Vérification et preuve automatique d'appartenance d'un mot à une grammaire formelle.

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 $\stackrel{(a,b)}{\rightarrow}$ la relation binaire définie sur $\Sigma^\star$ par

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

On note $\rightarrow := \bigcup_{d \in D} \stackrel{d}{\rightarrow}$ et on note $\stackrel{*}{\rightarrow}$ la cloture transitive et réflexive de $\rightarrow$.
Pour $x \in \Sigma^\star$, notons
$\delta(x) := \{y \in \Sigma^\star / x \stackrel{*}{\rightarrow} y\}$
$|x|_l := |\{i \in \mathbb{N} \mid x_i = l\}|$ 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 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 $m_1, \ldots, m_n$ 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 $\underline{\text{ aabbcc }}$ 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 = [(mot "aBSc",0,1);(mot "abc",2,3);
 (mot "aB",1,3);(mot "bb",2,4)]
```

3 Preuve automatique

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

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

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 :

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

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.

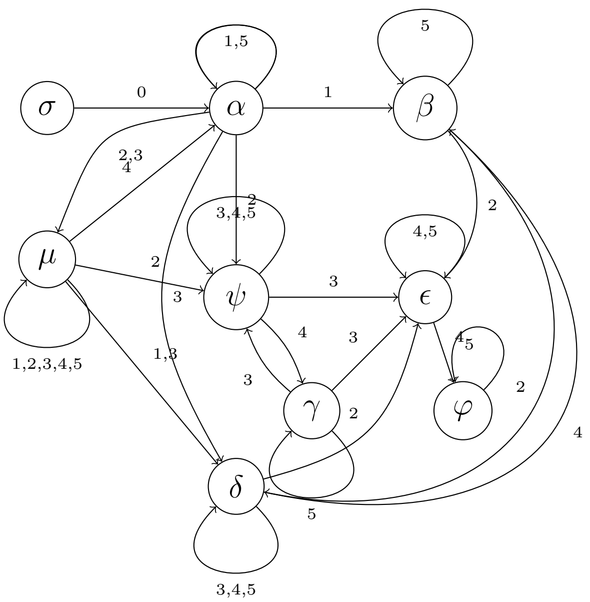
Si m' n'est pas accessible depuis x dans un graphe A majorant A_0, alors on peut interdire le parcours passant par x, `interdit x` renverra vrai.

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

Exemple avec la grammaire suivante :

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

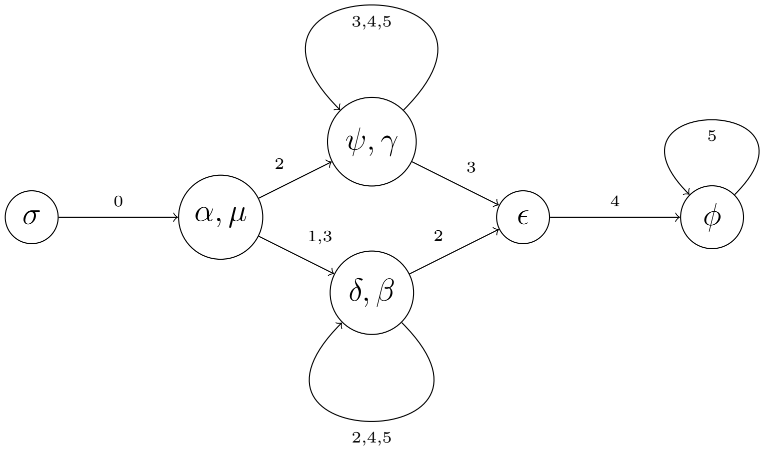
Voici un graphe A majorant A_0 (les arêtes sont étiquetées par les indices de règles de dérivation donnant l'arête):



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.

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

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

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` 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).

- $\to := \bigcup\limits_{d \in D} \overset{d}{\to}$, $x \to x'$ 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.

- $\overset{*}{\to} := \bigcup\limits_{n \in \mathbb{N}} \to^n$ 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 $x \in \Sigma^\star, l \in \Sigma, |x|_l := |\{1 \leq i \leq |x| \mid x_i = l\}|$ le nombre d'occurences de la lettre l dans x.

- Pour $x \in \Sigma^\star, l \in \Sigma, s_x(l)$ est un ensemble qui majore $|\delta(x)|_l = \{|y|_l \mid x \overset{*}{\to} y\}$

- $Q \subset (\mathcal{P}(\mathbb{N}))^\Sigma$ ensemble d'états / familles majorantes / sommets.

- $cat(x)$ unique état du graphe A_0 contenant le nombre d'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)

```
(*##### TYPES #####*)

type 'e caractere = T of 'e | Nt of int

type 'a regle = ('a array) * ('a array)

type 'e fg = {
  terminaux : ('e caractere) array ;
  nbnonterminaux : int ;
  axiome : 'e caractere ;
  reglesf : ('e caractere) regle array
}

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



(*##### UTILES #####*)

