Année **2023-2024** Session **2**

PROGRAMMATION FONCTIONNELLE
PG104

Filière : Informatique, 1ère Année

Date de l'examen : 28/06/2024 Durée de l'examen : 2h00

Sujet de : D. RENAULT

Documents ☐ autorisés ☒ non autorisés Calculatrice ☐ autorisée ☒ non autorisée

Cet examen contient deux types de questions, avec un symbole ✎ ou ⌨ :

— ✎ **Des questions de réflexion**

Celles-ci demandent de répondre à une question sur la programmation fonctionnelle. Il est demandé de répondre de manière succinte et argumentée dans les fichiers texte `ex[0-9].txt` fournis. Toute forme d'imprécision sera sanctionnée. Dans le fichier, il est demandé de répondre après la ligne `#### Reponse` sans modifier les marqueurs `####`.

```
#### Question 2.1

Quelle est la couleur du cheval blanc d'Henri IV ?

#### Reponse 2.1

Ecrire la reponse ici
```

— ⌨ **Des questions de programmation**

Les réponses demandent d'écrire du code Ecmascript respectant les standards, à l'intérieur des fichiers `ex[0-9].js`. Ces codes seront vérifiés avec les outils utilisés en cours (eslint, tsc). Ces outils sont mis à votre disposition sur la machine. Dans tous les cas, il est toujours mieux de justifier son code par des commentaires. Dans le fichier, il faut simplement écrire du code de manière à ce qu'il soit valide du point de vue de `node` :

```
function horseWhiteColor(aKing) {
    //
    // Write code here
    //
}

export { horseWhiteColor };
```

Un fichier `README.md` fournit de l'aide quant à l'utilisation des outils. La documentation du langage est accessible à l'URL `https://developer.mozilla.org`. Les autres domaines sont inaccessibles. Une attention particulière sera portée aux tests fournis avec le code.

Le script `verif.sh` permet de vérifier que la syntaxe des fichiers écrits est correcte.
Le barême est donné à titre indicatif. Les exercices sont *indépendants* les uns des autres.

Exercice 1: 3 points
✎ En programmation fonctionnelle, qu'est-ce qu'une *fermeture* ?
Donner un (ou plusieurs) exemple(s) de tels éléments, ainsi que des arguments pour leur utilisation.

✓ Selon le cours, une *fermeture* (en anglais *closure*) consiste en la donnée d'une fonction, ainsi que de l'environnement nécessaire à son fonctionnement. L'exemple de base donné en cours est celui d'un générateur aléatoire `rand` embarquant son état interne, qui est modifié à chaque appel. Les valeurs embarquées dans des fermetures le sont typiquement en tant que références dans le corps de la fonction. Elles ont une durée de vie au moins égale à celle de la fonction qui les embarque.
Les fermetures sont des outils très importants en programmation fonctionnelle. Elles assurent que les fonctions peuvent continuer à s'appliquer hors de leur contexte local, dans la mesure où elles transportent ce contexte avec elles. Cela en fait des *unités de code* transportables, ce qui favorise leur réutilisation, par exemple en appliquant les propriétés de la *première classe* (passer une fonction en paramètre, la renvoyer, la stocker dans une structure de données). Aussi, le fait d'enfermer un état interne inaccessible de l'extérieur permet d'en faire de petites unités modulaires, avec la fonction comme interface publique et l'état interne comme élément privé.

Rappel algorithmique
Un *tas* est une structure de données classique dans laquelle avec laquelle on peut interagir de la manière suivante :

— il est possible de produire un tas vide (`empty`) ;
— étant donné un tas et une valeur, il est possible d'insérer ladite valeur à l'intérieur du tas (fonction `insert`) ;
— étant donné un tas, il est possible d'obtenir la valeur *la plus petite* stockée à l'intérieur (fonction `findMin`) ;
— étant donné un tas, il est possible de retirer la valeur *la plus petite* stockée à l'intérieur (fonction `removeMin`).

