INFORMATIQUE THÉORIQUE ET APPLICATIONS

SOLANGE COUPET-GRIMAL

Prolog infinite trees and automata

Informatique théorique et applications, tome 25, n° 5 (1991), p. 397-418.

<http://www.numdam.org/item?id=ITA_1991__25_5_397_0>

Informatique théorique et Applications/Theoretical Informatics and Applications
(vol. 25, n° 5, 1991, p. 397 à 418)

PROLOG INFINITE TREES AND AUTOMATA (*)

by Solange COUPET-GRIMAL (1)

Abstract. – *This paper deals with an algorithm constructing the minimal deterministic finite state automaton recognizing a language defined by a rational expression. It relies on a representation in a normalized form of languages over finite alphabets by infinite trees and uses the powerful implementation of these trees in Prolog. It accomodates extended rational expressions with new operators including intersection and difference. It is also possible to get complete or non complete automata, according to what is needed. The result is a program which is neat, close to the mathematic formulation and very concise* (2 *pages*).

Résumé. – *Nous présentons dans ce papier un algorithme de construction de l'automate d'états finis déterministe et minimal reconnaissant un langage défini par une expression rationnelle. Il repose sur la représentation sous une forme normale des langages sur un alphabet fini par des arbres infinis et il utilise la puissante implémentation de ces arbres en Prolog. Il s'applique aux expressions rationnelles étendues, avec intersection et différence, et permet d'obtenir des automates complets ou non complets. Le résultat est un programme concis* (2 *pages*) *et élégant car très proche de la formulation mathématique.*

1. INTRODUCTION

Infinite rational trees have been the subject of many studies in theoretical computer science (see the article of Courcelles [10] about all these studies). Here we present an application of infinite trees in programming, using their implementation in Prolog II and Prolog III to give a new solution for a well known problem: the construction of a minimal deterministic finite automaton (DFA) from a regular expression.

We know many solutions for this problem. Let us cite Thomson's algorithm [16] which produces a nondeterministic automaton (NFA); the very elegant one due to Brzozowski [3], relying on the notion of a "*derivative*" of a regular expression, which constructs the minimal DFA and accomodates extended regular expressions with additional operators like intersection and

(*) Received December 1989, revised March 1990.
(1) Université de Provence, case M, 3, place Victor-Hugo, 13331 Marseille Cedex 3.

Informatique théorique et Applications/Theoretical Informatics and Applications
0988-3754/91/05 397 22/$4.20/

difference; the algorithm of MacNaughton and Yamada [13], which gives a DFA (nonminimal) and in which they mark all input symbols in a regular expression to make them distinct: for example, the marked version of $(a+b)*b(a+b)$ is $(a_1+b_2)*b_3(a_4+b_5)$, where a_1 and a_4 are different symbols; lastly, the very efficient algorithm of Berry and Sethi [2], using at once the derivatives of regular expressions and marking: it builds an NFA and, as the former one, it does not work for extended regular expressions.

The algorithm we give here is as general and as complete as possible, since it accomodates extended regular expressions and its displays on the screen the drawing of the minimal DFA. It is also possible to get complete or non complete automata, according to what is needed. The use of infinite trees permits us to represent languages over finite alphabets in a normalized form (although regular expressions are not normalized representations) and to profit by Prolog which provides a very powerful implementation of these trees. The program is very neat, close to the mathematic formulation, particularly concise (only one page) and is a solution for the problems set by Brzozowski [3] and which are still with us:

> "... we have assumed that it is always possible to recognize the equality of two regular expressions. If this is the case, then the state diagram constructed... is always minimal. However, it is often quite difficult to determine whether two regular expressions are equal. We... show that this difficulty can be overcome, and a state graph can always be constructed, but not necessarily with the minimal number of states. It should be pointed out that the other existing methods... have the same difficulties and, moreover, are limited to regular expressions with (+), (.) and (*) only."

(In Brzozowski's paper, expressions are said to be equal when they represent the same language.)

2. RATIONAL TREES AND REGULAR LANGUAGES

2.1. Rational trees

Informally, a tree is a set of nodes, each of them having a label and a position. The skeleton of a tree is the set consisting of all these positions. To make this more precise, we let A^* be the free monoid generated by a finite alphabet A.

Tree skeleton. *A tree skeleton is a set S for which there exists a nonnegative integer z such that:*

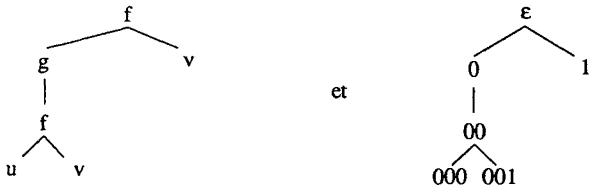
– *S is a nonempty subset of* $\{0, \ldots, z\}^*$;

– *every prefix of an element of S is in S;*

– *for every element p of S and every couple* (i, j) *of elements of* $\{0, \ldots, z\}$ *such that* $i < j$, *if pj belongs to S, then pi belongs to S.*

Tree: *Let F be a countable set and S be a tree skeleton. A tree over F whose skeleton is S is a mapping a from S to F; we write* $S = sk(a)$.

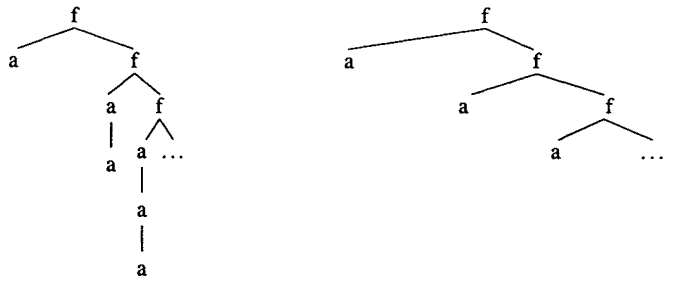
The figures below



represent respectively a tree a and its skeleton; the tree a whose skeleton is $\{\varepsilon, 0, 1, 00, 000, 001\}$ over the set of labels $\{f, g, u, v\}$ is defined by $a(\varepsilon) = f$, $a(0) = g$, $a(1) = v$, $a(00) = f$, $a(000) = u$, $a(001) = v$.

