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 code de manière à ce qu'il soit valide du point de vue de `node` :

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

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. 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 barème est donné à titre purement indicatif.
Les exercices sont tous *indépendants* les uns des autres.

Exercice 1: 3 points
✎ Considérons le code suivant, censé simuler la réception de données au format JSON et leur manipulation pour en extraire des données :

```
function fetch_() {
    return `{
        "user": "renault",
        "posts": [
            { "title": "the elementary nature of things",
              "contents": "...",
              "date": "15/06/22" },
            { "title": "simplicity: why not abuse of it ?",
              "contents": "...",
              "date": "17/06/22" }
        ]
    }`;
}

function get_(key) {
    return (e) => e[key];
}

function map_(fun) {
    return (a) => a.map(fun);
}

function do_(...args) {
    return args.reduce((cur, arg) => arg(cur), undefined);
}

do_(fetch_,
    JSON.parse,
    get_('posts'),
    map_(get_('date'))
); // → [ '15/06/22', '17/06/22' ]
```

Expliquer, en indiquant les numéros de ligne pour justifier, quelles techniques de programmation fonctionnelle ce code illustre.

✓ Les fonctions `get_` (l.15) et `map_` (l.19) sont des exemples de curryfication. Elles utilisent donc naturellement des fonctions anonymes, comme c'est le cas pour la fonction `do_` (l.23). Les fonctions `map_` (l.19) et `do_` (l.23) prennent des fonctions en paramètre et donc sont des fonctions d'ordre supérieur. La fonction `do_` (l.27-31) illustre une composition de fonctions.

Exercice 2: 3 points
⌨ Écrire une fonction `stringToInt` de manière *récursive terminale* et *pure* qui, étant donné une chaîne de caractères, produise la valeur de type `number` associé à cette chaîne de caractères. Cette fonction renvoie 0 sur une chaîne de caractères vide. Elle renvoie `undefined` si elle rencontre un caractère non numérique.

Indiquer en commentaire dans le code les appels récursifs terminaux.

Exemple :

```
stringToInt("");        // → 0
stringToInt("1234");    // → 1234
stringToInt("0.1");     // → undefined
```

Indication : la méthode `substring` de la classe `String` peut être utilisée pour obtenir une chaîne de caractères privée de son premier élément, la méthode `charCodeAt` pour obtenir le code ASCII d'un caractère.

✓ Une implémentation récursive terminale de cette fonction :

```
1  const charCode0 = "0".charCodeAt(0);
2
3  function stringToInt(aString) {
4      if (aString === "")
5          return 0;
6      else {
7          function stringToIntRec(aCpt, aString) {
8              if (aString === "")
9                  return aCpt;
10             else {
11                 const aTail = aString.substring(1);
12                 const aHead = aString.charCodeAt(0) - charCode0;
13                 if ((aHead < 0) || (aHead > 9))
14                     return undefined;
15                 else
16                     return stringToIntRec(aCpt * 10 + aHead, aTail);
17             }
18         }
19         return stringToIntRec(0, aString);
20     }
21 }
```

Parmi les éléments vérifiés (en plus de eslint et tsc) : si l'implémentation est impérative, si elle effectue des effets de bord, et si elle est récursive mais pas terminale. Les tests présents sont eux aussi pris en compte.
La fonction `stringToIntRec` donnée en correction est récursive terminale, son seul appel récursif se trouve l. 16. Parmi les éléments vérifiés (en plus de eslint et tsc) : si l'implémentation est impérative, si elle effectue des effets de bord, et si elle est simplement récursive non terminale.