(* Donne le trableau des lettres de la grammaire *)
let lettres gram = Array.append gram.terminaux (Array.mapi (fun i x -> Nt i) (Array.make (gram.nbnonterminaux) 0))

let implies a b = (not a) || b

(* Genere tous les n uplets dans {0,1} *)
let rec nuplets n =
  if n = 1 then [[0];[1]]
  else
  let autre = nuplets (n-1) in
  (List.map (fun l -> 0::l) autre )@( List.map (fun l -> 1::l) autre)

(* Produit cartesien *)
let cartesian l l' =
  List.concat (List.map (fun e -> List.map (fun e' -> (e,e')) l') l)

let enarray x = (List.map Array.of_list x)


(* Implementation de kmp *)
let kmppreprocess w =
 let n = Array.length w in
 let pos = ref 1 in
 let cnd = ref 0 in
```

```
 let t = Array.make (n+1) (-1) in
 while (!pos) < n do
  if w.(!pos) = w.(!cnd) then
   (t.(!pos) <- t.(!cnd);)
  else
  (
   t.(!pos) <- !cnd;
   while (!cnd >= 0 && w.(!pos) <> w.(!cnd) ) do
    cnd:=t.(!cnd);
   done;
  );
  pos:=(!pos)+1;
  cnd:=(!cnd)+1;
 done;
 t.(!pos) <- (!cnd);
 t;;

let kmp s w t =
 let ns = Array.length s in
 let nw = Array.length w in
 let j = ref 0 in
 let k = ref 0 in
 let res = ref [] in
 while (!j) < ns do
  if w.(!k) = s.(!j) then
  (
   j:=(!j)+1;
   k:=(!k)+1;
   if (!k)=nw then
   (
    res:=((!j)-(!k))::(!res);
    k:=t.(!k);
   );
  )
  else
  (
   k:=t.(!k);
   if (!k) < 0 then
   (
    j:=(!j)+1;
    k:=(!k)+1;
   );
  );
 done;
 !res;;

(* remplace x i l b remplace dans x le sous mot de longueur l
    qui commence a l'indice i par le mot b *)
let remplace x i l b =
```

```
  let n = Array.length x in
  let m = Array.length b in
  if i >= n || n < i+1 then failwith "OOH" else
  let res = Array.make (n-1+m) x.(0) in
  for j = 0 to (n-1+m-1) do
    if (j < i) then
    (
      res.(j) <- x.(j)
    ) else
    if (j >= i+m) then
    (
      res.(j) <- x.(j-m+1)
    )
    else
    (
      res.(j) <- b.(j-i)
    );
  done;
  res

let ajoute e l = if List.mem e l then l else e::l

let rec ajouteplein l1 l2 =
 match l1 with
  |[] -> l2
  |t::q -> (ajouteplein q (ajoute t l2))

(* Effectue un pretraitement (kmp) des membres de gauche des
    regles de derivation *)
let preprocessgf grf =
  Array.map
  (fun (a,b) ->
    (a,b,kmppreprocess a)
  )
  grf.reglesf

let nboccur m l = List.length ((List.filter (fun x -> x = l) ) (
    Array.to_list m))

(* Parcours en largeur *)
let rec bfs g dejaVus aVoir =
  match aVoir with
    |[] -> dejaVus
    |tete::queue -> if List.mem tete dejaVus
      then bfs g dejaVus queue
      else bfs g (tete::dejaVus) (queue@(g tete));;

(* Donne le nombre d'occurences des lettres de la grammaire dans
    le mot *)
```

```
let analysemot mot gram =
 let n = gram.nbnonterminaux in
 let m = Array.length gram.terminaux in
 let res = Array.make (n+m) 0 in
 Array.mapi (fun i l -> if i < n then nboccur mot (Nt i) else nboccur mot (gram.terminaux.(i-n)) ) res

let ordre a b = (Array.length a <= Array.length b ) && (
    let res = ref true in
    for i=0 to ((Array.length a ) - 1) do
     if b.(i) < a.(i) then res:=false;
    done;
    !res
  )

let moins a b =
 let res = Array.make (Array.length a) 0 in
 for i=0 to ((Array.length a) - 1) do
   res.(i) <- a.(i)-b.(i);
 done;
 res

(*
Un parcours en largeur qui
  -elimine des chemins passant par un sommet invalide
  -s'arrete des qu'il parcourt un sommet valide
PS :
  -dans avoir il y a des chemins
  -retourne un chemin.
*)
let rec parcoursmagique delta elimine termine dejavu avoir =
  let navoir = ref [] in
  let ndejavu = ref dejavu in
  if avoir = [] then None else (
  let res =
    List.find_opt
    (function
      |[] -> failwith "mauvais chemin"
      |s::q ->
        ndejavu := ajoute s (!ndejavu);
        if (termine s) then true else
        if (elimine s || List.mem s dejavu) then false else
        (
        navoir :=
          ajouteplein
          (
            List.map
            (fun v -> v::s::q)
            (delta s)
```

