TP 18

ALGORITHME DE COCKE-YOUNGER-KASAMI

On s'intéresse dans ce sujet à l'algorithme de Cocke-Younger-Kasami (dit CYK), qui est un algorithme d'analyse syntaxique ascendante.

1 Principe de l'algorithme

L'algorithme prend en entrée une grammaire $G = (\Sigma, V, P, S)$ **en forme normale de Chomsky** et un mot $w = w_0 \dots w_{n-1} \in \Sigma^*$. En sortie, on commencera par renvoyer un booléen indiquant si $w \in \mathcal{L}(G)$, puis l'on cherchera à calculer un arbre de dérivation (dans le cas où $w \in \mathcal{L}(G)$).

On note $V = \{X_0, \dots, X_{k-1}\}$ les variables de la grammaire, et l'on définit

$$t[l, d, i] = \begin{cases} \text{Vrai} & \text{si } X_i \Rightarrow^* w_d \dots w_{d+l-1} \\ \text{Faux} & \text{sinon} \end{cases}$$

► **Question 1** Pour quelles valeurs de l, d et i le booléen $t[l, d, i]$ est-il défini (et intéressant) ?

► **Question 2** À quelle condition (sur les $t[l, d, i]$) a-t-on $w \in \mathcal{L}(G)$?

► **Question 3** Que peut-on dire de $t[0, d, i]$?

► **Question 4** Expliquer comment initialiser simplement les $t[1, d, i]$,.

► **Question 5** Pour $l \geqslant 2$, exprimer $t[l, d, i]$ en fonction des $t[l', \cdot, \cdot]$ pour $l' < l$.

► **Question 6** En déduire le pseudo-code d'un algorithme permettant de déterminer si $w \in \mathcal{L}(G)$.

► **Question 7** Déterminer la complexité en temps et en espace de cet algorithme, en fonction de la longueur n de w et de la taille $|G|$ de la grammaire.

2 Implémentation

On définit les types suivants pour représenter une grammaire en forme normale de Chomsky :

```
type regle_unitaire = int * char
type regle_binaire = int * int * int

type cnf = {
  initial : int;
  nb_variables : int;
  unitaires : regle_unitaire list;
  binaires : regle_binaire list;
  mot_vide : bool
}
```

- On supposera que les variables sont numérotées consécutivement de 0 à `nb_variables` -1. Le champ `initial` indique le numéro du symbole initial.

- On considère que Σ est inclus dans l'ensemble des caractères ASCII : ainsi, la règle $X_2 \to d$ sera codée par le couple `(2, 'd')`.
- Une règle $X_i \to X_j X_k$ est codée par le triplet `(i, j, k)`.
- Le booléen `mot_vide` indique si $\varepsilon \in \mathcal{L}(G)$.

▸ **Question 8** Définir une variable de type `cnf` codant la grammaire G_0 suivante :

$$\begin{aligned} S &\to b \mid AB \mid BA \mid CA \\ A &\to a \mid AD \\ B &\to b \\ C &\to AB \\ D &\to a \end{aligned}$$

▸ **Question 9** Écrire une fonction `cyk_reconnait` déterminant si un mot u (donné sous forme d'une chaîne de caractères) appartient au langage d'une grammaire G (donné sous forme d'une variable de type `cnf`.)

```
val cyk_reconnait : cnf -> string -> bool
```

▸ **Question 10** Donner une description concise du langage généré par G_0, et tester votre fonction `cyk_reconnait` sur les exemples suivants : ε, a, b, $abaa$, $abaaba$, $baaaa$, $abba$.

On définit le type suivant pour représenter un arbre de dérivation (d'une grammaire en forme normale de Chomsky) :

```
type arbre =
  | Empty
  | Unaire of int * char
  | Binaire of int * arbre * arbre
```

Remarque

Le cas `Empty` correspond à une éventuelle dérivation $S \Rightarrow \varepsilon$. Il ne peut se produire qu'à la racine de l'arbre (puisque S n'apparaît à droite d'aucune règle).

