# CORRIGÉ DE L'ÉPREUVE D'INFORMATIQUE DES ENSI - 2000

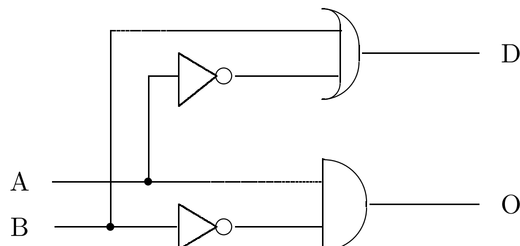
Programmation en Caml

## Partie I - Logique et calcul des propositions

**I.1** $P_1 = A, \quad P_2 = A \wedge B, \quad P_3 = \overline{B}.$

**I.2** $R = \big(P_1 \wedge \overline{P_2} \wedge P_3\big) \vee \big(\overline{P_1} \wedge P_2 \wedge \overline{P_3}\big) = \big(A \wedge \overline{A \wedge B} \wedge \overline{B}\big) \vee \big(\overline{A} \wedge A \wedge B \wedge B\big).$

**I.3** Puisque $A \wedge \overline{A} = 0$, il reste : $R = (A \wedge \overline{B}) \wedge (\overline{A} \vee \overline{B}) = \big(A \wedge \overline{B} \wedge \overline{A}\big) \vee \big(A \wedge \overline{B} \wedge \overline{B}\big) = \boxed{A \wedge \overline{B}.}$
L'action à effectuer est donc de baisser le levier $A$ uniquement.




## Partie II - Algorithmique et programmation en CaML

**II.1** Fonction `taille` de type `arbre -> int`

```
let rec taille a = match a with
               |         Vide -> 0
               | Noeud(_,g,d) -> 1 + taille g + taille d ;;
```

**II.2** Le nombre de nœuds d'un arbre binaire complet de profondeur $p$ est égal à :

$$C_p = 1 + 2 + 2^2 + \cdots + 2^p = \boxed{2^{p+1} - 1.}$$

**II.3** On a : $1 + (2^p - 1) = 2^p \leq n \leq 2^{p+1} - 1.$

**II.4** $2^p \leq n < 2^{p+1}$, donc $p \leq \log_2(n) < p + 1$, d'où $\boxed{p = \lfloor \log_2(n) \rfloor.}$

**II.5** Le nombre de nœuds placés à gauche de $n$ est égal à $\underline{n - 2^p}$.

**II.6** On en déduit que le nombre de nœuds dans le niveau $p + 1$ placés à gauche du fils gauche (s'il existe) du nœud de numéro $n$ est le double de la réponse précédente, donc $(2\,n) - 2^{p+1}$ et c'est $(2\,n + 1) - 2^{p+1}$ pour le fils droit.

**II.7** Le numéro du fils gauche du nœud $n$ situé au niveau $p$ est donc égal à :

$$C_p + (2\,n + 1) - 2^{p+1} = (2^{p+1} - 1) + (2\,n + 1) - 2^{p+1} = 2\,n.$$

Ainsi <u>le fils gauche a pour numéro $2\,n$ et le fils droit $2\,n + 1$</u>.

**II.8** On obtient immédiatement le numéro $\boxed{\lfloor n/2 \rfloor}$ pour le père du nœud de numéro $n$ pourvu que $n \geq 2$.

**II.9** Fonction `occurrence` de type `int -> int list`

```
let occurrence n =
        let rec aux k accu =
             if k= 1 then accu
             else if k mod 2 = 0 then aux (k/2) (0::accu)
                                 else aux (k/2) (1::accu)
        in aux n [] ;;
```

**II.10** On remarque que les fils gauches (resp. droits) ont pour occurrence celle de leur père suivie d'un **0** (resp. d'un **1**).
Comme il est plus facile en Caml de placer un élément en tête d'une liste plutôt qu'à la fin, on utilise une fonction récursive auxiliaire `aux` ayant comme argument `accu` qui sert à former la liste voulue dans la bon ordre. De plus la récursivité est terminale.