Les implémentations étudiées ici sont des tas-min (en anglais *min-heap*), au sens où on peut facilement en extraire la valeur *minimale* (à l'opposé des tas-max). Les tas sont typiquement utilisés pour implémenter des files de priorité, qui à leur tour peuvent servir dans des algorithmes tels que celui de Dijkstra.

Une fonction `anyToString` est fournie pour aider au débuggage, et convertit n'importe quelle valeur en chaîne de caractères pour affichage dans la console.

Exercice 2: 3 points
Cet exercice se propose étudier une structure de tas implémentée de manière impérative. Dans cette représentation les tas sont des arbres binaires dont les données sont stockées

dans un tableau.

> Vous avez à disposition une implémentation *impérative* des tas dans le fichier `given/heap.imperative.js`. Il est fortement encouragé de lire le code et d'écrire ses propres tests sur cette implémentation. La compréhension totale de ce code n'est pas indispensable à la réalisation de cet exercice.

1. ✎ Construire un tas à partir d'un tas vide, dans lequel on a inséré successivement les valeurs $\{7, 6, 5, 4, 3, 2, 1\}$ dans cet ordre.
   Donner le résultat de l'appel à la fonction `anyToString` sur le tas final.
2. ✎ En se basant sur l'exemple précédent, *et sur l'implémentation fournie*, expliquer en quoi cette implémentation utilise un arbre binaire pour stocker les valeurs.
3. ✎ En vous basant sur vos expérimentations, expliquer quelle est la profondeur de l'arbre interne d'un tas contenant n éléments.

✓ 2.1 Les valeurs ont été choisies de manière à ce que l'ordre des éléments ne soit pas trivial. Le code suivant produit le tas demandé :

```
import * as H from './given/heap.imperative.js';

const aHeap = H.empty();
[7, 6, 5, 4, 3, 2, 1].forEach((aVal) => {
    H.insert(aHeap, aVal);
});
console.log(anyToString(aHeap));
```

Le résultat escompté est : `{ elems: [1,4,2,7,5,6,3] }[7]`

2.2 Le code fourni explique que les données sont stockées dans un tableau, mais qu'il est possible d'utiliser des fonctions internes pour savoir comment les indices de ce tableau sont reliés en tant qu'un arbre binaire :

— la racine est l'élément d'index 0 ;
— le fils gauche (resp. droit) de l'élément d'index i est $2 * i + 1$ (resp. $2 * i + 2$),
— le parent de l'élément d'index $i > 0$ est $i/2$.

Bref, le tableau stocke ses données en utilisant la décomposition en base 2 de ses indices. Par exemple, le tas produit à la question précédente correspond à l'arbre :

```
      1
    /   \
   4     2
  / \   / \
  7 5   6 3
```

2.3 Le code rappelle que les données sont stockées de manière consécutive à l'intérieur du tableau. Cela assure que pour un tas à n éléments, l'arbre binaire associé contient :

— la racine à l'indice 0 ;
— les éléments à profondeur 1 aux indices 1 à 2 ;
— les éléments à profondeur 2 aux indices 3 à 6 ;
— ...
— les éléments à profondeur k aux indices $2^k - 1$ à $2^{k+1} - 2$.

Cela assure que la hauteur de l'arbre associé est exactement $\lfloor \log_2(n) \rfloor$.

Exercice 3: 7 points

Cet exercice se propose étudier une structure de tas implémentée de manière fonctionnelle. Dans cette représentation les tas sont des arbres binaires avec les propriétés suivantes :

- le tas vide est un arbre binaire spécial nommé `empty` qui est aussi une feuille (en tant qu'arbre binaire) ;
- un tas non vide est un arbre binaire tel que :
  - ses feuilles sont forcément égales à `empty` ;
  - la valeur d'un noeud est plus petite que celle de ses fils gauches et droits quand ils existent ;
  - les fils gauches et droits sont aussi des tas.

> Dans cet exercice, vous avez à disposition cette implémentation *fonctionnelle* des tas (à travers des fonctions `insert`, `findMin` et `removeMin`) dans le fichier `given/heap.functional.js`. Il est aussi fourni un ensemble de fonctions sur les arbres binaires dans le fichier `given/binary_tree.api.js`.

