Année **2022-2023** Session **1**

Programmation fonctionnelle
PG104

Filière : Informatique, 1ère Année

Date de l'examen : 25/05/2023 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[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 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[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 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 barême est donné à titre purement indicatif.
Les exercices sont tous *indépendants* les uns des autres.

Exercice 1: 2 points

✎ Quelles sont les propriétés des valeurs dites "citoyennes de 1ère classe" en programmation ? Donner un exemple de valeur qui *n'est pas* de 1ère classe dans un langage de votre choix, en justifiant.

✓ Selon le cours, une construction d'un langage de programmation est dite citoyenne de 1ère classe si le langage l'autorise à :

- être nommée et affectée dans une variable ;
- être créée à la demande dans n'importe quel contexte ;
- être passée comme argument à une fonction ;
- être retournée comme résultat d'une fonction ;
- apparaître dans n'importe quelle structure de données.

Le cours donne l'exemple des fonctions en C qui ne sont pas de première classe : elles ne peuvent pas être créées dans *n'importe quel* contexte, à l'opposé des lambdas en Javascript). De même, les classes en C++ ne peuvent pas être passées en paramètre ou retournées par des fonctions (au contraire des objets).

Exercice 2: 5 points

Lorsque l'on passe les arguments en ligne de commande à un programme, la liste des arguments est représentée par un tableau de chaînes de caractères. Dans une bibliothèque comme `getopt` (cf. `man 3 getopt`), on distingue les arguments *optionnels* (i.e les arguments débutant par un tiret "`-`") des arguments *obligatoires* (i.e tous les autres).

1. ⌨ Écrire une fonction `orderOpts` de manière *récursive terminale* et *pure* qui, étant donnée un tableau de chaînes de caractères `anArray`, renvoie un nouveau tableau dans lequel les paramètres optionnels sont placés au début, et les paramètres obligatoires sont placés à la fin. L'ordre des paramètres optionnels (resp. obligatoires) ne doit pas être modifié.

   Exemple :

```
orderOpts([]);                                    // → []
orderOpts(["./server", "play1.so", "play2.so"]);  // → ["./server", "play1.so", "play2.so"]
orderOpts(["./server", "--random", "play1.so"]);  // → ["--random", "./server", "play1.so"]
orderOpts(["./server", "--random", "--automatic"]);// → ["--random", "--automatic", "./server"]
```

2. ⌨ Écrire une fonction `makeGetOpts` qui, étant donné un tableau de chaîne de caractères `anArray`, renvoie une fonction `getopt`. La fonction `getopt`, à chaque fois qu'elle est appelée, doit renvoyer l'un des éléments de `anArray` dans l'ordre, et `undefined` si tous les éléments ont été renvoyés.

   Exemple :

```
const getopt = makeGetOpts(["a", "b", "c"]);
getopt() // → "a"
getopt() // → "b"
getopt() // → "c"
getopt() // → undefined
```

3. ✎ Est-il possible de faire que `makeGetOpts` soit pure ? Justifier.

✓

1. Une implémentation possible de `orderOpts` :

```
// 'orderOpts' takes an array 'anArray' of non-empty strings and
// returns a newarray where the optional parameters are first and the
// mandatory parameters are last. It is tail-recursive and pure.
function orderOpts(anArray) {
    function parseRec(anArrayToVisit, aMandatoryArray, anOptionArray) {
        if (anArrayToVisit.length === 0)
            return anOptionArray.concat(aMandatoryArray);
        else {
            const aHead = anArrayToVisit[0];
            const aTail = anArrayToVisit.slice(1);
            if (aHead[0] === '-')
                return parseRec(aTail, aMandatoryArray,
                                anOptionArray.concat(aHead));
            else
                return parseRec(aTail, aMandatoryArray.concat(aHead),
                                anOptionArray);
        }
    }
    return parseRec(anArray, [], []);
}
```

2. Une implémentation possible de `makeGetOpts` :

```
// 'makeGetOpts' takes an array 'anArray' of strings and returns a
// function that, when called, returns the elements inside 'anArray'
// one by one in order.
function makeGetOpts(anArray) {
    let aCursor = anArray;
    const getopt = () => {
        let aHead = aCursor[0];
        aCursor = aCursor.slice(1);
        return aHead;
    }
    return getopt;
}
```

3. La fonction renvoyée par `makeGetOpts` n'est par définition pas référentiellement transparente : elle ne renvoie pas les mêmes résultats sur les mêmes entrées. Elle fait donc nécessairement un effet de bord, et est donc impure.

