

TP : Langages réguliers et automates

Dans tout le TP, on va beaucoup travailler sur des listes. On rappelle ici quelques fonctions utiles. Comme les listes sont immuables, ces fonctions renvoient de nouvelles listes lorsqu'elles doivent en renvoyer.

fonction	effet
<code>List.hd</code> , <code>List.tl</code>	renvoient la tête et la queue d'une liste. En général on traite les listes récursivement, et on effectue plutôt un filtrage <code>[] -> ... x::y -> ...</code> , mais elles peuvent être utiles.
<code>@</code>	concaténation de listes (opérateur infixe).
<code>List.rev</code>	retourne une liste
<code>List.map</code>	de type <code>('a -> 'b) -> 'a list -> 'b list</code> , prend une fonction et une liste, et applique la fonction sur chaque élément de la liste pour en créer une nouvelle.
<code>List.mem</code>	de type <code>'a -> 'a list -> bool</code> , prend en entrée un élément et une liste et teste la présence de l'élément dans la liste.
<code>List.assoc</code>	de type <code>'a -> ('a * 'b) list -> 'b</code> , prend en entrée un élément x et une liste contenant des couples de la forme (y, z) . Si la fonction trouve un tel couple avec $x = y$, elle renvoie z . Si elle ne trouve pas de tel couple, elle lève l'expression <code>Not_found</code> .
<code>List.sort</code>	de type <code>('a -> 'a -> int) -> 'a list -> 'a list</code> . Trie une liste via une fonction de comparaison f . x doit se trouver avant y si $f\ x\ y$ est négatif. Le tri utilisé est le tri fusion.

Rappelons comment rattraper une exception. Voici une fonction qui prend un élément x et une liste de couples (y, z) et qui cherche si l'un des couples a son premier élément qui est x , en utilisant `assoc`. Ce n'est pas forcément la manière la plus naturelle de faire mais c'est pour l'exemple !

```
# let cherche a l= try let _=assoc a l in true with Not_found -> false ;;
cherche : 'a -> ('a * 'b) list -> bool = <fun>
```

Remarquez que `let _=assoc a l in true` a le type booléen si elle s'effectue sans erreur (on ne se sert pas du résultat de `assoc a l` ici, mais il est nécessaire de « l'oublier » pour pouvoir renvoyer un booléen derrière). Si l'exception `Not_found` est levée, on la rattrape en renvoyant `false`, qui a bien le type booléen. Dans le bloc associé à `try`, on peut avoir un code beaucoup plus complexe, ce qu'il faut comprendre c'est qu'on sort du bloc dès qu'une exception est levée.

Question 1. *Fonctions utiles.* On écrit ici des fonctions utiles dans la suite.

1. Écrire `union` de type `'a list -> 'a list -> 'a list` prenant en entrée deux listes supposées sans doublons, et renvoyant une liste sans doublon contenant les éléments présents dans une des deux listes.
2. Écrire `intersection` de type `'a list -> 'a list -> 'a list` prenant en entrée deux listes supposées sans doublons, et renvoyant une liste sans doublon contenant les éléments présents dans les des deux listes.
3. Écrire `pcar` de type `'a list -> 'b list -> ('a * 'b) list` prenant en entrée deux listes, et renvoyant la liste contenant tous les couples (x, y) pour x dans la première et y dans la deuxième.
4. Écrire `tri` un algorithme de tri sur les listes. (On utilisera `List.sort` appelé!).

1 Ensembles $\mathcal{P}(L)$, $\mathcal{S}(L)$ et $\mathcal{F}_2(L)$

On travaille sur les expressions rationnelles ne contenant pas $\emptyset$. Le but est de calculer les ensembles $\mathcal{P}$, $\mathcal{S}$ et $\mathcal{F}_2$ du langage associé, contenant les premières et dernières lettres ainsi que les facteurs de taille 2 des mots du langage associé à l'expression. On représente les expressions rationnelles comme ceci (présent dans l'annexe) :

```
type 'a erat= Eps | L of 'a | Plus of 'a erat * 'a erat | Conc of 'a erat * 'a erat | Etoile of 'a erat ;;
```

Question 2. À l'aide des fonctions `union` et `pcar`, écrire une fonction `calcule_car e`, renvoyant un quadruplet p, s, f, b , constitué des premières lettres des mots de $L(e)$, de ses dernières lettres, de ses facteurs de taille 2 et d'un booléen valant `true` si et seulement si ε est dans $L(e)$. Voici par exemple les ensembles associés à l'expression $e = (a + b)c^* + a^*b$ (l'expression e de l'annexe) :

```
# calcule_car e ;;
- : char list * char list * (char * char) list * bool =
([ 'a'; 'b'], [ 'a'; 'c'; 'b'],
 [ ('c', 'c'); ('b', 'c'); ('a', 'c'); ('a', 'a'); ('a', 'b')], false)
```

Rappel : cette fonction est un peu longue. Attention, pour la concaténation de deux langages, il faut discuter suivant si ε est dans l'un ou l'autre pour calculer les premières et dernières lettres.

