TD n°5 - Types inductifs : listes et arbres

▷ 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`).

▷ Les arbres ne font pas non plus partie de la bibliothèque standard d'Ecmascript. Pour pouvoir réaliser les exercices de cette feuille, les fonctions suivantes permettent de construire et déconstruire des arbres :

```
import { node, val, children } from '#src/utils/tree.functional.api.js';

let aTree = node(1, nil); // Simple tree with one node
console.logA(aTree);

// More complex tree with three nodes
let anotherTree = node('A', cons(node('B', nil), cons(node('C', nil), nil)));
console.logA(anotherTree);
```

Exercice 1: Grammaires

Dans cet exercice, on s'intéresse au lien entre les définitions des types de données et les prédicats permettant de les reconnaître effectivement dans le code. Par exemple, la fonction suivante permet de vérifier si un objet (en fait n'importe quelle valeur) possède un attribut donné :

```
import { hasKey } from '#src/utils/object.js';

hasKey({ name: "Alf" }, "name"); // → true
hasKey({ name: "Alf" }, "cat");  // → false
```

Pour rappel, une liste est une structure de données définie de manière inductive comme suit :

— la liste vide `nil` est une liste ;

— si e est une valeur et l est une liste, alors `cons(e, l)` est une liste.

1. Écrire un prédicat `isList` qui prend un paramètre l et renvoie un booléen valant `true` si et seulement ce paramètre est bien une liste.

Un arbre est une structure de données définie de manière inductive, et possédant deux attributs :

— une valeur val ;

— et une liste $children$ ne contenant que des arbres.

2. Écrire un prédicat `isTree` qui prend un paramètre t et renvoie un booléen valant `true` si et seulement ce paramètre est bien un arbre.

Exercice 2: Listes de listes

Dans cet exercice, on travaille avec des listes pouvant elle-même contenir des listes, et cela à une profondeur quelconque :

```
let aList1 = cons(1,
                  cons(listIota(5,9),
                       cons(4, nil)));      // → (|(|1|), (|5, 6, 7, 8|), 4|)
let aList2 = cons(cons(1, nil),
                  cons(cons(2, cons(3, cons(cons(4, nil), nil))),
                           nil));           // → (|(|1|), (|2, 3, (|4|)|)|)
console.log(`Disp aList1 : ${anyToString(aList1)}`);
console.log(`Disp aList2 : ${anyToString(aList2)}`);
```

1. Écrire une fonction `listFlatten` qui, étant donnée une liste l renvoie la liste aplatie des toutes les valeurs contenues à l'intérieur de l (à l'exception des listes vides `nil`).

   Exemple :

   ```
   listFlatten(aList1);          // → (|1, 5, 6, 7, 8, 4|)
   listFlatten(aList2);          // → (|1, 2, 3, 4|)
   ```

2. Écrire une version de la fonction `listFlattenTr` qui soit récursive terminale.

3. (*Bonus*) Écrire une fonction `listReverse` qui, étant donnée une liste l, renverse l'ordre des éléments de toutes les listes à l'intérieur de l.

   Exemple :

```
let aList = cons('a',
            cons('b',
            cons(cons('c',
                 cons('d', nil)),
            cons('e',
            cons('f', nil))))); // → (|a, b, (|c, d|), e, f|)
listReverse(aList);             // → (|f, e, (|d, c|), b, a|)
```

4. (*Bonus*) Écrire une fonction `listCount` qui prend en argument une valeur et une liste et calcule le nombre d'occurrences de cette valeur dans la liste.

   Exemple :

```
let aList = cons('a',
            cons(cons('b',
                 cons('a',
                 cons(cons('c',
                      cons('a', nil)), nil))),
            cons('d',
            cons('a', nil)))); // → (|a, (|b, a, (|c, a|)), d, a|)
countList('a', aList);         // → 4
```

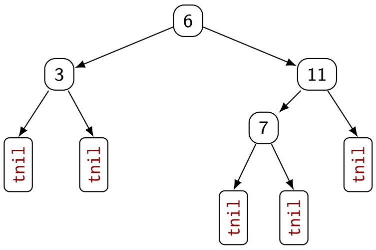
Exercice 3: Arbre binaire de recherche

Un arbre binaire de recherche (*binary search tree*) est un arbre binaire contenant des données ordonnées (par exemple des entiers). Il possède en plus la propriété suivante : pour tout nœud de l'arbre avec une valeur v, les valeurs des nœuds du sous-arbre gauche sont inférieures ou égales à v, et celles des nœuds du sous-arbre droit sont strictement supérieures à v.