> L'implémentation utilisée dans cet exercice est *complètement indépendante* de celle exposée dans l'exercice précédent. Néanmoins, elle utilise une représentation interne sous la forme d'un arbre binaire, qui est très similaire.

3.0.1 ⌨ Écrire une fonction *récursive* et *pure* `heapSize` permettant de calculer le nombre d'éléments dans un tas.

Exemples :

```
heapSize(H.empty);                 // → 0
heapSize(H.insert(H.empty, 'a')); // → 1;
```

Comme Javascript est dynamiquement typé, il existe de nombreuses situations dans lesquelles on désirerait savoir si une valeur donnée est bien un tas ou pas. Pour cela, on aimerait écrire une fonction `isHeap` renvoyant un booléen. Pour s'assurer de son fonctionnement, on se donne les cas de tests suivants :

```
isHeap(H.empty);                                                    // → true
isHeap(H.insert(H.empty, 'a'));                                     // → true
isHeap(H.insert(H.insert(H.empty, 'a'), 'b'));                      // → true
isHeap(H.insert(H.insert(H.insert(H.empty, 'b'), 'a'), 'c'));       // → true
isHeap(B.node(1, H.empty, H.empty));                                // → true
isHeap(1);                                                          // → false, not bintree
isHeap(true);                                                       // → false, not bintree
isHeap({});                                                         // → false, not bintree
isHeap(B.node(4, B.leaf(5), B.leaf(6)));                            // → false, bad leaves
isHeap(B.node(4, B.node(2, H.empty, H.empty), B.node(3, H.empty, H.empty)));// → false, bad order
```

3.0.2 ⌨ Écrire une fonction *récursive* et *pure* `isHeap` prenant en paramètre une valeur quelconque et retournant vrai si et seulement si cette valeur est bien un tas.

3.1 ✎ Est-il possible d'implémenter une fonction comme `heapSize` de manière récursive terminale ? Quelles difficultés cela présente t'il ?

3.2 ✎ Quels sont les risques liés au fait d'implémenter des fonctions de manière récursive non-terminale ? Dans l'exemple de `heapSize`, décrire un exemple de tas pour lequel de tels risques deviennent concrets.

✓ Ci-joint des implémentations possibles des fonctions `heapSize` et `isHeap`. Il n'était pas demandé d'écrire la fonction `isHeap` en une seule expression.

```
function heapSize(aHeap) {
    if (HA.isEmpty(aHeap))
        return 0;
    else
        return 1 + heapSize(BA.left(aHeap)) + heapSize(BA.right(aHeap));
}

function isHeap(aValue) {
    function isHeapB(aBinTree) {
        return ((BA.isLeaf(aBinTree)) && (HA.isEmpty(aBinTree))) ||
            ((!BA.isLeaf(aBinTree)) &&
             (isHeapB(BA.left(aBinTree))) &&
             (isHeapB(BA.right(aBinTree))) &&
             ((HA.isEmpty(BA.left(aBinTree))) || ((BA.val(BA.left(aBinTree)) > BA.val(aBinTree)))) &&
             ((HA.isEmpty(BA.right(aBinTree))) || ((BA.val(BA.right(aBinTree)) > BA.val(aBinTree))))
        );
    }
    return BA.isBinaryTree(aValue) && isHeapB(aValue);
}
```

Note (après examen)

Après l'examen, l'auteur a remarqué (sur un certain nombre de copies) qu'il était tout à fait possible d'implémenter de manière récursive terminale la fonction `heapSize`. Il suffisait pour cela d'appliquer récursivement la fonction `removeMin` jusqu'à ce que le tas soit vide. Comme chaque opération prend un temps logarithmique $\log(n)$, cela produit une fonction de complexité $\mathcal{O}(n \log(n))$, à comparer à la solution ci-dessus de complexité $\mathcal{O}(n)$.
Il est évident que *le choix d'une telle solution est mauvais*, et l'auteur a laissé passer cette possibilité en écrivant le sujet. La solution en question a été acceptée dans le cadre de l'examen, mais elle n'est pas raisonnable dans l'absolu.

