ENSEIRB-Matmeca
2024-2025

1ère année - Filière Info
PG104

TD n°7 - 1ère classe : map et reduce

▷ Les listes chaînées ne font pas partie de la bibliothèque standard d'Ecmascript. Afin de pouvoir réaliser les algorithmes récursifs de cette feuille, les fonctions suivantes permettent de construire et déconstruire des listes :

```
import { anyToString } from '#src/utils/printers.js';
import { head, tail, nil, cons, isEmpty } from '#src/utils/list.functional.api.js';

let aList = cons(1, cons(2, nil));
console.log(anyToString(aList));          // → (|1, 2|)
console.logA(aList);                      // → (|1, 2|)

let anotherList = cons(3, tail(aList));
console.logA(anotherList)                 // → (|3, 2|)
```

▷ La bibliothèque `list.functional.api.js` contient aussi des fonctions pour construire des listes (`listIota`) ainsi que les transformer en chaînes de caractères (`listToString`).

Exercice 1: Listes et Maps

1. Écrire une fonction récursive `listMap` prenant en argument une fonction `f` et une liste `l`, et renvoyant la liste des images par `f` des éléments de `l`.

   Exemple :

   ```
   const aList = cons(1, cons(-1, cons(2, nil)));
   listMap((x) ⇒ x+1, aList); // → (|2,0,3|)
   ```

2. Écrire une fonction `list2Squares` utilisant la fonction précédente prenant en argument une liste et renvoyant la liste des carrés des éléments de cette liste.

Exercice 2: Listes et Pliages

Lorsqu'il s'agit d'effectuer un parcours de tous les éléments d'une liste de manière fonctionnelle, pour les agréger dans un résultat, on parle de *pliage* (ou *fold*). Les noms des fonctions associées varient selon les langages de programmation : `reduce` en Javascript ou Java, `fold` en OCaml, Haskell, ou Scala ... Dans cet exercice, on se propose d'écrire ces fonctions sur les listes en Javascript.

Les pliages peuvent s'écrire en parcourant les éléments de la liste de gauche à droite, (*fold left*) ou de droite à gauche (*fold right*). Les élément de la liste sont pliés à l'aide d'une fonction agrégeante qui prend deux paramètres : l'état courant du pliage (aussi appelé *accumulateur*) et l'élément de la liste parcourue.

▷ Pour simplifier, on impose que l'accumulateur soit *systématiquement* le premier argument de la fonction agrégeante. Il s'agit de la convention utilisée en Ecmascript.

Les calculs effectués par les pliages de la fonction `f` sur la liste `[x1,... xn]` sont les suivants :

Fold Left

`f( ... f(f(init, x1), x2) ..., xn)`

Fold Right

`f(f(... (f(init, xn) ...), x2), x1)`

1. Définir une fonction `listFoldR` permettant d'appliquer récursivement une fonction de deux variables aux éléments d'une liste, en suivant le schéma de pliage à *droite*.
2. Définir une fonction `listFoldL` permettant d'appliquer récursivement une fonction de deux variables aux éléments d'une liste, en suivant le schéma de pliage à *gauche*.

Exercice 3: Pour quelques pliages de plus ...

1. Utiliser un pliage pour écrire une fonction `prodIterate` qui, étant donnée une fonction g et une liste chaînée contenant les valeurs $\{x_i\}$, calcule le produit $\prod_i g(x_i)$.

   Exemple :

```
let aNumberList1 = cons(4, cons(9, cons(25, nil)));
prodIterate(sqrt, aNumberList1); // → 30 = sqrt(4) * sqrt(9) * sqrt(25)
```

2. Utiliser un pliage pour écrire une fonction `listReverse` qui, étant donnée une liste chaînée l, produit une liste avec les mêmes éléments, mais dans l'ordre inversé.
3. Utiliser un pliage pour définir la fonction `listMapFold` qui effectue le même calcul que `listMap`, i.e prend en argument une fonction `g` et une liste `l`, et renvoie la liste des images par `g` des éléments de `l`.

   Exemple :

```
let aNumberList3 = cons(4, cons(9, cons(25, nil)));
listMapFold(sqrt, aNumberList3); // → (|2, 3, 4|)
```

4. Utiliser un pliage pour définir la fonction `listAppendFold` qui prend en paramètre deux listes chaînées l_1 et l_2 et renvoie la concaténation des deux.
5. Utiliser un pliage pour écrire une fonction `howMany` retournant le nombre d'éléments d'une liste `l` vérifiant le prédicat `pred`.

   Exemple :

```
let aNumberList2 = cons(1, cons(8, cons(2, cons(7, nil))));
howMany((x) ⇒ x > 5, aNumberList2); // → 2
```

Exercice 4: Vecteurs et matrices

Dans cet exercice, on se propose d'expérimenter avec les matrices et les vecteurs en les représentant à l'aide de listes. Un vecteur peut ainsi être représenté par la liste de ses n éléments :

$[a_1\ a_2\ \dots\ a_n]$

```
cons(a1, cons(a2, ... cons(an, nil)))
```

... et une matrice peut être représentée par la liste dont les éléments (sur n lignes et n colonnes) :

$$\begin{bmatrix} [a_{1,1}\ a_{1,2}\ \dots\ a_{1,n}] \\ \dots \\ [a_{n,1}\ a_{n,2}\ \dots\ a_{n,n}] \end{bmatrix}$$

```
cons(     cons(a1,1, cons(a1,2, ... cons(a1,n, nil))),
cons(     cons(a2,1, cons(a2,2, ... cons(a2,n, nil))),
...
cons(     cons(an,1, cons(an,2, ... cons(an,n, nil))),
nil))...)
```

```
// An example of vector
const aVector = cons(1, cons(2, cons(3, nil)));            // (|1,2,3|)
// An example of matrix
const aLine1 = cons(1, cons(2, cons(3,nil)));              // (|(|1,2,3|),
const aLine2 = cons(4, cons(5, cons(6,nil)));              //   (|4,5,6|),
const aLine3 = cons(7, cons(8, cons(9,nil)));              //   (|7,8,9|)|)
const aMatrix = cons(aLine1, cons(aLine2, cons(aLine3, nil)));

matrixDisp(aMatrix);
```

Les sources contiennent une fonction `matrixDisp` qui utilise les fonctions `listFoldL` et `listMap` pour permettre d'afficher les matrices. Pour rester cohérent dans les fonctions suivantes, utiliser aussi de préférence les fonctions `listMap`, `listFoldR` et `listFoldL` définies dans l'exercice précédent.

1. Écrire une fonction `listScalarProduct` qui retourne le produit scalaire de deux vecteurs.
2. Écrire une fonction `listMatVect` qui retourne le produit d'un vecteur par une matrice.
3. (*Bonus*) Écrire une fonction `listTranspose` qui retourne la transposée d'une matrice.
4. (*Bonus*) Écrire une fonction `listMatMat` qui retourne le produit de deux matrices.