▸ **Question 11** Écrire une fonction `cyk_analyse` qui prend les mêmes arguments que `cyk_reconnait` mais renvoie (une option sur) un arbre de dérivation possible pour le mot. On renverra `None` si le mot n'est pas dans le langage.

Remarque

En cas d'ambiguïté, on renverra *un* arbre de dérivation quelconque du mot.

```
val cyk_analyse : cnf -> string -> arbre option
```

▸ **Question 12** Écrire une fonction `cyk_compte` qui prend les mêmes arguments que `cyk_reconnait` et renvoie le nombre d'arbres de dérivation du mot fourni en argument.

```
val cyk_compte : cnf -> string -> int
```

3 Mise en forme normale de Chomsky

On propose les types suivants pour représenter une grammaire sans contexte quelconque :

```
type symbole = T of char | V of int
type regle = int * symbole list

type grammaire = {
  nb_variables : int;
  regles : regle list;
  initial : int
}
```

► **Question 13** Définir en OCaml la grammaire suivante :

$$\begin{aligned} S &\to aSb \mid aXb \\ X &\to YX \mid \varepsilon \\ Y &\to a \mid b \end{aligned}$$

► **Question 14** Écrire une fonction `start` correspondant à l'étape **START** de la mise en forme normale de Chomsky.

```
val start : grammaire -> grammaire
```

► **Question 15** Écrire de même des fonctions `term`, `bin`, `del` et `unit_rules` (de difficulté variable. . .).

► **Question 16** Écrire une fonction de mise en forme normale de Chomsky.

```
normalise : grammaire -> cnf
```

► **Question 17** Déterminer la complexité en temps de la fonction `normalise`.

Solutions

▶ **Question 1** l peut varier de 0 à n, d de 0 à $n-l$ et i de 0 à $k-1$.

▶ **Question 2** $w \in \mathcal{L}(G)$ si et seulement si $t[n, 0, i_0]$ est vrai, où $X_{i_0} = S$.

▶ **Question 3** La grammaire étant en forme normale de Chomsky, on a $t[0, d, i]$ faux si $i \neq i_0$, et $t[0, d, i_0]$ vrai si et seulement si la grammaire possède une règle $S \to \varepsilon$.

▶ **Question 4** On a $t[1, d, i]$ vrai si et seulement si $X_i \to w_d \in P$.

▶ **Question 5** La grammaire étant en forme normale de Chomsky, la seule manière d'obtenir $X_i \Rightarrow^* w_d \dots w_{d+l-1}$ avec $l \geqslant 2$ est d'avoir $X_i \Rightarrow X_j X_k$, $X_j \Rightarrow^* w_d \dots w_{d+l'-1}$ et $X_k \Rightarrow^* w_{d+l'} \dots w_{d+l-1}$ pour un certain $l' \in [1 \dots l-1]$ (et $X_j, X_k \in V$). On en déduit :

$$t[l, d, i] = \bigvee_{X_i \to X_j X_k \in P} \bigvee_{l'=1}^{l-1} t[l', d, j] \wedge t[l-l', d+l', k]$$

▶ **Question 6** On obtient le pseudo-code suivant (où $X_{i_0} = S$) :

Algorithme 1 CYK

```
fonction CYK(G, w)
    si w = ε alors
        renvoyer S → ε ∈ P
    t ← un tableau (n + 1) × n × k initialisé à false
    pour X_i → a ∈ P faire
        pour d = 0 à |w| − 1 faire
            si w_d = a alors
                t[1, d, i] ← true
    pour l = 2 à n faire
        pour d = 0 à n − l faire
            pour l' = 1 à l − 1 faire
                pour X_i → X_j X_k ∈ P faire
                    t[l, d, i] ← t[l, d, i] ∨ (t[l', d, j] ∧ t[l − l', d + l', k])
    renvoyer t[n, 0, i_0]
```

▶ **Question 7** En espace, on initialise un tableau de taille $(n+1) \times n \times k$, on a donc une complexité en espace en $O(kn^2) = O(|G| \cdot n^2)$. En temps, on a :

- la création du tableau en $O(kn^2)$;
- l'initialisation des $t[1, d, i]$ en $O(n \cdot |G|)$;
- la boucle principale en $O\left(\sum_{l=2}^{n} \sum_{d=0}^{n-l} \sum_{l'=1}^{l-1} |G|\right) = O(n^3 \cdot |G|)$.

