Concours Centrale - Supelec 2003

Corrigé de l'épreuve d'informatique

PARTIE I - Mots bien parenthésés

I.A.1) Comme $\varepsilon \in L_0$, $ab \in L_1$. On en déduit que $ab \in L_2$ (car $L_1 \subset L_2$) et que $a(ab)b \in L_2$. Ainsi, $ababbaa = ab.aabb$ est élément de L_2^2, donc de L_3 : $abaabb \in \mathcal{L}_P$.

I.A.2) Montrons par récurrence sur n que pour tout mot w de L_n, $|w|_a = |w|_b$ et que $w_1 = a$ et $w_{|w|} = b$ quand w est non vide.

- Comme L_0 est réduit au mot vide, l'hypothèse de récurrence est vérifiée au rang 0.
- Fixons $n \geq 0$ et supposons la propriété vérifiée au rang n. Soit alors $w \in L_{n+1}$, que nous pouvons supposer non vide. Nous sommes donc dans l'un des trois cas suivants :
  - $w \in L_n$ et l'hypothèse de récurrence affirme que w contient autant de a que de b, qu'il commence par un a et qu'il se termine par un b.
  - $w = uv$ avec $u, v \in L_n \setminus \{\varepsilon\}$ (si l'un des mots est vide, on se retrouve dans le cas précédent). Alors $|u|_a = |u|_b$, $|v|_a = |v|_b$, u et v commencent par a et se terminent par b : on en déduit que $|w|_a = |u|_a + |v|_a = |u|_b + |v|_b = |w|_b$ et que w commence par a et se termine par b.
  - $w = aub$ avec $u \in L_n$. La première (resp. dernière) lettre de w est donc un a (resp. un b) et comme $|u|_a = |u|_b$, on a encore $|w|_a = 1 + |u|_a = 1 + |u|_b = |w|_b$.

  La propriété est donc vérifiée au rang $n+1$, ce qui achève la preuve par récurrence.

I.A.3) Prouvons une nouvelle fois ce résultat par récurrence.

- si $w \in L_0$, $w = \varepsilon$ et la propriété est vérifiée, puisqu'il n'existe pas de i tel que $w_i = a$.
- soit $n \geq 0$ et supposons que la propriété soit vérifiée pour tout mot $w \in L_n$. Fixons alors un mot $w \in L_{n+1}$ et un i tel que $w_i = a$. Nous sommes une nouvelle fois dans l'un des trois cas suivants :
  - $w \in L_n$ et il suffit d'apliquer la propriété au rang n.
  - $w = uv$ avec $u, v \in L_n \setminus \{\varepsilon\}$. Si $i \leq |u|$, on sait qu'il existe j tel que $i < j \leq |u|$ et $u[i \dots j] \in \mathcal{L}_P$. On en déduit donc que $w[i \dots j] \in \mathcal{L}_P$. Si $i > |u|$, alors $v_{i-|u|} = a$ et il existe $j' > i - |u|$ tel que $v[i - |u| \dots j'] \in \mathcal{L}_P$. En posant $j = j' + |u|$, nous avons $i < j \leq |w|$ et $w[i \dots j] = v[i - |u| \dots j']$ est élément de $\mathcal{L}_P$.
  - $w = aub$ avec $u \in L_n$. Si $i = 1$, $j = |u|$ donne $w[i \dots j] = w \in \mathcal{L}_P$. Sinon, $u_{i-1} = a$ et il existe $j' > i - 1$ tel que $u[i - 1 \dots j'] \in \mathcal{L}_P$. En posant $j = j' + 1 > i$, nous avons donc $w[i \dots j] = u[i - 1, j'] \in \mathcal{L}_P$.

Le j n'est en général pas unique : avec $w = abaabb$ et $i = 1$, on peut choisir $j = 2$ ou $j = 6$.

I.B.1) Supposons qu'il existe un automate fini $\mathcal{A}$ reconnaissant $\mathcal{L}$. Notons q_0 l'état initial de $\mathcal{A}$ et $q.u$ l'état atteint quand on lit le mot u en partant de l'état q (quand cet état est défini). Nous allons montrer que pour tout entier n, l'état $q_n = q_0.a^n$ est défini et que les états q_n sont deux à deux distincts :

- Soit $n \in \mathbb{N}$. Comme $a^n b^n \in \mathcal{L}$, $q_0.(a^n b^n)$ est défini, et donc $q_0.a^n$ l'est à plus forte raison.
- Soit $n, m \in \mathbb{N}$ tels que $q_n = q_m$. Comme le mot $a^n b^n$ est reconnu par $\mathcal{A}$, $q_n.b^n = q_0.(a^n b^n)$ est un état final. On en déduit donc que $q_0.(a^m b^n) = q_m.b^n = q_n.b^n$ est un état final, i.e. que $a^m b^n \in \mathcal{L}$, ce qui impose $n = m$.

