TD n°9 - Techniques : contrôle de l'évaluation

> ⚠ Les exercices de cette feuille utilisent des structures de données différentes. Faire bien attention de réaliser chaque exercice dans un fichier différent.

Exercice 1: Tic, Tac, Toe

Le jeu de morpion (en anglais Tic-Tac-Toe, cf. `https://en.wikipedia.org/wiki/Tic-tac-toe`) est un exemple de jeu dans lequel l'ensemble des positions possibles est de taille tout à fait commensurable. L'ensemble des configurations différentes de jeu (appelées par la suite *états* ou *states* dans le code) est de taille 765, et le nombre total de parties est de 26830 si on les compte à rotation et symétrie près.

X		O
X	O	
X		

Proposons de représenter un état du jeu par un tableau de taille 9, listant les cases du plateau ligne par ligne. Ainsi, le plateau précédent peut être représenté par le tableau suivant :

```
[ 'x', '␣', 'o', 'x', 'o', '␣', 'x', '␣', '␣' ]
```

Le premier joueur à jouer est celui qui dessine une croix. Pour simplifier, les sources contiennent une fonction `stateToString` qui permet d'afficher un plateau de jeu.

```
let aState = [ 'x', '␣', 'o', 'x', 'o', '␣', 'x', '␣', '␣' ];
console.log(`A␣state␣:\n${stateToString(aState)}`);
```

```
-----
|x o|
|xo |
|x  |
-----
```

1. Décrire l'état initial de jeu `initialState`.
2. Écrire une fonction `stateEmptySlots` qui, étant donné un état du jeu, renvoie le tableau des indices des cases vides.

   Exemple :

```
stateEmptySlots([ 'x', '␣', 'o', 'x', 'o', '␣', 'x', '␣', '␣' ]);
// → [ 1, 5, 7, 8 ]
```

3. Écrire une fonction `stateNexts` qui, étant donnée un état et un joueur `'x'` ou `'o'`, construit le tableau contenant les états accessibles (i.e jouables) à partir de cet état.

   Exemple :

```
stateNexts(['x', 'o', 'x', '␣', '␣', '␣', 'x', 'o', 'x'], 'x');
// → [ [ 'x', 'o', 'x', 'x', ' ', ' ', 'x', 'o', 'x' ],
//     [ 'x', 'o', 'x', ' ', 'x', ' ', 'x', 'o', 'x' ],
//     [ 'x', 'o', 'x', ' ', ' ', 'x', 'x', 'o', 'x' ] ]
```

Posons nous maintenant la question de construire l'arbre de toutes les parties du jeu de tic-tac-toe. Il s'agit d'un arbre dont la racine est l'état initial `initialState`, et pour un sommet donné de cet arbre, les enfants sont les états que l'on peut atteindre par un coup valide du joueur courant.

> ▷ Pour simplifier cet exercice, on a remplacé les arbres fonctionnels des feuilles précédentes (pour lesquels les enfants sont une *liste* d'arbres) par une autre implémentation dans laquelle les enfants sont un *tableau* d'arbres. Les fonctions de manipulation (`node`, `val`, `children` ...) ont gardé les mêmes noms, mais gèrent maintenant des tableaux.

4. Écrire une fonction `stateDepth` qui, à partir d'un état donné `aState`, d'un joueur `aPlayer` et d'une profondeur `aDepth`, construit l'arbre des parties possibles de profondeur données pour ce joueur à partir de cet état (c'est à son tour de jouer).

   Remarque : le code contient une fonction `otherPlayer`.

Exercice 2: Contrôle de l'évaluation : arbre d'entiers

Les constructions précédentes permettent à elles deux de construire l'arbre de toutes les parties possibles à partir d'un état donné du jeu. Néanmoins, la construction de cet arbre pose plusieurs problèmes. En premier, il est nécessaire de construire l'arbre entier de toutes les parties d'un seul coup. Dans cet exercice, on se propose de construire une structure qui peut se déplier à la demande en faisant de l'*évaluation paresseuse*.

