Année **2023-2024** Session **1**

Programmation fonctionnelle
PG104

Filière : Informatique, 1ère Année

Date de l'examen : 24/05/2024 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.

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 1: 3 points

✎ Expliquer ce qu'est une structure de données *fonctionnelle*.

Donner un (ou plusieurs) exemple(s) de telles structures de données, ainsi que des arguments pour leur utilisation.

✓ Selon le cours, une structure de données est dite fonctionnelle si elle peut être implémentée de manière pure : toutes les opérations sur la structure se font à travers des fonctions pures. En particulier, aucun effet de bord n'est fait sur la structure, et elle est donc immutable (i.e ne subit pas de modification au cours de son utilisation). Les listes vues en cours de PG104 (à base de `car`, `cdr` et `cons`) sont des structures de données fonctionnelles, au même titre que les chaînes de caractères en Javascript.
Les structures de données fonctionnelles, de par leur immutabilité, sont bien adaptées à la programmation fonctionnelle, parce qu'elles encouragent l'utilisation de fonctions pures. La programmation avec des fonctions pures facilite la reproductibilité, et donc les tests, mais aussi les preuves. Elle permet d'appliquer des techniques de mémoïsation, de persistance et de parallélisation.

Exercice 2: 12 points
Cet exercice est tiré du livre d'*Okasaki* intitulé *Purely functional data structures*, chapitre 9.2.1 et porte sur l'implémentation de *binary random-access lists*, que nous appelerons par la suite des *bra-listes*. Il s'agit d'une structure de données fonctionnelle ayant la même interface que les listes vues en cours, mais qui permet aussi de faire des accès à n'importe quel élément (*lookup*) et des mises à jour (*update*) de manière efficace. Cet exercice se décompose en deux parties indépendantes :

— la première partie est une présentation simplifiée, implémentant un système de comptage en binaire en utilisant une simple liste nommée *bra-nombre* ;

— la seconde partie implémente quelques fonctions sur les *bra-listes*.

Première partie : comptage en binaire sur les bra-nombres Un *bra-nombre* est une liste d'entiers, telle que le i-ème élément de la liste vaut soit 0, soit 2^i. La valeur numérique d'un bra-nombre est la somme de ses éléments. L'idée derrière le bra-nombre est de représenter la décomposition en puissances de 2 d'un nombre entier. En particulier, le dernier élément d'un bra-nombre, quand il existe, est forcément non nul.

> Dans cet exercice, vous avez à disposition un ensemble de fonctions sur les listes vues en cours, implémentées dans un module `given/list.js`.

Exemples :

```
import * as L from './given/list.js';

const aBraNum0 = L.nil;                                          // the bra-num for 0
const aBraNum1 = L.cons(1, L.nil);                               // the bra-num for 1
const aBraNum2 = L.cons(0, L.cons(2, L.nil));                    // the bra-num for 2  = 0+2
const aBraNum3 = L.cons(1, L.cons(2, L.nil));                    // the bra-num for 3  = 1+2
const aBraNum8 = L.cons(0, L.cons(0, L.cons(0, L.cons(8, L.nil)))); // the bra-num for 8  = 0+0+0+8
const aBraNumF = L.cons(1, L.cons(2, L.cons(4, L.cons(8, L.nil)))); // the bra-num for 15 = 1+2+4+8
```

2.0.1 ⌨ Écrire une fonction *récursive terminale* et *pure* `braNumValue` permettant de transformer un bra-nombre en sa valeur numérique.

2.0.2 ⌨ Écrire une fonction *récursive terminale* et *pure* `braNumFromNumber` permettant de transformer un entier en le bra-nombre correspondant.

Note : une fonction `numMaxPowerOfTwo` est fournie pour aider au calcul.

Deuxième partie : les bra-listes La représentation précédente est une représentation jouet, permettant de comprendre la structure de données suivante. Commençons par fixer une valeur spéciale `none` (qui va jouer un rôle de marqueur, un peu comme `nil` pour les listes classiques).

Une *bra-liste* est une liste classique telle que le i-ème élément soit :

— soit `none` ;

— soit un arbre binaire complet de hauteur $i+1$ (et donc à 2^i feuilles).

