

1 Définitions

Les grammaires formelles

Une grammaire formelle est un quadruplet $G = (T, N_t, S, D)$ où :

- T est l'alphabet des terminaux
 - N_t est l'alphabet des non terminaux
 - $S \in N_t$ est l'axiome
- Notons $\Sigma := N_t \cup T$
- $D \subset (\Sigma^*)^2$, est l'ensemble des règles de dérivation.

Pour $(a, b) \in D$, soit $\xrightarrow{(a,b)}$ la relation binaire définie sur Σ^* par

$$\forall x, x' \in \Sigma^*, x \xrightarrow{(a,b)} x' \iff \exists u, v \in \Sigma^* / x = uav \text{ et } x' = ubv$$

On note $\rightarrow := \bigcup_{d \in D} \xrightarrow{d}$ et on note $\xrightarrow{*}$ la cloture transitive et réflexive de \rightarrow .

Pour $x \in \Sigma^*$, notons

$$\delta(x) := \{y \in \Sigma^* / x \xrightarrow{*} y\}$$

$|x|_l := |\{i \in \mathbb{N} \mid x_i = l\}|$ est le nombre d'occurrences de la lettre l dans x .

Alors le langage de la grammaire formelle G est le suivant :

$$\mathcal{L}(G) := \delta(S) \cap T^*$$

Nous allons supposer que N_t et T sont finis. Le mot preuve désignera une preuve d'appartenance d'un mot au langage de la grammaire.

2 Vérification de preuve

Une nouvelle structure de données pour les preuves

Nous pouvons nous restreindre à la définition d'un mot qui dérive d'un autre. On va alors fournir, comme preuve, une liste de mots m_1, \dots, m_n tels que $\forall i \in [1, n-1], m_i \rightarrow m_{i+1}$. Nous allons aussi renseigner de quelle manière le mot m_{i+1} dérive du mot m_i en fournissant l'indice de la règle de dérivation (a, b) et celui du début de a dans m_i qui sera remplacé par b . Nous aboutissons alors à :

```
type 'e preuveformelle = (('e caractere list)*int*int) list
```

Nous prendrons comme exemple la grammaire formelle $G = (T, N_t, S, D)$ où :

$T = \{\underline{a}, \underline{b}, \underline{c}\}$

$N_t = \{\underline{S}, \underline{B}\}$

$D = \{(\underline{S}, \underline{aBSc})_1, (\underline{S}, \underline{abc})_2, (\underline{Ba}, \underline{aB})_3, (\underline{Bb}, \underline{bb})_4\},$

alors aabbcc est dans $\mathcal{L}(G)$ car $\underline{S} \rightarrow_1 \underline{aBSc} \rightarrow_2 \underline{aBabcc} \rightarrow_3 \underline{aaBbcc} \rightarrow_4 \underline{aabbcc}$.

En Ocaml :

```
let unepreuve = [(mot "aBSc",0,1);(mot "abc",2,3);
  (mot "aB",1,3);(mot "bb",2,4)]
```

3 Preuve automatique

Nous allons générer successivement les $G_n := \{x \in \Sigma^* \mid S(\bigcup_{0 \leq k \leq n} \rightarrow^k)x\}$ à

l'aide de $G_{n+1} = \bigcup_{x \in G_n} \mathcal{S}(x)$ où $\mathcal{S}(x) := \{y \in \Sigma^* \mid x \rightarrow y\} = \bigcup_{d \in D} \{y \in \Sigma^* \mid$

$x \xrightarrow{d} y\}$ Pour ce faire, on trouve les successeurs d'un mot x par $d = (a, b)$ en recherchant le motif a dans x . (L'algorithme de Knuth-Morris-Pratt est adéquat). Nous avons donc un algorithme qui permet théoriquement de prouver que n'importe quel mot de la grammaire appartient bien à cette dernière. Cet algorithme termine si et seulement si le mot à prouver est prouvable (si il est dans la grammaire). Le problème étant la complexité de ce dernier, qui fonctionne comme une machine non déterministe où on ne coupe pas les instances arrivant à un état puit. La suite va donc consister à trouver, de manière heuristique ou non, des méthodes affirmant qu'à partir d'un mot m , on aura des difficultés à dériver en notre mot m' à prouver. Alors, nous allons arrêter les recherches de dérivations de S en m' qui passent par le mot m .

