Corrigé CCP 2017

Lycée Saint-Stanislas : MP

Partie I : Logique et calcul des propositions

1. On a $(X_1 \wedge Z_1) \vee (\neg X_1 \wedge \neg Z_1)$.
2. On a $X_1 = V \vee C$ et $Z_1 = \neg V$.
3. On a donc

$$\begin{aligned}(X_1 \wedge Z_1) \vee (\neg X_1 \wedge \neg Z_1) &\equiv ((V \vee C) \wedge \neg V) \vee (\neg V \wedge \neg C \wedge V) \\ &\equiv ((V \wedge \neg V) \vee (C \wedge \neg V)) \vee (\neg V \wedge \neg C) \\ &\equiv (C \wedge \neg V) \vee (\neg V \wedge \neg C)\end{aligned}$$

   La seconde proposition est nécessairement fausse, et donc le village se trouve dans les collines.

4. On a donc $\varphi = (X_2 \wedge Y_2 \wedge Z_2) \vee (\neg X_2 \wedge \neg Y_2 \wedge \neg Z_2)$.
5. On a $X_2 = G \wedge D$, $Y_2 = M \Rightarrow \neg D$ et $Z_2 = G \wedge \neg M$.
6. Regardons la table de vérité de φ

G	M	D	X_2	Y_2	Z_2	$X_2 \wedge Y_2 \wedge Z_2$	$\neg X_2 \wedge \neg Y_2 \wedge \neg Z_2$	φ
0	0	0	0	1	0	0	0	0
0	0	1	0	1	0	0	0	0
0	1	0	0	1	0	0	0	0
0	1	1	0	0	0	0	1	1
1	0	0	0	1	1	0	0	0
1	0	1	1	1	1	1	0	1
1	1	0	0	1	0	0	0	0
1	1	1	1	0	0	0	0	0

   On a donc $\neg G \wedge M \wedge D$ ou $G \wedge \neg M \wedge D$. Il faut donc emprunter le chemin de droite.

7. Si on sait que les trois participants ont menti, on peut regarder l'avant-dernière colonne du tableau, et donc on peut emprunter le chemin du milieu ou de droite.

1 Partie II : Algorithmique et programmation (Informatique pour tous)

8. Le programme affiche `9 0 2 0 1 7 15` puis `9 3 4 3 9 12 15`, et la variable `resultat` contient l'entier 3. La variable `exemple` contient `[1,4,7,9,12,15]`.
9. Commençons par montrer que la propriété proposée est un invariant de boucle.

   Au début, on a $d = 0$, $f = n$, et par (iii), il existe bien $i \in [d, f]$ tel que $p_i = v$.

   Supposons maintenant que la propriété est vérifiée avant une boucle, et montrons qu'elle est toujours vraie après.

   Puisqu'on est rentrés dans la boucle, on a donc $d < f$. Trois cas se présentent :

   - si $v < p_m$, alors f devient $m - 1$ qui est bien supérieur ou égal à d.
     Puisque la liste est triée par ordre croissant, il est clair que $\forall i > f,\ p_i > v$.
     Par hypothèse d'induction, on a toujours $\forall i < d,\ p_i \neq v$.
     Finalement, par (iii), il existe un rand $i \in [d, f]$ pour lequel $p_i = v$.
   - si $v > p_m$, alors le même raisonnement permet de conclure.
   - sinon, $v = p_m$, et dans ce cas on a bien $d = f$, et $p_d = p_m = v$.

On a donc bien un invariant de boucle.

En regardant la dernière itération, on a alors $r = d = f$, et donc

$$\exists i \; in[r, r], \; p_i = v,$$

et donc $v = p_r$.

10. Notons n la longueur de p. Si $n = 0$ ou 1, le résultat est évident, puisqu'on ne rentre pas dans la boucle.

    Montrons qu'il existe un moment pour lequel $f = d$. Supposons le contraire, *i.e.* que $f > d$ à tout moment du programme.

    Montrons que les valeurs $f - d$ diminuent strictement à chaque tour de boucle.

    Supposons qu'avant de rentrer dans la boucle, on a $f - d > 0$. Alors si $v < p[m]$, alors f devient $f - 1$, et donc $f - 1 - d < f - d$, et si $v > p[m]$, alors d devient $d + 1$, et donc $f - (d + 1) < f - d$. Si $v = p[m]$, alors $f - d$ devient nul, et $0 < f - d$.

