Interrogation écrite 4

INF 201 — IMA1— 04/04/2022 — 30 minutes

Exercice 1. (/4) Dans cet exercice il est **interdit** d'écrire des fonctions récursives, il faudra utiliser à la place les schémas d'ordre supérieur du module `List`. On souhaite générer à partir d'une liste la liste de ses sous-listes. Par exemple pour la liste `[1;2;3]` la liste des sous-listes est `[[1; 2; 3]; [2; 3]; [1; 3]; [3]; [1; 2]; [2]; [1]; []]`.

Question 1. Écrire une fonction `ajoute` qui ajoute un élément à toutes les listes d'une liste de liste en utilisant `List.map`

```
List.map = ('a -> 'b) -> 'a list -> 'b list
```

EXEMPLE: `ajoute [[1];[2]] 3 = [[1;3];[2;3]]`

```
let ajoute l e = List.map (fun x -> x@[e]) l;;
```

Question 2. En déduire une fonction `sous_listes` en utilisant `List.fold_left`

```
List.fold_left = ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
```

```
let sous_listes = List.fold_left (fun acc e -> (ajoute acc e)@acc) [[]];;
```

Exercice 2. (/8) Le but de cet exercice est d'implémenter le tri **fusion**. Ce tri **divise** la liste en deux parties a peu près égales qui sont triées chacune de leurs côtés puis **fusionnées**. On n'utilisera aucune fonction du module `List`. Voir figure.

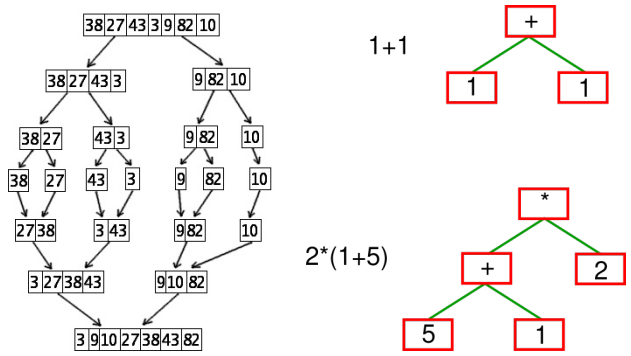


Figure 1. À gauche un exemple de `tri_fusion` sur des entiers (wikipedia.org). À droite représentations en arbre d'expressions algébriques (exercice 3): `N(X(1),Plus,X(1))` et `N(N(X(5),Plus,X(1)),Fois,X(2))`.

Question 3. Implémenter une fonction `fusionner` qui prend en entrée deux listes triées et les fusionne en **conservant l'ordre**, c'est-à-dire que la liste finale doit être triée également.

EXEMPLE: `fusionner [1;3;5];[2;4] = [1;2;3;4;5]`

Beaucoup de problèmes de compréhension, la fonction fusion ne doit pas concaténer puis trier la liste ! Elle doit fusionner en conservant l'ordre c'est à dire créer une liste qui est constitué par le min de l1 et l2.

```
let rec fusionner l1 l2 = match l1,l2 with
  |[],[] -> []
  |_,[] -> l1
  |[],_ -> l2
  |t1::q1,t2::q2 -> if t1 < t2 then t1::(fusionner q1 l2) else t2::(fusionner l1 q2);;
```

Question 4. Implémenter une fonction `diviser` qui sépare une liste en deux listes de longueur a peu près égales.

EXEMPLE: `diviser [1;2;3;4;5] = ([1;3;5],[2;4])`

Ici aussi beaucoup de confusions, pour une liste du type t1::t2::q on diviser q en l1,l2 puis on ajoute t1 à l1 et t2 à l2 ! Les cas terminaux ne présentent pas beaucoup de difficultés. **On ne peut pas concaténer des couples de listes !**

```
let rec diviser l = match l with
  |[] -> [],[]
  |[t] -> [t],[]
  |t1::t2::q -> let l1,l2 = diviser q in
                t1::l1,t2::l2;;
```

Question 5. Déduire des deux fonctions précédentes une fonction `tri_fusion` qui trie une liste.