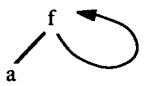
Finite or infinite tree: *A tree is called finite or infinite depending on whether its skeleton is finite or infinite.*

These are two infinite trees

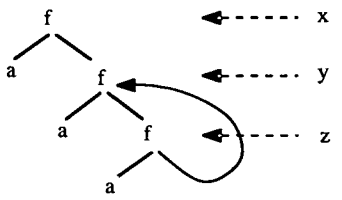


The first one has an infinite number of subtrees since all the subtrees in the left side are different: it is called nonrational. On the other hand, the second one has only two subtrees: itself and its left child; it is rational and thus it admits finite representations such as, for example, the diagrams below, corresponding to the systems of equations whose unknowns are x, y and z

and which are indicated under each diagram



{x = f(a,x) }



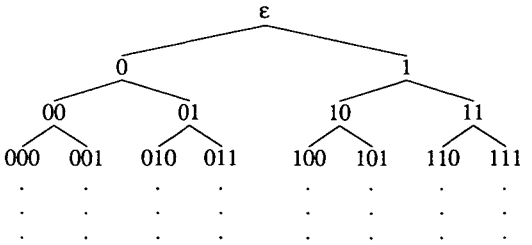
{x = f(a,y), y = f(a,z), z = f(a,y) }

Prolog infinite trees are rational infinite trees. The unification algorithm (so called by reference to the logical model) is in fact an algorithm for solving systems of equations and inequations on rational trees. A presentation of infinite trees in Prolog as well as several examples and a fundamental program can be found in [7]. This fundamental program produces the minimal representation s for a given tree a. It is now part of Prolog II and it can be called by *out-equ*(a, s). Other examples using infinite trees are given in [5] and [6].

In addition, we have to mention the evaluable predicates *eq* and *dif*: in Prolog *eq*(t, t') and *dif*(t, t') stand respectively for $t = t'$ and $t \neq t'$. Finally, *draw-equ*(a) [15] draws on the screen, the tree a in minimal form.

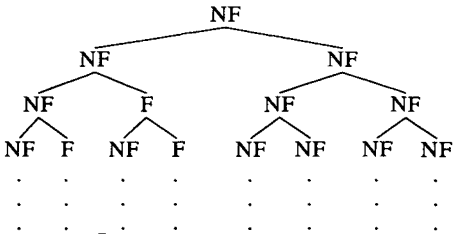
2.2. Languages in form of trees

For clarity, we will consider only languages over the alphabet $\{0, 1\}$, but it is very easy to generalize the following to the case of any finite alphabet. By noticing that the skeleton of the infinite binary trees:



is exactly the set of all the strings over $\{0, 1\}$, it is possible to characterize a language using only two labels. By convention, the strings in the language

are labeled F, and the other ones are labeled NF. For instance, the tree



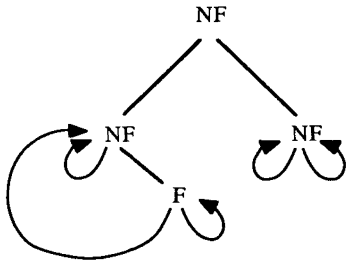
represents a language L which does not contain the empty string ε or any strings starting with 1 since the root and all the nodes of the right child are labeled NF. Accordingly strings in L must start with 0. Among these, only strings ending with 1 belong to the language since, in the left child, only right nodes are labeled F. Moreover, it should be noted that the left child of the tree of any language L is the tree of the "*derivatives of L by* 0", that is the language obtained by stripping from strings in L the leading 0's. More precisely, it is the set written $0^{-}L$ and defined by $0^{-}L=\{u/0u\in L\}$. Similarly, the right chid is the tree of $1^{-}L$. This involves the relation

$$\text{tree}(L) = \overset{c}{\underset{\text{tree}(0^{-}L)\ \text{tree}(1^{-}L)}{\diagup\ \diagdown}} \quad \text{where} \quad \begin{cases} c = F \text{ if } \varepsilon \in L \\ c = N\,F \text{ if } \varepsilon \notin L \end{cases}$$

Since a language is regular if and only if it admits a finite number of derivatives, we can deduce the following proposition.

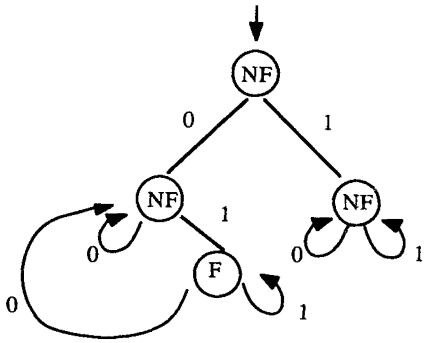
PROPOSITION: *A language L is regular if and only if the tree which represents it is rational.*

In the previous example, L is the set of all the strings starting with 0 and ending with 1. It is the regular language defined by $0(0+1)*1$. Its tree has exactly four subtrees. It is rational and its minimal diagram is



If we add to it some "decorations", we get the transition diagram of the minimal DFA which recognizes the language L (this explains the names of

the labels: F as final and NF as nonfinal).



Hence, to solve our problem is to construct the tree of a language given by a regular expression. This is the purpose of the following section. In addition, the minimization will be done by Prolog. This informal presentation relies on the following proposition [9].

PROPOSITION: *Let L be a regular language and a be its tree. The minimal* DFA *which recognizes L is defined in the following way:*

- *states are the subtrees of a;*
- *the initial state is a;*
- *the final states and nonfinal states are the subtrees of whose roots are labeled F and NF, respectively;*
- *for every subtree b of a, there is a transition under* 0 *from b to its left child and under* 1 *from b to its right child.*

3. CONSTRUCTING THE AUTOMATON

We will construct the automaton of a language L defined by a regular expression e. This automaton, denoted by $S(e)$, is the infinite tree associated with L. In the following we shall call it *the automaton of L* or *the tree of L*. As it has been mentionned already, we deal here with *extended* regular expressions, involving additional operators such as intersection and difference and whose syntax is defined as follows.

Regular expressions

(1) • φ *is a regular expression denoting the empty set.*

• ε *is a regular expression denoting the set* $\{\varepsilon\}$.

• *For every element a in* $\{0, 1\}$, *a is a regular expression denoting the set* $\{a\}$.

(2) • *If e,* e_1 *and* e_2 *are regular expressions denoting the languages L,* L_1 *and* L_2 *respectively, then:*

– $(e_1\, e_2)$ *is a regular expression denoting the product* $L_1 L_2$.