cf ANNEXE2 preprocessgf l.120 et succ l.198 qui calcule $\mathcal{S}(x)$

Un parcours en largeur particulier

Pour générer cette preuve, nous allons utiliser un parcours qui nous permettra de trouver dans un graphe un chemin plus court entre le sommet initial

(l'axiome S) et un sommet 'valide' (le mot que l'on cherche à prouver). Ce parcours devra aussi éviter les chemins passant par un sommet 'interdit' (voir la suite).

Alors on fournira à ce parcours une fonction `interdit` et une fonction `valide` toutes deux de type `sommet -> bool`.

Ce parcours est donc un parcours en largeur depuis un sommet x_0 qui garde en mémoire le chemin parcouru et qui s'arrête dès qu'un sommet s , tel que `valide s`, est parcouru. Il renvoie alors un chemin de x_0 à s . Lorsque le parcours passe par un sommet s tel que `interdit s`, il ne parcourt pas ensuite ses voisins.

cf ANNEXE2 `parcoursmagique` l.167

Enfin, un prouveur automatique d'appartenance d'un mot à une grammaire formelle

Il ne nous reste plus qu'à appliquer notre parcours sur un graphe où les sommets sont les mots de Σ^* , et les arêtes sont $\{(a, b) \in \Sigma^* \mid a \rightarrow b\} = \{(a, b) \in \Sigma^* \mid b \in \mathcal{S}(a)\}$.

cf ANNEXE2 `recherchepreuvenaif` l.210

Par la suite, nous allons apporter des améliorations, en fournissant des fonctions `interdit`.

4 Amélioration pour les grammaires croissantes

Une grammaire croissante est une grammaire telle que :

$$\forall (a, b) \in D, |a| \leq |b|$$

On a alors une première propriété très simple, $\forall x, x' \in \delta(S), x \xrightarrow{*} x' \implies |x| \leq |x'|$.

Alors, dans la recherche de dérivations de S vers m , on peut supprimer les "branches de recherche" qui partent d'un mot de longueur $> |m|$. Il ne reste qu'à faire le même parcours que précédemment avec la fonction suivante comme fonction `interdit`.

```
let interditcroiss x = Array.length x > Array.length m
```

5 Amélioration dans le cas général : déduction sur le nombre d'occurrence de chaque lettre

Rappelons, le problème : étant donné un mot $x \in \delta(S)$, comment le dériver en un mot $m \in \delta(x)$ donné ? Pour tout $l \in \Sigma, x \in \Sigma^*$, on va chercher un ensemble $s_x(l)$ qui majore $|\delta(x)|_l (= \{|y|_l \mid y \in \delta(x)\})$, l'ensemble des nombres d'occurrence de l dans les successeurs de x .

Ainsi, $x \xrightarrow{*} m \implies \forall l \in \Sigma, |m|_l \in s_x(l)$, la contraposée nous sera utile :

$$\exists l \in \Sigma / |m|_l \notin s_x(l) \implies \neg(x \xrightarrow{*} m)$$

Ce qui pourra nous permettre de réduire notre champ de recherche. En effet,

si pour un mot $x \in \delta(S)$, $\exists l \in \Sigma, |m|_l \notin s_x(l)$, alors $m \notin \delta(x)$.

Alors **interdit x** devra renvoyer vrai.

Pour calculer $s_x(l)$, nous allons utiliser un graphe que nous noterons A_0 .