Enfin ici aussi, alors que les briques étaient en place, beaucoup de confusions …

```
let rec tri_fusion l = match l with
  |[] -> []
  |[t] -> [t]
  |_ -> let l1,l2 = diviser l in fusionner (tri_fusion l1) (tri_fusion l2);;
```

Question 6. On admet pour cette question que les fonctions `diviser` et `fusionner` terminent. Montrer en spécifiant une mesure que la fonction `tri_fusion` termine.

On pose `mesure l = |l|` $\in\mathbb{N}$. `|l|=|l1@l2|=|l1|+|l2|` donc `|l|>|l1|` et `|l|>|l2|` l'appel récursif à tri_fusion est donc décroissant et s'effectue au sein de fonctions qui elles-même terminent. La mesure est décroissante et minorée donc convergente.

Exercice 3. (/8) En **notation polonaise inverse**, *les opérandes précèdent les opérateurs.* Par exemple au lieu d'écrire $3+4$ on écrira $3\ \ 4\ \ +$, ou bien encore au lieu d'écrire $(2+2)\times(3+4)$ on écrira $2\ \ 2\ \ +3\ \ 4\ \ +\ \ \times$. Cette façon d'écrire les expressions algébriques présente un énorme avantage: il n'y a plus d'ambiguité et les parenthèses deviennent donc inutile ! *On ne traitera dans cet exercice que l'addition et la multiplication.*

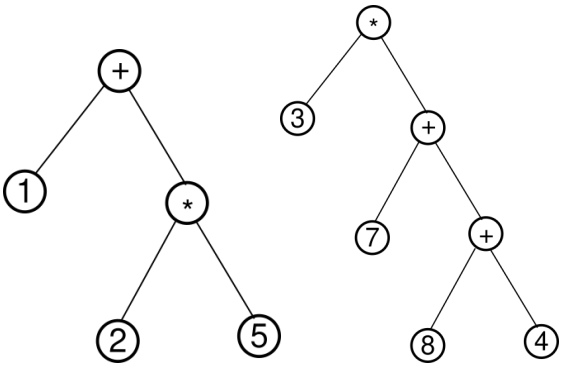
On introduit un type d'arbre pour représenter une expression algérique, ce type est différent du type vu en cours, la notion d'arbre vide n'existe pas ici, logique car il n'y a pas d'expression vide !

```
type operateur = Plus | Fois;;
type 'a arbre_expr =  N of ( 'a arbre_expr) * operateur * ('a arbre_expr) | X of int;;
```

Question 7. Exprimer $1+2\times5$ et $3\times(7+8+4)$ en NPI, en OCaml à l'aide du type `arbre_expr` et les dessiner.

$1\ \ 2\ \ 5\ \ \times+\ \ \rightarrow$ `N(X(1),Plus,N(X(2),Fois,X(5)))`

$3\ \ 7\ \ 8\ \ +4+\times\ \ \rightarrow$ `N(X(3),Fois,N(X(7),Plus,N(X(8),X(4))))`



Question 8. Écrire une fonction `eval` qui à partir d'un `arbre_expr` renvoie un entier qui représente le valeur du calcul.

Bien réussie dans l'ensemble.

```
let rec eval a = match a with
  |X(n) -> n
  |N(Plus, a1, a2) -> (eval a1) + (eval a2)
  |N(Fois, a1, a2) -> (eval a1) * (eval a2)
;;
```

Question 9. Écrire une fonction `arbre_vers_npi` qui transforme un arbre en une expression algébrique NPI. On pourra recourir à l'opérateur `@`. La fonction renverra une **liste** de `char`, on supposera l'existence d'une fonction `nombre_vers_car` qui transforme un entier en son char associé. Exemple: `nombre_vers_car 9 = '9'`.

```
let rec arbre_vers_npi a = match a with
  |X(n) -> [(nombre_vers_car n)]
  |N(Plus, a1, a2) -> (arbre_vers_npi a1)@(arbre_vers_npi a2)@['+']
  |N(Fois, a1, a2) -> (arbre_vers_npi a1)@(arbre_vers_npi a2)@['*']
;;
```