Sur la vérification de preuve et la preuve automatique d'appartenance d'un mot à une grammaire.

Ulysse Durand

Contents

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 $\Sigma := N_t \cup T$
- $D \subset (\Sigma^\star)^2$, est l'ensemble des règles de dérivation.

Pour $(a, b) \in D$, soit $\overset{(a,b)}{\to}$ la relation binaire définie sur $\Sigma^\star$ par

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

On note $\to := \bigcup_{d \in D} \overset{d}{\to}$ et on note $\overset{*}{\to}$ la cloture transitive et réflexive de $\to$.
Pour $x \in \Sigma^\star$, notons
$\delta(x) := \{y \in \Sigma^\star / x \overset{*}{\to} y\}$
$|x|_l := |\{i \in \mathbb{N} \mid x_i = l\}|$ est le nombre d'occurences de la lettre l dans x.

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

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

Nous allons supposer que N_t est dénombrable et T est fini.

cf ANNEXE2 typesetutiles debut types l.

2 Vérification de preuve - grammaire formelle

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 $m_1, \ldots, m_n$ tels que $\forall i \in [1, n-1], m_i \to m_{i+1}$. 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 = (int*int*('e caractere list)) 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 $\underline{\text{aabbcc}}$ 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 = [(0,1,mot "aBSc");(2,3,mot "abc");(1,3,mot "aB");(2,4,mot "bb")]
```

cf ANNEXE2 verifderiv l.
cf ANNEXE2 verifpreuve l.

3 Preuve automatique - grammaire formelle

Nous allons générer successivement les $G_n := \{x \in \Sigma^\star \mid S(\bigcup_{0 \leq k \leq n} \rightarrow^k)x\}$ à l'aide de $G_{n+1} = \bigcup_{x \in G_n} \mathcal{S}(x)$ où $\mathcal{S}(x) := \{y \in \Sigma^\star \mid x \rightarrow y\} = \bigcup_{d \in D} \{y \in \Sigma^\star \mid x \xrightarrow{d} y\}$ 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 code preprocessgf et succ qui calcule $\mathcal{S}(x)$

Un parcours en largeur particulier

Nous allons utiliser un parcours qui nous permettra de trouver dans un graphe un chemin plus court entre le sommet initial et un sommet 'valide'. Ce parcours devra aussi éviter les chemins passant par un sommet 'interdit'.

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 renvoit 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.

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 $\Sigma^\star$, et les arêtes sont $\{(a,b) \in \Sigma^\star \mid a \to b\} = \{(a,b) \in \Sigma^\star \mid b \in \mathcal{S}(a)\}$.

cf ANNEXE2 chercherderivationnaif l.

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 précédement 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'occurence 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 $l \in \Sigma, x \in \Sigma^\star$, 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'occurence 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 :

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

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 $q' \in Q$.

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

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.

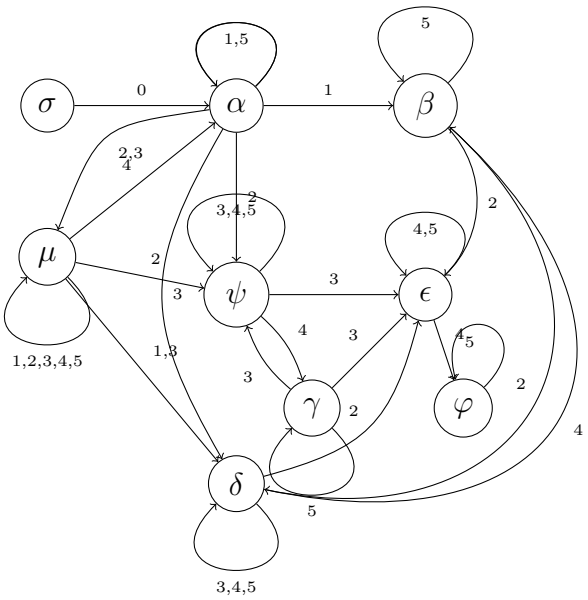
Si m' n'est pas accessible depuis x dans un graphe A majorant A_0, alors on peut interdire le parcours passant par x, `interdit x` renverra vrai.

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

Exemple avec la grammaire suivante :

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

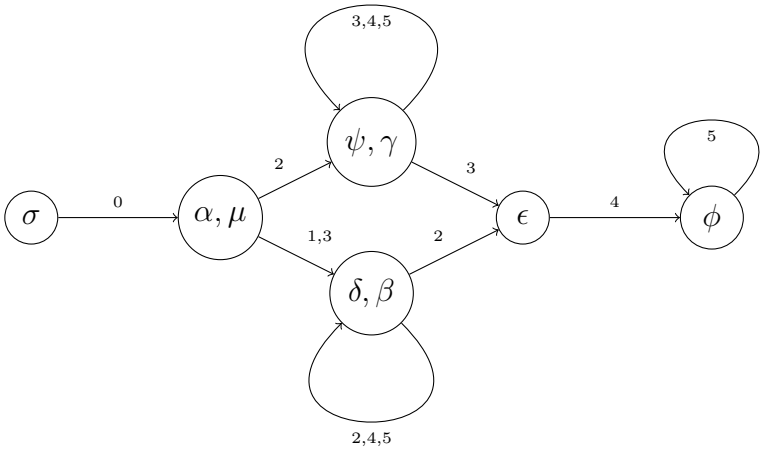