Au total, on obtient du $O(n^3 \cdot |G|)$.

▶ **Question 8**

```
let g0 = {
  initial = 0;
  nb_variables = 5;
  unitaires = [(0, 'b'); (1, 'a'); (2, 'b'); (4, 'a')];
  binaires = [(0, 1, 2); (0, 2, 1); (0, 3, 1); (1, 1, 4); (3, 1, 2)];
  mot_vide = false;
}
```

▶ **Question 9** On traduit directement le pseudo-code donné plus haut, en traitant séparément le cas du mot vide.

```
let cyk_reconnait g entree =
  let n = String.length entree in
  let m = g.nb_variables in

  let tab = Array.make (n + 1) [||] in
  for l = 0 to n do
    tab.(l) <- Array.make_matrix n m false
  done;

  let traite_regle_unitaire (x, c) =
    for i = 0 to n - 1 do
      if entree.[i] = c then tab.(1).(i).(x) <- true
    done in
  List.iter traite_regle_unitaire g.unitaires;

  for l = 2 to n do
    for debut = 0 to n - l do
      for l_gauche = 1 to l - 1 do
        let traite_regle_binaire (a, b, c) =
          tab.(l).(debut).(a) <-
            tab.(l).(debut).(a)
            || (tab.(l_gauche).(debut).(b)
                && tab.(l - l_gauche).(debut + l_gauche).(c)) in
        List.iter traite_regle_binaire g.binaires
      done;
    done;
  done;

  if n = 0 then g.mot_vide
  else tab.(n).(0).(g.initial)
```

▶ **Question 10** On observe successivement que :

- A engendre a^+ ;
- C engendre a^+b ;
- S engendre $b \mid a^+b \mid ba^+ \mid a^+ba^+$

En simplifiant, on a donc $\mathcal{L}(G_0) = a^*ba^*$. On vérifie ensuite que l'on obtient bien ce qui est attendu.

► **Question 11** On remplace notre tableau (tri-dimensionnel) de booléens par un tableau (tri-dimensionnel) d'options sur des arbres :

- si $X_i \Rightarrow^* w_d \dots w_{d+l-1}$, alors `tab.(l).(d).(i) = Some t`, où t est un arbre de dérivation convenable ;
- sinon, `tab.(l).(d).(i) = None`.

La structure du code est essentiellement inchangée.

```
let cyk_analyse (g : cnf) entree =
  let n = String.length entree in
  let m = g.nb_variables in

  let tab = Array.make (n + 1) [||] in
  for l = 0 to n do
    tab.(l) <- Array.make_matrix n m None
  done;

  let traite_regle_unitaire (x, c) =
    for i = 0 to n - 1 do
      if entree.[i] = c then tab.(1).(i).(x) <- Some (Unaire (i, c))
    done in
  List.iter traite_regle_unitaire g.unitaires;

  for l = 2 to n do
    for deb = 0 to n - l do
      for l_g = 1 to l - 1 do
        let traite_regle_binaire (a, b, c) =
          match tab.(l_g).(deb).(b), tab.(l - l_g).(deb + l_g).(c) with
          | Some gauche, Some droit ->
            tab.(l).(deb).(a) <- Some (Binaire (a, gauche, droit))
          | _ -> () in
        List.iter traite_regle_binaire g.binaires
      done;
    done;
  done;

  if n = 0 && g.mot_vide then Some Empty
  else if n = 0 then None
  else tab.(n).(0).(g.initial)
```

