Option informatique

Partie I — Logique et calcul des propositions

Question I.1
On note a (*resp.* b) la proposition : *l'interrupteur A doit être fermé* (*resp. l'interrupteur B doit être fermé*).
Alors $p_1 = a$, $p_2 = a \wedge b$ et $p_3 = \neg b$.

Question I.2
La règle du jeu s'exprime alors sous la forme $p = (p_1 \wedge \neg p_2 \wedge p_3) \vee (\neg p_1 \wedge p_2 \wedge \neg p_3)$.

Question I.3
Dressons la table de vérité de p :

a	b	p_1	p_2	p_3	p
V	V	V	V	F	F
V	F	V	F	V	V
F	V	F	F	F	F
F	F	F	F	V	F

Tableau 1 Table de vérité de p

Il convient donc de fermer l'interrupteur A et de laisser ouvert l'interrupteur B.

Question I.4
On a donc $p = a \wedge \neg b$ et $\neg p = \neg a \vee b$, d'où le circuit ci-dessous :

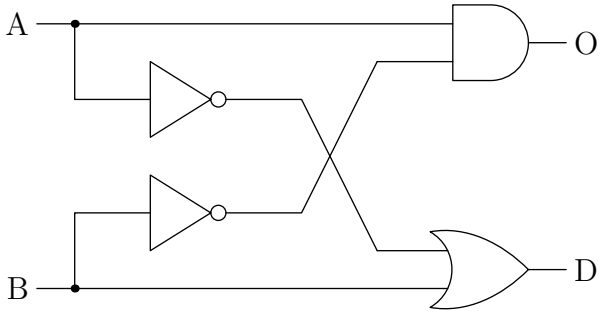


Figure 1 Un circuit électronique

Partie II — Automates et langages

Question II.1
L'expression régulière `a(ba|c)*` dénote le langage reconnu par $\mathcal{E}_1$.
L'expression régulière `(a|b|cc*a)(bc*a)*` dénote le langage reconnu par $\mathcal{E}_2$.

Question II.2
On obtient l'automate de la figure 2.

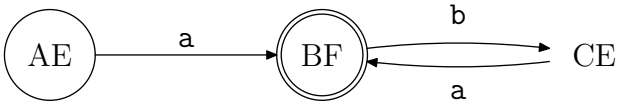


Figure 2 Un automate produit

Question II.3
Le langage reconnu par cet automate est dénoté par l'expression régulière `a(ba)*`.

Question II.4
Bien sûr, l'ensemble d'états $Q_1 \times Q_2$ est fini. De plus, la définition de $\delta_{1\times 2}$ peut se réécrire : $\forall (o_1, o_2) \in Q_1 \times Q_2, \forall x \in X, \delta_{1\times 2}((o_1, o_2), x) = (\delta_1(o_1, x), \delta_2(o_2, x))$, qui est bien définie en tant qu'application de $(Q_1 \times Q_2) \times X$ dans $Q_1 \times Q_2$.
C'est dire que l'automate $\mathcal{A} = \mathcal{A}_1 \times \mathcal{A}_2$ est déterministe et complet.

Question II.5
Il suffit de raisonner par récurrence sur $|m|$ pour démontrer la propriété demandée. En effet, si $|m| = 1$, on a simplement affaire à la définition de $\delta_{1\times 2}$ et la vérification est encore plus triviale si $|m| = 0$.
Montrons maintenant que le résultat est encore vrai pour un mot $m = e.m'$ en supposant qu'il est acquis pour le mot m'.
Soit donc $o = (o_1, o_2)$ un état quelconque dans $Q_1 \times Q_2$. Par définition de $\delta^*_{1\times 2}$, si on note $d = (d_1, d_2)$ l'état $\delta^*_{1\times 2}(o, e.m')$, on peut écrire que $(d_1, d_2) = \delta^*_{1\times 2}(q, m')$ où on a posé $q = (q_1, q_2) = \delta_{1\times 2}(o, e)$ c'est-à-dire $q_1 = \delta_1(o_1, e)$ et $q_2 = \delta_2(o_2, e)$.
Alors par hypothèse de récurrence $d_1 = \delta^*_1(q_1, m')$ et $d_2 = \delta^*_2(q_2, m')$. Mais alors on a bien $d_1 = \delta^*_1(o_1, m)$ et $d_2 = \delta^*_2(o_2, m)$, ce qui enclenche la récurrence.

