Année 2021-2022 — Session 1

Programmation fonctionnelle
PG104

Filière : Informatique, 1ère Année

Date de l'examen : 23/05/2022 — Durée de l'examen : 2h00

Sujet de : D. Renault

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

Cet examen contient deux types de questions :

— **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[1-5].txt` fournis. Toute forme d'imprécision sera sanctionnée. Dans le fichier, il suffit de répondre après la ligne `#### Reponse`.

```
#### Question H4

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

#### Reponse H4

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[1-5].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 codede manière à ce qu'il soit valide du point de vue de `node` :

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

Un script `verif.sh` vous est fourni afin de vérifier simplement que la syntaxe des fichiers écrits est correcte. Pour les questions de programmation, les paquetages `eslint` et `typescript` sont installés, et un fichier `README.txt` fournit de l'aide quant à leur utilisation. La documentation du langage est accessible à l'URL `https://developer.mozilla.org`. Les autres domaines sont inaccessibles.

Le barême est donné à titre purement indicatif.

Les exercices sont tous *indépendants* les uns des autres.

Exercice 1: 2 points

✎ En quoi consiste la programmation avec des fonctions pures ?

Donnez sur un exemple simple les intérêts de ce style de programmation.

✓ Une fonction est dite *pure* si c'est une fonction au sens mathématique : elle produit les mêmes résultats sur les mêmes entrées, sans effets de bords. Le cours contient de nombreux exemples, le plus récurrent étant celui de la fonction `count_chars`, dans sa version récursive pure et sa version impérative impure. La reproductibilité des résultats permet de raisonner plus facilement sur les fonctions (tests, preuves) et d'appliquer des optimisations (appels récursifs terminaux, mémoïsation). Le fait de s'interdire les effets de bords assure que les objets manipulés sont indépendants les uns des autres, évitant les bugs dûs à l'aliasing, et permettant des techniques comme la persistance (permettant par exemple de faire des retours vers le passé) et la parallèlisation (par exemple dans des techniques comme le map-reduce).

Exercice 2: 3 points

⌨ Écrire une fonction `isBalanced` de manière récursive qui, étant donné une chaîne de caractères, vérifie que celle-ci forme une expression bien parenthésée (i.e chaque parenthèse fermante est associée à une unique parenthèse ouvrante qui la précède dans la chaîne). Les caractères différents de `'('` et `')'` sont ignorés.

Exemple :

```
isBalanced("");            // → true
isBalanced("(1+2)*(4-3)"); // → true
isBalanced(")))");         // → false
```

Indication : la méthode `substring` de la classe `String` peut être utilisée pour obtenir une chaîne de caractères privée de son premier élément.

Une implémentation possible (et récursive terminale) de cette fonction :

```
// ‘isBalanced‘ takes ‘aString‘ parameter and returns a boolean
// telling whether ‘aString‘ has balanced parentheses.
function isBalanced(aString) {

    function isBalancedRec(aString, aLevel) {
        if (aLevel < 0)
            return false;
        else if (aString === "")
            return aLevel === 0;
        else {
            const aChar = aString[0];
            const aSubstring = aString.substring(1);
            switch (aChar) {
            case "(": return isBalancedRec(aSubstring, aLevel+1);
            case ")": return isBalancedRec(aSubstring, aLevel-1);
            default:  return isBalancedRec(aSubstring, aLevel);
            }
        }
    }

    return isBalancedRec(aString, 0);
}

isBalanced(""); // → true
```

Parmi les éléments vérifiés (en plus de eslint et tsc) : si l'implémentation est impérative et si elle effectue des effets de bord. Bonus si l'implémentation est récursive terminale et que l'auteur le fait remarquer.

Exercice 3: 5 points
Considérons le code suivant, permettant de construire un curseur sur un tableau donné. La fonction `logCursor` montre comment utiliser le curseur pour parcourir les éléments du tableau un à un :