Voici un graphe A majorant A_0 (les arêtes sont étiquetées par les indices de règles de dérivation donnant l'arête):



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

q	α	β	γ	δ	ϵ	φ	ψ	μ	σ
$q(a)$	$\mathbb{N}^*$	$\mathbb{N}^*$	$\mathbb{N}^*$	$\mathbb{N}^*$	$\mathbb{N}^*$	$\mathbb{N}^*$	$\mathbb{N}^*$	$\mathbb{N}^*$	$\{0\}$
$q(b)$	$\mathbb{N}^*$	$\mathbb{N}^*$	$\{0\}$	$\mathbb{N}^*$	$\{0\}$	$\{0\}$	$\{0\}$	$\mathbb{N}^*$	$\{0\}$
$q(c)$	$\mathbb{N}^*$	$\{0\}$	$\mathbb{N}^*$	$\{0\}$	$\{0\}$	$\{0\}$	$\mathbb{N}^*$	$\mathbb{N}^*$	$\{0\}$
$q(k)$	$\{0\}$	$\{0\}$	$\{0\}$	$\mathbb{N}^*$	$\mathbb{N}^*$	$\{0\}$	$\mathbb{N}^*$	$\mathbb{N}^*$	$\{0\}$
$q(S)$	$\{0\}$	$\{0\}$	$\{0\}$	$\{0\}$	$\{0\}$	$\{0\}$	$\{0\}$	$\{0\}$	$\mathbb{N}^*$

Le graphe réduit suivant sera utile :



Ainsi, nous pouvons tout de suite affirmer qu'il est impossible de dériver aaabakab en akkcckaaakck (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.

$$A_{q,d} := \{(a,b) \in A_q \mid \exists x, x' \in \Sigma^\star / x \xrightarrow{d} x' \text{ et } cat(x) = q \text{ et } cat(x') = q'\}$$

$$A_{q,(a,b)} \subset \begin{cases} \emptyset \text{ si } cat(a) \neq q \\ \{(q,q') \in Q^2 \mid \forall l \in \Sigma, (cat(b)(l) = \mathbb{N}^* \implies q'(l) = \mathbb{N}^*) \text{ et} \\ (q(l) = \{0\} \text{ et } cat(b)(l) = \{0\} \implies q(l) = \{0\})\} \text{ sinon} \end{cases}$$

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` renvoit 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.
- $\Sigma := N_t \cup T$
- $x \overset{(a,b)}{\to} x' \iff \exists u, v \in \Sigma^\star / x = uav$ et $x' = ubv$ si x' dérive directement depuis x par (a, b).
- $\to := \bigcup_{d \in D} \overset{d}{\to}$, $x \to x'$ si x' dérive directement depuis x. $(x' \in \mathcal{S}(x))$
- Pour $x \in \Sigma^\star$, $\mathcal{S}(x) := \{y \in \Sigma^\star \mid x \to y\}$ ensemble des mots directements dérivables depuis x.
- $\overset{*}{\to} := \bigcup_{n \in \mathbb{N}}$ est la cloture transitive et réflexive de $\to$, $x \overset{*}{\to} x'$ si x' dérive depuis x. $(x' \in \delta(x))$
- Pour $x \in \Sigma^\star$, $\delta(x) := \{y \in \Sigma^\star \mid x \overset{*}{\to} y\}$ ensemble des mots dérivables depuis x.
- $\mathcal{L}(G) := \delta(S) \cap T^\star$ le langage de la grammaire.
- Pour $x \in \Sigma^\star$, $|x|_l := |\{1 \leq i \leq |x| \mid x_i = l\}|$ le nombre d'occurences de la lettre l dans x.
- $Q \subset (\mathcal{P}(\mathbb{N}))^\Sigma$ ensemble d'états / familles majorantes / sommets.
- $cat(x)$ unique état du graphe A_0 contenant le nombre d'occurence 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 array) list
15
16
17
18
19
20
21 (*##### UTILES #####*)
22
23 (* Implementation de kmp *)
24 let kmppreprocess w =
25  let n = Array.length w in
26  let pos = ref 1 in
27  let cnd = ref 0 in
28  let t = Array.make (n+1) (-1) in
29  while (!pos) < n do
30   if w.(!pos) = w.(!cnd) then
31    (t.(!pos) <- t.(!cnd);)
32   else
33   (
34    t.(!pos) <- !cnd;
35    while (!cnd >= 0 && w.(!pos) <> w.(!cnd) ) do
36     cnd:=t.(!cnd);
37    done;
38   );
39   pos:=(!pos)+1;
40   cnd:=(!cnd)+1;
41  done;
42  t.(!pos) <- (!cnd);
43  t;;
44
45 let kmp s w t =
46  let ns = Array.length s in
```

```
47   let nw = Array.length w in
48   let j = ref 0 in
49   let k = ref 0 in
50   let res = ref [] in
51   while (!j) < ns do
52    if w.(!k) = s.(!j) then
53    (
54     j:=(!j)+1;
55     k:=(!k)+1;
56     if (!k)=nw then
57     (
58      res:=((!j)-(!k))::(!res);
59      k:=t.(!k);
60     );
61    )
62    else
63    (