Question II.6
Il suffit d'écrire que, par définition de $L(\mathcal{A}_1 \times \mathcal{A}_2)$, un mot m est dans ce langage si et seulement si $\delta^*_{1\times 2}((i_1, i_2), m) = (d_1, d_2) \in T_1 \times T_2$, ce qui, d'après la question précédente, peut se réécrire sous la forme $\delta^*_1(i_1, m) = d_1 \in T_1 \wedge \delta^*_2(i_2, m) = d_2 \in T_2$ ou encore $m \in L(\mathcal{A}_1) \wedge m \in L(\mathcal{A}_2)$.

Question II.7
Autrement dit : $L(\mathcal{A}_1 \times \mathcal{A}_2) = L(\mathcal{A}_1) \cap L(\mathcal{A}_2)$.

Partie III — Algorithmique et programmation en CaML

Question III.1
On écrit sans difficulté le programme 1.

```
let rec insertion_ens v = function
  | (t :: q) as l -> if t = v then l else t :: (insertion_ens v q)
  | [] -> [ v ] ;;
```

Programme 1 La fonction `insertion_ens`

Question III.2
Dans le meilleur cas, l'élément à insérer figurait déjà dans la liste en première position, et la fonction termine dès cette constatation. Dans le pire des cas, l'élément à insérer ne figurant pas dans la liste, notre fonction visite entièrement cette liste : la complexité est linéaire en la taille de celle-ci.

Question III.3
On écrit le programme 2.

```
let rec elimination_ens v = function
  | t :: q -> if t = v then q else t :: (elimination_ens v q)
  | [] -> [] ;;
```

Programme 2 La fonction `elimination_ens`

Question III.4
Le meilleur cas est celui où l'élément à retirer figurait déjà dans la liste, en première position : coût nul (aucun appel récursif ne sera effectué). Le pire cas est celui où l'élément à retirer ne figurait pas dans la liste : il y a alors autant d'appels récursifs que d'éléments dans la liste, la complexité est linéaire.

Question III.5
On peut écrire $|A| = \begin{cases} 0, & \text{si } A = \emptyset \ ; \\ 1 + \max(|\mathcal{G}(A)|, |\mathcal{D}(A)|), & \text{sinon.} \end{cases}$

Question III.6
Un arbre à n éléments sera de profondeur maximale, égale à n, dans le cas d'un arbre-ligne, c'est-à-dire un arbre dont tous les nœuds ont un fils gauche vide, par exemple.
La profondeur minimale p sera atteinte dans le cas d'un arbre-tas, où tous les niveaux jusqu'à la profondeur $p-1$ sont remplis et où seule la profondeur p n'est pas forcément pleine. Une récurrence évidente montre alors que pour les premiers niveaux il y a 2^{k-1} nœuds à la profondeur $k < p$ et donc au total $1 + 2 + \cdots + 2^{p-2} = 2^{p-1} - 1$ nœuds à profondeur au plus égale à p.
Ainsi a-t-on $1 \leqslant n - (2^{p-1} - 1) \leqslant 2^{p-1}$ donc $2^{p-1} \leqslant n \leqslant 2^p - 1$ ou encore $p - 1 = \lfloor \lg n \rfloor$.
La profondeur minimale est donc égale à $1 + \lfloor \lg n \rfloor$.