✓ 3.1 Le cours affirme que toute fonction récursive peut se mettre sous la forme d'une fonction récursive terminale, mais que c'est non trivial. La formulation de la question sous-entend néanmoins que cela peut présenter des difficultés.

Dans le cas présent, la fonction `heapSize` effectue deux appels récursifs. Potentiellement, elle doit parcourir un arbre binaire complet.

L'implémenter de façon récursive terminale, c'est l'implémenter comme une réécriture de ses paramètres. Cette réécriture doit descendre l'arbre le long de sa branche gauche, remonter, puis redescendre sa branche droite. C'est la partie consistant à remonter qui est difficile, parce que la donnée seule d'un noeud de l'arbre ne permet pas d'obtenir son parent. Il faudrait donc pour remonter se rappeler à chaque étape du chemin jusqu'à la racine. Il s'agit en réalité d'une technique classique appelée `zipper` et mise au point par un français, Gérard Huet (cf. `https://en.wikipedia.org/wiki/Zipper_(data_structure)`)

3.2 Les fonctions récursives non terminales ne peuvent pas être facilement optimisées par un compilateur. Elles effectuent donc leurs appels récursifs dans la pile d'appel. Le risque principal consiste à atteindre la taille maximale de la pile d'appel, et de produire un dépassement de pile (*stack overflow*).

Dans le cas présent, le nombre d'appels récursifs empilés est de l'ordre de la profondeur de l'arbre associé au tas. Or cette profondeur est logarithmique en le nombre d'éléments. Pour engendrer un *stack overflow*, il faudrait appliquer la fonction sur un tas contenant 2^n éléments avec n de l'ordre de la taille de la pile. Et c'est irréaliste en pratique. En effet, la taille d'uune pile est de l'ordre de 8 mégaoctets. En cours on a vu qu'elle pouvait empiler de l'ordre de 10^4 appels récursifs. Et pour créer un tas de profondeur 100 (soit 100 fois moins que la limite), il faudrait qu'il contienne 2^{100} éléments, ce qui correspond à 2^{57} gigaoctets de données.

Exercice 4: 6 points

Les tas sont des structures de données qui permettent d'implémenter de manière efficace des tris. En effet, les opérations `insert` et `removeMin` sur les tas se font en temps logarithmique, ce qui permet de trier la liste en un temps optimal. L'algorithme que nous proposons d'implémenter ici est d'une grande simplicité. En partant d'une liste de valeurs `aList` :

— insérer les éléments de `aList` un par un dans un tas `aHeap` initialement vide ;

— extraire les éléments de `aHeap` un par un et les placer dans une nouvelle liste triée.

Dans cet exercice, nous allons utiliser l'implémentation des tas de manière fonctionnelle (qui ont été vus dans l'exercice 3).

Vous avez à disposition une implémentation *fonctionnelle* des tas `given/heap.functional.js`, l'implémentation des listes vues en cours `given/list.api.js`, ainsi qu'un ensemble de fonctions utilitaires dans `given/list.firstclass.js` et `given/list.utils.js`.

4.0.1 ⌨ Écrire une fonction *récursive terminale* et *pure* `heapToList` qui prend en paramètre un tas `aHeap` et renvoie la liste des élements dans le tas triée dans l'ordre croissant.

Remarque : il est possible d'utiliser la fonction `listReverse` qui est récursive terminale.

4.0.2 ⌨ Écrire une fonction *récursive terminale* et *pure* `heapSort` qui prend en paramètre une liste `aList` et renvoie une liste contenant les mêmes éléments et triée dans l'ordre croissant.

✓ Ci-joint des implémentations possibles des fonctions `heapToList` et `heapSort`. La fonction `listToHeap` n'était pas demandée, mais son écriture simplifie la compréhension de `heapSort`.

```
function listToHeap(aList) {
    return LF.listReduce(aList, HA.insert, HA.empty);
}

function heapToList(aHeap) {
    function heapToListTR(aHeap, aList) {
        if (HA.isEmpty(aHeap))
            return LU.listReverse(aList);
        else
            return heapToListTR(HA.removeMin(aHeap), LA.cons(HA.findMin(aHeap), aList));
    }
    return heapToListTR(aHeap, LA.nil);
}

function heapSort(aList) {
    const aHeap = listToHeap(aList); // Insert into heap
    return heapToList(aHeap);        // Remove from heap
}
```

Exercice 5: 3 points

Au final, le sujet fournit deux implémentations des tas : l'une est fonctionnelle (`heap.functional.js`) et l'autre est impérative (`heap.imperative.js`).