– $(e_1 + e_2)$ *is a regular expression denoting* $L_1 \cup L_2$.

– e^* *is a regular expression denoting the reflexive transitive closure* L^* *of L.*

– $(e_1 - e_2)$ *is a regular expression denoting* $L_1 - L_2$.

– $(e_1 \cap e_2)$ *is a regular expression denoting* $L_1 \cap L_2$.

These are the only regular expressions.

3.1. The method of construction

The construction of the automaton $S(e)$ associated with the regular expression e is recursive and directed by the syntax of e. We define the operations *union*, *concatenation*, *closure*, *intersection* and *difference* on the set of automata so that if e_1, e_2 and e are three regular expressions and s_1, s_2 and s are the automata associated with them, then:

union (s_1, s_2)	is the automaton associated with	$(e_1 + e_2)$	$(\mathscr{R})$
concatenation (s_1, s_2)	is the automaton associated with	$(e_1 . e_2)$	
fermeture (s)	is the automaton associated with	e^*	
intersection (s_1, s_2)	is the automaton associated with	$(e_1 \cap e_2)$	
différence (s_1, s_2)	is the automaton associated with	$(e_1 - e_2)$.	

Assuming that s is known or that s_1 and s_2 are known, we have to construct the tree resulting from one of these five operations. Thus, we will be able to produce the automaton of a regular expression recursively from the four basic automata, recognizing respectively $\varnothing$, $\{\varepsilon\}$, $\{0\}$ and $\{1\}$. This can be expressed in Prolog by rules such as the two following. The first one constructs the automaton s of $\varnothing$ and the second one constructs the automaton s from $(e_1 + e_2)$.

automaton (*empty*, *s*) → *eq* (*s*, *non-final* (*s*, *s*));

automaton (*plus* (*e*1, *e*2), *s*) →

automaton (*e*1, *s*1) *automaton* (*e*2, *s*2) *union* (⟨ *s*1, *s*2 ⟩, *s*);

First, we will present *union*, *intersection* and *difference* which are treated in the same natural way.

3.2. The operations union, intersection, difference

Let us construct s, the automaton of a language L resulting from the *union*, *intersection* or *difference* of two languages L_1 and L_2 recognized by the automata s_1 and s_2. It is a top-down construction: after finding its root, we calculate its left and right children. We saw previously that the children of s are the derivatives $0^- L$ and $1^- L$. Moreover the nature of the root (F or NF) depends on whether s recognizes the empty string or not. Thus we are led to define the laws $+$, $\cap$ and $-$ on the set $\{F, NF\}$ so that, when applied to the roots of s_1 and s_2, they give the root of s for each of the three operations union, intersection and difference. Then we will define the derivatives of L from those of L_1 and L_2.

DEFINITION: *For every element c in $\{F, NF\}$ we define:*

$$F+c=F, \qquad NF \cap c=NF, \qquad c-F=NF;$$

$$NF+c=c, \qquad F \cap c=c, \qquad c-NF=c.$$

PROPERTIES OF THE DERIVATIVES: *Let L_1 and L_2 be two regular languages. For every symbol a in $\{0, 1\}$:*

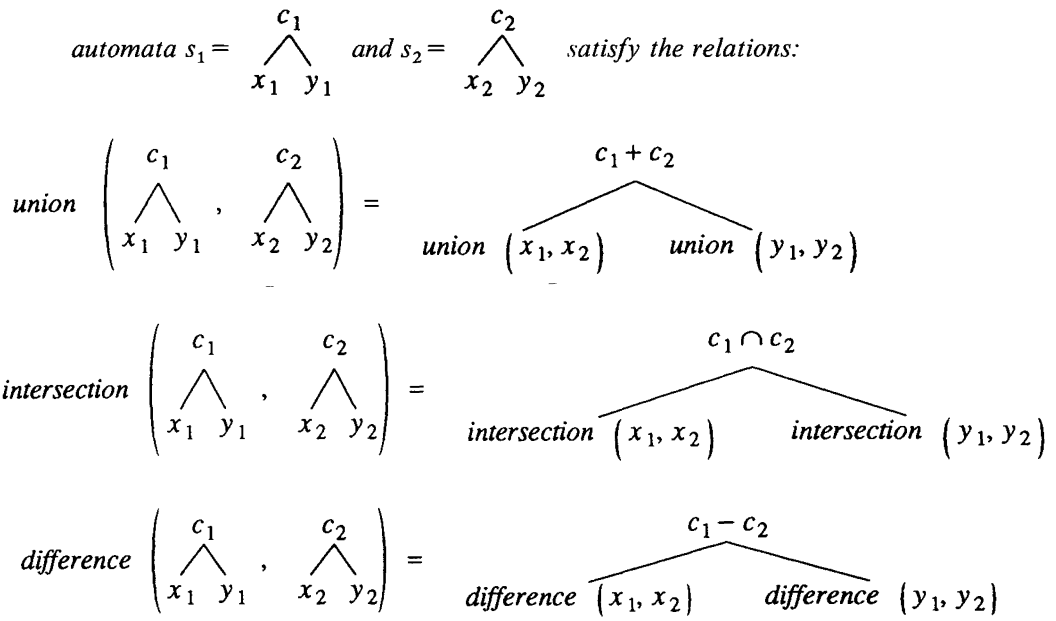
$$a^-(L_1 \cup L_2)=a^- L_1 \cup a^- L_2$$

$$a^-(L_1 \cap L_2)=a^- L_1 \cap a^- L_2$$

$$a^-(L_1-L_2)=a^- L_1-a^- L_2.$$

A proof of these results (given with the formalism of regular expressions) can be found in [3]. The definitions and properties above involve the following proposition.