Question 3. Écrire une fonction `linearisation e` de type `'a erat -> int erat * 'a list` prenant en entrée une expression rationnelle, et remplaçant les caractères dans l'ordre de leur apparition par un entier (à partir de 0). La fonction doit aussi renvoyer la liste de couples (entier, lettre de `e`) permettant de reconstruire l'expression initiale.

```
# linearisation e ;;
- : int erat * (int * char) list =
(Plus (Conc (Plus (L 0, L 1), Etoile (L 2)), Conc (Etoile (L 3), L 4)),
 [(0, 'a'); (1, 'b'); (2, 'c'); (3, 'a'); (4, 'b')])
```

Indication : utiliser une fonction auxiliaire récursive `aux f p liste` où `f` est une expression rationnelle et `p` un entier, dont le but est de remplacer les lettres de `f` (ε n'est pas une lettre) par `p, p + 1, ..., p + k - 1` (si `f` possède `k` lettres) pour obtenir `f'` et de renvoyer le triplet `f', p + k, l2` où `l2` correspond à `liste` à laquelle on a ajouté les couples (i, x_i) pour $i \in [p, p + k - 1]$.

2 Automates

On souhaite travailler avec des automates. Pour englober les cas déterministes et non déterministes, et se garder la possibilité de déterminer un automate (non déterministe) en construisant l'automate des parties, on est amené à définir le type très général suivant :

```
type ('a, 'b) automate = {init: 'a list ; finaux: 'a list ; delta: (('a * 'b) * 'a) list } ;;
```

Pour $\mathcal{A} = (Q, \Sigma, I, F, \delta)$ un tel automate, `'a` est le type des états (éléments de Q), `'b` est le type des lettres (éléments de Σ). Bien que ce ne soit pas optimal¹, on a choisi d'encoder les ensemble d'états initiaux et terminaux par des listes, et de stocker l'ensemble des transitions dans une liste, qui va donc contenir des éléments de la forme $((q, x), q')$, où $q' \in \delta(q, x)$.

On considérera que l'utilisateur sait ce qu'il fait lorsqu'il considère qu'un automate est déterministe (un seul état initial, non-ambiguïté des transitions).

Question 4. En se rappelant de ce qu'on a fait en TD, écrire une fonction `gen_automates_div n` de signature `int -> (int, int) automate` prenant en entrée un entier naturel $n > 0$ et permettant de générer un automate déterministe reconnaissant les entiers divisibles par n , donnés par leur écriture en base 10 (lue classiquement de gauche à droite). On considérera que le mot vide représente 0.

```
#gen_automate_div 3 ;;
- : (int, int) automate =
{init = [0]; finaux = [0];
 delta =
 [(2, 9), 2; (2, 8), 1; (2, 7), 0; (2, 6), 2; (2, 5), 1; (2, 4), 0;
  (2, 3), 2; (2, 2), 1; (2, 1), 0; (2, 0), 2; (1, 9), 1; (1, 8), 0;
  (1, 7), 2; (1, 6), 1; (1, 5), 0; (1, 4), 2; (1, 3), 1; (1, 2), 0;
  (1, 1), 2; (1, 0), 1; (0, 9), 0; (0, 8), 2; (0, 7), 1; (0, 6), 0;
  (0, 5), 2; (0, 4), 1; (0, 3), 0; (0, 2), 2; (0, 1), 1; (0, 0), 0]}
```

Question 5. Écrire une fonction `accepte a m` de signature `('a, 'b) automate -> 'b list -> bool` prenant en entrée un automate, et un mot donné comme une liste et retournant un booléen. Si l'automate est déterministe (un seul état initial, non-ambiguïté des transitions), votre fonction doit renvoyer `true` si et seulement si le mot est reconnu par l'automate². Vous utiliserez `List.hd`, `List.assoc` et `List.mem`, et rattraperez l'exception `Not_found` de `assoc`.