```
const aCursor1 = (() => {
    let anArray = [1, 2, 3, 6, 7, 72 ];
    let anIndex = 0;
    return {
        isEmpty: () => anIndex >= anArray.length,
        next:    () => anIndex++,
        data:    () => anArray[anIndex],
    }
})();

function logCursor(aCursor) {
    while(!aCursor.isEmpty()) {
        console.log(aCursor.data());
        aCursor.next();
    }
}

logCursor(aCursor1); // Logs : 1 2 3 6 7 72
```

1. ✎ Indiquer les composants de `aCursor1` qui sont accessibles au toplevel et ceux qui ne le sont pas.
2. ✎ Quel nom donne t-on à une construction comme celle-ci séparant sa part externe de sa part interne ?
3. ✎ Proposer un type pour chacun des composants externes de `aCursor1`. Quel nom donne t-on à une telle description ?

Dans la suite, on nomme *curseur* toute construction suivant le même principe de fonctionnement que `aCursor1`. En particulier, un curseur doit pouvoir être passé sans erreur à la fonction `logCursor`.

4. ⌨ Écrire une fonction `makeArrayCursor` qui, à partir d'un tableau donné, construise un curseur énumérant les éléments du tableau.
5. ✎ Quel nom donne-t'on à la technique de programmation faisant passer de `aCursor1` à `makeArrayCursor` ?
6. ⌨ Écrire une fonction `makeListCursor` qui, à partir d'une liste donnée, construise un curseur énumérant les éléments de la liste. Pour cela, il est possible d'utiliser `nil`, `cons`, `head`, `tail`, et `isEmpty`.
7. ⌨ Écrire une fonction `zipCursors` qui, étant donnés deux curseurs `cursor1` et `cursor2`, construise un curseur produisant les paires (sous la forme de tableaux à deux éléments) des éléments produits par `cursor1` et `cursor2`. Le curseur résultant est vide dès que l'un des curseurs `cursor1` ou `cursor2` est vide.

✓ La formulation de l'exercice évite à dessein les termes utilisés dans le cours, ce que ne fait pas la correction suivante :

1. Les éléments *publics* sont les trois fonctions `isEmpty`, `next` et `data`. Les éléments privés sont le tableau `anArray` et l'entier `anIndex`. La séparation entre public et privé se fait grâce au mécanisme de *portée*.
2. Une construction séparant sa partie privée de sa partie publique est un *module*. Le terme d'*encapsulation* est aussi acceptable. Noter que l'*abstraction* est une propriété logicielle et pas une construction.
3. La liste des composants externes et de leurs types constitue une *interface*. Ici, cette interface serait :

   ```
   isEmpty : (void) ⇒ bool
   next:     (void) ⇒ number
   data:     (void) ⇒ number
   ```

   Il était acceptable de considérer que la méthode `next` faisait simplement un effet de bord, et renvoyait `void`, même si c'était incorrect du point de vue de Typescript.

5. Il s'agit d'une *généralisation*, ici du tableau utilisé à l'intérieur du curseur. On peut aussi parler de *refactoring*, mais c'est beaucoup moins précis.

Le reste du code est fourni ci-après :

```
function makeArrayCursor(anArray) {
    let anIndex = 0;
    return {
        isEmpty: () ⇒ anIndex >= anArray.length,
        next:    () ⇒ anIndex++,
        data:    () ⇒ anArray[anIndex],
    }
}

function makeListCursor(aList) {
    let aPointer = aList;
    return {
        isEmpty: () ⇒ isEmpty(aPointer),
        next:    () ⇒ aPointer = tail(aPointer),
        data:    () ⇒ head(aPointer),
    }
}