Les sommets sont des éléments de $Q \subset (\mathcal{P}(\mathbb{N}))^\Sigma$ tels que :

$$\begin{aligned} \forall q, q' \in Q, \exists l \in \Sigma / q(l) \cap q'(l) &= \emptyset \\ \text{et } \forall m \in \delta(S), \exists q \in Q / \forall l \in \Sigma, |m|_l &\in q(l) \end{aligned}$$

Ainsi à tout mot $x \in \delta(S)$, on peut associer un unique état q tel que $\forall l \in \Sigma, |x|_l \in q(l)$, notons cet état $cat(x)$

Les arêtes du graphe sont les dérivation possibles d'un famille majorante $q \in Q$ à une autre $q' \in Q$.

$$\begin{aligned} (q, q') \in A_0 &\iff \exists x, x' \in \Sigma^* / x \rightarrow x' \text{ et } \forall l \in \Sigma, |x|_l \in q(l) \text{ et } |x'|_l \in q'(l) \\ &\iff \exists x, x' \in \Sigma^* / x \rightarrow x' \text{ et } cat(x) = q \text{ et } cat(x') = q' \end{aligned}$$

Ainsi, si x correspond à un état q , et x' à un état q' , alors $x \xrightarrow{*} x' \implies q'$ est accessible depuis q dans tout graphe A majorant A_0 .

Donc $\forall l \in \Sigma, s_x(l)$ est l'union des $q(l)$ où q est accessible depuis $cat(x)$ dans un graphe A majorant A_0 .

Si m' n'est pas accessible depuis x dans un graphe A majorant A_0 , alors on peut interdire le parcours passant par x , **interdit x** renverra vrai.

Les états q sous la forme $q \in Q = \{\{0\}, \mathbb{N}^*\}^\Sigma$

Exemple avec la grammaire suivante :

$$D := \{(S, \underline{abc})_0, (\underline{abc}, \underline{ab})_1, (\underline{b}, \underline{k})_2, (\underline{c}, \underline{ak})_3, (\underline{kak}, \underline{aa})_4, (\underline{a}, \underline{aaa})_5\}$$

Voici un graphe A majorant A_0 (les arêtes sont étiquetées par les indices de règles de dérivation donnant l'arête):

Et voici ce à quoi correspondent les différents états (sommets) du graphe :

Le graphe réduit suivant sera utile :

Ainsi, nous pouvons tout de suite affirmer qu'il est impossible de dériver aaabakab en akkckkaaakck (en effet, ψ n'est pas accessible depuis δ)

Intéressons nous au calcul des arrêtes A du graphe.

Considérons d'abord les arêtes partant d'un état q donné, puis celles correspondant à une dérivation d donnée.

$$A_{q,d} := \{(q, q') \in Q^2 \mid \exists x, x' \in \Sigma^*/x \xrightarrow{d} x' \text{ et } cat(x) = q \text{ et } cat(x') = q'\}$$

$$A_{q,(a,b)} \subset \left\{ (q, q') \in Q^2 \mid \left\{ \begin{array}{l} cat(a) \preceq q \text{ et } q - cat(a) \preceq q' \text{ et} \\ \forall l \in \Sigma, \\ (cat(b)(l) = \mathbb{N}^* \implies q'(l) = \mathbb{N}^*) \text{ et} \\ (q(l) = \{0\} \text{ et } cat(b)(l) = \{0\} \implies q'(l) = \{0\}) \end{array} \right\} \right\}$$

Pour $(q \preceq q') \iff \forall l \in \Sigma, \max q(l) \leq \min q'(l)$ et

$$(q - q')(l) = \begin{cases} \{0\} & \text{si } q'(l) = \mathbb{N} \text{ ou } q(l) = \{0\} \\ \mathbb{N} & \text{sinon} \end{cases}$$

On peut alors majorer l'ensemble des arêtes du graphe.

Maintenant, une fois le graphe majorant associé à notre grammaire calculé, on peut faire notre fonction `interdit`, `interdit x` renvoie vrai si et seulement si x n'est pas accessible depuis S dans le graphe majorant. (On peut précalculer le graphe réduit où les sommets sont les composantes fortement connexes et sa matrice d'accessibilité avec l'algorithme Floyd Warshall par exemple, pour réduire les temps de calcul).

Il peut être pertinent d'étendre l'étude au cas $Q \subset \{2\mathbb{N}, 2\mathbb{N} + 1\}^\Sigma$.