Question III.7
Dans toute la suite, nous appelons *feuille* un nœud qui n'a pas de fils gauche ni droit, et quand nous parlerons de *branche* nous évoquerons une branche reliant la racine à une feuille.
Considérons une branche la plus longue d'un arbre bicolore A de n nœuds. Sa longueur est égale à la profondeur $p = |A|$ de l'arbre, et parmi ces p nœuds, il y a exactement $\mathcal{R}(A)$ nœuds gris, donc $p - \mathcal{R}(A)$ nœuds blancs. En particulier $\mathcal{R}(A) \leqslant p = |A|$. Mais le fils d'un nœud blanc (sauf peut-être le dernier s'il s'agit d'une feuille) qui est dans la branche considérée est forcément gris : on en déduit que $\mathcal{R}(A) \geqslant (p - \mathcal{R}(A)) - 1$ ou encore que $p = |A| \leqslant 2\mathcal{R}(A) + 1$.
On a bien montré $\mathcal{R}(A) \leqslant |A| \leqslant 2\mathcal{R}(A) + 1$.

On peut montrer la minoration $n \geqslant 2^{\mathcal{R}(A)} - 1$ par récurrence sur n : la relation est évidente pour un arbre à un seul nœud, pour lequel $\mathcal{R}(A) = 0$ ou 1.

Supposons la relation vraie pour tous les arbres bicolores de taille au plus égale à n, on considère un arbre A bicolore de taille $n+1$: $\mathcal{G}(A)$ et $\mathcal{D}(A)$ sont tous deux des arbres bicolores, dont le rang vaut $\mathcal{R}(A)$ si la racine de A est blanche, et $\mathcal{R}(A)-1$ si la racine de A est grise.
Soit p et q les tailles respectives de $\mathcal{G}(A)$ et $\mathcal{D}(A)$.
Alors $n+1 = p+q+1 \geqslant (2^{\mathcal{R}(A)-1}-1)+(2^{\mathcal{R}(A)-1}-1)+1 = 2^{\mathcal{R}(A)}-1$, ce qui enclenche la récurrence.

Question III.8
Remarque : il y a une erreur d'énoncé, la majoration demandée est fausse. Pour un arbre possédant un seul nœud, on a $|A| = n = 1$ et $|A| \leqslant 2\lfloor \lg n \rfloor$ est faux.

La minoration $1 + \lfloor \lg n \rfloor \leqslant |A|$ a été démontrée pour tout arbre binaire, bicolore ou pas.
D'après la question précédente, $|A| \leqslant 2\mathcal{R}(A)+1$ et $2^{\mathcal{R}(A)} \leqslant n+1$ donc $\mathcal{R}(A) \leqslant \lfloor \lg(n+1) \rfloor$, et finalement

$$1 + \lfloor \lg n \rfloor \leqslant |A| \leqslant 2\lfloor \lg(n+1) \rfloor + 1.$$

On n'a pas démontré ce qui était demandé (et qui est faux), mais la majoration obtenue montre que la profondeur est un $O(\lg n)$, ce qui est le seul renseignement utile !

Question III.9
Dans le programme 3, on suppose que l'arbre fourni en argument à la fonction `rang` est bicolore. On peut donc se contenter de descendre dans une branche au choix pour calculer son rang.

```
let rec rang = function
    | Vide -> 0
    | Noeud(Gris,g,_,_) -> rang g + 1
    | Noeud(Blanc,g,_,_) -> rang g ;;
```

Programme 3 La fonction `rang`

Question III.10
Nous proposons deux versions de la fonction `validation_bicolore`.
La première version, du programme 4, utilise une exception pour interrompre le traitement dès qu'on repère une violation des règles de formation des arbres bicolores. La fonction auxiliaire `vérifie` déclenche l'exception si l'arbre fourni en argument n'est pas correctement colorié, et renvoie son rang sinon. Pour ce faire, elle s'appelle récursivement sur les sous-arbres gauche et droit. S'ils sont tous les deux corrects, elle vérifie encore qu'ils ont le même rang, et que si la racine de l'arbre est blanche, ses fils sont de racines grises.
Il suffit pour conclure que `vérifie a` s'effectue sans déclencher d'exception (on ignore le rang qu'elle a calculé) pour garantir la correction de l'arbre argument.
On a écrit, pour alléger l'écriture, une fonction auxiliaire `racine_grise` qui vérifie qu'un arbre est de racine grise.
La deuxième version, du programme 5, n'utilise pas d'exception. Elle travaille de façon analogue mais cette fois la fonction auxiliaire `vérifie` renvoie un couple (r, b), où r est le rang de l'arbre (comme dans la version précédente) et b est un booléen qui indique si l'arbre est correctement colorié.
On observera les tests (instructions `if`) imbriqués qui évitent des appels récursifs inutiles dès qu'on a repéré une violation des règles de coloriage.