Les arbres binaires sont soit vides, soit ont deux enfants distingués (`leftChild` et `rightChild`). Pour assurer cette propriété, nous allons définir un arbre binaire vide `tnil`. De plus, nous ajoutons systématiquement des noeuds avec la valeur `tnil` aux feuilles de nos arbres non vides. Partons du code suivant :

```
// A special tree representing the empty binary tree
const tnil = node(null, nil);
// A function to test if a tree is the empty binary tree
function binaryTreeIsEmpty(t) { return t === tnil; }
```

Voici un exemple d'arbre binaire :




```
let aTree = binaryNode(6,
                       binaryLeaf(3),
                       binaryNode(11,
                                  binaryLeaf(7),
                                  tnil));
```

Pour vous aider à afficher les arbres, une fonction `binaryTreeDisp` est fournie dans les sources du TD, mais la fonction `anyToString` fonctionne aussi. Il reste donc à adapter l'API actuelle sur les arbres.

1. Écrire les fonctions de l'API manquantes, *en réutilisant* les fonctions de l'API sur les listes que sont `node`, `cons` et `nil`, `head`, `tail` et `children`.
   - `binaryNode(v, l, r)` qui renvoie un arbre binaire dont la valeur au sommet est `v` et les fils gauches et droits sont `l` et `r` ;
   - `binaryLeaf(v)` qui réutilise la fonction précédente pour construire une feuille ;
   - `leftChild(t)` et `rightChild(t)` qui renvoient les sous-arbres gauche et droit de l'arbre binaire non vide `t`.
2. Écrire une fonction `binaryTreeInsert` qui, étant donné un arbre binaire de recherche potentiellement vide t et une valeur v, construit l'arbre binaire de recherche t auquel on a ajouté une feuille v.

Exercice 4: Listes et ordre supérieur

Dans cet exercice, on s'intéresse à l'écriture de fonctions utilisant les méthodes d'ordre supérieur suivantes : `map`, `reduce` et `filter`.

> ▷ Pour ne pas avoir à recoder ces fonctions sur les listes (pour l'instant), on se propose de travailler ici avec les tableaux Ecmascript.

Pour rappel, la documentation fournit :

— "The `map()` method creates a new array populated with the results of calling a provided function on every element in the calling array." The function can also take the index of the element.

```
const array1 = [1, 4, 9, 16];
const map1 = array1.map((elem,ind) => [elem * 2, ind]);
console.log(map1); // → [[2, 0], [8, 1], [18, 2], [32, 3]]
```

— "The `reduce()` method executes a reducer function (that you provide) on each element of the array, resulting in single output value."

```
const array1 = [1, 2, 3, 4];
const reducer = (accumulator, currentValue) => accumulator + currentValue;

console.log(array1.reduce(reducer));    // → 0 + 1 + 2 + 3 + 4 = 10
console.log(array1.reduce(reducer, 5)); // → 5 + 1 + 2 + 3 + 4 = 15
```

— "The `filter()` method creates a new array with all elements that pass the test implemented by the provided function."

```
const words = ['spray', 'limit', 'elite', 'exuberant', 'destruction', 'present'];

const result = words.filter(word => word.length < 6); // → [ 'spray', 'limit', 'elite' ]
```

1. Écrire une fonction `scalarProduct` qui, étant données deux tableaux (de même taille) passés en paramètres, retourne le produit scalaire des vecteurs qu'ils représentent.
2. Écrire la fonction `divisors` qui étant donné un entier n produit la liste de tous ses diviseurs. On pourra s'aider du code suivant qui construit le tableau des entiers de 1 à `N` : `Array.from({length: N}, (_, index) => index + 1);`.