Ceci contredit la finitude de l'automate : L n'est pas reconnaissable par automate fini.

I.B.2) La stabilité de Rec(A) par intersection est un résultat de cours. La preuve classique est la suivante : soient L_1 et L_2 deux langages reconnaissables par automate fini sur l'alphabet A, et soit $\mathcal{A}_1 = (Q_1, q_1, F_1, \delta_1, A)$ et $\mathcal{A}_2 = (Q_2, q_2, F_2, \delta_2, A)$ des automates déterministes reconnaissant respectivement L_1 et L_2. Q_i désigne l'ensemble des états de $\mathcal{A}_i$, q_i est l'état initial, F_i est l'ensembles des états finals et δ_i est la fonction (partielle) de transition. Soit alors $\mathcal{A} = (Q, q, F, \delta, A)$ l'automate déterministe défini par :

- $Q = Q_1 \times Q_2$ et $q = (q_1, q_2)$;
- $F = F_1 \times F_2$;
- pour tout $e = (e_1, e_2) \in Q$ et pour tout $a \in A$, $\delta(e, a)$ est défini si et seulement si $\delta_1(e_1, a)$ et $\delta_2(e_2, a)$ sont définis, avec dans ce cas $\delta(e, a) = (\delta_1(e_1, a), \delta_2(e_2, a))$.

Un mot u sur A est reconnu par $\mathcal{A}$ si et seulement si $\delta(q, u)$ est défini et est élément de F, i.e. si et seulement si pour $i \in \{1, 2\}$, $\delta_i(q_i, u)$ est défini et est élément de F_i : le langage reconnu par l'automate fini $\mathcal{A}$ est donc $L_1 \cap L_2$.

I.B.3) Soit $\mathcal{L}'$ le langage représenté par l'expression rationnelle $a^* b^*$. Comme $\mathcal{L} = \mathcal{L}' \cap \mathcal{L}_P$ et $\mathcal{L}' \in \mathrm{Rec}\,(\{a, b\})$, le langage $\mathcal{L}_P$ n'est pas reconnaissable par automate fini, par contraposée du résultat précédent.

I.C.1) Notons $(\mathcal{P})$ la condition : $|w|_a = |w|_b$ et $|u|_a \geq |u|_b$ pour tout préfixe u de w.
Montrons tout d'abord que $(\mathcal{P})$ est nécessaire pour que w appartienne à $\mathcal{L}_P$. Comme on a déjà vu que $|w|_a = |w|_b$ pour tout $w \in \mathcal{L}_P$, il reste à montrer (par induction) que $|u|_a \geq |u|_b$ pour tout préfixe u de $w \in \mathcal{L}_P$.

- Si $w \in L_0$, $w = \varepsilon$ et le seul préfixe de w est ε, qui vérifie bien $|\varepsilon|_a \geq |\varepsilon|_b$.
- Soit $n \in \mathbb{N}$ et supposons la propriété vraie pour tout mot $w \in L_n$. Fixons $w \in L_{n+1}$, que nous pouvons supposer ne pas appartenir à L_n, et soit u un préfixe de w. Nous avons alors trois cas possibles :
  - $w = w_1 w_2$ avec $w_1, w_2 \in L_n$ et u est un préfixe de w_1 : par hypothèse de récurrence, $|u|_a \geq |u|_b$.
  - $w = w_1 w_2$ avec $w_1, w_2 \in L_n$ et $u = w_1 v$ où v est un préfixe (non vide) de w_2 : par hypothèse de récurrence, on sait que $|v|_a \geq |v|_b$, et donc
  $$|u|_a = |w_1|_a + |v|_a = |w_1|_b + |v|_a \geq |w_1|_b + |v|_b = |u|_b.$$
  - $w = avb$ avec $v \in L_n$. Si u est vide ou si $u = w$, on a bien $|u|_a \geq |u|_b$. Sinon, on peut écrire $u = at$ avec t préfixe de v. Par hypothèse de récurrence, $|t|_a \geq |t|_b$ et donc
  $$|u|_a = 1 + |t|_a \geq 1 + |t|_b > |t|_b = |u|_b.$$

  Ainsi, tout mot de $\mathcal{L}_P$ vérifie la propriété $(\mathcal{P})$.

Montrons ensuite que la condition est suffisante par induction sur la longueur du mot w.

- Si w est le mot vide, il est bien élément de $\mathcal{L}_P$.