6 ANNEXE 1 : Notations et rappel des définitions

- On considère la grammaire formelle $G = (T, N_t, S, D)$ où T est l'ensemble des terminaux, N_t l'ensemble de non-terminaux, S , l'axiome et D l'ensemble des règles de dérivation.
- $\Sigma := N_t \cup T$
- $x \xrightarrow{(a,b)} x' \iff \exists u, v \in \Sigma^* / x = uav \text{ et } x' = ubv \text{ si } x' \text{ dérive directement depuis } x \text{ par } (a, b).$
- $\rightarrow := \bigcup_{d \in D} \xrightarrow{d}, x \rightarrow x' \text{ si } x' \text{ dérive directement depuis } x. (x' \in \mathcal{S}(x))$
- Pour $x \in \Sigma^*, \mathcal{S}(x) := \{y \in \Sigma^* \mid x \rightarrow y\}$ ensemble des mots directement dérivables depuis x .
- $\xrightarrow{*} := \bigcup_{n \in \mathbb{N}} \rightarrow^n$ est la clôture transitive et réflexive de \rightarrow , $x \xrightarrow{*} x'$ si x' dérive depuis x . ($x' \in \delta(x)$)
- Pour $x \in \Sigma^*, \delta(x) := \{y \in \Sigma^* \mid x \xrightarrow{*} y\}$ ensemble des mots dérivables depuis x .
- $\mathcal{L}(G) := \delta(S) \cap T^*$ le langage de la grammaire.
- Pour $x \in \Sigma^*, l \in \Sigma, |x|_l := |\{1 \leq i \leq |x| \mid x_i = l\}|$ le nombre d'occurrences de la lettre l dans x .
- Pour $x \in \Sigma^*, l \in \Sigma, s_x(l)$ est un ensemble qui majore $|\delta(x)|_l = \{|y|_l \mid x \xrightarrow{*} y\}$
- $Q \subset (\mathcal{P}(\mathbb{N}))^\Sigma$ ensemble d'états / familles majorantes / sommets.
- $cat(x)$ unique état du graphe A_0 contenant le nombre d'occurrence de chaque lettre de x .
- A_0 graphe des dérivations possibles entre familles majorantes.
- A graphe majorant A_0 . $((q, q') \in A_0 \implies (q, q') \in A).$

7 ANNEXE 2 : Implémentation logicielle (OCaml)

```
1  (##### TYPES #####)
2
3  type 'e caractere = T of 'e | Nt of int
4
5  type 'a regle = ('a array) * ('a array)
6
7  type 'e fg = {
8    terminaux : ('e caractere) array ;
9    nbnonterminaux : int ;
10   axiome : 'e caractere ;
11   reglesf : ('e caractere) regle array
12 }
13
14 type 'e preuveformelle = (('e caractere list)*int*int) list
15
16
17
18 (##### UTILES #####)
19
20 (* Donne le tableau des lettres de la grammaire *)
21 let lettres gram = Array.append gram.terminaux (Array.mapi (fun
22   i x -> Nt i) (Array.make (gram.nbnonterminaux) 0))
23
24
25 let implies a b = (not a) || b
26
27 (* Genere tous les n uplets dans {0,1}* *)
28 let rec nuplets n =
29   if n = 1 then [[0];[1]]
30   else
31     let autre = nuplets (n-1) in
32     (List.map (fun l -> 0::l) autre )@( List.map (fun l -> 1::l)
33       autre)
34
35
36 (* Produit cartésien *)
37 let cartesian l l' =
38   List.concat (List.map (fun e -> List.map (fun e' -> (e,e')) l
39     ') l)
40
41
42 (* Implementation de kmp *)
43 let kmppreprocess w =
44   let n = Array.length w in
45   let pos = ref 1 in
46   let cnd = ref 0 in
```

```

44  let t = Array.make (n+1) (-1) in
45  while (!pos) < n do
46    if w.(!pos) = w.(!cnd) then
47      (t.(!pos) <- t.(!cnd);)
48    else
49      (
50        t.(!pos) <- !cnd;
51        while (!cnd >= 0 && w.(!pos) <> w.(!cnd) ) do
52          cnd:=t.(!cnd);
53        done;
54      );
55    pos:=(!pos)+1;
56    cnd:=(!cnd)+1;
57  done;
58  t.(!pos) <- (!cnd);
59  t;;
60
61  let kmp s w t =
62    let ns = Array.length s in
63    let nw = Array.length w in
64    let j = ref 0 in
65    let k = ref 0 in
66    let res = ref [] in
67    while (!j) < ns do
68      if w.(!k) = s.(!j) then
69        (
70          j:=(!j)+1;
71          k:=(!k)+1;
72          if (!k)=nw then
73            (
74              res:=((!j)-(!k))::(!res);
75              k:=t.(!k);
76            );
77          )
78      else
79        (
80          k:=t.(!k);
81          if (!k) < 0 then
82            (
83              j:=(!j)+1;
84              k:=(!k)+1;
85            );
86          );
87      done;
88      !res;;
89
90  (* remplace x i l b remplace dans x le sous mot de longueur l
    qui commence a l'indice i par le mot b *)
91  let remplace x i l b =