Question III.11
La fonction `validation_bicolore`, dans ses deux versions, a la même complexité : le cas le pire est celui d'un arbre bien colorié, pour lequel elle devra explorer l'arbre en entier. Il y aura alors n appels récursifs, où n est le nombre de nœuds de l'arbre.

Question III.12
On propose sur le même modèle que `validation_bicolore` le programme 6. Cette fois, la fonction récursive auxiliaire `vérifie`, si elle ne trouve pas d'erreur, renvoie un couple d'entiers précisant les valeurs minimale et maximale des nœuds de l'arbre analysé.
Ici encore, on pourrait éviter le recours aux exceptions en ajoutant un booléen à la valeur retournée par la fonction auxiliaire.

```
exception Incorrect ;;

let racine_grise = function
  | Noeud(Gris,_,_,_) -> true
  | _ -> false ;;

let validation_bicolore a =
  let rec vérifie = function
    | Vide -> 0
    | Noeud(c,g,_,d) ->
       let rg = vérifie g and rd = vérifie d in
         if rg = rd then
           if c = Gris then rg + 1
           else if racine_grise g && racine_grise d then rg else raise Incorrect
         else raise Incorrect
  in
    try let _ = vérifie a in true with Incorrect -> false ;;
```

Programme 4 La fonction `validation_bicolore`, version avec exception

```
let validation_bicolore a =
  let rec vérifie = function
    | Vide -> (0,true)
    | Noeud(c,g,_,d) ->
       let (rg,b) = vérifie g in
         if b then let (rd,b) = vérifie d in
           if b && rg = rd then
             if c = Gris then (rg+1,true)
             else (rg,racine_grise g && racine_grise d)
           else (0,false)
         else (0,false)
  in
    match vérifie a with (_,b) -> b ;;
```

Programme 5 La fonction `validation_bicolore`, version sans exception

Question III.13
L'insertion aux feuilles dans un arbre binaire de recherche figure au programme de l'option... On obtient le programme 7, page 6.

Question III.14
La complexité de la fonction écrite est égale à la profondeur de la branche réellement explorée. Pour un arbre de recherche quelconque, on a vu qu'elle pouvait varier de 1, dans le meilleur cas, à n, dans le pire cas.
Si l'arbre utilisé est bicolore, on a une meilleure complexité dans le pire cas, puisque la profondeur de l'arbre est alors limitée à $2\lfloor \lg(n+1) \rfloor + 1$.

Question III.15
Il est clair que la condition **P1** n'est pas violée. La condition **P3** non plus, puisque l'on ajoute un nœud de couleur blanche.
En revanche, la condition **P2** est violée si le nœud blanc ajouté se retrouve fils d'un autre nœud blanc.