- Soit $n \in \mathbb{N}$ et supposons que tout mot w de longueur au plus n vérifiant $(\mathcal{P})$ est élément de $\mathcal{L}_P$. Soit alors un mot w de longueur $n+1$ vérifiant $(\mathcal{P})$. Notons $\varphi : \{0, 1, \ldots, n+1\} \to \mathbb{Z}$ qui à tout i associe $|u|_a - |u|_b$, où u est le préfixe de w de longueur i : $u = w[1 \ldots i]$. Par hypothèse, φ est à valeurs dans $\mathbb{N}$ et $\varphi(0) = \varphi(n+1) = 0$. Comme la différence entre $\varphi(i)$ et $\varphi(i+1)$ est toujours égale à 1 ou -1, ceci impose en particulier les conditions $\varphi(1) = 1$ et $\varphi(n) = 1$, i.e. $w_1 = a$ et $w_{n+1} = b$. Deux cas se présentent alors :
  - pour tout i compris entre 1 et n, $\varphi(i) \geq 1$. En notant $v = w[2 \ldots n]$, nous montrons facilement que v vérifie la propriété $(\mathcal{P})$. Par exemple, si u est un préfixe de v de longueur i :
    $$|u|_a - |u|_b = |au|_a - 1 - |au|_b = \varphi(i+1) - 1 \geq 0.$$
    Comme $|v| \leq n$, l'hypothèse de récurrence permet d'affirmer que v est élément de $\mathcal{L}_P$, puis que $w = aub$ l'est également.
  - il existe i tel que $1 \leq i \leq n$ et $\varphi(i) = 0$. Posons alors $w_1 = w[1 \ldots i]$ et $w_2 = w[i+1 \ldots n+1]$. On montre une nouvelle fois facilement que w_1 et w_2 vérifient $(\mathcal{P})$. Comme ces mots sont de longueurs au plus n, ils sont par hypothèse de récurrence éléments de $\mathcal{L}_P$, de même que $w = w_1 w_2$.

I.C.2) La fonction ci-dessous calcule $\varphi(i)$ pour i croissant de 1 jusqu'à $|w|$, éventuellement en s'arrêtant quand on obtient une valeur négative. Si la boucle est entièrement parcourue, le mot w est bien parenthésé si et seulement si $\varphi(|w|)$ est nul. Sinon, le mot est mal parenthésé.

```
function parenthese(w:string):boolean;
var phi,i: integer;
    bool: boolean;
begin
phi := 0;
i := 0;
bool := true;
while bool and i<length(w) do
   begin
   i := i+1;
   if w[i]='a' then
      phi := phi+1
   else
      begin
      phi := phi-1;
      bool := (phi >= 0);
      end;
   end;
if bool then
   bool := (phi=0);
parenthese := bool;
end;
```

```
let parenthese w =
   let l = string_length w in
   let rec aux a b n =
      if n=l then a=b
      else (a>=b) && (if w.[n]='a' then aux (a+1) b (n+1)
      else aux a (b+1) (n+1))
   in aux 0 0 0;;
```

La complexité est clairement linéaire (dans le pire des cas) en la taille de la chaîne w, puisque la boucle porte sur l'indice i qui décrit, dans le pire des cas, l'intervalle $[\![0, |w|]\!]$.

PARTIE II - Fermeture d'une parenthèse

L'énoncé fait une erreur grossière dans la représentation des vecteurs en Caml, le format de l'exemple donné étant celui d'une liste : il faut donc lire dans l'énoncé $[|5;2;1;4;3;0;7;6|]$ *au lieu de* $[5;2;1;4;3;0;7;6]$.

II.A.1) Ce premier algorithme fonctionne comme celui donné à la question précédente : pour chaque i tel ques $w_i = a$, on calcule la différence entre le nombre de a et le nombre de b dans les mots $w[i \dots j]$ en incrémentant j jusqu'à ce que cette différence soit nulle. Dès que j est calculé, on définit la valeur des cases d'indices i et j du tableau `resultat`. Cela donne :

```
procedure association(w:string;var resultat:tableau);
var i,j,phi:integer ;
begin
for i:=1 to length(w) do
   begin
   if w[i]='a' then
      begin
      j := i;
      phi := 1;
      while phi<>0 do
         begin
         j := j+1;
         if w[j]='a' then
            phi := phi+1
         else
            phi := phi-1;
         end;
      resultat[i] := j;
      resultat[j] := i;
      end;
   end;
end;
```

```
let association w =
   let l = string_length w in
   let t = make_vect l 0 in
      for i = 0 to l-2 do
         if w.[i] = '(' then begin
            let rec aux a b j =
               if a=b then j else
                  if w.[j+1]='(' then
                     aux (a+1) b (j+1)
                  else
                     aux a (b+1) (j+1)
            in let j = aux 1 0 i
            in t.(i) <- j; t.(j) <- i
         end
      done;
   t;;
```