PROPOSITION: *Let c_1 and c_2 be two elements in $\{F, NF\}$ and x_1, y_1, x_2, y_2 any automata. The operations* union, intersection *and* difference *applied to the*



Let us prove for example the first relation; the others can be obtained in a similar way. *union* (s_1, s_2) recognizes the empty string ε if and only if at least one of the two automata s_1 and s_2 recognizes ε, that is to say if and only if at least one of the two roots c_1 or c_2 is F. By the definition of the law + on $\{F, NF\}$, we deduce that the root of *union* (s_1, s_2) is $c_1 + c_2$. Moreover, if L_1 and L_2 are the languages recognized by s_1 and s_2 respectively, we know that x_1 is the automaton that recognizes $0^- L_1$; similarly x_2 recognizes $0^- L_2$. Hence, by the definition of *union*, *union* (x_1, x_2) recognizes $0^- L_1 \cup 0^- L_2$, which is, according to the properties of the derivatives, $0^- (L_1 \cup L_2)$. This implies that *union* (x_1, x_2) is the left child of the tree *union* (s_1, s_2). In the same way, it can be shown that *union* (y_1, y_2) is the right child of *union* (s_1, s_2).

If the trees we deal with were not infinite, the three relations would provide recursive definitions for the operations *union*, *intersection* and *difference:* we would construct the result for a couple (s_1, s_2) from the results of the children of s_1 and s_2. In this way, we would get a recursive algorithm (without terminal case), whose translation into Prolog could be obtained immediatly. In the program below F and NF are coded with the identifiers *final* and *non-final*. In addition, *union* $(\langle s_1, s_2 \rangle, s)$, and *difference* $(\langle s_1, s_2 \rangle, s)$ are assertions if and only if s is respectively the result of the *union*, the *intersection*

and the *difference* of s_1 and s_2. Finally we recall that in Prolog II the tree  is coded by $\langle c, x, y \rangle$ if c is a variable.

```
union (< < c1,x1,y1 > , < c2,x2,y2 >> , <c,x,y > )  ->
 plus (< c1,c2 >, c)
union (< x1,x2 >,x)
union (< y1,y2 >,y);

intersection  (< < c1,x1,y1 > , < c2,x2,y2 >> , <c,x,y > )  ->
 inter (< c1,c2 >, c)
 intersection (< x1,x2 >,x)
 intersection (< y1,y2 >,y);

difference(< < c1,x1,y1 > , < c2,x2,y2 >> , <c,x,y > )  ->
 minus (< c1,c2 >, c)
 difference (< x1,x2 >,x)
 difference (< y1,y2 >,y);

 plus (<final , c >,final ) ->;
 plus (<non-final , c >,  c) -> ;

 inter (<final , c >,c ) ->;
 inter (<non-final , c >,non-final) ->;

 minus (<c,final>, non-final ) ->;
 minus  (<c, non-final  >,c)->;
```

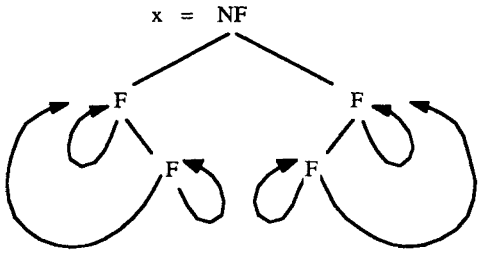
However, it is clear that here there is no terminal case since the two trees $s_1 =$  and $s_2 =$  that we are traversing are infinite; they do not have any leaves. Such a program calculates indefinitely all the nodes of the leftmost branch of the resulting automaton.

HOW TO MAKE THE ALGORITHM TERMINATE: Let f be one of the three operations *union*, *intersection* or *difference*. The termination of our algorithm relies on the fact that the trees s_1 and s_2 both have a finite number of subtrees. The program above, when constructing the infinite leftmost branch of $s = f(s_1, s_2)$, calculates, in each step, a tree $t = f(t_1, t_2)$ where t_1 and t_2 are subtrees of s_1 and s_2 respectively and t is a child of the tree calculated in the preceding step. Since the trees s_1 and s_2 have a finite number of subtrees, after a finite number of steps, this program will necessarily calculate a tree $t' = f(t_1, t_2)$ after undertaking the same calculation $t = f(t_1, t_2)$ in a previous step for an ancestor t of t'. This will be the terminal case for the recursion. Informally,

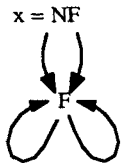
t' is replaced by an arrow pointing to its ancestor t. This contributes simultaneously to the determination of t and t'. More rigorously, the treatment of the current branch terminates by adding the equation $t = t'$ to the system of equations that determines s and that we are building by traversing s_1 and s_2 in preorder. This is called (by reference to the logical model) the unification of t with t'.

On the following page, we give two trees r and l' representing respectively the languages consisting of all the words ending with 0 and 1. We mean to construct the union x of r and l'. We see that these two trees have only two subtrees: r and l for the first one and r' and l' for the second one. They are followed by schemas showing the different steps of the construction of the tree $x = \text{union}(r, l')$. The equations in boldface are the unifications which ensure the termination. The right child of x, that is to say the tree $z = \text{union}(r, r')$, is calculated in the same way.

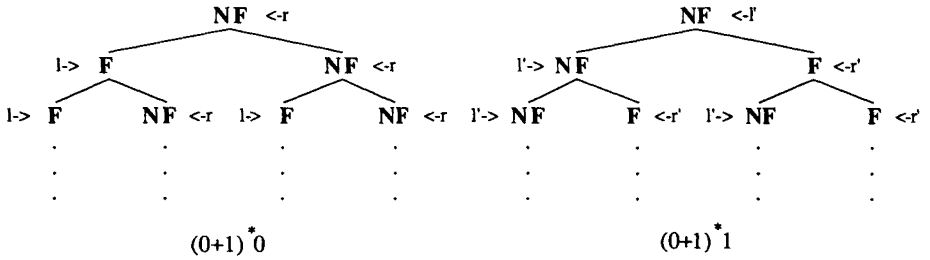
It is possible to verify that in the end, we obtain:

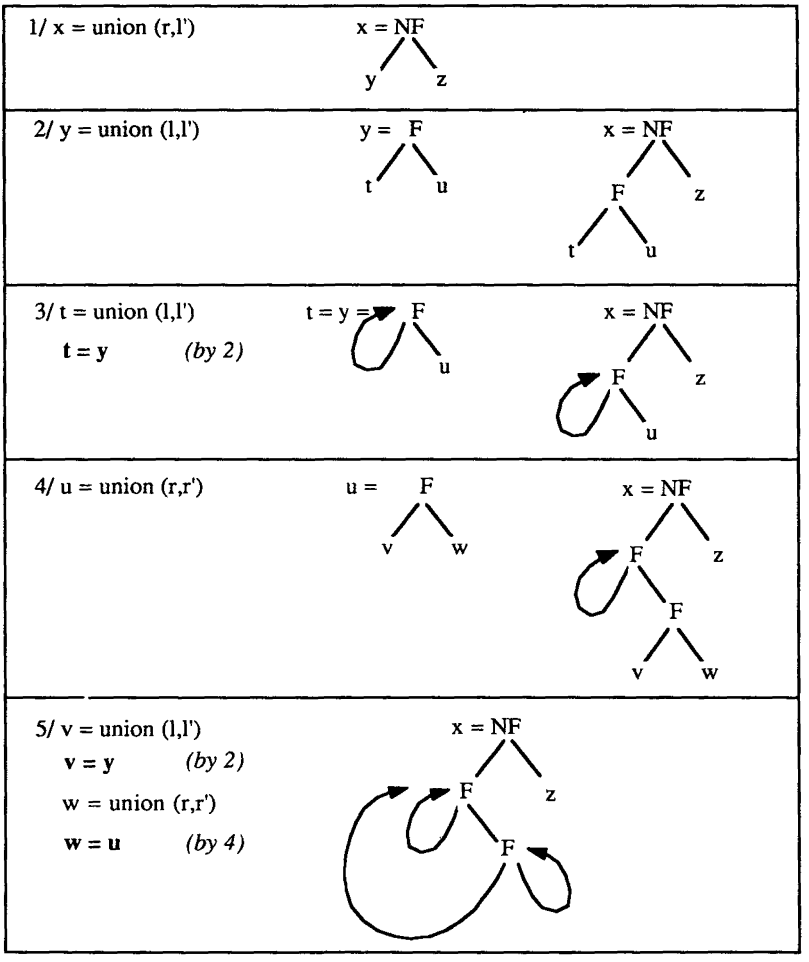


which is minimized by Prolog resulting in



This principle for terminating the recursion is rather difficult to explain, but it can be expressed very easily in Prolog. We record in a list all the equations of the type $r = \text{union}(a_1, a_2)$ that have been treated already or, more simply, all the couples $\langle a, r \rangle$ (argument-result) where $a = (a_1, a_2)$. This list l will be an additional parameter in the rules *union*, *intersection*..., and at the beginning, it will be the empty list, traditionally denoted by *nil*.


NF <-r
l-> F
NF <-r
l-> F
NF <-r
l-> F
NF <-r
(0+1)*0
NF <-l'
l'-> NF
F <-r'
l'-> NF
F <-r'
l'-> NF
F <-r'
(0+1)*1


1/ x = union (r,l')
x = NF
y z
2/ y = union (l,l')
y = F
t u
x = NF
F z
t u
3/ t = union (l,l')
t = y (by 2)
t = y = F
u
x = NF
F z
u
4/ u = union (r,r')
u = F
v w
x = NF
F z
F
v w
5/ v = union (l,l')
v = y (by 2)
w = union (r,r')
w = u (by 4)
x = NF
F z
F

The rules *member* below permit us, with a given couple $\langle a, r \rangle$ (first parameter), to know if the argument a occurs in a couple $\langle a, r' \rangle$ of the list (second parameter). If it actually occurs, the result r we are looking for is unified with r' and the third parameter takes the value *true;* in the opposite case, it takes the value *false.*

```
member(<a,r>,nil,false) ->;
member(<a,r>,<a,r'>.l,true) -> eq(r,r');
member(<a,r>,<a',r'>.l,b)-> dif(a,a') member(<a,r>,l,b);
```

The two following rules deal with the termination of the recursion in accordance with the value (*true* or *false*) of the first parameter.

```
iff(true,p) ->;
iff(false,p) -> p;
```

Finally, it is easy to be convinced that the problem can be solved by transforming the rules

automaton and *union*, ... in the following way:

```
automaton(plus(e1,e2),s) ->     automaton(e1,s1)
                                automaton(e2,s2)
                                already-met(union(<s1,s2>,s,nil));

union(<<c1,x1,y1>,<c2,x2,y2>>,<c,x,y>,l) ->     plus(<c1,c2>,c)
                                                already-met(union(<x1,x2>,x,l))
                                                already-met(union(<y1,y2>,y,l));
```

and adding the rule

```
already-met(f(a,r,l)) ->  member(<a,r>,l,b)   iff(b,f(a,r,<a,r>.l>));
```

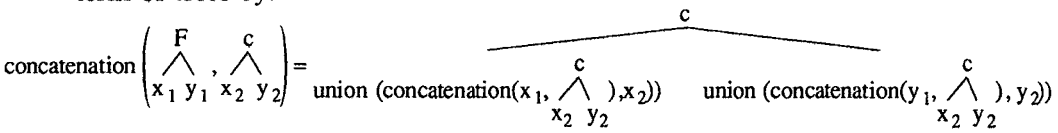
In the same way, we modify the rules *intersection* and *difference* (*cf.* the whole program in the end). Now, let us consider the case of *concatenation* and *closure* which raise more problems.

3.3. Concatenation

The same approach as in the preceding section would lead us to calculate the derivatives of the product of two languages L_1 and L_2. Now, it can be shown that for every symbol a in the alphabet:

$$a^{-}(L_1 L_2) = (a^{-} L_1) L_2 \qquad \text{if} \quad \varepsilon \notin L_1$$
$$a^{-}(L_1 L_2) = (a^{-} L_1) L_2 \cup a^{-} L_2 \qquad \text{if} \quad \varepsilon \in L1.$$

One recognizes that the problem is more difficult, since the operator "a^-" does not commute with the product. The last relation can be translated in term of trees by:



where c denotes the root of s. A recursive algorithm relying on this equality appeals to a non terminal recursion. This poses a problem for termination. As a matter of fact, we have to calculate the *union* of two trees; the first one, resulting from an operation *concatenation* is not completely known. A new approach led us to introduce an auxiliary operation, called *concatenation'*, with three arguments.

DEFINITION: *Let L_1, L_2 and R be three languages. We define*

$$concatenation'(L_1, R, L_2) = R \cup (L_1 L_2).$$

Now, it is possible to express the product with this operation since

$$L_1 L_2 = concatenation'(L_1, \varnothing, L_2).$$

Consequently, to solve our problem, it is sufficient to give a construction of the automaton recognizing the language L resulting from the operation *concatenation'* when applied to any three languages. This construction relies on the usual considerations: it uses a necessary and sufficient condition for L to contain the string ε and a caracterization of the derivatives of L.