Question III.16
La racine v_2 est blanche, et ses deux fils sont de racines grises. En outre, sous un nœud gris, peu importe la couleur des nœuds fils. Enfin, F_1, F_2, F_3 et F_4 sont bicolores, donc finalement la condition **P2** est bien vérifiée par l'arbre obtenu par correction blanche. La condition **P1** est également trivialement vérifiée.

```
let validation_abr a =
  let rec vérifie = function
      | Vide -> (0,0) (* valeurs inutilisables *)
      | Noeud(_,Vide,v,Vide) -> (v,v)
      | Noeud(_,Vide,v,d) -> let (md,Md) = vérifie d in
         if v < md then (v,Md) else raise Incorrect
      | Noeud(_,g,v,Vide) -> let (mg,Mg) = vérifie g in
         if Mg < v then (mg,v) else raise Incorrect
      | Noeud(_,g,v,d) -> let (mg,Mg) = vérifie g and (md,Md) = vérifie d in
         if Mg < v && v < md then (mg,Md) else raise Incorrect
  in
    try let _ = vérifie a in true with Incorrect -> false ;;
```

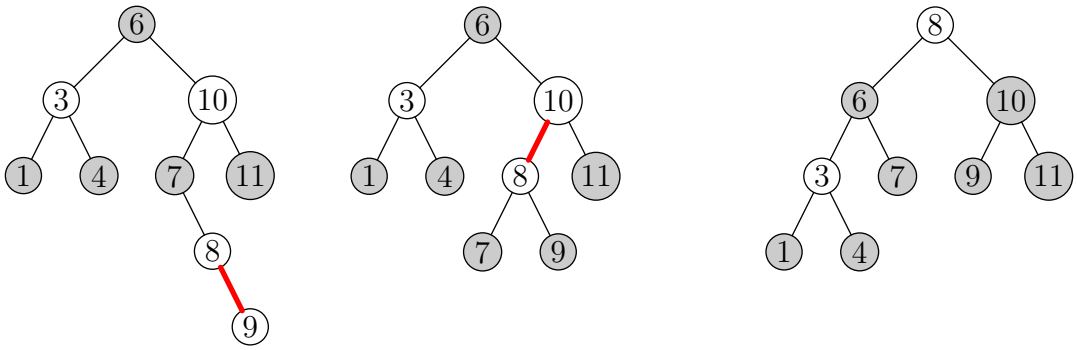
Programme 6 La fonction `validation_abr`

```
let rec insertion_abr v = function
    | Vide -> Noeud(Blanc,Vide,v,Vide)
    | Noeud(c,g,r,d) as a ->
       if v < r then Noeud(c,insertion_abr v g,r,d)
       else if v > r then Noeud(c,g,r,insertion_abr v d)
       else a ;;
```

Programme 7 La fonction `insertion_abr`

Toute branche issue de la racine v_2 est constitué ou bien de v_1 puis d'une branche de F_1 ou de F_2, ou bien de v_3 puis d'une branche de F_3 ou de F_4. On comptera donc toujours $n+1$ nœuds gris sur cette branche, et la condition **P3** est également vérifiée.

Question III.17
On obtient les étapes consécutives de la figure 3.



après l'insertion — première correction — deuxième correction

Figure 3 L'insertion et les deux corrections blanches

Question III.18
On dira dans la suite qu'un entier x sépare deux arbres G et D si, pour tout nœud g de G et tout nœud d de D on a $g < x < d$.
Supposons que les arbres F_1, F_2, F_3 et F_4 sont des arbres binaires de recherche bicolores et que v_1 sépare F_1 et F_2, v_2 sépare F_2 et F_3, et v_3 sépare F_3 et F_4.
Alors la question III.16 a établi que la forme transformée est un arbre bicolore. Les hypothèses de séparation permettent d'affirmer que c'est également un arbre de recherche.
Une correction blanche permet donc de fournir un sous-arbre bicolore ne violant plus la règle **P3**.

Plus précisément, si a est un arbre de recherche bicolore à une violation près de la règle **P3**, cette violation apparaissant à la profondeur p, l'arbre a' obtenu par correction blanche est toujours un arbre de recherche, et si la règle **P3** est violée, ce ne peut être qu'à la profondeur $p-1$.
Autrement dit, la correction blanche permet de remonter la filiation blanc-blanc fautive jusque vers la racine.