Exercice 3: 5 points

Considérons un jeu de pierre-papier-ciseaux. Les trois possibilités du jeu sont codées par les entiers `0`, `1` et `2`. Le code suivant définit une fonction `whoWins` qui calcule le vainqueur suivant les coups des deux joueurs.

```
// ROCK = 0, PAPER = 1, SCISSORS = 2
const RULES = {
    0: { 0: -1, 1: 1,  2: 0  },
    1: { 0: 0,  1: -1, 2: 1  },
    2: { 0: 1,  1: 0,  2: -1 },
};
// Given two moves, return 0 if moveA wins, 1 if moveB wins, -1 otherwise
function whoWins(moveA, moveB) { return RULES[moveA][moveB]; }
```

Pour jouer avec ce code, nous proposons d'utiliser le générateur aléatoire (*random number generator*) vu en cours :

```
let seed = undefined;
const m = 65537; const a = 75;

// Initialize the RNG seed
function srand(aInit) {
    seed = aInit;
}

// Return a new pseudo-random value
function rand() {
    seed = (a * seed) % m;
    return seed;
};
```

Un joueur est représenté par une fonction `play` qui génère un coup possible. Par exemple :

```
function playFair() {
    return rand() % 3;
}
```

```
function playCheat() {
    return (seed+1) % 3;
}
```

Le code suivant (qu'il n'est pas nécessaire de comprendre précisément) représente un petit serveur de jeu, qui prend deux fonctions comme les précédentes, et les fait se rencontrer un certain nombre de fois :

```
function server(n, play0, play1) {
    function makePlayer(p) { return { play: p, score: 0 }; }
    srand(new Date().getSeconds());                          // Init the RNG
    const players = [ makePlayer(play0), makePlayer(play1) ]; // Create the players
    for (let i = 0; i < n; i++) {                            // Loop n times
        const moves  = players.map((p) => p.play());         //     Compute their moves
        const winner = whoWins(moves[0], moves[1]);          //     Compute the winner
        if (winner >= 0)                                     //     If there's a winner
            players[winner].score ++;                        //         Increase the winner's score
    }
    console.log(players);
}
```

1. ✎ Quelle est la portée, dans le code, des identificateurs suivants : `seed`, `players`, `winner` ?
2. ✎ Pour quelle raison, dans l'une des parties proposées dans le fichier, l'un des joueurs gagne-t'il systématiquement ? (donner une explication générale, et une explication concernant les problèmes propres au code)

Considérons une manière de modifier le code pour empêcher la situation précédente.

3. ⌨ Écrire une fonction `makeRng` qui prend en paramètre un entier (servant de graine initiale) et renvoie une fonction qui, à chaque fois qu'on l'appelle, renvoie des valeurs différentes (en utilisant l'algorithme de génération de nombre aléatoires ci-dessus).

   De manière importante, il ne doit pas être possible d'accéder à la graine de ce générateur aléatoire depuis l'extérieur de la fonction.

Exemple d'utilisation :

```
const aRng = makeRng(12);
[1,2,3,4,5].forEach((i) => console.log(aRng())); // Displays : 900 1963 16151 31659 15093
```

4. ⌨ Écrire une fonction `makePlayFairSafe` qui ne prenne pas de paramètres, et renvoie un joueur (au même titre que `playFair` et `playCheat`) utilisant son propre générateur aléatoire produit par `makeRng` pour produire des coups valides.

   Tester cette fonction avec `server`, et montrer que le joueur `playCheat` ne gagne plus systématiquement.

5. ✎ Quelle propriété de la programmation fonctionnelle utilise t'on principalement dans ces deux dernières fonctions ? Justifier.

✓

1. `seed` a une portée globale, à partir de son point de définition (l.9-39). `players` est accessible dans l'ensemble de la fonction `server`, à partir de son point de définition (l.30-39). `winner` est accessible dans la boucle `for` de la fonction `server` (l.34-37).
2. Le joueur `playCheat` triche en renvoyant un résultat dépendant directement de la variable `seed`. Si il joue juste après un autre joueur faisant appel à `rand`, il peut estimer le coup qu'il a joué, et donc jouer un coup qui lui est favorable.

   En terme de code, la variable `seed` est accessible aux deux joueurs. Pire, les deux joueurs partagent le même générateur aléatoire. Cela permet à l'un des joueurs d'analyser ce générateur pour tenter de deviner des informations concernant le code de son adversaire.