    La suite des $f - d$ est donc une suite d'entiers strictement décroissante, ce qui est impossible, et donc il existe bien un moment où $f = d$, ce qui arrête le programme.

11. Prenons $p = [\![0, 2^n - 1]\!]$ pour $n \in \mathbb{N}^*$, et $v = 0$ ou $v = 2^n - 1$. Alors l'algorithme de dichotomie sera en complexité $O(n) = O(\texttt{len}(p))$.

Partie III : Automates et langages

12.

```
type lettre = char;;
type alphabet = lettre list;;
type mot = lettre list;;
type langage = mot list;;
```

13.

```
let rec prefixer u l =
  match l with
  |[] -> []
  |t :: q -> (u@t)::(prefixer u q);;
```

14. La fonction `prefixer` fait autant d'appels récursifs que la longueur du langage. Chaque conténation demande $|u|$ appels récursifs, et on a donc une complexité en $O(|u||l|) = O(|l|)$.

15. Les mots seront représentés par des chaînes de caractères, et les langages par des listes de chaînes de caractères.

16.

```
def prefixer(u,l):
  res = l[:]
  n = len(l)
  for i in range(n):
    res[i] = u+res[i]
  return l
```

17. On retrouve encore une complexité en $O(|l|)$, dûe à la boucle inconditionnelle.

18. Pour l'automate $\mathcal{E}_1$, on note que D est un état puit. Il est ensuite clair que le langage reconnu est $L(\mathcal{E}_1) = a(b^*(ab)^*)^* + bb(b^*(ab)^*)^* = (a + bb)(b + ab)^*$.

    De la même fqçon, H est un état puit pour $\mathcal{E}_2$. On a alors $L(\mathcal{E}_2) = ab(a + cb)^* + c(a + bc)^* = (ab + c)(a + cb)^*$.

19.

```
type etat = int
type transition = etat * lettre * etat
type automate = {q : etat list; x:alphabet; i:etat list; t:etat list;
  gamma:transition list}

let expl_III_1 = {
  Q = [0;1;2;3];
```

```
  X = ['a';'b';'c'];
  I = [0];
  T = [1];
  gamma = [ (0,'a',1); (0,'c',3); (0,'b',2); (1,'b',1); (1,'a',2);
  (1,'c',2); (2,'a',3);(2,'b',1);(2,'c',3);(3,'a',3); (3,'b',3);(3,'c',3)]}
```

20. On vérifie les différentes inclusions

```
let rec inclus u v =
  match u with
  |[] -> true
  |t::q -> List.mem t v && inclus q v

let rec gamma_valide gamma q x =
  match gamma with
  |[] -> true
  |(u,l,v)::t -> List.mem u q &&
                 List.mem v q &&
                 List.mem l x &&
                 gamma_valide t q x

let valider a =
  inclus a.i a.q &&
  inclus a.i a.q &&
  gamma_valide a.gamma a.q a.x
```

21. On va à chaque lecture de lettre calculer la liste des états accessibles en lisant le préfixe du mot jusqu'à cette lettre.

```
let rec etape l liste_etats gamma =
  match liste_etats with
  |[] -> []
  |t :: q -> match gamma with
             |[] -> []
             |(u,s,v)::gamma' ->
               let liste_arrivee = etape l liste_etats gamma' in
               if List.mem v liste_arrivee ||
                    s <> l ||
                      not (List.mem u liste_etats)
               then liste_arrivee
               else v :: liste_arrivee;;

let rec disjoint l1 l2 =
  match l1 with
  |[] -> true
  |t::q -> if List.mem t l2 then false else disjoint q l2;;

let accepter mot a =
  let rec aux mot a etat =
    match mot with
    |[] -> not (disjoint etat a.t)
    |l :: reste -> aux reste a (etape l etat a.gamma)
```

22. La fonction `etape` parcourt la liste des transitions complète. La fonction `accepter` appelle la fonction `etape` pour chaque lettre du mot, ce qui donne une complexité en $O(|m||gamma|)$.

23. En Python, on peut de la même façon représenter un automate par une liste `[Q,X,I,T,gamma]` où `Q` est une liste d'entiers, `X` une liste de caractères, `I` et `T` des listes d'entiers, et `gamma` une liste de triplets.