1. Des structures de dictionnaire, implémentées par exemple avec des fonctions de hachage ou des arbres binaires de recherche seraient plus efficaces.

2. Le comportement si l'automate n'est pas déterministe n'a pas d'importance.

```
#accepte (gen_automate_div 2) [1; 2; 3] ;;
- : bool = false
#accepte (gen_automate_div 3) [1; 2; 3] ;;
- : bool = true
```

Question 6. Écrire une fonction `separe_entier n` prenant en entrée un entier $n \geq 0$ et retournant la liste de ses chiffres en base 10, de gauche à droite. On pourra renvoyer la liste vide pour zéro. Testez vos codes précédents !

```
#separe_entier 123 ;;
- : int list = [1; 2; 3]
```

Dans la suite, on souhaite déterminer un automate a priori non déterministe.

Question 7. Écrire une fonction `rajoute_sans_doublon x q` de signature `'a -> 'a list -> 'a list` prenant en entrée un élément et une liste, et retournant la même liste, augmentée de `x` s'il n'était pas déjà présent.

Question 8. Écrire une fonction `alphabet a` de signature `('a, 'b) automate -> 'b list` qui prend en paramètre un automate et retourne son alphabet (l'ensemble des étiquettes de ses transitions), sous forme de liste.

```
#alphabet (gen_automate_div 3) ;;
- : int list = [0; 1; 2; 3; 4; 5; 6; 7; 8; 9]
```

Question 9. Écrire une fonction `delta2 aut partie x` de signature

```
('a, 'b) automate -> 'a list -> 'b -> 'a list
```

prenant en entrée un automate `aut`, un ensemble d'états `partie` et une lettre `x`, et retournant l'ensemble des états que l'on peut atteindre depuis un état de `partie` avec la lettre `x`. On triera la partie obtenue (on a écrit un tri en début de TP) :

```
#delta2 (gen_automate_div 3) [0;1] 1 ;;
- : int list = [1; 2]
```

Question 10. En déduire une fonction `delta3 aut partie alpha` de signature

```
('a, 'b) automate -> 'a list -> 'b list -> (('a list * 'b) * 'a list) list
```

prenant en entrée un automate, un ensemble d'états $\mathcal{P}$ et un alphabet (supposé être celui de l'automate) et retournant l'ensemble des transitions de la forme $((\mathcal{P}, x), \mathcal{P}')$ pour x dans l'alphabet (et $\mathcal{P}'$ donné par la fonction précédente).

```
#delta3 (gen_automate_div 3) [0;1] [0; 1; 2; 3; 4; 5; 6; 7; 8; 9] ;;
- : ((int list * int) * int list) list =
[[([0; 1], 0), [0; 1]; ([0; 1], 1), [1; 2]; ([0; 1], 2), [0; 2];
([0; 1], 3), [0; 1]; ([0; 1], 4), [1; 2]; ([0; 1], 5), [0; 2];
([0; 1], 6), [0; 1]; ([0; 1], 7), [1; 2]; ([0; 1], 8), [0; 2];
([0; 1], 9), [0; 1]]
```

Question 11. Déduire des questions précédentes une fonction `automate_des_parties` de signature

```
('a, 'b) automate -> ('a list, 'b) automate
```

prenant en entrée un automate et retournant son automate des parties. Il y a encore pas mal de choses à écrire ! Dans le principe de l'exploration, on s'inspirera d'un parcours de graphe, en particulier on ne calculera que les transitions issues d'états accessibles.

Dans la suite, on va tester tout ça !

Question 12. Écrire une fonction `automate_miroir` de signature `('a, 'b) automate -> ('a, 'b) automate` prenant en entrée un automate et renvoyant un automate reconnaissant le langage transposé (il suffit d'échanger états initiaux et terminaux et de changer le sens des transitions).