PROPOSITION: *If $\varepsilon \in L_1$ then*

$$concatenation'(L_1, R, L_2) = concatenation'(L_1 - \{\varepsilon\}, R \cup L_2, L_2).$$

If $\varepsilon \notin L_1$ then:

- $\varepsilon \in concatenation'(L_1, R, L_2)$ *if and only if* $\varepsilon \in R$;
- $$a^- \, concatenation'(L_1, R, L_2) = concatenation'(a^- L_1, a^- R, L_2)$$

for every a in the alphabet.

Proof: Let us assume that $\varepsilon \in L_1$. Hence $L_1 L_2 = L_2 \cup (L_1 - \{\varepsilon\}) L_2$ and therefore concatenation' $(L_1, R, L_2) = R \cup (L_1 L_2) = (R \cup L_2) \cup (L_1 - \{\varepsilon\}) L_2$ $= \text{concatenation}' (L_1 - \{\varepsilon\}, R \cup L_2, L_2)$; this proves the first equality. Now, let us assume that ε is not in L_1. Thus, it is not in the product $L_1 L_2$.

Therefore it is in the union $R \cup (L_1 L_2)$ if and only if it is in R. Finally, for every symbol a in the alphabet $a^-(R \cup L_1 L_2) = a^- R \cup (a^- L_1) L_2$ by the properties of the derivatives of the union and the concatenation of two languages. From this, we deduce the result immediately. The following corollary is just a translation in terms of trees.

COROLLARY: *If c denotes any element in* $\{F, NF\}$ *and* x, y, x_1, y_1, x_2, y_2, r, t *any automata, the following relations hold:*

1. $concatenation'\left(\begin{array}{c} F \\ \wedge \\ x\ \ y \end{array}, r, t\right) = concatenation'\left(\begin{array}{c} NF \\ \wedge \\ x\ \ y \end{array}, union\ (r, t), t\right)$

2. $concatenation'\left(\begin{array}{c} NF \quad\quad c \\ \wedge \quad\quad \wedge \\ x_1 \quad y_1\ x_2\ y_2 \end{array}, t\right) = \overbrace{concatenation'(x_1, x_2, t) \quad concatenation'(y_1, y_2, t)}^{c}$

We can notice that, by these relations, the *union* is computed before a recursive call to concatenation' (relation 1), with two known arguments. With the considerations of the preceding section about the termination of the algorithm, the relations can be expressed in Prolog by:

```
automaton(conc(e1,e2),s) ->
   automaton(e1,s1)
   automaton(e2,s2)
   automaton(empty,r)
   already-met(concatenation'(<s1,r,s2>,s,nil));

concatenation'(<non-final(x1,y1),<c,x',y'>,s2>,<c,x,y>,l) ->
   already-met(concatenation'(<x1,x',s2>,x,l))
   already-met(concatenation'(<y1,y',s2>,y,l));
concatenation'(<final(x1,y1),r,s2>,s,l) ->
   already-met(union(<r,s2>,r',nil))
   already-met(concatenation'(<non-final(x1,y1),r',s2>,s,l));
```

To prove that the algorithm terminates, it remains to show that if s, r and t are three automata, the computation of *concatenation'* (s, r, t) only generates a finite number of new calculations of the form *concatenation'* (s', r', t'). We show that each argument (s', r', t') is such that:

- *s' is a subtree of s, with possibly a different root.*
- $t = t'$.
- *r' is the union of a subtree of r and a finite number (maybe zero) of subtrees of t.*

These three conditions are obviously satisfied for the initial argument (s, r, t). Let us assume that they are satisfied for an argument (s', r', t') in a given step. Two cases may occur:

- if we apply the first rule, the new argument (s'', r'', t'') of *concatenation'* is such that $t''=t$. Moreover, s'' equals s' (with possibly a different root). Thus it is a subtree of s (with possible a different root). Finally, r'' is the union of r' and t. Therefore it is actually the union of a subtree of r and some subtrees of t.

- By applying the second rule, we are led to calculate *concatenation'* (x_1, x_2, t) and *concatenation'*(y_1, y_2, t) where x_1 and y_1 are the children of s' and x_2 and y_2 those of r'. x_1 and y_1 are subtrees of s. In addition, by hypothesis, r' is the union of a subtree r'' of r and some subtrees $t_1, t_2, \ldots, t_n$ of t. Hence if $L(X)$ denotes the language recognized by an automaton X, then $L(r')=L(r'') \cup L(t_1) \cup \ldots \cup L(t_n)$, and thus for every symbol a in $\{0, 1\}$, $a^- L(r')=a^- L(r'') \cup a^- L(t_1) \ldots \cup a^- L(t_n)$. This implies that the children of r', which are the automaton of the derivatives of $a^- L(r')$, are the union of a child of r'' and a child of $t_1, t_2, \ldots, t_n$, that is to say of a child of r and some subtrees of t.

The algorithm terminates since s, r and t have a finite number of subtrees.

3.4. Closure

Here we will construct, from the automaton of a language L, the automaton of its transitive reflexive closure L^*. As previously, we are led to study the derivatives of $a^- L^*$. Now, it can be proved that, for every symbol a of $\{0, 1\}$

$$a^- L^* = (a^- L) L^*,$$

and we immediately deduce the relation for automata:

$$closure\left(\begin{array}{c} c \\ \wedge \\ x\ y \end{array}\right) = \overbrace{concatenation\left(x, closure\left(\begin{array}{c} c \\ \wedge \\ x\ y \end{array}\right)\right) \quad concatenation\left(y, closure\left(\begin{array}{c} c \\ \wedge \\ x\ y \end{array}\right)\right)}^{F}$$

But there is the same problem as in the preceding section. A recursive algorithm, relying on this relation, to calculate $t=closure\,(s)$ would lead to the computation of $t'=closure\,(s)$. After unifying t' with t, we would compute

concatenation (x, t') with one of the two arguments, t', which is not completely determined. This follows from the fact that the recursion is not terminal.

Let us define again a new operation *closure'* as follows.