Les noeuds et les feuilles des arbres peuvent stocker des valeurs. Sur les noeuds internes, la valeur stockée est le nombre de feuilles de l'arbre à partir de ce noeud. Sur les feuilles, on stocke les données à l'intérieur de la structure. Dans cet exercice, les données seront *uniquement des chaînes de caractères*. Comme pour les bra-nombres, une bra-liste est telle que son dernier élément ne peut pas être `none`.

> Dans cet exercice, vous avez à disposition (entre autres) un ensemble de fonctions sur les arbres binaires vus en cours, implémentées dans un module `given/binary_tree.js`.

Exemples :

la bra-liste vide — la bra-liste à 1 élément $\{hop\}$ — la bra-liste à 5 éléments $\{a, b, c, d, e\}$

Les *éléments* de la bra-liste sont définis comme les chaînes de caractères stockées dans les feuilles. La *taille* d'une bra-liste est le nombre d'éléments qu'elle contient. L'*indice* d'un élément dans une bra-liste est le nombre de feuilles apparaissant avant cet élément dans le même arbre ou dans les arbres précédents dans la liste. En reprenant l'exemple de la bra-liste à 5 éléments ci-dessus, `'a'` a l'indice 0, `'b'` a l'indice 1, et `'d'` a l'indice 3.

Cette structure de données se base sur l'idée d'une décomposition des éléments de la bra-liste en puissances de 2, comme pour les bra-nombres.

Le code suivant montre les mêmes bra-listes implémentées en Javascript :

```
import * as L from './given/list.js';
import * as B from './given/binary_tree.js';

const aBraList0 = L.nil;                          // the empty bra-list
const aBraList1 = L.cons(B.leaf('x'), L.nil);     // the bra-list containing {x}
const aBraList5 = L.cons(B.leaf('a'),
                         L.cons(none,
                                L.cons(B.node(4,
                                              B.node(2, B.leaf('b'), B.leaf('c')),
                                              B.node(2, B.leaf('d'), B.leaf('e'))),
                                       L.nil))); // the bra-list containing {a,b,c,d,e}
```

Plusieurs opérations sur les bra-listes sont déjà implémentées dans le module `given/bra_list.js`. La plupart d'entre elles ont des noms équivalents à celles sur les listes. En particulier, sont

fournis une fonction `isBraList` pour tester si une valeur est bien une bra-liste, ainsi qu'une fonction `braToString` pour convertir une bra-liste en chaîne de caractères.

Attention : les bra-listes et les listes ont la même interface, mais les fonctions ne renvoient pas les mêmes valeurs.

```
BL.head(aBraList5); // → 'a'
L.head(aBraList5);  // → B.leaf('a')
```

2.1.1 ⌨ Écrire une fonction pure `braSize` prenant en entrée une bra-liste et retournant le nombre d'éléments à l'intérieur. La fonction `braTreeSize` peut être utilisée.

Cherchons à écrire une fonction `braLookup` qui, étant donné un indice i, renvoie le i-ème élément d'une bra-liste. Pour ce faire, on propose de travailler en deux temps :

— Étant donné un arbre binaire complet de hauteur k, trouver comment obtenir la i-ème feuille de cet arbre (question 2.1.2, fonction `braLookupTree`) ;

```
const aBTree = B.node(4,
                      B.node(2, B.leaf('b'), B.leaf('c')),
                      B.node(2, B.leaf('d'), B.leaf('e')));
braLookupTree(aBTree, 0); // → 'b';
braLookupTree(aBTree, 2); // → 'd';
```

— Étant donné une bra-liste et un indice i, trouver l'arbre dans la bra-liste qui contient le i-ème élément (question 2.1.3, fonction `braLookup`).

```
braLookup(aBraList5, 0); // → 'a'
braLookup(aBraList5, 3); // → 'd'
```

2.1.2 ⌨ Écrire une fonction pure `braLookupTree` prenant en entrée un arbre binaire et un indice `anIndex` et retournant la `anIndex`-ème feuille de cet arbre. Cette fonction renvoie une erreur en cas d'indice incorrect.