5.1 ✎ Quel est le nom utilisé dans les cours pour parler de plusieurs implémentations d'une même structure de données partageant une interface commune ?

5.2 ✎ En fait, il n'est pas possible de remplacer directement l'une des implémentations par l'autre pour implémenter `heapSort` [1]. Pourquoi ?

1. On dit qu'elles ne sont pas *substituables*.

✓

5.1 La notion évoquée est celle de *type abstrait de données*, vue en particulier dans les cours d'ateliers de programmation et de programmation fonctionnelle.

5.2 Les deux implémentations ne sont pas substituables, malgré le fait qu'elles aient (en apparence) la même interface. En effet, l'implémentation impérative agit par effet de bord sur le tas passé en paramètre, alors que l'implémentation fonctionnelle est pure.

Cela se traduit par le fait que les types de retour des fonctions comme `insert` et `removeMin` ne sont pas les mêmes, et que `empty` n'est pas de la même nature.

En pratique, les implémentations d'algorithmes comme `heapSort` seront différentes. Voici par exemple l'implémentation avec les tas impératifs :

```
function heapToListI(aHeap) {
    let aList = LA.nil;
    while (!HI.isEmpty(aHeap)) {
        aList = LA.cons(HI.findMin(aHeap), aList);
        HI.removeMin(aHeap);
    }
    return LU.listReverse(aList);
}

function heapSortI(aList) {
    const aHeap = HI.empty();
    LF.listForEach(aList, (aVal) ⇒ HI.insert(aHeap, aVal));
    return heapToListI(aHeap);
}
```

Au final, les deux implémentations sont assez différentes.

Recommandations globales

Cette section ne contient pas de questions pour l'examen, mais un certain nombre de recommandations et de consignes générales.

Outils disponibles

— Tous les outils utilisables usuellement en TD sont disponibles pour programmer. Cela contient en particulier `node`, `emacs`, `code`, `firefox` et `evince`.

— Un certain nombre d'outils standard pour le développement en Javascript sont aussi installés, ainsi que les dépendances dans le répertoire `node_modules` : `tsc`, `eslint` et `jest`.

— Des fichiers de configuration pour les outils précédents sont disponibles dans le répertoire `exam`, et les scripts suivants utilisables avec `npm run` :

  — `npm run tsc` pour le compilateur Typescript,

  — `npm run eslint` pour le linter ESLint et

  — `npm run jest` pour la bibliothèque de tests Jest.

Questions de texte

— Il est demandé de répondre avec précision aux questions. Toute forme d'imprécision sera considérée comme un malus potentiel.

— Si vous estimez que vous êtes imprécis, pensez à prendre un exemple pour étoffer votre explication.

— En cas d'utilisation d'un exemple, pensez à relier cet exemple aux définitions vues en cours.

Questions de code

— Tout fichier syntaxiquement incorrect, ou dont l'utilisation directe avec `node` ne termine pas sans erreur ne sera pas considéré pour la correction. Le script `verif.sh` ne fait qu'une vérification syntaxique (avec `node --check`).

— Aucun autre fichier que ceux demandés dans les exercices ne sera lu ni considéré pour la correction. Écrire des tests dans un autre fichier (avec `jest` ou toute autre forme de test) peut aider pour développer, mais ils ne seront pas considérés comme des rendus de l'examen, sauf s'ils sont insérés dans les fichiers réponses.

— Aucun import n'est nécessaire dans le code fourni, hormis ceux déjà présents dans les fichiers fournis. Si jamais des fonctions dépendaient l'une de l'autre, elles seraient présentes dans le même fichier. Modifier les imports existant ou ajouter ses propres imports, c'est prendre le risque que le code final ne soit pas considéré.