Partie III - Automates et langages

1 Mots et langages

Q12.

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

Implicitement, l'énoncé semble ne considérer que des langages finis.

Q13.

```
(* Concaténation *)
let rec concat l1 l2 =
  match l1 with
  | [] -> l2
  | t :: q -> t :: (concat q l2)
;;

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

Q14. Le nombre d'appel récursifs dans `concat` est $|l1|$, ce qui donne pour `prefixer` une complexité de $|p| \cdot |l| = O(|l|)$.

Q15. On choisit de représenter les mots par des chaînes de caractères et les langages (supposés finis) par des listes de chaînes de caractères.

Q16.

```
def prefixer(p, l):
    """renvoie le langage dans lequel tous les mots du langage l ont été
préfixés par le mot p"""
    L = []
    for mot in l:
        L.append(p + mot)
    return L
```

Q17. De nouveau, on obtient une complexité en $O(|l|)$.

2 Automate fini

Q18. $L(\mathscr{E}1) = \left(a \middle| b^2\right)(ab|b)^*$ et $L(\mathscr{E}2) = (ab|c)(cb|a)^*$.

Q19. Comme semble le suggérer l'énoncé, les transitions seront modélisées par une liste de triplets. Ce n'est pas optimal en terme de complexité, mais cela permet de compléter ce qui a été vu en TP d'option.

```
type etat = int;;
type transition = int * lettre * int;;
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, 'b', 2); (0, 'c', 3); (1, 'a', 2); (1, 'b', 1); (1, 'c', 3);
    (2, 'a', 3); (2, 'b', 1); (2, 'a', 3); (3, 'a', 3); (3, 'b', 3); (3, 'c', 3)
  ]
};;
```

Q20.

```
let rec est_dans x liste =
  match liste with
  | [] -> false
  | t :: q -> (t = x) || (est_dans x q)
;;

let rec est_inclus liste1 liste2 =
  match l1 with
  | [] -> true
  | t :: q -> (est_dans t liste2) & (est_inclus q liste2)
;;

let rec valider_transitions etats alphabet transitions =
  match transitions with
  | [] -> true
  | (e1, lettre, e2) :: q -> est_dans e1 etats & est_dans e2 etats
    & est_dans lettre alphabet & valider_transitions etats alphabet q
;;

let valider a =
  est_inclus a.I a.Q
  & est_inclus a.T a.Q
  & valider_transitions a.Q a.X a.gamma;;
```

Q21.

```
(* renvoie la liste de tous les états accessibles á partir des etats_orig par la lettre
   dans l'automate a *)
let rec etats_suivants lettre etats_orig gamma =
  match etats_orig, gamma with
  | _, [] -> []
  | [], _ -> []
  | _, (e1, lettre1, e2) :: q ->
    let etats_dest = etats_suivants lettre etats_orig q in
    if est_dans e2 etats_dest || lettre1 <> lettre || not est_dans e1 etats_orig then
      etats_dest
    else
      e2 :: etats_dest
;;

(* renvoie la liste des états dans lesquels on se trouve aprés avoir lu le mot m á partir
   de l'un des états de la liste etats *)
let rec itere m etats gamma =
  match m with
  | [] -> etats
  | t :: q -> itere q (etats_suivants t etats gamma) gamma
;;

(* teste si l'intersection des listes n'est pas vide *)
let rec rencontre liste1 liste2 =
  match liste1 with
  | [] -> false
  | t :: q -> (est_dans t liste2) || (rencontre q liste2)
;;

let accepter m a =
  rencontre (itere m a.I a.gamma) a.T
;;
```

Q22. La fonction `etats_suivants` a une complexité... pas très belle!! Pour chaque lettre de `m`, on parcourt `gamma`. Soit $O(|\texttt{m}|\cdot|\texttt{gamma}|)$ en négligeant les `est_dans` et `rencontre`...

Q23. Un automate est modélisé par une liste `automate` telle que si `Q, X, I, T, gamma = automate`,

- `Q` est la liste des états (entiers),
- `X` est l'alphabet (liste de caractères),
- `I` est la liste des états initiaux,
- `T` est la liste des états terminaux,
- `gamma` est la liste des triplets `(o, e, d)` représentant une transition.

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


Q24.

```
def valider(a):
    "teste si l'automate a est correctement dêfini."
    Q, X, I, T, gamma = a
    # Vérification des états initiaux
    for etat in I:
        if etat not in Q:
            return False
    # Vérification des états terminaux
    for etat in T:
        if etat not in Q:
            return False
    # Vérification des transitions
    for orig, etiq, dest in gamma:
        if orig not in Q or dest not in Q or etiq not in X:
            return False

    return True
```

Q25.