Calculons la complexité dans le pire des cas de cet algorithme, en fonction de la valeur $n = |w|/2$. Pour chaque i compris entre 1 et $|w|$ tel que $|w_i| =)$, le nombre d'opération est constant. Par contre, pour i compris entre 1 et $|w|$ tel que $|w_i| = (,$ le nombre d'opérations effectuées est de l'ordre de $r(i)-i+1$ où $r(i)$ est l'indice de la parenthèse fermante associée à w_i (l'indice de la boucle `while` décrit $[\![i, r(i)]\!]$). Le nombre d'opérations effectuées est donc de l'ordre de $N(w) = \sum_{\substack{1\leq i\leq |w| \\ w_i=(}} r(i) - i + 1$. On devine que le pire des cas est obtenu pour le mot $w = (^n)^n = \underbrace{(\ldots\ldots(}_{n \text{ lettres}}\underbrace{)\ldots\ldots)}_{n \text{ lettres}}$, pour lequel $N(w)$ vaut $n(n+1)$. Nous pouvons montrer ce résultat par récurrence sur n :

- Si $n = 1$, $w = ab$ et $N(w) = n(n+1)$;
- Soit $n \in \mathbb{N}$ et supposons que pour tout mot bien parenthésé w de longueur $2k \leq 2n$, on ait $N(w) \leq k(k-1)$. Fixons alors un mot bien parenthésé w de longueur $2(n+1)$. Une nouvelle fois, deux cas se présentent :
  - ou bien $w = aub$ avec $u \in \mathcal{L}_P$, auquel cas $N(w) = N(u)+2n+2 \leq n(n+1)+2n+2 = (n+1)(n+2)$;
  - ou bien $w = w_1w_2$ avec $w_1, w_2 \in \mathcal{L}_P$ et non vides. En notant $2k$ la longueur de w_1 (avec $1 \leq k \leq n$), nous avons par hypothèse de récurrence :

    $$N(w) = N(w_1)+N(w_2) \leq k(k+1)+(n+1-k)(n+1-k+1) = k(k-2(n+1))+(n+1)(n+2) = P_n(k).$$

    Le trinome P_n atteint son minimum au milieu de l'intervalle $[1, n]$ et la valeur maximale de $P_n(k)$ est obtenue pour $k = 1$ ou $k = n$, ce qui nous donne

    $$N(w) \leq P_n(1) = (n+1)(n+2) - 2n < (n+1)(n+2).$$

Ainsi, nous avons montré que $N(w) \leq n(n+1)$ pour tout mot bien parenthésé de longueur $2n$, avec égalité si et seulement si $w = (^n)^n$, ce qui prouve que l'algorithme a un temps de calcul dans le pire des cas de l'ordre de $n^2/2$.

Remarque : l'énoncé nous demande d'exhiber un cas limite, donnant la complexité dans le pire des cas, ce qui nous impose la preuve précédente, un peu fastidieuse. On aurait peut-être pu se contenter de majoration plus large, en remarquant que le mot $(^n)^n$ donne un temps de calcul de l'ordre de $n^2/2$, et que dans tous les cas, le temps de calcul est au maximum quadratique (on a deux boucles imbriquées de longueur au plus $2n$), obtenant un temps dans le pire des cas en $\Theta(n^2)$.

II.A.2) Quand on lit une parenthèse fermante en j-ème position dans w, la pile contient à son sommet l'indice i de la parenthèse ouvrante associée à la parenthèse fermante w_j. Il suffit donc de définir les entrées entrées d'indices i et j de `parenthese` : `parenthese[i]=j` et `parenthese[j]=i`.

```
procedure association(w:string;var resultat:tableau);
var i,j:integer; p:pile;
begin
creer_pile(p);
for j:=1 to length(w) do
   begin
   if w[j]='(' then
      empiler(j,p)
   else
      begin
      i := depiler(p); resultat[i] := j; resultat[j] := i;
      end;
   end;
end;
```

```
let association w =
   let p = creer_pile () in
   let l = string_length s in
   let t = make_vect l 0 in
      for i = 0 to l-1 do
         if w.[i] = '(' then
            empiler i p
         else
            let k = depiler p in t.(i) <- k; t.(k) <- i
      done;
   t;;
```

L'analyse se fait facilement :

- pour chaque indice j tel que $w_j = ($, l'algorithme empile une valeur dans `p` ;
- pour chaque indice j tel que $w_j =)$, l'algorithme dépile `p` et définit la valeur de 3 variables.