```



```

92   let n = Array.length x in
93   let m = Array.length b in
94   if i >= n || n < i+1 then failwith "OOH" else
95   let res = Array.make (n-l+m) x.(0) in
96   for j = 0 to (n-l+m-1) do
97     if (j < i) then
98       (
99         res.(j) <- x.(j)
100      ) else
101      if (j >= i+m) then
102        (
103          res.(j) <- x.(j-m+1)
104        )
105      else
106        (
107          res.(j) <- b.(j-i)
108        );
109   done;
110   res
111
112 let ajoute e l = if List.mem e l then l else e::l
113
114 let rec ajouteplein l1 l2 =
115   match l1 with
116   | [] -> l2
117   | t::q -> (ajouteplein q (ajoute t l2))
118
119 (* Effectue un pretraitement (kmp) des membres de gauche des
    regles de derivation *)
120 let preprocessgf grf =
121   Array.map
122   (fun (a,b) ->
123     (a,b,kmppreprocess a)
124   )
125   grf.reglesf
126
127 let nboccur m l = List.length ((List.filter (fun x -> x = l) ) (
    Array.to-list m))
128
129 (* Parcours en largeur *)
130 let rec bfs g dejaVus aVoir =
131   match aVoir with
132   | [] -> dejaVus
133   | tete::queue -> if List.mem tete dejaVus
134     then bfs g dejaVus queue
135     else bfs g (tete::dejaVus) (queue@(g tete));;
136
137 (* Donne le nombre d'occurences des lettres de la grammaire dans
    le mot *)

```

```

138 let analysemot mot gram =
139   let n = gram.nbsonterminaux in
140   let m = Array.length gram.terminaux in
141   let res = Array.make (n+m) 0 in
142   Array.mapi (fun i l -> if i < n then nboccur mot (Nt i) else
        nboccur mot (gram.terminaux.(i-n)) ) res
143
144 let ordre a b = (Array.length a <= Array.length b ) && (
145   let res = ref true in
146   for i=0 to ((Array.length a ) - 1) do
147     if b.(i) < a.(i) then res:=false;
148   done;
149   !res
150 )
151
152 let moins a b =
153   let res = Array.make (Array.length a) 0 in
154   for i=0 to ((Array.length a) - 1) do
155     res.(i) <- a.(i)-b.(i);
156   done;
157   res
158
159 (*
160  Un parcours en largeur qui
161  -elimine des chemins passant par un sommet invalide
162  -s'arrete des qu'il parcourt un sommet valide
163  PS :
164  -dans avoir il y a des chemins
165  -retourne un chemin.
166  *)
167 let rec parcouresmagique delta elimine termine dejavu avoir =
168   let navoir = ref [] in
169   let ndejavu = ref dejavu in
170   if avoir = [] then None else (
171     let res =
172       List.find_opt
173       (function
174        |[] -> failwith "mauvais chemin"
175        |s::q ->
176          ndejavu := ajoute s (!ndejavu);
177          if (termine s) then true else
178          if (elimine s || List.mem s dejavu) then false else
179          (
180            navoir :=
181              ajouteplein
182              (
183                List.map
184                (fun v -> v::s::q)
185                (delta s)