   Comme solution, il faudrait que les deux joueurs ne puissent accéder aux données aléatoires de l'autre. Le mieux serait donc de fournir un générateur aléatoire pour chaque joueur.

5. Si l'on ne fait que considérer les deux fonctions à écrire, on utilise principalement la *propriété de 1ère classe* des fonctions, en particulier celle qui permet à des fonctions de renvoyer des fonctions. Il est aussi possible de voir le fait de représenter des joueurs et des générateurs aléatoires par des fonction comme de la *représentation des données par des fonctions*.

   Si l'on considère l'ensemble de l'exercice en général (ce qui n'était pas la question), alors il met en valeur des problèmes d'encapsulation des données et donc de modularité, en proposant de rendre les variables comme `seed` privées pour que les joueurs ne la partagent pas. Noter que la modularité n'est pas une propriété propre à la programmation fonctionnelle, et que parler de modularité dans cette question pourrait s'appliquer aussi bien dans un cadre impératif.

Le reste du code est fourni ci-après :

```
function makeRng(aInit) {
    let seed = aInit;
    const m = 65537; const a = 75;
    return () => {
        seed = (a * seed) % m;
        return seed;
    };
}

const aRng = makeRng(12);
[1,2,3,4,5].forEach((i) => console.log(aRng()));

function makePlayFairSafe() {
    let aRng = makeRng(12);
    return () => aRng() % 3;
}
const playFairSafe = makePlayFairSafe();

// The following code shows that playFairSafe can win against playCheat
server(10, playFairSafe, playCheat);
```

Exercice 4: 5 points

Considérons un ensemble de fonctions permettant de manipuler des listes et des arbres, comme fourni ci-après :

```
function cons(_car, _cdr) { return { car: _car, cdr: _cdr }; }
const nil = {};
function car(cons) { return cons['car']; }
function cdr(cons) { return cons['cdr']; }

// Functions on lists
function head(l)    { return car(l); }
function tail(l)    { return cdr(l); }
function isEmpty(l) { return l === nil; }

function node(_val, _children) {
    return { val: _val, children: _children };
}
function val(tree)      { return tree['val']; }
function children(tree) { return tree['children']; }
function leaf(_val)     { return node(_val, nil); }
```

Les listes sont des listes chaînées classiques, et les arbres sont des noeuds possédant une valeur (`val`) ainsi qu'une liste potentiellement vide de fils (`children`), chacun de ces fils étant aussi un arbre. Sont fournies avec le code (entre autres) une fonction `treeDisp` permettant d'afficher un arbre, et une fonction `makeIdd` permettant de construire un arbre à partir d'une valeur entière [1].

1. ⌨ Écrire une fonction `treeSum` qui, étant donné un arbre, renvoie la somme des valeurs des noeuds de cet arbre.
2. ⌨ Écrire une fonction `compareTrees` qui, étant donné deux arbres, renvoie un booléen affirmant si les deux arbres sont égaux (même nombre de fils à chaque noeud, même valeur sur les sommets dans le même ordre).
3. ⌨ Écrire une fonction `treeForEach` qui soit l'équivalent de la fonction `listForEach` (présente dans le fichier `lists.js`) pour les arbres.