Le temps de calcul est donc linéaire, ce qui est bien meilleur que le temps quadratique obtenu avec le pemier algorithme. Ce temps est (en ordre de grandeur) optimal : un tableau ne peut se remplir en un temps négligeable devant sa taille.

II.B.1) Prouvons l'existence d'un couple (i, j) tel que $w \sim)^i (^j$ par induction sur la longueur de w.

- Le résultat est évident si w est de longueur nulle : $(i, j) = (0, 0)$ convient.
- Soit $n \in \mathbb{N}$ et supposons l'existence démontrée pour tout mot de longueur au plus n. Soit w de longueur $n+1$. Nous avons $w = au$ avec $a \in \{(,)\}$ et $|u| = n$, puis par hypothèse de récurrence, il existe i' et j' tels que $u \sim)^{i'} (^{j'}$. Nous avons quelques cas particuliers à étudier :
  - si $a =)$, $w \sim)^i (^j$ avec $i = i' + 1$ et $j = j'$;
  - si $a = ($ et si $i' = 0$, alors $w \sim)^i (^j$ avec $i = 0$ et $j = j' + 1$;
  - si $a = ($ et si $i' \geq 1$, $w \sim ())^{i'-1} (^{j'} \sim)^i (^j$ avec $i = i' - 1$ et $j = j'$.

Prouvons maintenant l'unicité du couple (i, j). Pour cela, notons φ la fonction qui, à un mot w sur l'alphabet $\{(,)\}$ associe la différence $|w|_(- |w|_)$, et ψ la fonction qui à un mot w associe le minimum des $\varphi(u)$ quand u décrit l'ensemble de préfixes de w. Nous allons montrer que si deux mots w_1 et w_2 sont équivalents, alors $\varphi(w_1) = \varphi(w_2)$ et $\psi(w_1) = \psi(w_2)$. Pour cela, il suffit de démontrer le résultat quand on peut écrire $w_1 = u()v$ et $w_2 = uv$ (on passe d'un mot à un mot équivalent en appliquant cette transformation un nombre fini de fois). Nous avons dans ce cas :

- $|w_1|_(= 1 + |w_2|_($ et $|w_1|_) = 1 + |w_2|_)$, donc $\varphi(w_1) = \varphi(w_2)$;
- $\psi(w_1) = \min(\psi(u), 1 + \varphi(u), \varphi(u), \varphi(u) + \psi(v))$ et $\psi(w_2) = \min(\psi(u), \varphi(u) + \psi(v))$. Comme $\psi(u) \leq \varphi(u) \leq 1 + \varphi(u)$, on a bien $\psi(w_1) = \min(\psi(u), \varphi(u) + \psi(v)) = \psi(w_2)$.

Ainsi, si un mot w est à la fois équivalent aux mots $w_1 =)^{i_1} (^{j_1}$ et $w_2 =)^{i_2} (^{j_2}$, alors

$$\begin{cases} j_1 - i_1 = \varphi(w_1) = \varphi(w_2) = j_2 - i_2, \\ -i_1 = \psi(w_1) = \psi(w_2) = -i_2, \end{cases}$$

d'où l'on déduit l'égalité des mots w_1 et w_2.

En fait, nous venons de démontrer que le représentant irréductible d'un mot w est le mot $)^i(^j$ où $i = -\psi(w)$ et $j = \varphi(w) - \psi(w)$. Ceci prouve en particulier que deux mots sont équivalents si et seulement ils ont mêmes images par φ et ψ.

