will then try to generate them using a set of default rules : see next paragraph ;

- a list, maybe empty, of terminals used to improve syntactical error recovery.

Two attributes are predefined :

- ptext, which is attached only to terminals, returns their text as defined by lexical analysis, under the form of an interned Lisp atom ;

- source-index, which applies to both terminals and non-terminals, returns the index in the source text of the beginning of the terminal sentence generated by the symbol ; this allows, together with the predefined function "put-error", to write messages (e-g, error ones) in the listing produced by the analysis, with well-placed markers.

These attributes imply of course no semantic dependencies.

For each semantic rule, we can specify a list of synchronization attributes : these are attribute occurences which we want that they will be computed before the defined attribute. We force the evaluation of the formers before the one of the latter, by placing their "definition" in the evaluation function before the effective computation. On the other hand, they are added to the list of dependencies for the strong non-circularity test. This allows the user to have a more explicit control of the order of attribute evaluation. But this can make the grammar not strongly non-circular, and cost much computation time.

At last, we can write an empty definition for an attribute which will be unused in some production. Evaluating an attribute with an empty definition will cause a run-time error, but normally this will not happen if these empty definitions are used deliberately.

2) Generating missing declarations and definitions.

An attribute has one of four types :

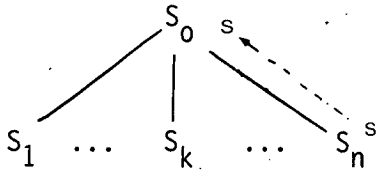
- synthesized ;
- inherited ;
- global : it is used as an inherited attribute, but has not the same behaviour with respect to missing definitions generation ;
- mixt : a mixt attribute "attr" is a pair composed of a synthesized attribute "s.attr" and an inherited are "h.attr", which behave normally except for missing definitions generation ; of course the two components of this pair are attached to the same non-terminals.

To get the list of all attribute declarations, we first use the ones supplied by the user, and then we scan the semantic rules, looking at each attribute occurence : if the attribute is already known, no problem. Else we declare it, with a type which depends on the non-terminal and on the place where we find that occurence. If it is the left hand side of a semantic rule, then if the non-terminal is the left hand side of the production then it will synthesized, else inherited. If that occurence appears in the right hand side of a semantic rule, it is the opposite. The two components of a mixt attribute carry their proper type, and we declare both of them at the same time. Then we add the non-terminal to the list of those to which the attribute will be attached. If the attribute is known, but the non-terminal does not belong to its list, then we do the addition.

Then we loop over every simple production A = B with no significant semantics and attach to A each inherited, global or mixt attribute of B, and attach to B each synthesized attribute of A, and that until convergence.

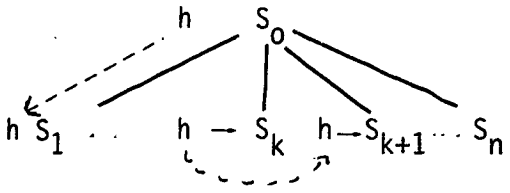
Then we are ready to generate the missing definitions (semantic rules). The following default rules are taken from Lorho [L 77]. In the sequel, the "father" will refer to the left hand side non-terminal of the production, and the "sons" to the right hand side ones.

A synthesized attribute of the father, the definition of which is missing, will be defined as equal to the corresponding attribute (the one with same name) of the right most son if it exists ; else it is



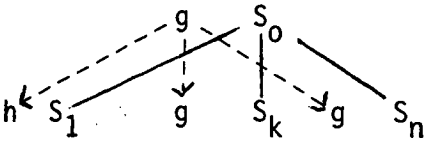
or error

An inherited attribute of a son is equal to the corresponding attribute (if this one exists) of its father or left brother, according as this son is the leftmost or another ; else error :



or error

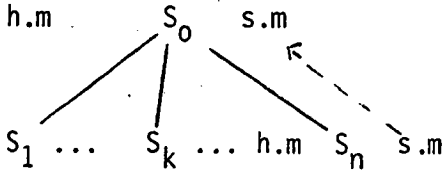
A global attribute of a son will be equal to the corresponding attribute of its father if this one exists ; else error :



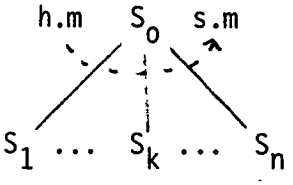
or error

The purpose of a global attribute is thus to be transmitted to each node of a subtree. It is the case for instance of the attribute "symtab" of the example of paragraph 3-4.