```

```

186         )
187         (!navoir);
188         false)
189     )
190     avoir
191     in
192     match res with
193     |None -> parcoursmagique delta elimine termine (!ndejavu) (!
194         navoir)
195     |Some x -> Some x
196 )
197 (* Retourne les mots vers lesquels x peut derivier une fois *)
198 let succ ppregles x =
199     let res =
200         List.flatten
201         (
202             Array.toList (
203                 Array.map
204                 (fun (a,b,pp) ->
205                     List.map
206                     (fun i ->
207                         remplace x i (Array.length a) b
208                     )
209                     (kmp x a pp)
210                 )
211                 ppregles
212             )
213         )
214     in
215     ajouteplein res []
216
217 (* Retourne les mots vers lesquels x peut derivier une fois et
218     comment il derive *)
219 let succbis ppregles (x,-,-) =
220     let res =
221         List.flatten
222         (
223             Array.toList (
224                 Array.map
225                 (fun numregle (a,b,pp) ->
226                     List.map
227                     (fun i ->
228                         (remplace x i (Array.length a) b,numregle,i)
229                     )
230                     (kmp x a pp)
231                 )
232                 ppregles
233             )

```

```

233 )
234 in
235   ajouteplein res []
236
237 (* Cherche une derivation de x vers m *)
238 let recherchesuitemots x m ppregles =
239   parcoursmagique
240   (succ ppregles)
241   (fun x -> false)
242   (fun x -> x = m)
243   []
244   [[x]]
245
246 (* La meme fonction mais fourni une preuve dans le type
    preuveformelle *)
247 let recherchepreuvenaif gram ppregles x m =
248   parcoursmagique
249   (succbis ppregles)
250   (fun (y,a,b) -> false)
251   (fun (y,a,b) -> y = m)
252   []
253   [[x,0,0]]
254
255 (* Donne cat(m) pour des etats Q donnees par fctcat et pour la
    grammaire gram*)
256 let categorisemot gram fctcat m = Array.map fctcat (analysemot m
    gram)
257
258 (* Fonction categorisante binaire *)
259 let fct_cat_bin n = if n > 0 then 1 else 0
260
261 (* Fonction cat pour notre categorisation *)
262 let cat gram = categorisemot gram fct_cat_bin
263
264 (* Verifie si il existe un mot de categorie q derivable via la
    derivation d dans la grammaire gram *)
265 let estpossible q d gram =
266   let (a,b) = d in
267   ordre (cat gram a) q
268
269 (* Construit le graphe A *)
270 let grapheA gram =
271   fun q ->
272     let leslettres = lettres gram in
273     List.concat_map
274     (fun (i,(a,b)) ->
275       let leres =
276         List.filter
277         (fun qp ->

```

```

278         (ordre (cat gram a) q) &&
279         (ordre (moins q (cat gram a)) qp) &&
280         (
281             List.for_all
282             (fun l ->
283                 (
284                     implies ((cat gram b).(1) = 1) (qp.(1) = 1)
285                 )&&(
286                     implies (q.(1) = 0 && (cat gram b).(1) = 0) (qp
287                         .(1) = 0)
288                 )
289             (Array.to_list (Array.mapi (fun i x -> i) (Array.
290                 make (Array.length leslettres) 0)))
291         )
292         (enarray (nuplets (Array.length leslettres)))
293         (*in (i, leres)*)
294         in leres
295     )
296     (Array.to_list (Array.mapi (fun i x -> (i, x) ) gram.reglesf)
297 )
298 (* Finalement, la fonction interdit voulue *)
299 let interditfort gram m x =
300     not (List.mem (cat gram m) (bfs (grapheA gram) [] [cat gram x]
301 ) )
302
303 (* Recherche une preuve que l'on peut derivier x en m en
304     utilisant la simplification de complexite *)
305 let recherchepreuve gram ppregles x m =
306     parcoursmagique
307     (succbis ppregles)
308     (fun (y,a,b) -> let res = interditfort gram m y in if res then
309         print_int 1 ; res)
310     (fun (y,a,b) -> y = m)
311     []
312     [[x,0,0]]
313
314
315 (##### IMPLEMENTATION DU CODE #####)
316
317 (* Un exemple de grammaire formelle *)
318 let exfg = {
319     terminaux = [| T 'a' ; T 'b' ; T 'c' ; T 'k' |];
320     nbnonterminaux = 1;
321     axiome = Nt 0;