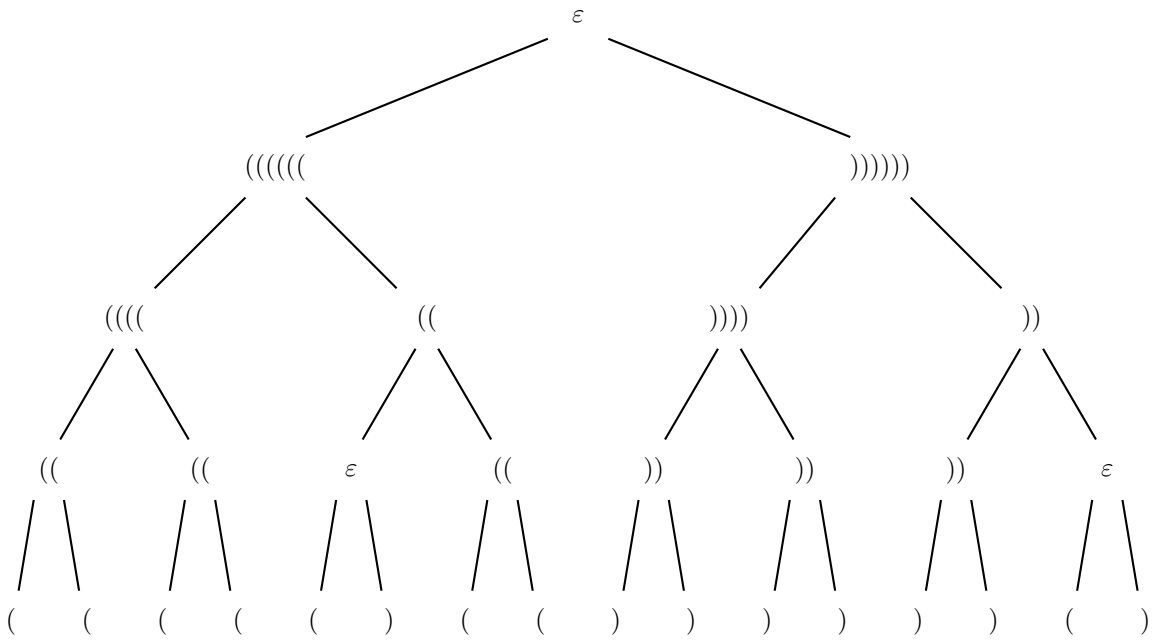
II.B.2) En posant $w_1 =)^{i_1}(^{j_1}$ et $w_2 =)^{i_2}(^{j_2}$, nous avons :

- si $j_1 \leq i_2$, $w_1 w_2 \sim)^{i_1+i_2-j_1}(^{j_2}$;
- sinon, $w_1 w_2 \sim)^{i_1}(^{j_1-i_2+j_2}$.

II.B.3) On sait qu'un mot w est bien parenthésé si et seulement si $\varphi(w) = \psi(w) = 0$ (voir question I.C.1). D'après les remarques faites à la question II.B.1), on en déduit qu'un mot est bien parenthésé si et seulement si sa forme irréductible est le mot vide.

II.B.4) On a vu à la question 2) que le calcul du représentant irréductible du concaténé de deux mots irréductibles se fait en temps constant (une comparaison, une addition, une soustraction et deux affectations). Le temps nécessaire à la construction de l'arbre réduit d'un mot w est donc de l'ordre du nombre de nœuds de cet arbre : le temps de construction est donc de l'ordre de $2^{p+1} - 1$, i.e. de l'ordre de $|w|$.

II.B.5) On obtient sans problème (mais avec de bons yeux pour compter les parenthèses) l'arbre ci-dessous.



L'arbre réduit associé à w_1

II.B.6) Nous supposons ici que chaque mot réduit u contenu dans l'arbre est représenté par le couple (i, j) tel que $u =)^i(^j$. Soit p la position de la parenthèse ouvrante dont on cherche à calculer la fermante associée. On

initialise trois variables :

$$i := 0,\ j := 1,\ m := 1$$

et on se place au nœud N correspondant à la p-ème feuille. Tout au long du calcul, (i, j) sera le code du mot contenu au nœud N et m sera la position de la parenthèse étudiée parmi les i parenthèses ouvrantes présentes dans le mot réduit (la position étant calculée à partir de la droite).

Ensuite, on va remonter dans l'arbre réduit jusqu'à ce que la parenthèse étudiée disparaisse, puis redescendre dans l'arbre en suivant la parenthèse fermante correspondante. Plus précisément, chaque étape de la remontée se fait ainsi : en notant (i', j') le couple associé au père du nœud courant N, et $(i", j")$ le couple associé au frère de N, on distingue trois cas :

- si N est le fils droit de son père, on ne modifie par m et pose $i := i'$ et $j := j'$; en effet, on a ajouté des parenthèses ouvrantes à gauche, et notre parenthèse qui se trouvait à la m-ième position parmi les j parenthèses ouvrantes se trouve toujours à la m-ième position parmi les j' parenthèses ouvrantes. Le nœud N est bien sûr remplacé par son père.
- si N est le fils gauche de son père et si $m > i"$, la parenthèse étudiée ne sera pas supprimée et on pose $m := m - i" + j"$, $i := i'$ et $j := j'$: en effet, les $i"$ premières parenthèse ouvrantes ont été fermées, puis $j"$ nouvelles parenthèses ouvrantes sont placées à droite. Le nœud N est ici encore replacé par son père.
- si N est le fils gauche de son père et si $m \leq i"$, la parenthèse est enfin fermée : on va pouvoir commencer la descente, en posant : $i := i"$, $j := j"$ et en conservant la valeur de m. Cette fois-ci, on remplace N par son frère (droit) et m désigne maintenant la position de la parenthèse fermante associée à la parenthèse ouvrante initiale, mais repérée à partir de la gauche.