2.1.3 ⌨ Écrire une fonction pure `braLookup` prenant en entrée une bra-liste et un indice `anIndex` et retournant l'élément à l'indice `anIndex` dans la bra-liste. Cette fonction renvoie une erreur en cas d'indice incorrect.

2.2 ✎ Quelle est le nombre (au pire) d'arbres dans une bra-liste à n éléments ?

2.3 ✎ Évaluer le complexité algorithmique (au pire) d'une recherche dans une bra-liste à n éléments.

Il est possible d'implémenter une fonction `braUpdate(aBbra, anIndex, aValue)` qui réalise pratiquement le même code que `braLookup`, mais construit une nouvelle bra-liste dans laquelle on a remplacé l'élément à l'indice `anIndex` par `aValue`. Et cela sans effets de bords.

2.4 ✎ Quelle est le nom de la propriété évoquée en cours exprimant le fait qu'avec des structures de données fonctionnelles, on peut réutiliser les parties des structures déjà existantes lorsqu'on en crée de nouvelles ?

Correction de la première partie : les bra-nombres

Une implémentation possible pour `braNumValue` et `braNumFromNumber` :

```
function braNumValue(aBraNum) {
    function braVTR(aBraNum, aResult) {
        if (L.isEmpty(aBraNum))
            return aResult;
        else
            return braVTR(L.tail(aBraNum), L.head(aBraNum) + aResult);
    }
    return braVTR(aBraNum, 0);
}

function braNumFromNumber(aNumber) {
    function braFNTR(aNumber, aResult, aPower) {
        if (aPower === 1)
            return aNumber === 1 ? L.cons(1, aResult) : L.cons(0, aResult);
        else if (aNumber >= aPower)
            return braFNTR(aNumber - aPower, L.cons(aPower, aResult), aPower / 2);
        else
            return braFNTR(aNumber, L.cons(0, aResult), aPower / 2);
    }
    if (aNumber === 0)
        return L.nil;
    else
        return braFNTR(aNumber, L.nil, numMaxPowerOfTwo(aNumber));
}
```

Correction de la seconde partie : les bra-listes

Une implémentation possible pour `braSize`, `braLookupTree` et `braLookup` :

```
function braSize(aBra) {
    if (BL.braIsEmpty(aBra))
        return 0;
    else {
        const tree = L.head(aBra);
        if (tree !== BL.none) {
            if (B.isLeaf(tree))
                return 1 + braSize(L.tail(aBra));
            else
                return BL.braTreeSize(tree) + braSize(L.tail(aBra));
        } else {
            return braSize(L.tail(aBra));
        }
    }
}

function braLookupTree(aTree, anIndex) {
    if (B.isLeaf(aTree)) {
        if (anIndex > 0)
            throw new Error("braLookupTree: index too large for leaf");
        else
            return B.val(aTree);
    } else {
        const w = B.val(aTree);
        if (anIndex < Math.floor(w / 2))
            return braLookupTree(B.left(aTree), anIndex);
        else
            return braLookupTree(B.right(aTree), anIndex - Math.floor(w / 2));
    }
}

function braLookup(aBra, anIndex) {
    if (BL.braIsEmpty(aBra))
        throw new Error("braLookup: index too large for bra");
    else {
        const aTree = L.head(aBra);
        if (aTree !== BL.none) {
            const aSize = BL.braTreeSize(aTree);
            if (anIndex < aSize)
                return braLookupTree(aTree, anIndex);
            else
                return braLookup(L.tail(aBra), anIndex - aSize);
        } else {
            return braLookup(L.tail(aBra), anIndex);
        }
    }
}
```

✓ 2.2 Si on voit une bra-liste en tant que liste d'arbres de taille n, le i-ème élément de cette liste contient potentiellement 2^i valeurs. Cela signifie que cette même liste contient au mieux $\sum 2^k = 2^n - 1$ valeurs. Et réciproquement, si on doit stocker moins de $2^n - 1$ valeurs, on a besoin d'une bra-liste de longueur $\leq n$. Si on ramène cela à la question énoncée, une bra-liste à n éléments contient au pire $\log_2(n)$ arbres.