DEFINITION: *Let* L_1 *and* L_2 *be two languages. We define:*

$$closure'(L_1, L_2) = L_2(L_1^*).$$

Moreover, the closure of a language L can be expressed with closure': $L^* = closure'(L_1, \{\varepsilon\})$. Thus an algorithm permitting the construction of the automaton of closure'(L_1, L_2) for any languages L_1 and L_2 given by their automata solves our problem. For this purpose, we use the following proposition.

PROPOSITION: $\varepsilon \in closure'(L_1, L_2)$ *if and only if* $\varepsilon \in L_2$. *In addition, for every symbol a in the alphabet:*

$$a^- closure'(L_1, L_2) = closure'(L_1, a^- L_2) \qquad if \quad \varepsilon \notin L_2$$

$$a^- closure'(L_1, L_2) = closure'(L_1, a^-(L_1 \cup L_2)) \qquad if \quad \varepsilon \in L_2.$$

This proposition can be proved easily by using the properties of the derivatives of products and closures. It can be expressed in terms of automata by the corollary:

COROLLARY: *Let* x, y, x', y' *be any automata. The operation* closure' *satisfies the following relations.*

1. $closure'\left(s, \overset{NF}{\underset{x\ y}{\wedge}}\right) = \overset{NF}{\underset{closure'(s,x)\ \ closure'(s,y)}{\wedge}}$

2. $closure'\left(s, \overset{F}{\underset{x\ y}{\wedge}}\right) = \overset{F}{\underset{closure'(s,x')\ \ closure'(s,y')}{\wedge}}$ where $union\left(s, \overset{F}{\underset{x\ y}{\wedge}}\right) = \overset{F}{\underset{x'\ y'}{\wedge}}$

From these two relations we can deduce, with the usual considerations about the termination of the algorithm, the Prolog rules below (where " $\wedge$ " stands for ε).

```
automaton(star(e),s) ->  automaton(e,s1)  automaton("^",n)  already-met(closure'(<s1,n>,s,nil));

closure'(<s,non-final(x',y')>,non-final(x,y),l) ->
  already-met(closure'(<s,x'>,x,l))
  already-met(closure'(<s,y'>,y,l));
closure'(<s,final(x',y')>,final(x,y),l) ->
   already-met(union(<final(x',y'),s>,final(x'',y''),nil))
   already-met(closure'(<s,x''>,x,l))
   already-met(closure'(<s,y''>,y,l));
```

As for the concatenation, it can be proved [9] that the computation of *closure'* (s, n) only generates a finite number of new calculations of the form *closure'* (s_1, s_2) and that ensures the termination of the algorithm.

4. CONCLUSION

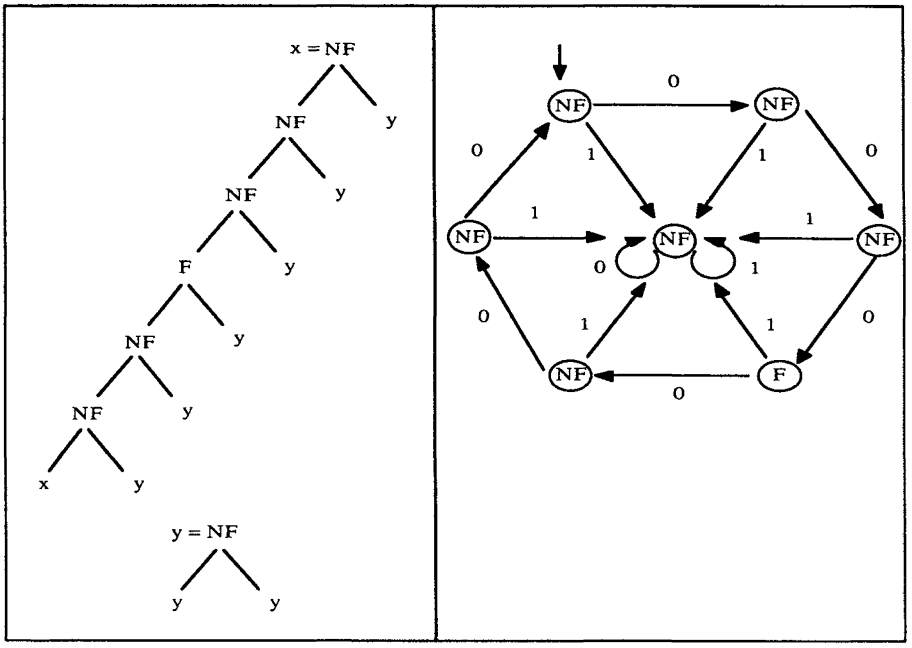
The whole program can be found in the following section. We observe that it is very clear and concise: Prolog, which memorizes, compares, unifies, minimizes, draws the infinite trees, takes charge of most of the work. Moreover, it is easy to generalize it [9] to any finite alphabet and to noncomplete automata. With this more general version and the compiler Prolog II$^+$ we tested the following examples on a Sun 3.60. The table below gives, expressed in seconds, the respective times for the computation of the automaton, the computation of the automaton including the display on the screen of its minimal system of equations and the computation of the automaton including the drawing of its minimal form. In the last column, there is the number of states of the minimal automaton. For example, it is 2^{n+1} for the expression $(a+b)^* a (a+b)^n$. The expression $((ba^*)^n b + ab^* ab)^*$ is an example given in [4] and [12].

regular expression	computation of the automaton	with the minimal system	with the drawing	number of states
$(aaa)^*$- $(aa)^*$	0,05	0,2	0,8	6
$(aaa)^* \cap (aa)^*$	0,05	0,2	0,7	6
$(a+b)^*a\ (a+b)$	0,08	0,1	0,4	4
$(a+b)^*a\ (a+b)^2$	0,4	0,7	1,2	8
$(a+b)^*a\ (a+b)^3$	11,5	13	13,8	16
$(ba^*b+ab^*ab^*)^*$	0,2	0,7	1,5	8
$((ba^*)^2b+ab^*ab^*)^*$	1	2,6	4,6	15
$((ba^*)^3b+ab^*ab^*)^*$	10	16,4	19,7	28

To calculate the automaton corresponding to (000)*-(00)* we ask Prolog the question:

> *automaton* (*minus* (*star* (*conc* (*conc* ("0", "0"), "0")),
star (*conc* ("0", "0"))), *s*) *draw-equ*(*s*);

Prolog answers by showing on the screen the drawing of the infinite tree in the left side of the figure on the following page, which represents the automaton whose transition diagram is the hexagone in the right side.