Chaque étape de la descente se fait de la manière suivante, en notant (i', j') et $(i", j")$ les couples associés respectivement aux fils gauche et droit du nœud courant :

- si $i' \geq m$, on pose simplement $i := i'$, $j := j'$ et le nœud N est remplacé par son fils gauche.
- sinon, on a nécessairement $j' < i"$ et on pose $m := m - i' + j'$, $i := i"$ et $j := j"$, le nœud N étant remplacé par son fils droit.

l'algorithme s'arrêtant une fois atteint une feuille, qui correspond à la fermante cherchée.

Le temps de calcul est bien de l'ordre (dans le pire des cas) de la hauteur de l'arbre, i.e. de $\ln |w|$, le passage d'un nœud au suivant se faisant en un temps constant.

Dans le cas cité en exemple, avec $p = 4$, nous obtenons le parcours de l'arbre réduit représenté page suivante, dans lequel la parenthèse étudiée est tracée en gras et est un peu plus grande que les autres : la parenthèse cherchée est donc la onzième lettre de w_1.

II.B.7) Il est donc possible de calculer toutes les associations ouvrantes-fermantes en un temps de l'ordre de $|w| \ln |w|$, en effectuant le calcul détaillé dans la question précédente pour chaque parenthèse ouvrante. Comme $|w| \ll |w| \ln |w| \ll |w|^2$, ce temps est asymptotiquement compris entre les temps de calculs obtenus dans la partie II.A.

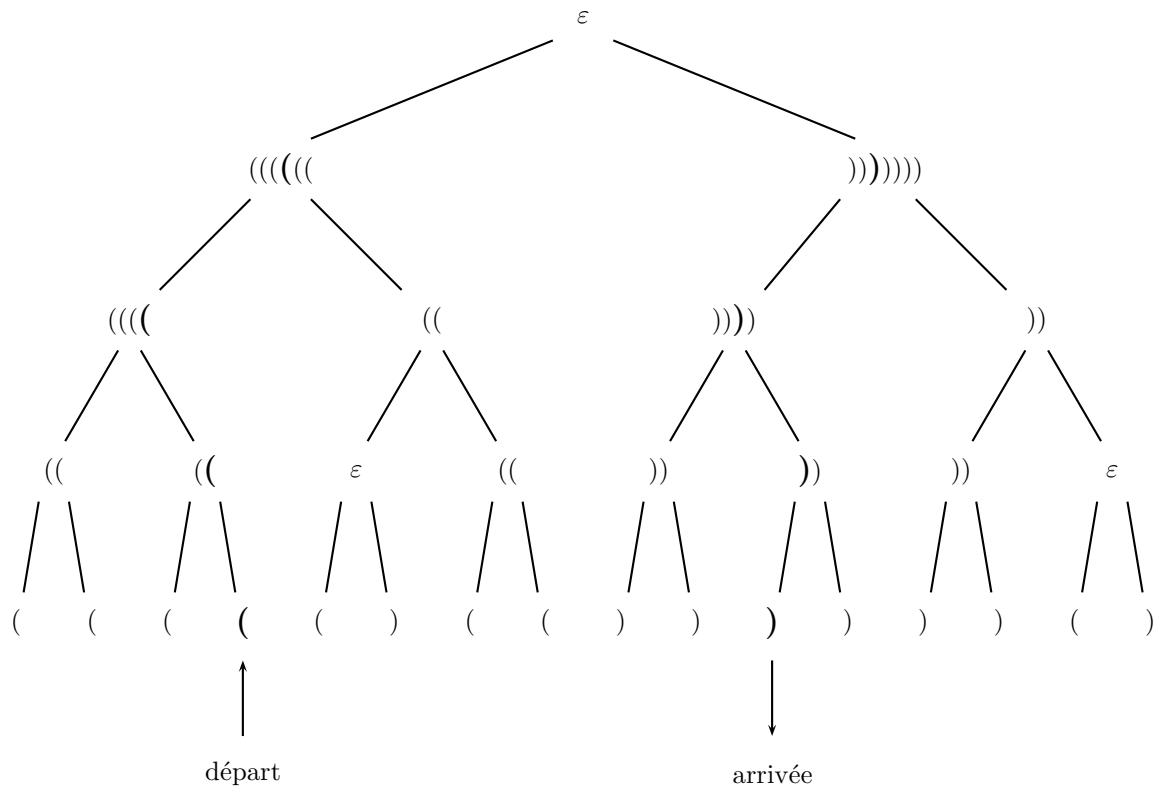
II.B.8) Notons N le nombre de processeurs. Dans un premier temps, on définit les feuilles en utilisant en parallèle les N processeurs : cela demande un temps T_0, égal au produit du temps de définition d'une feuille par $\lceil 2^p/N \rceil$ (où $\lceil x \rceil$ désigne la partie entière supérieure de x). Ensuite, on construit les feuilles de l'avant dernier étage (toujours en parallèle), ce qui demande un temps T_1, égal au produit du temps de définition d'un nœud par $\lceil 2^{p-1}/N \rceil$. On continue ainsi jusqu'à la racine de l'arbre, ce qui donne un temps total de calcul égal à