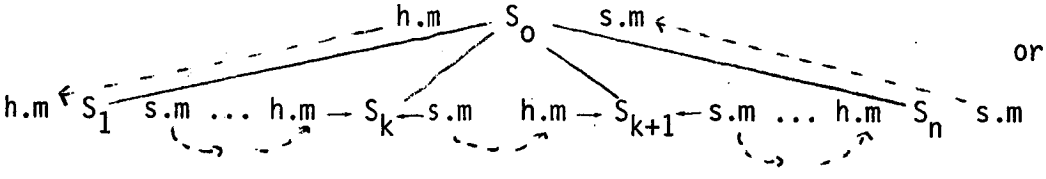
The synthesized component of a mixt attribute of the father is equal to the synthesized component of the corresponding attribute of the rightmost son if this one exists ; else it is equal to the inherited component of the father 's attribute.



or



The inherited component of a mixt attribute of the leftmost son is equal to the inherited components of the corresponding attribute of the father if it exists ; else error. The inherited component of a mixt attribute of another son is equal to the synthesized component of its left brother if it exists ; else error ;



or error

The purpose of a mixt attribute is thus to circulate from left to right, in a list for instance, being modified at each non-terminal. It is the case for instance of the attribute "symbol-table" in a declaration list.

To define the "rightmost son" or the "left brother", we do not take into account purely syntactical non-terminals having no attributes, e.g <GOTO> which could derive either in "GO TO" or in "GOTO".

These rules decrease in a very substantial manner (more than 50%) the number of semantic rules that the user must write for "usual" attribute grammars. As a special case, for simple productions, no semantic rule need to be written.

3) The constructor

The constructor reads in an attribute grammar description, checks its strong non-circularity and constructs the corresponding evaluation functions.

It is written in PL/1 (about 3700 lines), using meta-compilation techniques described in [BR 81].

It has been used heavily for now a couple of months, and seems much satisfying.

4) The run-time system

It is a quite small Lisp subsystem which :

- loads the file containing the evaluation functions (after compilation or not) ;
- calls the scanner and parser of SYNTAX and builds up the derivation tree ;
- calls the function "semantic-value", passing that latter tree as parameter ;
- gets back the root attributes and outputs them.

A number of features (options, predefined variables, ...) make its use pleasant.

The Lisp structure of the parse tree is as follows :
non-terminal :

```
(production-number
   source-index
   assoc - list of the attributes (initially nil)
   son - 1
   ...
   son - n)
```

terminal :

(ptext source-index)

Thus, the functions "subtree" and the two attributes "ptext" and "source-index" translate to a simple sequence of "car" and "cdr".

V - Conclusion

We presented four algorithms to build efficient evaluators for strongly non-circular attribute grammars, working by recursion according to the structure of the parse tree.

They implement a dynamic evaluation by need, thus decreasing substantially the number of attributes to compute to get the semantic value of a text. To our knowledge, it is the first method including that feature without introducing non-determinism.

The Lisp implementation which was completed is quite satisfying and proves the validity of the method, either with respect to computing time or with respect to memory usage.

Acknowledgments : this work would have been impossible without the help of Bruno Courcelle and Paul Franchi-Zannettacci, Pierre Boullier, Bernard Lorho and all the members of the "Langages et Traducteurs" project at INRIA.

VI - Références

[ADJ 77] ADJ group (Goguen et al.), "Initial Algebra Semantics and Continuous Algebras", JACM 24, (1977), pp 68 - 95.

[Bou 80] P. Boullier, "Génération Automatique d'Analyseurs Syntaxiques avec Rattrapage d'Erreurs", Journées Francophones sur la Production Assistée de Logiciels, Genève (1980).

[BR 81] P. Boullier and K. Ripken, "Building an ADA Compiler Following Meta-Compilation Methods", Séminaires INRIA "Langages et Traducteurs" 1978 - 1981, INRIA, Rocquencourt (1981), pp 99 - 140.

[CF 82] B. Courcelle and P. Franchi-Zannettacci, "Attribute Grammars and Recursive Program Schemes", Theoretical Computer Science, 17 (1982), pp 163 - 191 and 235 - 257.

[DJL 83] P. Deransart, M. Jourdan and B. Lorho, "Speeding up Circularity Tests for Attribute Grammars", report RR - 211, INRIA, Rocquencourt (1983).

[F 82] P. Franchi-Zannettacci, "Attributs Sémantiques et Schémas de Programmes", Thèse d'Etat, Université de Bordeaux I (1982).

[Gue 81] I. Guessarian, "Algebraic Semantics", Lecture Notes in Computer Science 99, Springer-Verlag (1981).

[JG 83] F. Jalili and J.T. Gallier, "A General Incremental Evaluator for Attribute Grammars", draft, University of Pennsylvannia, Philadelphia (1983).