Attention ! L'énoncé semble supposer qu'un arbre comportant une violation de la règle **P3** est nécessairement d'une des quatre formes initiales qu'il décrit. En effet, la règle **P3** semble avoir pour corollaire que tout nœud blanc a pour père un nœud gris. C'est oublier un cas particulier important : si la racine de l'arbre est un nœud blanc, il se peut que la filiation blanc-blanc fautive remonte jusqu'à la racine, et dans ce cas l'arbre fautif voit sa racine et la racine d'un de ses deux fils de la même couleur blanche. Il n'y a pas de correction blanche envisageable au sens de ce que prévoit l'énoncé. Mais il y a une correction facile à imaginer : il suffit de colorier la racine en gris pour résoudre le problème ! C'est d'ailleurs dans ce seul cas que le rang de l'arbre bicolore augmentera d'une unité.

Question III.19
$|A| - \mathcal{R}(A)$ est un majorant du nombre de nœuds blancs présents sur une branche (liant la racine à une feuille).
Or après l'insertion d'une clé, la violation de **P3** n'interviendra qu'en présence d'un nœud blanc sur la branche le long de laquelle on remonte, provoquant une correction blanche. On opèrera donc au plus $|A| - \mathcal{R}(A)$ corrections blanches.

Remarque : il y a encore une erreur d'énoncé, puisque ce majorant n'est pas strict. Il suffit d'ailleurs d'observer le cas de l'insertion examinée dans la question III.17 : l'arbre de départ était de profondeur 4 et de rang 2. Et il a fallu $4 - 2 = 2$ corrections blanches pour le réparer !

Question III.20
La seule difficulté dans la rédaction du programme 8 est d'ordre syntaxique !

```
let correction_blanche = function
  | Noeud(Gris,Noeud(Blanc,Noeud(Blanc,f1,v1,f2),v2,f3),v3,f4)
    -> Noeud(Blanc,Noeud(Gris,f1,v1,f2),v2,Noeud(Gris,f3,v3,f4))
  | Noeud(Gris,Noeud(Blanc,f1,v1,Noeud(Blanc,f2,v2,f3)),v3,f4)
    -> Noeud(Blanc,Noeud(Gris,f1,v1,f2),v2,Noeud(Gris,f3,v3,f4))
  | Noeud(Gris,f1,v1,Noeud(Blanc,f2,v2,Noeud(Blanc,f3,v3,f4)))
    -> Noeud(Blanc,Noeud(Gris,f1,v1,f2),v2,Noeud(Gris,f3,v3,f4))
  | Noeud(Gris,f1,v1,Noeud(Blanc,Noeud(Blanc,f2,v2,f3),v3,f4))
    -> Noeud(Blanc,Noeud(Gris,f1,v1,f2),v2,Noeud(Gris,f3,v3,f4))
  | a -> a ;;
```

Programme 8 La fonction `correction_blanche`

Question III.21
Le programme 9 se déduit aisément du programme 7, page 6 : il suffit d'appeler la fonction `correction_blanche` qu'on vient d'écrire après chaque appel récursif, puisqu'on a pris la précaution, dans le programme 8, de retourner tel quel un arbre qui n'a pas besoin de correction blanche.
En revanche, il faut penser à tester si la filiation blanc-blanc fautive apparaît à la racine de l'arbre à la fin de l'appel : c'est pourquoi les appels récursifs ont été ici confiés à une fonction auxiliaire `insère`.

```
let insertion_abr v a =
  let rec insère = function
    | Vide -> Noeud(Blanc,Vide,v,Vide)
    | Noeud(c,g,r,d) as a ->
       if v < r then correction_blanche (Noeud(c,insère g,r,d))
       else if v > r then correction_blanche (Noeud(c,g,r,insère d))
       else a
  in
    match insère a with
      | Noeud(Blanc,(Noeud(Blanc,_,_,_) as g),r,d) -> Noeud(Gris,g,r,d)
      | Noeud(Blanc,g,r,(Noeud(Blanc,_,_,_) as d)) -> Noeud(Gris,g,r,d)
      | a -> a ;;
```

Programme 9 La fonction `insertion_abr` finale