```

```

321   reglesf = [|
322     [|Nt 0|], [|T 'a' ; T 'b' ; T 'c'|] ;
323     [|T 'a' ; T 'b' ; T 'c'|], [|T 'a' ; T 'b'|] ;
324     [|T 'b'|], [|T 'k'|] ;
325     [|T 'c'|], [|T 'a' ; T 'k'|] ;
326     [|T 'k' ; T 'a' ; T 'k'|], [|T 'a' ; T 'a'|] ;
327     [|T 'a'|], [|T 'a' ; T 'a' ; T 'a'|]
328   |]
329 }
330
331 let exreglespp = preprocessgf exfg
332
333 let m = [|T 'a' ; T 'k' ; T 'k' ; T 'c' ; T 'c' ; T 'k' ; T 'a'
          ; T 'a' ; T 'a' ; T 'k' ; T 'c' ; T 'k'|]
334
335 let x = [|T 'a' ; T 'a' ; T 'a' ; T 'b' ; T 'a' ; T 'k' ; T 'a'
          ; T 'b'|]
336
337 let reszero = interditfort exfg m x
338
339 let resun = recherchepreuvenaif exfg exreglespp [|Nt 0|] [|T 'a'
          ; T 'a' ; T 'a' ; T 'k' ; T 'a' ; T 'a' ; T 'a' ; T 'k'|]
340
341 let resdeux = recherchepreuve exfg exreglespp [|Nt 0|] [|T 'a' ;
          T 'a' ; T 'a' ; T 'k' ; T 'a' ; T 'a' ; T 'a' ; T 'k'|]

```

8 ANNEXE 3 : Resultat de l'exécution logicielle

Voici la fin de ce que le programme exécuté en toplevel retourne :

```
1  val reszero : bool = true
2  val resun : (char caractere array * int * int) list option =
3    Some
4      [([|T 'a'; T 'a'; T 'a'; T 'k'; T 'a'; T 'a'; T 'a'; T 'k'|] ,
5        5, 4);
5        ([|T 'a'; T 'a'; T 'a'; T 'k'; T 'a'; T 'k'|] , 3, 4);
6        ([|T 'a'; T 'a'; T 'a'; T 'k'; T 'c'|] , 2, 3);
7        ([|T 'a'; T 'a'; T 'a'; T 'b'; T 'c'|] , 5, 0);
8        ([|T 'a'; T 'b'; T 'c'|] , 0, 0); ([|Nt 0|] , 0, 0)]
9  111val resdeux : (char caractere array * int * int) list option
    =
10    Some
11      [([|T 'a'; T 'a'; T 'a'; T 'k'; T 'a'; T 'a'; T 'a'; T 'k'|] ,
12        5, 4);
12        ([|T 'a'; T 'a'; T 'a'; T 'k'; T 'a'; T 'k'|] , 3, 4);
13        ([|T 'a'; T 'a'; T 'a'; T 'k'; T 'c'|] , 2, 3);
14        ([|T 'a'; T 'a'; T 'a'; T 'b'; T 'c'|] , 5, 0);
15        ([|T 'a'; T 'b'; T 'c'|] , 0, 0); ([|Nt 0|] , 0, 0)]
```

On a donc seulement trois instants dans le parcours où l'amélioration a été utile pour cette grammaire (les trois '1' affichés ligne 9 lors de l'exécution de `interditfort`).