[Jaz 81] M. Jazayeri,"A Simpler Construction for Showing the Intrisically Exponential Complexity of the Circularity Problem for Attribute Grammars", JACM 28 (1981), pp 715 - 720.

[Jou 82] M. Jourdan, "Un Evaluateur Efficace pour les Grammaires Attribuées Fortement Non-Circulaires", DEA report, Université de Paris VII (1982).

[Kat 80] T. Katamaya, "Transformation of Attribute Grammars into Procedures", report CS - K - 8001, Department of Computer Science, Tokyo Institute of Technology (1980).

[Knu 68] D. Knuth, "Semantics of Context-Free Languages", Math-Systems Theory 2 (1968), pp 127 - 145.

[KW 76] K. Kennedy and S. Warren, "Automatic Generation of Efficient Evaluators for Attribute Grammars", Procs. of 3rd ACM POPL, Atlanta (1976), pp 32 - 49.

[L 77] B. Lorho, "Semantic Attributes Processing in the System DELTA", in "Methods of Algorithmic Language Implementation", Ershov and Koster (eds.), Springer Verlag (1977), pp 21 - 40.

[LP 75] B. Lorho and C. Pair, "Algorithms for Checking Consistency of Attribute Grammars", in "Proving and Improving Programs", Colloque INRIA, Arc - et - Senans (1975), pp 29 - 54.

[MN 82] H. Meijer and A. Nijholt, "Translator Writing Tools Since 1970 : A Selective Bibliography", ACM SIGPLAN Notices 17,10 (1982), pp 62 - 72.

[O 82] A. Obaïd, "Evaluateurs Optimisés pour les Grammaires d'Attributs Fortement Non-Circulaires", Thèse de 3ème cycle, Université de Bordeaux I (1982).

[Räi 79] K. J. Räihä, "Dynamic Allocation of Space for Attribute Instances in Multi-Pass Evaluators for Attribute Grammars", ACM SIGPLAN Notices 14,8 (1979), pp 26 - 38.

[Räi 80] K. J. Räihä, "Bibliography on Attribute Grammars", SIGPLAN Notices 15,3 (1980), pp 35 - 44.

APPENDIX

A full example

This example is the universally known example of binary numbers [Knu68]. It has the advantage to be very short, easy to understand and significant.

Here is the description of the attribute grammar in the format for our system. The non-terminals are written between angle brackets (< and >), and the attributes begin with an ampersand (&).

```
        * An auxiliary function:
        (defun 2** (n)
               (expt 2.0 n))
        $
        * Attributes declarations; s = synthesized, h = inherited.
        s#&v(<N>, <L>, <B>);
        h#&s(<L>, <B>);
        s#&d(<L>);
        $

1       <N> = <L> . <L> ;
        &v(<N>) = (plus &v(<L>) &v(<L>'))
        &s(<L>) = 0
        &s(<L>') = (minus &d(<L>'))

2       <N> = <L> ;
            &v(<N>) = &v(<L>) ; this rule was generated by the
                              ; system.
        &s(<L>) = 0

3       <L> = <L> <B> ;
        &v(<L>) = (plus &v(<L>') &v(<B>))
        &s(<B>) = &s(<L>)
        &s(<L>') = (1+ &s(<L>))
        &d(<L>) = (1+ &d(<L>'))

4       <L> = <B> ;
            &s(<B>) = &s(<L>)
            &v(<L>) = &v(<B>)
        &d(<L>) = 1.

5       <B> = 0 ;
        &v(<B>) = 0

6       <B> = 1 ;
        &v(<B>) = (2** &s(<B>))

        $
        $
```

Here is the result of the strong non-circularity test, i.e. the argument selector:

```
On the non-terminal <N> :
    the attribute &v depends on no inherited attribute.

On the non-terminal <L> :
    the attribute &v depends on: &s
    the attribute &d depends on no inherited attribute.

On the non-terminal <B> :
    the attribute &v depends on: &s
```