5. The Program

```
automaton(empty,s) ->
  eq(s,non-final(s,s));
automaton("^",final(s,s)) ->
  automaton(empty,s);
automaton("0",non-final(final(s,s),s)) ->
  automaton(empty,s);
automaton("1",non-final(s,final(s,s))) ->
  automaton(empty,s);
automaton(conc(e1,e2),s) ->
  automaton(e1,s1)
  automaton(e2,s2)
  automaton(empty,r)
```

```
  already-met(concatenation'(<s1,r,s2>,s,nil));
automaton(plus(e1,e2),s) ->
  automaton(e1,s1)
  automaton(e2,s2)
  already-met(union(<s1,s2>,s,nil));
automaton(star(e),s) ->
  automaton(e,s1)
  automaton("^",n)
  already-met(closure'(<s1,n>,s,nil));
automaton(et(e1,e2),s) ->
  automaton(e1,s1)
  automaton(e2,s2)
  already-met(intersection(<s1,s2>,s,nil));
automaton(minus(e1,e2),s) ->
  automaton(e1,s1)
  automaton(e2,s2)
  already-met(difference(<s1,s2>,s,nil));

"CONCATENATION"

concatenation'(<non-final(x1,y1),<c,x',y'>,s2>,<c,x,y>,l) ->
  already-met(concatenation'(<x1,x',s2>,x,l))
  already-met(concatenation'(<y1,y',s2>,y,l));
concatenation'(<final(x1,y1),r,s2>,s,l) ->
  already-met(union(<r,s2>,r',nil))
  already-met(concatenation'(<non-final(x1,y1),r',s2>,s,l));

"UNION"

union(<<c1,x1,y1>,<c2,x2,y2>>,<c,x,y>,l) ->
  plus(<c1,c2>,c)
  already-met(union(<x1,x2>,x,l))
  already-met(union(<y1,y2>,y,l));

plus(non-final(c),c) ->;
plus(final(c),final) ->;

"CLOSURE"

closure'(<s,non-final(x',y')>,non-final(x,y),l) ->
  already-met(closure'(<s,x'>,x,l))
  already-met(closure'(<s,y'>,y,l));
closure'(<s,final(x',y')>,final(x,y),l) ->
  already-met(union(<final(x',y'),s>,final(x'',y''),nil))
  already-met(closure'(<s,x''>,x,l))
  already-met(closure'(<s,y''>,y,l));

"INTERSECTION"

intersection(<<c1,x1,y1>,<c2,x2,y2>>,<c,x,y>,l) ->
  inter(<c1,c2>,c)
  already-met(intersection(<x1,x2>,x,l))
  already-met(intersection(<y1,y2>,y,l));
```

```
inter(final(c),c) ->;
inter(non-final(c),non-final) ->;

"DIFFERENCE"

difference(<<c1,x1,y1>,<c2,x2,y2>>,<c,x,y>,l) ->
  minus(<c1,c2>,c)
  already-met(difference(<x1,x2>,x,l))
  already-met(difference(<y1,y2>,y,l));

minus(<c,final>,non-final) ->;
minus(<c,non-final>,c) ->;

"DIVERS"

already-met(<f,a,r,l>) ->
  member(<a,r>,l,v)
  iff(v,<f,a,r,<a,r>.l>);

member(<x,s>,nil,faux) ->;
member(<x,s>,<x,s'>.l,vrai) -> eq(s,s');
member(<x,s>,<y,t>.l,v) -> dif(x,y) member(<x,s>,l,v);

iff(vrai,p) ->;
iff(faux,p) -> p;
```

REFERENCES

1. A. V. AHO, J. D. ULLMAN, The theory of parsing, translation and compiling, Prentice Hall, series in automatic computation.
2. G. BERRY, R. SETHI, From regular expressions to deterministic automata (note), *Theoret. Comput. Sci.*, 1986, 48 : 1, pp. 117-126.
3. J. A. BRZOZOWSKI, Derivatives of regular expressions, *J. A.C.M.*, 1964, 11 : 4, pp. 481-494.
4. J. M. CHAMPARNAUD, Automate : un système de manipulation des automates finis, Rapport interne 85-48, LITP, Université Paris VII, 1985.
5. J. COHEN, F. GIANNESINI, Parser generation and grammar manipulations using Prolog's infinite trees, *Logic Programming*, octobre 1984, pp. 253-265.
6. J. COHEN, T. J. HICKEY, Parsing and compiling using prolog, *A.C.M. Trans. Progr. Lang. Systems*, 1987, 9 : 2, pp. 125-163.
7. A. COLMERAUER, Prolog and Infinite Trees, *Logic Programming*, W. K. CLARK, S. A. TARNLUND Ed.), Academic Press, New York, 1982, pp. 231-251.
8. A. COLMERAUER, Introduction to Prolog III, *Esprit 87-Archivement and Impact Proceedings of the 4th Annual Esprit Conference*, 1987, 1 : 1, Bruxelles.
9. S. COUPET-GRIMAL, Deux arguments pour les arbres infinis en Prolog, *Thèse d'université*, Université Aix-Marseille-II, 1988.
10. B. COURCELLE, Fundamental properties of infinite tees, *Theor. Comput. Sci.*, 1985, 25, pp. 95-169.

11. G. COUSINEAU, Les arbres à feuilles indicées : un cadre algébrique pour l'étude des structures de contrôle, *Thèse d'état*, Université de Paris-VII, 1977.
12. E. LEISS, Regpack, an interactive package for regular languages and finite automata, Research report CS-77-32, Université de Waterloo, octobre 1977.
13. R. MCNAUGHTON, H. YAMADA, Regular expressions and state graphs for automata, *I.R.E.E. Trans. Comput.*, 1960, EC-9 : 1, pp. 38-47.
14. M. NIVAT, Langages algébriques sur le magma libre et sémantique des schémas de programmes, in *automata, languages and programming*, M. NIVAT Ed., Amsterdam, 1973.
15. J. F. PIQUE, Drawing trees and their equations in Prolog, *Proceedings of the second international logic programming conference*, Uppsala University, 1984.
16. K. THOMSON, Regular expression search algorithm, *Comm. A.C.M.*, 1968, 11 : 6, pp. 419-422.