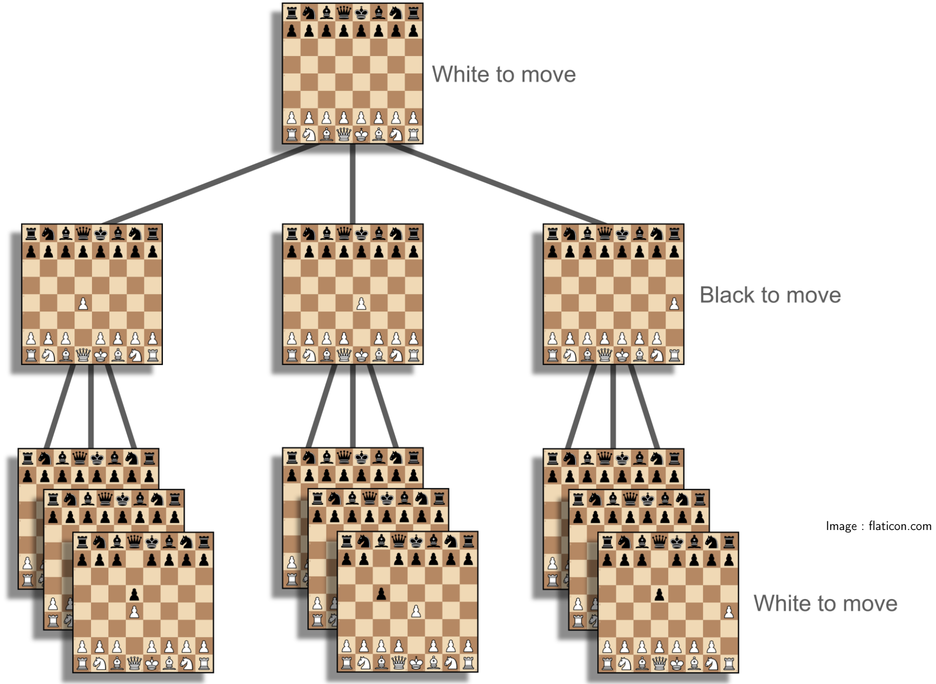
Remarque : dans tous les cas, il est conseillé d'écrire une fonction sur les *listes d'arbres* en même temps que la fonction à réaliser.

Exercice 5: 4 points

Sur un jeu comme les échecs, étant donné une position de jeu, l'arbre des positions est défini comme l'arbre dont les noeuds sont des positions de jeu, et les fils d'un noeud v sont les positions accessibles depuis v pour le joueur en cours.

---

1. Se référer à l'examen de 2022 session 1 pour plus de détails sur cette construction. Ces détails ne sont pas nécessaires pour la réalisation de cet exercice.



En s'appuyant sur cet exemple, donner une définition de ce qu'est l'*évaluation paresseuse*. En quoi cette technique se marie t'elle bien avec la programmation fonctionnelle ?

✓ L'évaluation paresseuse est une *stratégie d'évaluation* dans laquelle on *repousse* le plus tard possible l'évaluation des calculs. Prenons l'exemple du jeu d'échecs, où un joueur décide de chercher une position gagnante à partir de la position courante. Il peut construire l'arbre des positions à partir de ce point de départ, mais le construire en entier est irréalisable du fait du nombre de positions possibles. Il va donc construire cet arbre paresseusement, en ne calculant pas tous ses noeuds (par exemple en utilisant la technique de congélation). Ainsi, il pourra regarder l'arbre à profondeur 2, ou 3, ou n ... suivant le temps dont il dispose pour se décider. Les calculs construisant cet arbre sont repoussés au moment où le joueur en a besoin. La programmation paresseuse assure aussi qu'une fois les calculs réalisés, ils sont *mémorisés* ce qui permet de ne pas les refaire deux fois. Cela lui permet par exemple, après avoir calculé l'arbre à profondeur 2, de décider dans un second temps de le calculer à profondeur 3.

L'évaluation paresseuse ne peut s'effectuer que dans des conditions où les calculs ne dépendent pas du moment où ils sont évalués (sous peine de ne plus être déterministe). C'est en particulier le cas lorsque les fonctions réalisant ces calculs sont *pures*, et donc *référentiellement transparentes*. C'est tout à fait le cas de notre jeu d'échecs, où le calcul des positions accessibles à partir d'une position donnée est une fonction pure.

L'exemple du jeu d'échecs montre l'utilisation d'une *structure de données paresseuse* (sous la forme d'un arbre partiellement calculé). Une telle structure congèle (i.e n'évalue pas) une partie de son contenu, histoire de pouvoir repousser son évalution. En cela, il s'agit d'une technique de *contrôle de l'évaluation*.