function zipCursors(aCursor1, aCursor2) {
    return {
        hasNext: () ⇒ aCursor1.hasNext() && aCursor2.hasNext(),
        next:    () ⇒ { aCursor1.next(); aCursor2.next(); },
        data:    () ⇒ [ aCursor1.data(), aCursor2.data() ],
    }
}
```

Exercice 4: 5 points
Un IDD (pour *Integer Dichotomy Diagrams*) est une représentation d'un nombre entier de potentiellement grande taille sous une forme arborescente [1]. Pour représenter de tels entiers, on remarque que tout entier naturel s'écrit de manière unique :

— soit comme l'entier 0 ;

— soit comme l'entier 1 ;

— soit comme $h \times 2^{2^p} + l$ avec $1 \leq h < 2^{2^p}$ et $0 \leq l < 2^{2^p}$.

Dans ce dernier cas, on note $\langle h, p, l \rangle$ ce triplet. Quelques exemples :

— l'entier 42 s'écrit $\langle 2, 2, 10 \rangle$ car $42 = 2 \times 2^{2^2} + 10 = 2 \times 16 + 10$,

— l'entier 10 s'écrit $\langle 2, 1, 2 \rangle$ car $10 = 2 \times 2^{2^1} + 2 = 2 \times 4 + 2$,

— l'entier 2 s'écrit $\langle 1, 0, 0 \rangle$ car $2 = 1 \times 2^{2^0} + 0 = 1 \times 2 + 0$.

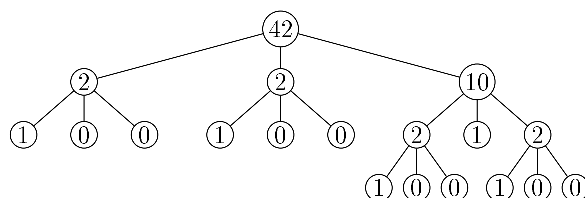
Il est admis que le représentation de n sous la forme $\langle h, p, l \rangle$ est unique. De plus, à p fixé, $\langle h, p, l \rangle$ varie dans l'intervalle $[2^{2^p}; 2^{2^{p+1}}[$. Ainsi, si $p = 0$, les $\langle h, p, l \rangle$ possibles sont les entiers entre 2 et 3, tandis que si $p = 1$, ce sont les entiers entre 4 et 16.

Dans le code est fourni :

— une fonction `fromIdd` qui étant donné `h`, `p` et `l`, renvoie la valeur de `n` correspondant à $\langle h, p, l \rangle$.

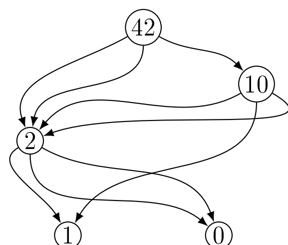
— une fonction `toIdd` qui étant donné un entier `n` ≥ 2, renvoie un tableau `[h, p, l]` correspondant à $\langle h, p, l \rangle$.

L'IDD correspondant au nombre n est un arbre ternaire dont la racine est n, et les trois fils sont les IDDs des trois nombres h, p et l tels que $n = \langle h, p, l \rangle$. Ainsi, pour l'entier 42 :



1. ⌨ Écrire une fonction `makeIdd` qui, étant donné un entier `n`, construit l'arbre précédent. Pour cela, il est possible d'utiliser `nil`, `cons`, `node` et `leaf`.
2. ✎ Quelles propriétés de la programmation fonctionnelle sont-elles envisageables sur de telles structures de données ?

Une proposition consiste à réutiliser les sous-arbres utilisés dans la construction des arbres précédents qui correspondent aux mêmes entiers. Ainsi, pour l'entier 42 :



42 a pour fils 2, 2 et 10
10 a pour fils 2, 1 et 2
2 a pour fils 1, 0 et 0

1. Cet exercice est inspiré à la fois d'un devoir d'agrégation d'informatique et de l'article qui lui correspond intitulé *Shared Integer Dichotomy* de J. Vuillemin, 2014, évidemment très largements simplifiés.

Pour cela, on fournit un dictionnaire contenant les arbres déjà construits :