**II.11** Accès à l'étiquette d'un nœud.

```
let rec consulter c a = match a with
            |         Vide -> failwith "Erreur"
            | Noeud(r,g,d) -> if c = [] then r
                              else if hd(c) =0 then consulter (tl c)  g
                                               else consulter (tl c ) d ;;
```

**II.12** Insertion d'un nœud.

```
let rec inserer v c a = match a with
            |         Vide -> Noeud(v,Vide,Vide)
            | Noeud(r,g,d) -> if hd(c) = 0
                              then Noeud(r, inserer v (tl c) g, d)
                              else Noeud(r, g, inserer v (tl c) d) ;;
```

**II.13** L'insertion dans un arbre vide est immédiate (fin de la récursivité).
Sinon on insère $v$ dans l'arbre gauche (resp. droit) si la liste `c` commence par **0** (resp. par **1**) et on enlève l'élément de tête de la liste `c`.

**II.14** Le nombre d'appels récursifs effectués est égal à la profondeur $p = \lfloor \log_2(n) \rfloor$ où $n$ désigne la taille de l'arbre `a`.

**II.15** Fonction `inserer_tas` de type `int -> int list -> arbre -> arbre`

```
let rec inserer_tas v c a = match a with
      |         Vide -> Noeud(v,Vide,Vide)
      | Noeud(r,g,d) -> if hd(c) = 0

         then if r <= v then Noeud(r, inserer_tas v (tl c) g, d)
                        else Noeud(v, inserer_tas r (tl c) g, d)
         else if r <= v then Noeud(r, g, inserer_tas v (tl c) d)
                        else Noeud(v, g, inserer_tas r (tl c) d) ;;
```

**II.16** Si on doit insérer $v$ dans l'arbre gauche par exemple et si $r > v$, alors on place $v$ à la place de $r$ et on insère alors $r$ dans l'arbre gauche. Puisque $r$ était plus petit que la racine de l'arbre droit `d` (supposé non vide), il en sera de même a fortiori avec la nouvelle racine $v$ de l'arbre `a`. On obtient donc bien un tas une fois terminé.

**II.17** Là encore la complexité en nombre de comparaisons d'étiquettes est égale à la profondeur de l'arbre `a`.

**II.18** Fonction `construire` de type `int list -> arbre`

```
let rec construire l = match l with
          |  []  -> Vide
          | t::q -> let a = construire q in
                    inserer_tas t (occurrence (taille a + 1)) a ;;
```

**II.19** On parcours la liste `l` : pour chaque élément de cette liste à insérer, on calcule dans l'arbre obtenu précédemment l'"occurrence" du numéro d'insertion (qui est égal à la taille de l'arbre augmentée de 1).

**II.20** Si $n$ désigne la taille de la liste `l`, le nombre de comparaisons d'éléments de cette liste est égal à:

$$\sum_{k=1}^{n} \lfloor \log_2(k) \rfloor \leq \sum_{k=1}^{n} \log_2(k) \leq \int_1^{n+1} \log_2 t \, \mathrm{d}t = \underline{O(n \log_2(n))}.$$

**II.21** Fonction `aplatir` de type `Arbre -> int list`

```
let rec aplatir a =
    if a = Vide then []
    else let c = occurrence (taille a)
         in (consulter [] a)::(aplatir (extraire c a)) ;;
```

**II.22** La fonction `consulter` utilisée pour déterminer l'étiquette de la racine de $a$ a un cot constant. Les fonctions `occurrence` et `extraire` ont chacune une complexité en $O\big(\log_2(\textit{taille } a)\big)$, donc la complexité de la fonction `aplatir` est en $\displaystyle\sum_{k=1}^{n} O(\log_2 k) = O\big(\sum_{k=1}^{n} \log_2 k\big) = \underline{O(n \log_2 n)}$ où $n$ désigne la taiile de l'arbre $a$ passé en argument.

**II.23** `let trier l = aplatir(construire l) ;;`

**II.24** Les fonctions `aplatir` et `construire` ayant une complexité en $\underline{O(n \log_2 n)}$, il en est de même de la fonction `trier`.