Question 13. En déduire une fonction `gen_automate_div_inv n` de signature `int -> (int list, int) automate` prenant en entrée un entier n et retournant un automate *déterministe* capable de reconnaître l'ensemble des entiers divisibles par n lus de droite à gauche.

```
#gen_automate_div_inv 2 ;;
- : (int list, int) automate =
{init = [[0]]; finaux = [[0]; [0; 1]];
 delta = [[([0; 1], 9), [0; 1]; ([0; 1], 8), [0; 1]; ([0; 1], 7), [0; 1]; ([0; 1], 6), [0; 1];
([0; 1], 5), [0; 1]; ([0; 1], 4), [0; 1]; ([0; 1], 3), [0; 1]; ([0; 1], 2), [0; 1]; ([0; 1], 1), [0; 1];
([0; 1], 0), [0; 1]; ([], 9), [], ([], 8), [], ([], 7), [], ([], 6), [], ([], 5), [], ([], 4), [];
([], 3), []; ([], 2), []; ([], 1), []; ([], 0), []; ([0], 9), []; ([0], 8), [0; 1]; ([0], 7), [];
([0], 6), [0; 1]; ([0], 5), []; ([0], 4), [0; 1]; ([0], 3), []; ([0], 2), [0; 1]; ([0], 1), [];
([0], 0), [0; 1]]}
```

Question 14. On rappelle que `rev` permet de retourner une liste. Écrire une fonction `separe_entier_rev n` qui renvoie la liste des chiffres de n lu de droite à gauche, et tester vos fonctions!

Question 15. Bonus : écrire une fonction `emondage` permettant d'émonder un automate déterministe.

3 Automate associé à une expression rationnelle

Question 16. Écrire une fonction `automate_lin p s f` permettant, étant donné deux listes d'entiers et une liste de couples d'entiers, de créer un automate déterministe local standard reconnaissant le langage local associé aux ensembles. Elle doit avoir pour signature `int list -> int list -> (int * int) list -> (int, int) automate`. On supposera les entiers naturels, et on pourra utiliser -1 comme état initial.

```
#e2 (* obtenue par linéarisation de e vue plus haut *) ;;
- : int erat =
Plus (Conc (Plus (L 0, L 1), Etoile (L 2)), Conc (Etoile (L 3), L 4))
#let p,s,f=calculer_p e2,calculer_s e2, calculer_f e2 in automate_lin p s f ;;
- : (int, int) automate =
{init = [-1]; finaux = [0; 1; 2; 4];
 delta =
[(-1, 4), 4; (-1, 3), 3; (-1, 1), 1; (-1, 0), 0; (3, 4), 4; (3, 3), 3;
(0, 2), 2; (1, 2), 2; (2, 2), 2]}
```

Question 17. Écrire une fonction `remplacer : ('a, 'b) automate -> ('b * 'c) list -> ('a, 'c) automate` remplaçant les transitions étiquetées par x de l'automate initial par des transitions étiquetées par y , avec (x,y) se trouvant dans la liste passée en entrée (sans surprise, on utilisera `assoc`, et on pourra utiliser `map` en définissant d'abord la fonction qui va bien).

Question 18. Implémenter maintenant l'algorithme de Berry-Sethi permettant de calculer l'automate de Glushkov associé à une expression rationnelle (sans ε ni $\emptyset$), sous la forme d'une fonction `berry_sethi` de signature `'a erat -> (int list, 'a) automate`.

```
#let e=Plus (Conc (Plus (L 'a', L 'b'), Etoile (L 'c')), Conc (Etoile (L 'a'), L 'b')) ;;
#berry_sethi e ;;
- : (int list, char) automate =
{init = [[-1]]; finaux = [[0; 3]; [2]; [4]; [1; 4]]; delta =
[[([1; 4], 'c'), [2]; ([1; 4], 'a'), []; ([1; 4], 'b'), []; ([4], 'c'), []; ([4], 'a'), [];
([4], 'b'), []; ([3], 'c'), []; ([3], 'a'), [3]; ([3], 'b'), [4]; ([2], 'c'), [2]; ([2], 'a'), [];
([2], 'b'), []; ([0; 3], 'c'), [2]; ([0; 3], 'a'), [3]; ([0; 3], 'b'), [4]; ([], 'c'), [];
([], 'a'), []; ([], 'b'), []; ([-1], 'c'), []; ([-1], 'a'), [0; 3]; ([-1], 'b'), [1; 4]]}
```

Si vous avez écrit une fonction d'émonder, vous pouvez l'utiliser :

```
#emondage (berry_sethi e) ;;
- : (int list, char) automate =
{init = [[-1]]; finaux = [[0; 3]; [2]; [4]; [1; 4]]; delta =
[[([-1], 'b'), [1; 4]; ([-1], 'a'), [0; 3]; ([0; 3], 'b'), [4]; ([0; 3], 'c'), [2];
([2], 'c'), [2]; ([3], 'b'), [4]; ([3], 'a'), [3]; ([1; 4], 'c'), [2]]}
```