64     k:=t.(!k);
65     if (!k) < 0 then
66     (
67      j:=(!j)+1;
68      k:=(!k)+1;
69     );
70    );
71   done;
72   !res;;
73
74  (*remplace x i l b remplace dans x le sous mot de longueur l qui
        commence a l'indice i par le mot b*)
75  let remplace x i l b =
76    let n = Array.length x in
77    let m = Array.length b in
78    if i >= n || n < i+l then failwith "OOH" else
79    let res = Array.make (n-l+m) x.(0) in
80    for j = 0 to (n-l+m-1) do
81      if (j < i) then
82      (
83        res.(j) <- x.(j)
84      ) else
85      if (j >= i+m) then
86      (
87        res.(j) <- x.(j-m+l)
88      )
89      else
90      (
91        res.(j) <- b.(j-i)
92      );
93    done;
94    res
```

```
95
96  let ajoute e l = if List.mem e l then l else e::l
97
98  let rec ajouteplein l1 l2 =
99   match l1 with
100    |[] -> l2
101    |t::q -> (ajouteplein q (ajoute t l2))
102
103
104 (* Effectue un pretraitement (kmp) des membres de gauche des
        regles de derivation *)
105 let preprocessgf grf =
106   Array.map
107   (fun (a,b) ->
108     (a,b,kmppreprocess a)
109   )
110   grf.reglesf
111
112 let nboccur m l = List.length ((List.filter (fun x -> x = l) ) (
        Array.to_list m))
113
114 (* Donne le nombre d'occurences des lettres de la grammaire dans
         le mot *)
115 let analysemot mot gram =
116  let n = gram.nbnonterminaux in
117  let m = Array.length gram.terminaux in
118  let res = Array.make (n+m) 0 in
119  Array.mapi (fun i l -> if i < n then nboccur mot (Nt i) else
        nboccur mot (gram.terminaux.(i-n)) ) res
120
121 let categorisemot gram fctcat m = Array.map fctcat (analysemot m
        gram)
122
123 (*
124 Un parcours en largeur qui
125   -elimine des chemins passant par un sommet invalide
126   -s'arrete des qu'il parcourt un sommet valide
127 PS :
128   -dans avoir il y a des chemins
129   -retourne un chemin.
130 *)
131 let rec parcoursmagique delta elimine termine dejavu avoir =
132   let navoir = ref [] in
133   let ndejavu = ref dejavu in
134   if avoir = [] then None else (
135   let res =
136     List.find_opt
137     (function
138       |[] -> failwith "mauvais chemin"
```

```
139         |s::q ->
140           ndejavu := ajoute s (!ndejavu);
141           if (termine s) then true else
142           if (elimine s || List.mem s dejavu) then false else
143           (
144           navoir :=
145             ajouteplein
146             (
147               List.map
148               (fun v -> v::s::q)
149               (delta s)
150             )
151             (!navoir);
152           false)
153       )
154       avoir
155       in
156     match res with
157       |None -> parcoursmagique delta elimine termine (!ndejavu) (!
              navoir)
158       |Some x -> Some x
159     )
160
161   (* Retourne les mots vers lesquels x peut deriver une fois *)
162   let succ ppregles x =
163     let res =
164     List.flatten
165     (
166       Array.to_list (
167         Array.map
168         (fun (a,b,pp) ->
169           List.map
170           (fun i ->
171             remplace x i (Array.length a) b
172           )
173           (kmp x a pp)
174         )
175         ppregles
176       )
177     )
178     in
179     ajouteplein res []
180
181   (* Retourne les mots vers lesquels x peut deriver une fois *)
182   let succbis ppregles (x,_,_) =
183     let res =
184     List.flatten
185     (
186       Array.to_list (
```

```
187         Array.mapi
188         (fun numregle (a,b,pp) ->
189           List.map
190           (fun i ->