▶ **Question 12** Il suffit encore d'une petite modification du code. En notant $\varphi(l, d, i)$ le nombre d'arbres de dérivation pour $X_i \Rightarrow^* w_d \ldots w_{d+l-1}$, on a :

$$\varphi(l, d, i) = \sum_{X_i \to X_j X_k \in P} \sum_{l'=1}^{l-1} \varphi(l', d, j)\varphi(l - l', d + l', k)$$

```
let cyk_compte (g : cnf) entree =
  let n = String.length entree in
  let m = g.nb_variables in

  let tab = Array.make (n + 1) [||] in
  for l = 0 to n do
    tab.(l) <- Array.make_matrix n m 0
  done;

  let traite_regle_unitaire (x, c) =
    for i = 0 to n - 1 do
      if entree.[i] = c then tab.(1).(i).(x) <- 1
    done in
  List.iter traite_regle_unitaire g.unitaires;

  for l = 2 to n do
    for debut = 0 to n - l do
      for l_gauche = 1 to l - 1 do
        let traite_regle_binaire (a, b, c) =
          tab.(l).(debut).(a) <-
            tab.(l).(debut).(a)
            + (tab.(l_gauche).(debut).(b)
               * tab.(l - l_gauche).(debut + l_gauche).(c)) in
        List.iter traite_regle_binaire g.binaires
      done;
    done;
  done;

  if n = 0 && g.mot_vide then 1
  else if n = 0 then 0
  else tab.(n).(0).(g.initial)
```

▶ **Question 13**

```
let g1 = {
  initial = 0;
  nb_variables = 3;
  regles = [(0, [T 'a'; V 0; T 'b']); (0, [T 'a'; V 1; T 'b']);
           (1, [V 2; V 1]); (1, []); (2, [T 'a']); (2, [T 'b'])]
}
```

Cette grammaire (ambiguë) génère $a(a|b)^*b$.

▶ **Question 14** On pourrait ne créer un nouveau symbole que si nécessaire, mais pour simplifier on le fait systématiquement.

```
let start g =
  {nb_variables = g.nb_variables + 1;
   regles = (n, [V g.initial]) :: g.regles;
   initial = n
  }
```

▶ **Question 15** C'est un peu long, et plus ou moins délicat suivant les étapes...

TERM On crée un tableau `tab` qui indique pour chaque caractère s'il faut créer une variable lui correspondant, et l'on maintient à jour une référence `next` qui indique le prochain numéro de variable « libre ». Ensuite :

- pour chaque règle $X \to \alpha$ avec $|\alpha| > 1$, on crée les variables correspondant aux terminaux présents si elles n'existent pas encore, et l'on remplace cet terminaux par la variable correspondante ;
- ensuite, on ajoute les règles $N_a \to a$ pour toutes les variables nouvellement créées ;
- la valeur finale de `next` donne le nouveau nombre de variables.

```
let term g =
  let tab = Array.make 256 (-1) in
  let next = ref g.nb_variables in

  let rec traite mot =
    match mot with
    | [] -> []
    | V i :: xs -> V i :: traite xs
    | T c :: xs ->
      let i = int_of_char c in
      if tab.(i) = -1 then (tab.(i) <- !next; incr next);
      V tab.(i) :: traite xs in

  let transforme_regle (v, mot) =
    if List.length mot <= 1 then (v, mot)
    else (v, traite mot) in

  let regles' = ref (List.map transforme_regle g.regles) in
  for i = 0 to 255 do
    if tab.(i) <> -1 then (
      regles' := (tab.(i), [T (char_of_int i)]) :: !regles'
    )
  done;

  {nb_variables = !next;
  regles = !regles';
  initial = g.initial}
```

La complexité est linéaire en la taille de la grammaire (somme des tailles des règles), tout comme l'augmentation de la taille.

BIN La fonction `binarise` prend en entrée une règle $X \to \alpha$ et renvoie une liste de règles qui vont la remplacer :

- si $|\alpha| \leqslant 2$, on renvoie simplement la liste $[X \to \alpha]$;
- sinon, on a nécessairement $\alpha = X_1 \dots X_p$ (on suppose qu'on a déjà effectué **TERM**), et l'on renvoie $[X \to X_1Y_1, Y_1 \to X_2Y_2, \dots, Y_{p-2} \to X_{p-1}X_p]$ (où les Y_i sont fraîches).