```
const iddMem = {
  0: leaf(0),
  1: leaf(1),
};
```

3. ✎ Expliquer comment il est possible d'utiliser la représentation des arbres vue en cours (à travers les fonctions `node` et `leaf`) pour représenter cette structure.
4. ⌨ Écrire une fonction `makeIddMem` qui, étant donné un entier `n`, construit l'arbre précédent en sauvegardant les sous-arbres construits dans le tableau `iddMem`.
5. ✎ Quel est le nom de la technique de programmation utilisée ici ?
   Expliquer son intérêt.

2. Les valeurs dans les arbres ainsi construits sont constantes. Il est envisageable d'utiliser des techniques basées sur de la programmation fonctionnelle pure.
3. Les noeuds de l'arbre sont des objets Javascript, qui sont manipulés par *référence*. Il est ainsi possible d'*aliaser* un même objet plusieurs fois, ici en utilisant la même référence pour plusieurs noeuds de l'arbre. En C, l'équivalent consisterait à utiliser des pointeurs vers des structures en mémoire. Noter que la structure obtenue n'est plus un arbre, mais un DAG.
5. Il s'agit d'une *mémoïsation*. Un intérêt évident est la possibilité de réduire de manière importante le nombre de noeuds construits (rien que pour l'entier 42, la première construction en compte 18, et la seconde 5). Un autre intérêt consiste à disposer d'un *cache* des sous-arbres, qui permet à son tour d'optimiser la réutilisation des calculs précédents.

Le reste du code est fourni ci-après :

```
function makeIdd(n) {
    if (n === 0)
        return leaf(0);
    else if (n === 1)
        return leaf(1);
    else {
        const [th, tp, tl] = to_idd(n).map(makeIdd);
        return node(n, cons(th, cons(tp, cons(tl, nil))));
    }
}

const iddMem = {
    0: leaf(0),
    1: leaf(1),
};

function makeIddMem(n) {
    if (n in iddMem)
        return iddMem[n];
    else {
        const [th, tp, tl] = to_idd(n).map(makeIddMem);
        iddMem[n] = node(n, cons(th, cons(tp, cons(tl, nil))));
        return iddMem[n];
    }
}
```

Exercice 5: 5 points

✎ Considérons une interface graphique conçue pour éditer un document au format texte. Les opérations sur ce document se font sur un *buffer*, sur lequel on peut réaliser l'insertion d'un caractère, la suppression d'un caractère, et le remplacement d'un caractère par un autre. Pour les besoins de cette interface, il est aussi demandé de pouvoir gérer un historique des opérations réalisées, en permettant de revenir en arrière.

Rappeler la définition de ce qu'est un type abstrait de données. Proposer un ensemble de types abstraits de données permettant de représenter un tel projet. Expliquer en quoi la programmation fonctionnelle peut aider à sa réalisation.

En reprenant les définitions du cours, un *type abstrait de données* est une représentation d'un composant logiciel sous la forme d'un ensemble de valeurs, une représentation abstraite de cet ensemble de valeurs (interface), et une ou plusieurs implémentations de cette interface.
Dans le cas présent, il semble intéressant de disposer d'un type abstrait de données pour représenter le buffer de texte, et d'un type abstrait de données pour représenter l'historique des opérations.

```
type buffer; // a type representing the state of the text buffer

buffer_insert   : buffer * int * char → void // insert a char at position
buffer_replace  : buffer * int * char → char // replace a char at position
buffer_delete   : buffer * int        → char // delete a char at position
```

```
type op;       // a type representing an operation in the GUI
type history;  // a type representing the history

history_push   : history * op   → history      // push an operation in the history
history_revert : history * buffer → history // revert the top operation from the history
```

L'historique est ici une pile d'opérations qui peut être construite de manière fonctionnelle pure, permettant de garder en mémoire toutes les opérations réalisées jusqu'à présent, en profitant des propriétés de la *persistance*. Le buffer est lui modifié de manière impérative, pour garder la version courante du fichier édité. Le couple `[buffer, history]` constitue l'interface graphique, qui peut correspondre à au autre TAD.
Il est aussi envisageable de garder un historique complet des états du fichier édité (plutôt que simplement la liste des transformations), même si l'efficacité en mémoire est critiquable. On peut argumenter que l'historique doit être modifié à chaque nouvelle opération, ce qui peut se faire en modifiant l'historique courant (avec un effet de bord, de manière impérative) ou simplement en construisant un nouvel historique à partir de l'historique précédent (et donc de manière fonctionnelle).
Bonus à toute personne ayant insisté sur la possibilité d'implémentations multiples, pouvant être laissées au choix de la personne composant le code, et justifiant l'utilisation de l'abstraction et de la modularité (qui sont des qualités s'appliquant à tous les styles de programmation, pas seulement fonctionnelle).