```
            )
            (!navoir);
          false)
      )
      avoir
      in
    match res with
      |None -> parcoursmagique delta elimine termine (!ndejavu) (!
         navoir)
      |Some x -> Some x
    )

(* Retourne les mots vers lesquels x peut deriver une fois *)
let succ ppregles x =
  let res =
  List.flatten
  (
    Array.to_list (
      Array.map
      (fun (a,b,pp) ->
        List.map
        (fun i ->
          remplace x i (Array.length a) b
        )
        (kmp x a pp)
      )
      ppregles
    )
  )
  in
  ajouteplein res []

(* Retourne les mots vers lesquels x peut deriver une fois et
   comment il derive *)
let succbis ppregles (x,_,_) =
  let res =
  List.flatten
  (
    Array.to_list (
      Array.mapi
      (fun numregle (a,b,pp) ->
        List.map
        (fun i ->
          (remplace x i (Array.length a) b,numregle,i)
        )
        (kmp x a pp)
      )
      ppregles
    )
```

```
  )
  in
  ajouteplein res []

(* Cherche une derivation de x vers m *)
let recherchesuitemots x m ppregles =
  parcoursmagique
  (succ ppregles)
  (fun x -> false)
  (fun x -> x = m)
  []
  [[x]]

(* La meme fonction mais fourni une preuve dans le type
    preuveformelle *)
let recherchepreuvenaif gram ppregles x m =
  parcoursmagique
  (succbis ppregles)
  (fun (y,a,b) -> false)
  (fun (y,a,b) -> y = m)
  []
  [[x,0,0]]

(* Donne cat(m) pour des etats Q donnes par fctcat et pour la
    grammaire gram*)
let categorisemot gram fctcat m = Array.map fctcat (analysemot m
    gram)

(* Fonction categorisante binaire *)
let fct_cat_bin n = if n > 0 then 1 else 0

(* Fonction cat pour notre categorisation *)
let cat gram = categorisemot gram fct_cat_bin

(* Verifie si il existe un mot de categorie q derivable via la
    derivation d dans la grammaire gram *)
let estpossible q d gram =
  let (a,b) = d in
  ordre (cat gram a) q

(* Construit le graphe A *)
let grapheA gram =
  fun q ->
    let leslettres = lettres gram in
    List.concat_map
    (fun (i,(a,b)) ->
      let leres =
      List.filter
        (fun qp ->
```

```
            (ordre (cat gram a) q) &&
            (ordre (moins q (cat gram a)) qp) &&
            (
              List.for_all
              (fun l ->
                (
                  implies ((cat gram b).(l) = 1) (qp.(l) = 1)
                )&&(
                  implies (q.(l) = 0 && (cat gram b).(l) = 0) (qp.(l) = 0)
                )
              )
              (Array.to_list (Array.mapi (fun i x -> i) (Array.make (Array.length leslettres) 0)))
            )
          )
          (enarray (nuplets (Array.length leslettres)))
        (*in (i,leres)*)
        in leres
      )
      (Array.to_list (Array.mapi (fun i x -> (i,x) ) gram.reglesf))

(* Finalement, la fonction interdit voulue *)
let interditfort gram m x =
  not (List.mem (cat gram m) (bfs (grapheA gram) [] [cat gram x]) )

(* Recherche une preuve que l'on peut deriver x en m en utilisant la simplification de complexite *)
let recherchepreuve gram ppregles x m =
  parcoursmagique
  (succbis ppregles)
  (fun (y,a,b) -> let res = interditfort gram m y in if res then print_int 1 ; res)
  (fun (y,a,b) -> y = m)
  []
  [[x,0,0]]



(*##### IMPLEMENTATION DU CODE #####*)

(* Un exemple de grammaire formelle *)

let exfg = {
  terminaux = [| T 'a' ; T 'b' ; T 'c' ; T 'k' |];
  nbnonterminaux = 1;
  axiome = Nt 0;
```

```
    reglesf = [|
      [|Nt 0|],[|T 'a' ; T 'b' ; T 'c'|] ;
      [|T 'a' ; T 'b' ; T 'c'|],[|T 'a' ; T 'b'|] ;
      [|T 'b'|],[|T 'k'|] ;
      [|T 'c'|],[|T 'a' ; T 'k'|] ;
      [|T 'k' ; T 'a' ; T 'k'|],[|T 'a' ; T 'a'|] ;
      [|T 'a'|],[|T 'a' ; T 'a' ; T 'a'|]
    |]
}

let exreglespp = preprocessgf exfg

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'|]

let x = [|T 'a' ; T 'a' ; T 'a' ; T 'b' ; T 'a' ; T 'k' ; T 'a'
    ; T 'b'|]

let reszero = interditfort exfg m x

let resun = recherchepreuvenaif exfg exreglespp [|Nt 0|] [|T 'a'
    ; T 'a' ; T 'a' ; T 'k' ; T 'a' ; T 'a' ; T 'a' ; T 'k'|]

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'execution logicielle

Voici la fin de ce que le programme executé 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, 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'|],
         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'execution de `interditfort`).