## Partie III - Automates et langages

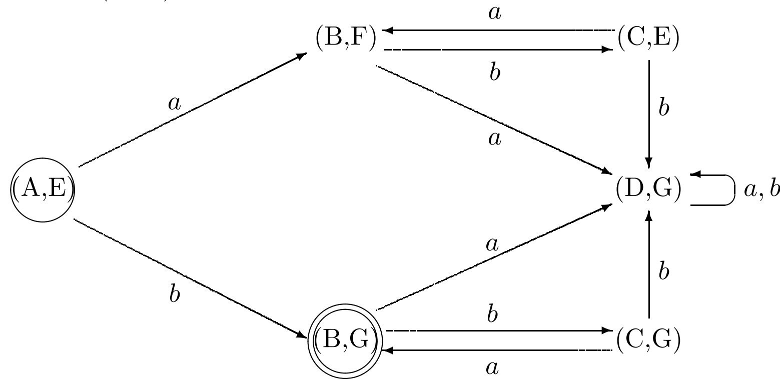
**III.1** $D$ étant un rebut, $\boxed{E_1 = (a+b)(ba)^*.}$

**III.1** $G$ étant un rebut, $\boxed{E_2 = a(ba)^*.}$

**III.3** On cherche d'abord un automate complet et accessible équivalent à $\mathcal{E}_1 \ominus \mathcal{E}_2$.

| | a | b |
|---|---|---|
| (A,E) | (B,F) | (B,G) |
| (B,F) | (D,G) | (C,E) |
| (C,E) | (B,F) | (D,G) |
| (B,G) | (D,G) | (C,G) |
| (C,G) | (B,G) | (D,G) |
| (D,G) | (D,G) | (D,G) |

(A,E) est l'élément initial et $(B,G)$ le seul état terminal.




**III.4** Une expression régulière du langage reconnu par $\mathcal{E}_1 \ominus \mathcal{E}_2$ est $\underline{b(ba)^*}$.
On constate que $L(\mathcal{E}_1 \ominus \mathcal{E}_2) = L(\mathcal{E}_1) \setminus L(\mathcal{E}_2)$.

**III.5** Si $\delta_1$ et $\delta_2$ sont définies sur $X \times Q_1$ et $X \times Q_2$ respectivement, alors $\delta_{1\ominus 2}$ est définie sur $X \times (Q \times Q_2)$, donc $A_1 \ominus A_2$ est un automate <u>complet</u>.

**III.6** Montrons par <u>induction structurelle</u> que $\underline{\forall\, m \in X^*,\ \delta^*_{1\ominus 2}\big(m, (q_1, q_2)\big) = \big(\delta^*_1(m, q_1), \delta^*_2(m, q_2)\big)}$ (**1**).

. C'est vrai si $m = \Lambda$ car $\delta^*_{1\ominus 2}\big(\Lambda, (q_1, q_2)\big) = (q,q_2) = \big(\delta^*_1(\Lambda, q_1), \delta^*_2(\Lambda, q_2)\big)$.

Supposons (**1**) vraie pour un mot $m$ et montrons qu'elle est aussi vraie pour le mot $m.x$ où $x$ est quelconque dans $X$.

$$\begin{aligned}
\underline{\delta^*_{1\ominus 2}\big(m.x, (q_1, q_2)\big)} &= \delta_{1\ominus 2}\Big(x, \delta^*_{1\ominus 2}\big(m, (q_1, q_2)\big)\Big) && \text{par définition de } \delta^*_{1\ominus 2} \\
&= \delta_{1\ominus 2}\Big(x, \big(\delta^*_1(m, q_1), \delta_2(m, q_2)\big)\Big) && \text{d'apès l'hypothèse d'induction} \\
&= \Big(\delta_1\big(x, \delta^*_1(m, q_1)\big), \delta_2\big(x, \delta^*_2(m, q_2)\big)\Big) && \text{par définition de } \delta_{1\ominus 2} \\
&= \underline{\big(\delta^*_1(m.x, q_1), \delta^*_2(m.x, q_2)\big)} && \text{par définition de } \delta^*_1 \text{ et } \delta^*_2.
\end{aligned}$$

**III.7**
$$\begin{aligned}
\underline{m \in L(A_1 \ominus A_2)} &\iff \delta_{1\ominus 2}\big(m, (q_1, q_2)\big) \in T_1 \times (Q_2 \setminus T_2) \\
&\iff \delta^*_1(m, q_1) \in T_1 \ \wedge\ \delta^*_2(m, q_2) \notin T_2 \\
&\underline{\iff m \in L(A_1) \ \wedge\ m \notin L(A_2)}.
\end{aligned}$$

**III.8** On a donc $\boxed{L(A_1 \ominus A_2) = L(A_1) \setminus L(A_2).}$

### Fin du corrigé