Nous allons utiliser les valeurs paresseuses de la manière suivante :

```
import { evaluated, freeze, isFrozen, thaw, value } from '#src/utils/lazy.js';

const aFrozenValue = freeze(() => 7);   // Create a lazy frozen value
isFrozen(aFrozenValue);                 // → true                    (it is indeed frozen)
thaw(aFrozenValue);                     // Thaw it                  (this is a side-effect)
isFrozen(aFrozenValue);                 // → false                  (it has been thawed)
value(aFrozenValue);                    // → 7
const anEvaluatedValue = evaluated(5);  // Create a lazy value that is in fact evaluated
isFrozen(anEvaluatedValue);             // → false                  (it has never been frozen)
```

Et nous allons remplacer les arbres utilisés jusqu'à maintenant par des arbres acceptant de congeler leur descendance :

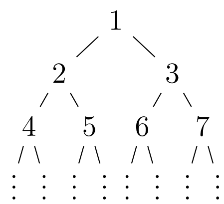
```
import { leaf, node, treeThawAtDepth, treeDisp } from '#src/utils/tree.lazy.api.js';

let normalTree = node("root",
                      evaluated([ leaf("unique son") ]));
console.logA(normalTree);

let frozenTree = node("root",
                       freeze(() => [ leaf("1st son"),
                                      leaf("2nd son") ]));
console.logA(frozenTree);
```

1. Quel est l'affichage obtenu lorsque l'on regarde les valeurs à l'intérieur de `frozenTree` ? Comment peut-on faire pour "déplier" l'arbre ?

Pour la suite, on fournit une fonction `nodeThaw` qui décongèle un seul sommet de l'arbre, c'est-à-dire qui n'applique qu'une seule fois la fonction `thaw` à la liste de ses enfants. On fournit aussi une fonction `treeThawAtDepth` qui applique la décongélation récursivement à tous les sommets jusqu'à une profondeur donnée. (cette fonction n'est utilisée ici que pour vérifier les valeurs à l'intérieur d'un arbre congelé).

Utilisons le mécanisme du contrôle de l'évaluation pour construire des arbres qui peuvent se déplier à la demande. Considérons l'arbre *infini* des entiers construit de la manière suivante :

- La racine a pour valeur 1 ;
- Un noeud de valeur v a deux fils de valeurs $2 * v$ et $2 * v + 1$.



2. Écrire une fonction *récursive* `makeFiniteIntegerTree` qui prend une valeur `val` à la racine et une profondeur `depth`, et qui construit l'arbre des entiers avec racine `val` jusqu'à la profondeur donnée, complètement évalué.
3. Recopier la fonction précédente en une nouvelle fonction `makeInfiniteIntegerTree`, en prenant bien soin de renommer les appels récursifs.
   Faire que cette nouvelle fonction *congèle* la liste des fils juste avant de les calculer.
   Utiliser la fonction `treeThawAtDepth` pour décongeler cet arbre à une profondeur donnée, et vérifier son contenu.
4. A quoi sert le paramètre `depth` dans la fonction `makeInfiniteIntegerTree` ? Le supprimer.

Exercice 3: Contrôle de l'évaluation : Tic, Tac, Toe

Les constructions de l'exercice précédent sont académiques. Mettons les en application pour construire l'arbre des parties du jeu de morpion, avec contrôle de l'évaluation pour n'évaluer que les noeuds de l'arbre qu'à la demande.

> ▷ Pour simplifier, les sources de cet exercice contiennent une fonction `stateWinner` qui renvoie les gagnants d'un état donné, et une fonction `stateIsFinal` qui calcule si une partie est terminée.

1. Écrire une fonction *récursive* `makeGameTree` qui prend en paramètre un état `state` et un joueur `player`, et qui construise l'arbre congelé du jeu de morpion à partir de cet état. On pensera à utiliser la fonction `stateNexts`.

   Remarque : il est possible de repartir de la fonction `stateDepth` de l'exercice précédent.

2. Écrire une fonction `playRandomGame` qui prend un arbre des parties (potentiellement congelé), et joue une partie dans cet arbre en le décongelant jusqu'à obtenir une fin de partie (on pourra utiliser la fonction `stateIsFinal`).