On remplace ensuite chaque règle par la liste des règles qui lui correspondent (en concaténant).

```ocaml
let bin g =
  let next = ref g.nb_variables in

  let rec binarise (v, droite) =
    match droite with
    | [] | [_] | [_; _] -> [(v, droite)]
    | a :: b :: xs ->
      let nv_v = !next in
      let nv_regle = (v, [a; V nv_v]) in
      incr next;
      nv_regle :: binarise (nv_v, b :: xs) in

  let rec traite_regles = function
    | [] -> []
    | r :: rs -> binarise r @ traite_regles rs in

  let regles' = traite_regles g.regles in
  {
    nb_variables = !next;
    regles = regles';
    initial = g.initial
  }
```

À nouveau, la complexité et l'augmentation de la taille sont linéaires (une règle $X \to X_1 \dots X_p$ de taille $p+1$ est remplacée par $p-1$ règles de taille 3, donc au total de l'ordre de $3p$).

DEL On commence par calculer les variables annulables. Pour ce faire, on définit E_i comme l'ensemble des $X \in V$ telles que $X \Rightarrow^j \varepsilon$ avec $j \leqslant i$. On calcule successivement les E_i en utilisant la remarque du cours, en s'arrêtant quand $E_{i+1} = E_i$ (la suite est alors stationnaire). La suite des $|E_i|$ est croissante par construction et majorée par $|V|$, il y a donc au plus $|V|$ étapes, donc chacune demande de parcourir toutes les règles et se fait donc en temps $O(|V|)$: le calcul des variables annulables se fait en temps $O(|V|^3) = O(|G|^2)$.

```ocaml
let calcul_annulables g =
  let annulables = Array.make g.nb_variables false in
  let changement = ref true in
  let traite_regle (v, droite) =
    let rec aux = function
      | [] -> changement := true; annulables.(v) <- true
      | V x :: reste when annulables.(x) -> aux reste
      | _ -> () in
    if not annulables.(v) then aux droite in
  while !changement do
    changement := false;
    List.iter traite_regle g.regles
  done;
  annulables
```

Une fois connues les variables annulables, on tire parti du fait que la grammaire est déjà binarisée pour ajouter simplement les règles permettant de compenser la suppression de toutes les ε-productions. On n'oublie pas de rajouter la production $S \to \varepsilon$ si S est annulable (cette production n'existait pas nécessairement dans la grammaire, même dans ce cas).

```
let del g =
  let annulables = calcul_annulables g in
  let efface v (x, y) =
    let u = ref [] in
    let f x y =
      match x with
      | V v' when annulables.(v') -> u := (v, [y]) :: !u
      | _ -> () in
    f x y;
    f y x;
    !u in
  let rec traite_regles regles =
    match regles with
    | [] -> []
    | (v, []) :: reste -> traite_regles reste
    | (v, [x]) :: reste -> (v, [x]) :: traite_regles reste
    | (v, [x; y]) :: reste ->
      efface v (x, y) @ (v, [x; y]) :: traite_regles reste
    | _ -> failwith "commencez par binariser !" in
  let regles' =
    let u =
      if annulables.(g.initial) then (g.initial, []) :: traite_regles g.regles
      else traite_regles g.regles in
    List.sort_uniq compare u in
  {nb_variables = g.nb_variables;
  regles = regles';
  initial = g.initial}
```

Cette étape se fait en temps linéaire en la taille de la grammaire, et on crée au plus deux nouvelles règles (de taille 2) pour chaque règle. Finalement, l'étape **DEL** :

- a une complexité en $O(|G|^2)$ (du fait du calcul des variables annulables) ;
- cause une augmentation linéaire de la taille de la grammaire.