    Pour l'exemple

```
expl_III_1 = [
  [0,1,2,3],
  ['a','b','c'],
```

```
  [0],
  [1],
  [ [0,'a',1], [0,'c',3], [0,'b',2], [1,'b',1], [1,'a',2],
    [1,'c',2], [2,'a',3],[2,'b',1],[2,'c',3],[3,'a',3], [3,'b',3],[3,'c',3]]]
```

24. On écrit une nouvelle fonction `inclus`, puis on vérifie

```
def inclus(u,v):
  for i in u:
    if i not in v:
      return False
  return True

def gamma_valide(gamma,Q,X):
  for u,l,v in gamma:
    if u notin Q or v notin q or l notin X:
      return False
  return True

def valider(a):
  Q,X,I,T,gamma = a
  return (inclus I Q) and (inclus T Q) and gamma_valide gamma
```

25. TODO

26. On a

$$\begin{aligned} abc\|_S abd &= a(bc\|_S abd) + a(abc\|_S bd) \\ &= aa(bc\|_S bd) + aa(bc\|_S bd) \\ &= aa(bc\|_S bd) \\ &= aab(c\|_S d) \\ &= aab(c(\varepsilon\|_S d) + d(c\|_S \varepsilon)) \\ &= aab(cd + dc) \end{aligned}$$

27. Soient $m_1, m_2 \in X^*$. Si $m_1 = \varepsilon = m_2$, alors $m_1\|_S m_2 = \{\varepsilon\}$, et il est clair que l'implication réciproque est vraie.

    Réciproquement, supposons $m_1 \neq \varepsilon$: $m_1 = an_1$, $a \in X$.

    On regarde ensuite différent cas :

    - si $m_2 = \varepsilon$ et $a \in S$, alors $m_1\|_S m_2 = \emptyset$ qui ne contient pas le mot vide
    - si $m_2 = \varepsilon$ et $a \notin S$, alors $m_1\|_2 m_2 = a(n_1\|_S \varepsilon)$, qui est soit vide, soit contient des mots d'au moins une lettre
    - si $m_2 = bn_2$, $b \in X$, alors dans tous les cas; $m_1\|_2 m_2$ est soit vide, soit tous ses mots admettent a ou b pour préfixe.

28. Soient $s \in S$, $m, m_1, m_2 \in X^*$. Pour le sens réciproque, c'est une application directe de la cinquième règle de lq liste.

    Pour le sens direct, supposons donc $sm \in m_1\|_S m_2$. On note qu'à l'exception de cette cinquième règle, toutes les autres donnent des ensembles soit vide, soit préfixés par des lettres qui ne sont pas dans S. L'unique façon d'avoir un mot commençant par s est donc d'avoir appliqué cette cinquième règle, et on a bien l'implication directe.

29. Soit $x \in X\backslash S$, et soient $m, m_1, m_2 \in X^*$.

    Un mot commençant par une lettre non synchronisante ne peut provenir que des règles 4, 7 ou 8 ; cette lettre est alors nécessairement au début de m_1 ou m_2.

30. On commence par définir l'union de deux langages pour implémenter la dernière règle.

```
let rec union l1 l2 =
  match l1 with
  |[] -> l2
  |t::q -> if List.mem t l2 then union q l2
           else t::(union q l2);;

let rec synchro_mot m1 m2 syn =
  match m1,m2 with
  |[],[] -> [[]]
```

```
  |s::m,[] when List.mem s syn -> []
  |[],s::m when List.mem s syn -> []
  |x::m,[] -> prefixer [x] (synchro_mot m [] syn)
  |[],x::m -> prefixer [x] (synchro_mot [] m syn)
  |s1::m1',s2::m2' when List.mem s1 syn && List.mem s2 syn ->
    if s1 = s2 then prefixer [s1] (synchro_mot m1' m2' syn)
    else []
  |s::m1',x::m2' when List.mem s syn -> prefixer [x] (synchro_mot (s::m1') m2' syn)
  |x::m1',s::m2' when List.mem s syn -> prefixer [x] (synchro_mot (s::m2') m1' syn)
  |x1::m1',x2::m2' -> union (prefixer [x1] (synchro_mot m1 (x2::m2') syn))
                       (prefixer [x2] (synchro_mot (x1::m1') m2' syn));;
```

31. TODO

32. On peut prendre

```
let rec synchro_lang l1 l2 syn =
  match l1,l2 with
  |[],_ -> []
  |_,[] -> []
  |m1::l1',m2::l2' ->
    union (synchro_mot m1 m2 syn)
      (union (synchro_lang l1' l2 syn) (synchro_lang l1 l2' syn));;
```

33. TODO

34. TODO

35.