```
def etats_suivants(lettre, etats_orig, a):
    """renvoie la liste des états que l'on peut atteindre avec la lettre
á partir d'une liste d'état d'origine, dans l'automate a"""
    gamma = a[4]
    etats_dest = []
    for orig, etiq, dest in gamma:
        if etiq == lettre and orig in etats_orig and dest not in etats_dest:
            etats_dest.append(dest)
    return etats_dest

def accepter(m, a):
    "teste si le mot m est accepté par l'automate a"
    Q, X, I, T, gamma = a

    etats = I # on part des états initiaux
    for lettre in m: # on met á jour les états au fur et á mesure
        etats = etats_suivants(lettre, etats, a) # de la lecture des lettres

    # on vérifie si on a l'un des état terminaux
    for etat in T:
        if etat in etats:
            return True
    return False
```

3 Synchronisation de mots

Q26.

$$\begin{aligned} abc \,||_S\, abd &= a(bc \,||_S\, abd) \cup a(abd \,||_S\, bd) \\ &= aa(bc \,||_S\, bd) \cup a^2(bd \,||_S\, bd) \\ &= aa(bc \,||_S\, bd) \\ &= aab(c \,||_S\, d) \\ &= aab(c(\varepsilon \,||_S\, d) \cup d(c \,||_S\, \varepsilon)) \\ &= \{aabcd, aabdc\}. \end{aligned}$$

Q27. L'implication $\Leftarrow$ est immédiate. Pour l'autre, on observe les règles et on voit que dès que m_1 ou m_2 n'est pas le mot vide, alors $m_1 \,||_S\, m_2$ est soit vide, soit contient des mots ayant au moins un caractère (x ou s ou x_1 ou x_2).

Q28. L'implication $\Leftarrow$ est une conséquence de la 5^e règle. Pour l'autre sens, on observe que c'est la seule règle permettant d'avoir un mot commençant par une lettre s synchronisante, d'où l'implication.

Q29. De la même manière, les deux dernières règles et la 4^e conduisent au cas où un mot commence par une lettre non synchronisante, lettre qui se trouve nécessairement au début de m_1 ou de m_2. (à détailler...)

Q30.

```
let rec union liste1 liste2 =
  match liste1 with
  | [] -> liste2
  | t :: q ->
    if est_dans t liste2 then
      union q liste2
    else
      t :: union q liste2
;;

let rec synchro_mot m1 m2 s =
  match m1, m2 with
  | [], [] -> [[]]
  | [], t :: _ when est_dans t s -> []
  | [], t :: q -> prefixer [t] (synchro_mot q [] s)
  | _, [] -> synchro_mot m2 m1 s
  | t1 :: q1, t2 :: q2 when est_dans t1 s ->
    if t2 = t1 then
      prefixer [t1] (synchro_mot q1 q2 s)
    else if est_dans t2 s then
      []
    else
      prefixer [t2] (synchro_mot m1 q2 s)
  | t1 :: q1, t2 :: q2 when est_dans t2 s -> prefixer [t1] (synchro_mot q1 m2 s)
  | t1 :: q1, t2 :: q2 ->
    union (prefixer [t1] (synchro_mot q1 m2 s)) (prefixer [t2] (synchro_mot m1 q2 s))
;;
```

Q31.

```
def union(L1, L2):
    L = L1[:]
    for elem in L2:
        if elem not in L1:
            L.append(elem)
    return L

def synchro_mot(m1, m2, s):
    "renvoie le langage m1 ||s m2"
    if len(m1) == 0 and len(m2) == 0:
        return [""]
    if len(m1) < len(m2):
        return synchro_mot(m2, m1, s)

    if len(m2) == 0:
        if m1[0] in s: # ici m1 ne peut être videt1 :: q1
            return []
        else:
            return prefixer(m1[0], synchro_mot(m1[1:], "", s))

    # ici m1 et m2 ne peuvent être vides
    if m1[0] in s:
        if m1[0] == m2[0] :
            return prefixer(m1[0], synchro_mot(m1[1:], m2[1:], s))
        elif m2[0] in s:
            return []
        else:
            return prefixer(m2[0], synchro_mot(m1, m2[1:], s))

    # ici m1[0] n'est pas dans s
    if m2[0] in s:
        return prefixer(m1[0], synchro_mot(m1[1:], m2, s))

    return union(prefixer(m1[0], synchro_mot(m1[1:], m2, s)),
                 prefixer(m2[0], synchro_mot(m1, m2[1:], s)))
```

Q32.

```
let synchro_lang l1 l2  s =
    match l1, l2 with
    | [], _ -> []
    | _, [] -> []
    | m1 :: q1, m2 :: q2 ->
        union (union (synchro_lang l1 q2 s) (synchro_lang q1 l2 s))
              (synchro_mot m1 m2 s)
```

```
def synchro_lang(L1, L2, s):
    if L1 == []:
        return []
    elif L2 == []:
        return []
    else:
        m1 = L1.pop()
        m2 = L2.pop()
        return union(union(synchro_lang(L1 + [m1], L2, s),
                           synchro_lang(L1, L2 + [m2], s)),
                     synchro_mot(m1, m2, s))
```