UNIT On commence par créer le graphe unitaire de G, dont les sommets sont les variables et qui possède un arc (X, Y) si et seulement $X \to Y \in P$.

```
let graphe_unitaire g =
  let n = g.nb_variables in
  let graphe = Array.make n [] in
  let traite_regle (v, droite) =
    match droite with
    | [V v'] ->
      graphe.(v) <- v' :: graphe.(v)
    | _ -> () in
  List.iter traite_regle g.regles;
  graphe
```

On calcule ensuite la clôture transitive de ce graphe : il s'agit de la matrice $|V| \times |V|$ telle que `cloture.(i).(j)` soit vrai si et seulement si X_j est accessible depuis X_i dans le graphe unitaire. On effectue pour cela un parcours depuis chaque sommet du graphe.

```
let cloture_transitive graphe =
  let n = Array.length graphe in
  let cloture = Array.make_matrix n n false in
  let calcule_ligne v =
    let vus = Array.make n false in
    let rec explore i =
      if not vus.(i) then (
        vus.(i) <- true;
        cloture.(v).(i) <- true;
        List.iter explore graphe.(v)
      ) in
    explore v in
  for v = 0 to n - 1 do
    calcule_ligne v
  done;
  cloture
```

Ensuite, en notant $\mathcal{U}(X)$ la clôture unitaire de X (les variables Y telles que $X \Rightarrow^* Y$) :

- on crée un tableau permettant d'accéder rapidement à toutes les règles de la forme $X_i \to \alpha$ (pour un i donné) ;
- on élimine toutes les règles unitaires $X \to Y$;
- pour chaque variable X et chaque $Y \in \mathcal{U}(X)$, on ajoute toutes les règles $X \to \alpha$ où α n'est pas une variable et $Y \to \alpha \in P$.

```
let remove_unit g =
  let cloture = cloture_transitive (graphe_unitaire g) in
  let n = g.nb_variables in
  let tab_regles = Array.make n [] in
  let ajoute (v, droite) =
    match droite with
    | [V _] -> ()
    | _ -> tab_regles.(v) <-
        droite :: tab_regles.(v) in
  List.iter ajoute g.regles;
  let regles = ref [] in
  for v = 0 to n - 1 do
    for v' = 0 to n - 1 do
      if cloture.(v).(v') then (
        let f droite = regles := (v, droite) :: !regles in
        List.iter f tab_regles.(v')
      )
    done
  done;
  {initial = g.initial;
  regles = List.sort_uniq compare !regles;
  nb_variables = g.nb_variables}
```

Le calcul de la clôture unitaire demande $|V|$ parcours du graphe unitaire, et chacun de ces parcours se fait en temps $|V|$. Ensuite, si la grammaire comporte p règles, on peut ajouter jusqu'à p règles pour chaque variable (ou presque), si le graphe unitaire est fortement connexe. Comme p et $|V|$ sont en $O(|G|)$, on a une complexité en temps en $O(|G|^2)$ et la nouvelle taille de la grammaire est en $O(|G|^2)$.

▶ **Question 16** On applique successivement toutes les transformations (dans l'ordre, on a écrit chaque fonction en supposant que les transformations précédentes avaient été effectuées). La conversion finale ne pose aucun problème.

```
let normalise g =
  let g' = g |> start |> term |> bin |> del |> remove_unit in
  let unitaires = ref [] in
  let binaires = ref [] in
  let mot_vide = ref false in
  let traite (v, droite) =
    match droite with
    | [] -> assert (v = g'.initial); mot_vide := true
    | [T c] -> unitaires := (v, c) :: !unitaires
    | [V x; V y] -> binaires := (v, x, y) :: !binaires
    | _ -> assert false in
  List.iter traite g'.regles;
  {initial = g'.initial;
  nb_variables = g'.nb_variables;
  unitaires = !unitaires;
  binaires = !binaires;
  mot_vide = !mot_vide}
```

► **Question 17** Avec toutes les remarques faites précédemment, on a une complexité en temps en $O(|G|^2)$ et $|G'| = O(|G|^2)$ (avec G' la grammaire normalisée).