191             (remplace x i (Array.length a) b,numregle,i)
192           )
193           (kmp x a pp)
194         )
195         ppregles
196       )
197     )
198     in
199     ajouteplein res []
200
201   (* Cherche une derivation de x vers m *)
202   let chercherderivationnaif x m ppregles =
203     parcoursmagique
204     (succ ppregles)
205     (fun x -> false)
206     (fun x -> x = m)
207     []
208     [[x]]
209
210   let lememeavecderivations x m ppregles =
211     parcoursmagique
212     (succbis ppregles)
213     (fun (y,a,b) -> false)
214     (fun (y,a,b) -> y = m)
215     []
216     [[x,0,0]](* Fonction categorisante binaire *)
217   let fct_cat_bin n = if n > 0 then 1 else 0
218
219   (* Verifie si il existe un mot de categorie q derivable via la
        derivation d dans la grammaire gram *)
220   let estpossible q d gram =
221     let (a,b) = d in
222     let cata = categorisemot gram fct_cat_bin a in
223     q = cata
224
225   (* Donne l'ensemble des indices des derivations faisables depuis
        l'etat q *)
226   let lesucc gram q =
227     List.map
228     fst
229     (
230       List.filter
231       (
232         fun (i,d) -> estpossible q d gram
233       )
```

```
234       (
235         Array.to_list
236         (
237           Array.mapi (fun i d -> (i,d)) gram.reglesf
238         )
239       )
240     )
241
242   let vrailesucc gram q =
243
244
245   (* Donne, pour la derivation i, l'etat-couple vers lequel on
        arrive *)
246   let etatsdederiv gram i =
247     let (a,b) = gram.reglesf.(i) in
248     (i,categorisemot gram fct_cat_bin b)
249
250   (* Donne, les couple de categories des derivations d'une
        grammaire*)
251   let pretraitegram gram =
252     Array.map
253     (fun (a,b) -> (categorisemot gram fct_cat_bin a,categorisemot
          gram fct_cat_bin b))
254     gram.reglesf
255
256   (*  *)
257
258
259   (* Donne une table d'association des etats accessibles *)
260   let pretraitebisgram gram =
261     let avoir = ref [categorisemot gram fct_cat_bin [|Nt 0|]] in
262     let dejavu = ref [] in
263     while (!avoir != []) do
264       let t::q = !avoir in
265       if not (List.mem t (!dejavu)) then (
266         let suivants = lesucc gram t in
267         let vraisuivants = (
268           List.map
269           (fun i -> categorisemot gram fct_cat_bin (snd gram.
                reglesf.(i)))
270           suivants
271         ) in
272         avoir := ajouteplein vraisuivants q;
273         dejavu := t::(!dejavu);
274       );
275     done;
276     Array.of_list (List.rev (!dejavu))
277
278
```

```
279
280 (* Un exemple de grammaire formelle *)
281 let exfg = {
282   terminaux = [| T 'a' ; T 'b' ; T 'c' ; T 'k' |];
283   nbnonterminaux = 1;
284   axiome = Nt 0;
285   reglesf = [|
286     [|Nt 0|],[|T 'a' ; T 'b' ; T 'c'|] ;
287     [|T 'a' ; T 'b' ; T 'c'|],[|T 'a' ; T 'b'|] ;
288     [|T 'b'|],[|T 'k'|] ;
289     [|T 'c'|],[|T 'a' ; T 'k'|] ;
290     [|T 'k' ; T 'a' ; T 'k'|],[|T 'a' ; T 'a'|] ;
291     [|T 'a'|],[|T 'a' ; T 'a' ; T 'a'|]
292   |]
293 }
294 let unmachin = succ (preprocessgf exfg) [|T 'a' ; T 'b' ; T 'c'
      |]
295
296 let resultat = lememeavecderivations [|Nt 0|] [|T 'a' ; T 'a' ;
     T 'a' ; T 'k'|] (preprocessgf exfg)
297
298 let printcarac c = match c with
299   |T c -> print_char c
300   |Nt i -> print_int i
301
302 let printmot = List.iter printcarac
303
304 let pretraited = pretraitebisgram exfg
305
306 let test = estpossible [|0;1;1;1;0|] exfg.reglesf.(3) exfg
```