2.3 Si on cherche un élément dans une bra-liste à n éléments, il faut parcourir une liste de longueur au pire $\log_2(n)$ pour trouver l'arbre binaire contenant cet élément. De plus, cet arbre est au pire de profondeur $\log_2(n)$. Il faut trouver une feuille dans cet arbre étant donné son indice dans l'ordre préfixe. Cela se fait aussi en temps logarithmique. Au final, l'opération `braLookup` a une complexité temporelle au pire de $\log_2(n)$ opérations pour une bra-liste à n éléments.

2.4 Il s'agit de la *persistance*. Cette propriété permet ici d'avoir que les mises à jour sur les bra-listes se font en temps logarithmique en la taille de la bra-liste. Évidemment, cette propriété ne fonctionne que si les opérations sur la bra-liste se font sans modifications des structures déjà existantes, mais c'est la définition d'une structure de données fonctionnelle. Noter que la *pureté* n'est pas une réponse correcte à cette question, même si la persistance est une conséquence de la pureté.

Exercice 3: 5 points

Considérons la question [1] de pouvoir réaliser un programme capable d'effectuer des calculs de la manière suivante :

```
init(1).add(2).mul(3).div(4).val(); // Should return 2.25 = ((1 + 2) * 3) / 4
```

Parmi les réponses proposées, et si on omet les réponses tirant parti du paradigme objet, il a été proposé le code suivant :

```
function init(start) {
    let self = {
        value: start,
        add: num ⇒ { self.value += num; return self; },
        sub: num ⇒ { self.value -= num; return self; },
        mul: num ⇒ { self.value *= num; return self; },
        div: num ⇒ { self.value /= num; return self; },
        val: () ⇒ self.value,
    };
    return self;
}
init(1).add(2).mul(3).div(4).val(); // → 2.25
```

1. ✎ En quoi ce code met-il en valeur la programmation fonctionnelle ?
   (*Remarque* : il y a au moins 3 arguments allant dans le sens de cette question, mais ils ne sont pas tous aussi évidents les uns que les autres)
2. ✎ En quoi ce code *ne* met-il *pas* en valeur la programmation fonctionnelle ?
3. ✎ Pourriez-vous proposer une ou plusieurs modifications améliorant les critiques de la question précédente ?

1. Cette question est inspirée de StackOverflow (https://stackoverflow.com/questions/54318398/how-to-create-chaining-of-objects-in-js), qui a été posée lors d'un un entretien d'embauche.

✓

1. Le code en question met en valeur la programmation fonctionnelle, et cela sur au moins trois points :

   — les valeurs manipulées sont des dictionnaires de fonctions, ce qui illustre la *propriété de première classe* des fonctions vue en cours ;

   — les valeurs construites encouragent la *composition* des appels de fonctions pour produire un résultat ;

   — (moins évident) la valeur manipulée est une valeur récursive, dans la mesure où les fonctions qu'elle contient font référence à elle-même.

   Stricto sensu, il s'agit d'un codage (classique en réalité) de l'objet (au sens de la programmation orientée objet) sous la forme d'une valeur écrite avec le paradigme fonctionnel. La variable `self` fait référence au `this` des langages objets. L'exemple met en valeur le fait qu'un objet est une valeur naturellement récursive.

2. Le code fait des effets de bords sur `self.value`. Les fonctions utilisées sont impures, ce qui est raisonnable du point de vue du paradigme objet, mais critiquable du point de vue du paradigme fonctionnel.

3. Pour rendre le code pur, il faudrait s'assurer de créer une nouvelle valeur à chaque transformation. C'est en réalité très simple à réaliser, une fois qu'on a compris qu'on pouvait utiliser `init` comme constructeur des valeurs :

```
function init(start) {
    let self = {
        value: start,
        add: num ⇒ init(self.value + num),
        sub: num ⇒ init(self.value - num),
        mul: num ⇒ init(self.value * num),
        div: num ⇒ init(self.value / num),
        val: () ⇒ self.value,
    };
    return self;
}
init(1).add(2).mul(3).div(4).val(); // → 2.25
```

Annexe

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.

— Si vous ne faites que donner 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 (par exemple avec `jest`) 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, 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 est juste un risque que le code final ne soit pas considéré.