$T_w = T\sum_{k=0}^{p}\lceil 2^k/N\rceil$ où T désigne le temps de définition d'un nœud. Nous obtenons ainsi une majoration du temps de construction :

$$T_w \leq T\sum_{k=0}^{p}\left(\frac{2^k}{N}+1\right) = T\left(p+1+\frac{2|w|-1}{N}\right) = \Theta(p) = \Theta(\ln|w|)$$

puisque $N = \Theta(|w|)$. On peut donc construire l'arbre réduit de w en temps $\Theta(\ln(|w|))$ quand on dispose de $\Theta(|w|)$ processeurs.

Remarque : il y a une erreur grossière dans l'énoncé, l'auteur confondant les O et les Θ. Dire que l'on dispose de $O(|w|)$ processeurs n'a évidemment aucun sens (par exemple, 1 est un $O(|w|)$!) Il faut donc lire $\Theta(|w|)$ au lieu de $O(|w|)$, comme cela est d'ailleurs dit au début de la question.



Parcours de l'arbre pour le calcul de la quatrième fermante de w_1

PARTIE III - Parenthésage hétérogène

III.A On reprend une définition identique à celle donnée dans la partie I : on pose $L'_0 = \{\varepsilon\}$, puis pour $n \in \mathbb{N}$:

$$L'_{n+1} = L'_n \cup {L'_n}^2 \cup (L'_n) \cup [L'_n],$$

(L'_n) (resp. $[L'_n]$) représentant les mots de la forme (w) (resp. $[w]$) pour w décrivant L'_n. L'ensemble $\mathcal{L}'_P$ est alors la réunion de ces L'_n.

III.B On a clairement $\mathcal{L}_P = \mathcal{L}'_P \cap \{(,)\}^*$. Comme $\{(,)\}^* \in \mathrm{Rec}(A)$ et $\mathcal{L}_P \notin \mathrm{Rec}(A)$, $\mathcal{L}'_P$ n'est pas reconnaissable par automate fini, ni donc rationnel (d'après le théorème de Kleene).

Pour tester l'appartenance d'un mot w à $\mathcal{L}'_P$, nous allons encore utiliser une pile `p`. Une fois la pile initialisée, on lit chaque lettre $w[i]$:

- si $w[i]$ est ouvrant, on empile i dans `p` ;
- si $w[i]$ est fermant et si `p` est vide, on est assuré que w n'est pas bien parenthésé ;
- si $w[i]$ est fermant et si `p` est non vide, on dépile `p` ; si $w[$`depile(p)`$]$ et $w[i]$ ne sont pas du même type, on est une nouvelle sûr que w n'est pas bien parenthésé. Sinon, on peut passer à la valeur suivante de i.
- enfin, si on a parcouru toutes les lettres de w, w est bien parenthésé si et seulement si p est vide.

Le booléen `bool` permet d'arrêter le calcul dès que l'on a un décelé un défaut de parenthésage.

```
function parenthese_color(w:string):boolean;
var: i:integer; p:pile; bool:boolean;
begin
creer_pile(p);
i := 0; bool := true ;
while bool and i<length(w) do
   begin
   i := i+1;
   if w[i]='(' or w[i]='[' then
      empiler(i,p);
   else
      if est_vide(p) then
         bool := false
      else
         begin
         if w[depiler(p)]='(' then
            bool := w[i]=')'
         else
            bool := w[i]=']';
         end;
   end;
if bool then bool := est_vide(p);
parenthese_color := bool;
end;

let parenthese_color w =
   let l = string_length w in
   let p = creer_pile () in
   let rec aux i =
      if i=l then est_vide p else
      let c = w.[i] in
         if c='(' then begin empiler ')' p; aux (i+1) end
         else if c = '[' then begin empiler ']' p; aux (i+1) end
         else not (est_vide p) && c = depiler p && aux (i+1)
   in aux 0;;
```

III.C La question ne présente aucun intérêt : il suffit de reprendre la procédure `association` de la question II.A.2) puisque seul le caractère ouvrant ou fermant des parenthèses doit être pris en compte, en remplaçant

`if w[j] = '(' then` par `if w[j] = '(' or w[j] = '[' then`

`if w.[i] = '(' then` par `if w.[i] = '(' || w.[i] = '[' then`