ENSEIRB-Matmeca
2024-2025

1ère année - Filière Info
PG104

TD n°10 - Modularité et Tests

Exercice 1: 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 :

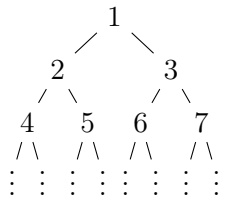
```
import { leaf, node, treeThawAtDepth, treeDisp } from '#src/utils/tree.lazy.api.js';

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 2: 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`).

Exercice 3: Ça tombe pile

Dans cet exercice, il s'agit de découper un code en différents modules qui interagissent les uns avec les autres. Le code en question utilise la réalisation des listes utilisée dans les exercices précédents pour implémenter un type abstrait de données de pile (cf. `https://en.wikipedia.org/wiki/Stack_(abstract_data_type)`). Pour fixer les choses, on propose l'interface de programmation suivante pour ces piles, nommées par la suite `stack` :

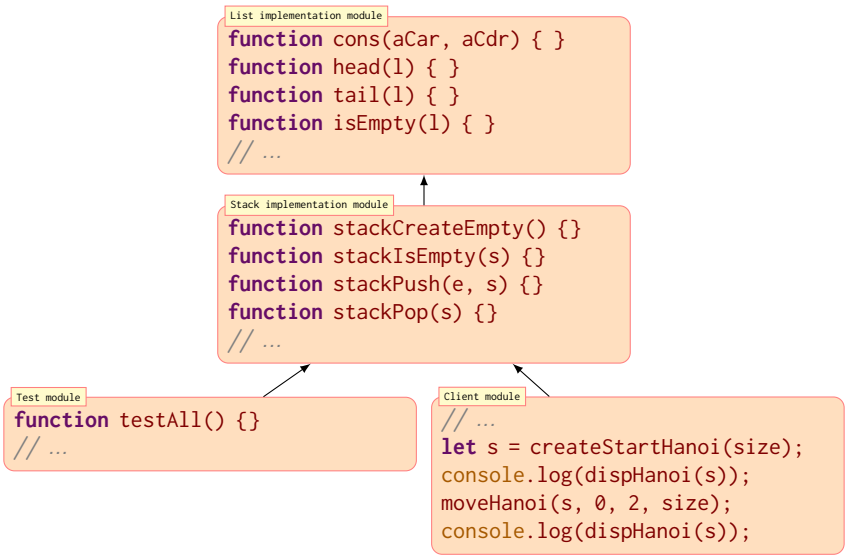
```
// Returns an empty stack
function stackCreateEmpty() {}
// Checks that the stack `s` is empty
function stackIsEmpty(s) {}
// Returns a new stack where the element `e` has been pushed on top of the stack `s`
function stackPush(e, s) {}
// Returns a new stack where the top of the stack `s` has been popped
// Throws an error if `s` is empty
function stackPop(s) {}
// Returns the element at the top of the stack `s`
// Throws an error if `s` is empty
function stackPeek(s) {}
// Returns a string representing the content of the stack `s`
function stackDisplay(s) {}
```

Le code fourni dans les sources contient une implémentation du jeu des tours de Hanoi (cf. https://en.wikipedia.org/wiki/Tower_of_Hanoi), dans laquelle on utilise trois piles pour déplacer des ensembles de disques. Il n'est pas nécessaire de comprendre le code en question pour faire cet exercice, simplement de voir qu'il utilise une réalisation du TAD `stack`.

1. A partir des sources assemblées dans un unique fichier, compléter le code manquant pour terminer la réalisation du type `stack`.

   Remarque : les implémentations des fonctions du TAD sont très simples, n'ont pas besoin de récursivité, et devraient tenir en une seule et courte ligne, sauf pour les fonction renvoyant des erreurs qui devraient tenir en 4 lignes.

Il est maintenant temps de découper le code de manière modulaire. Le découpage suivant est proposé :



Le système de modules utilisé dans cet exercice sera celui des Modules Ecmascript (ceux utilisant les mots-clés `import` et `export`). Dans la suite, nous allons découper découper le code en modules, *i.e* en fichiers séparés :

2. Écrire un module `list.impl.js` représentant le type abstrait de données `list`.

   Faire le nécessaire pour que ce module *exporte* les fonctions précédentes en utilisant le mot-clé `export`.

3. Écrire un module `stack.impl.js` implémentant le type abstrait de données `stack`. Ce module doit utiliser les fonctions du module précédent à l'aide d'un appel à `import`.

4. Relier le module précédent à un module `stack.client.js` qui contient le code des tours de Hanoi. Vérifier que vos implémentations sont bien raccordées en exécutant le code des tours de Hanoi.

5. Lancer la commande suivante pour vérifier que votre code vérifier bien les standards de codage du cours :

   ```
   npx eslint
   ```

▷ Afin de tester le code produit, la bibliothèque Jest (`https://jestjs.io`) sera utilisée. Son installation se fait très simplement avec la commande :

```
npm install jest
```

Les pages `https://jestjs.io/docs/en/getting-started` et `https://zetcode.com/javascript/jest/` donnent quelques exemples d'utilisations simples de ce paquetage.

```
npx jest
```

Avec la configuration proposée, `jest` ne considère que les fichiers `*.test.js`.

6. En utilisant la bibliothèque `Jest`, écrire un module `stack.test.js` contenant les jeux de test permettant de tester toutes les fonctionnalités de `Stack`, de manière *indépendante* du code client manipulant les tours de Hanoi.
7. Lancer la commande suivante pour calculer le taux de couverture de vos tests :

   ```
   npx jest --coverage
   ```

   Noter que cette commande, en plus de vous afficher les résultats des tests, vous créer un rapport sous la forme d'un fichier HTML dans le répertoire `coverage/lcov-report/index.html`.
8. Compléter les tests en vous assurant d'avoir une couverture complète du code ($> 90\%$ pour chaque fichier).

   Comment faire pour que ces tests soient écrits de manière pure ? Que gagne-t'on ?