And here is the set of Lisp functions generated for this grammar.

```
(declare (*expr put_error) ; Lisp declarations supplied by the system
         (*lexpr put-error)
         (fixnum prodnum)
         (special ???-msg working-dir-name process-dir-name
                  translator-name stripped-program-entry-name))

(defun 2** (n) (expt 2.0 n)) ; the auxiliary function is just copied.
```

```
(defun &v (tree param-list)
      param-list                                  ; if unused
      (let ((couple (assq '&v (caddr tree))))
           (cond (couple (cdr couple))            ; already computed?
                 (t (let
                      ((valeur                    ; else compute it
                         (let ((prodnum (car tree)))
                              (cond ; "case" according to the production
                                ((= prodnum 1.)
                                 (plus
                                   (&v (cadddr tree)  ; first son
                                       (list (cons '&s 0.)))
                                                  ; parameter-list
                                   (&v (cadddr (cdr (cdr tree)))
                                                  ; third son
                                       (list      ; parameter-list
                                         (cons '&s
                                               (minus
                                                 (&d
                                                   (cadddr
                                                     (cdr (cdr tree)))
                                                   (list)))))))))
                                ; &d depends on no attribute, so empty param-list
                                ((= prodnum 2.)
                                 (&v (cadddr tree) (list (cons '&s 0.))))
                                ((= prodnum 3.)
                                 (plus
                                   (&v (cadddr tree)
                                       (list
                                         (cons '&s
                                               (1+
                                                 (cdr
                                                   (assq '&s
                                                         param-list))))))
                                                  ; to retrieve parameter &s
                                   (&v (cadddr (cdr tree))
                                       (list (assq '&s param-list)))))
                                ((= prodnum 4.)
                                 (&v (cadddr tree)
                                     (list (assq '&s param-list))))
                                ((= prodnum 5.) 0.)
                                ((= prodnum 6.)
                                 (2** (cdr (assq '&s param-list))))
                                (t (error ???-msg))))))
                                                  ; if the tree is inconsistent
                        (let ((couples-list (caddr tree)))
                             (cond ((car couples-list) ; store in the tree
                                    (rplacd couples-list
                                            (cons (cons '&v valeur)
                                                  (cdr couples-list))))
                                   (t (rplaca (cddr tree)
                                              (ncons (cons '&v
                                                           valeur))))))
                        valeur)))))                   ; return the value
```

```
(defun &d (tree param-list)
      param-list
      (let ((couple (assq '&d (caddr tree))))
           (cond (couple (cdr couple))
                 (t (let ((valeur
                           (let ((prodnum (car tree)))
                                (cond ((= prodnum 3.)
                                       (1+ (&d (cadddr tree)
                                               (list))))
                                      ((= prodnum 4.) 1.)
                                      (t (error ???-msg))))))
                          (let ((couples-list (caddr tree)))
                               (cond ((car couples-list)
                                      (rplacd couples-list
                                              (cons (cons '&d valeur)
                                                    (cdr couples-list))))
                                     (t (rplaca (cddr tree)
                                                (ncons (cons '&d
                                                             valeur))))))
                          valeur)))))

(defun root_attributes (tree) ; this function returns the semantic
      (list (cons '&v (&v tree nil)))) ; value of "tree"
```

Now, here is a source text:

```
1  1101.01
```

Let us check that the grammar does compute the expected result:

```
runfnc essai.bin -"sc
;;; fnc-mj, version 3.6 du 20/04/83
LOAD    0.106
INIT    0.125 .
SCANNER    0.022 .
PARSER    0.101 .
ATTRIBUTES EVALUATION    0.015 .
FINAL    0.471 .

Et voici les resultats :
>user_dir_dir>Langages>Jourdan>binaire>essai.bin
&v =

13.25      ; OK!!
```

Here is the (not decorated) derivation tree corresponding to this text. The "nil" of each node of the tree is the slot to store the attributes.

```
(1 1 nil
   (3 1 nil
      (3 1 nil
         (3 1 nil
            (4 1 nil
               (6 1 nil
                  (/1 1)))                    ; a terminal
            (6 2 nil
               (/1 2)))
         (5 3 nil
            (/0 3)))
      (6 4 nil
         (/1 4)))
   (/. 5)
   (3 6 nil
      (4 6 nil
         (5 6 nil
            (/0 6)))
      (6 7 nil
         (/1 7))))
```

Here is the decorated tree:

```
(1 1 ((&v . 13.25))
   (3 1 ((&v . 13.0))
      (3 1 ((&v . 12.0))
         (3 1 ((&v . 12.0))
            (4 1 ((&v . 3.0))
               (6 1 ((&v . 8.0))
                  (/1 1)))
            (6 2 ((&v . 4.0))
               (/1 2)))
         (5 3 ((&v . 0))
            (/0 3)))
      (6 4 ((&v . 1.0))
         (/1 4)))
   (/. 5)
   (3 6 ((&d . 2) (&v . 0.25))
      (4 6 ((&d . 1) (&v . 0))
         (5 6 ((&v . 0))
            (/0 6)))
      (6 7 ((&v . 0.25))
         (/1 7))))
```

Notice that in the binary number before the decimal point, i.e. the first subtree of the root, the attribute &d is not computed, since it is useless. Notice also that inherited attributes (&s) are not stored in the tree.

Imprimé en France

par

l'Institut National de Recherche en Informatique et en Automatique