   Noter que si l'on se permet de modifier la spécification de la fonction (et donc de répondre à une autre question), on peut rendre la fonction pure, par exemple dans la réponse suivante :

```
function makeGetOptsPure(anArray) {
    function getoptPure(num) {
        if (num < anArray.length) {
            return [() => getoptPure(num + 1), anArray[num]];
        }
        return [() => getoptPure(num + 1), undefined];
    }
    return () => getoptPure(0);
}
```

Exercice 3: 4 points

Considérons le code suivant censé gérer l'affichage d'un ensemble de parties d'un jeu de société quelconque :

```
const games = { list: [], index: undefined };

function makeGame(player0, player1, winner) {
    return { player0: player0, player1: player1, winner: winner };
}

function parseGames() {
    return [
        makeGame("Hercule", "Hydra",          "Hercule" ),
        makeGame("David",   "Goliath",        "David" ),
        makeGame("Thor",    "The Destroyer", "Thor" ),
    ];
}

function initGames() {
    games.list = parseGames();
    games.index = 0;
}

function currentGame() {
    return games.list[games.index];
}

function goToNextGame() {
    if (games.index+1 < games.list.length)
        games.index += 1;
}

function displayGame(aGame) {
    console.log(`What a win for ${aGame.winner} !`);
}

initGames();
console.log(currentGame());
goToNextGame();
console.log(currentGame());
```

1. ✎ Dans ce code, quelles sont les fonctions pures et les fonctions impures ?
2. ✎ En quoi est-ce que la disposition de la variable `games` est critiquable dans ce code ? Quelle est la caractéristique de cette variable qui pose problème ?
3. ⌨ Modifier le code en utilisant des techniques de programmation fonctionnelle pour atténuer ces critiques (n.b : aucune forme de programmation objet ou modulaire n'est autorisée pour cette question)

✓

1. Techniquement, aucune de ces fonctions n'est pure :
   - Les fonctions `makeGame` et `parseGames` renvoient un objet, qui est différent à chaque appel. Elles sont impures. Néanmoins, les réponses associées ont été ignorées.
   - Les fonctions `initGames` et `goToNextGame` sont impures, car elles font des effets de bords sur `games`.
   - La fonction `currentGame` est impure, car elle est dépendante de `games` qui lui est externe (à comparer à une fonction comme `rand`)
   - La fonction `displayGame` est impure car elle fait un effet de bord en réalisant un affichage sur la console.

   La correction a compensé les réponses fausses lorsque les réponses justes étaient justifiées.
2. La variable `games` est une variable de *portée* globale. Cela lui permet d'être accessible de partout, mais cela autorise aussi n'importe quelle fonction à y accéder. Cette critique est aggravée par le fait que son contenu est modifiable (malgré le `const`).

   C'est critiquable, car cela demande de la méthode de la part du programmeur pour ne pas la lire et la modifier indûment. Par exemple, il est tout à fait possible de modifier le champe `list` de `games` sans modifier `index`, amenant à un état douteux. Plus généralement, le côté global empêche le compilateur de réaliser des optimisations sur le code, ou de gérer deux listes de parties en même temps.
3. Une technique fonctionnelle allégeant les critiques précédentes consiste à limiter le partage de la variable `games` aux seules fonctions qui ont besoin de la manipuler.

```
function makeGame(player0, player1, winner) {
    return { player0: player0, player1: player1, winner: winner };
}

function parseGames() { return [ makeGame("Hercule", "Hydra",          "Hercule" ),
                                 makeGame("David",   "Goliath",        "David" ),
                                 makeGame("Thor",    "The␣Destroyer", "Thor" ),  ];}

function displayGame(aGame) { console.log(`What␣a␣win␣for␣${aGame.winner}␣!`); }

function setupGames() {
    const games = { list: [], index: undefined };
    function initGames()    {
        games.list = parseGames(); games.index = 0; }
    function currentGame()  {
        return games.list[games.index]; }
    function goToNextGame() {
        if (games.index+1 < games.list.length)
            games.index += 1; }
    return { initGames: initGames, currentGame: currentGame,
             goToNextGame: goToNextGame };
}

const { initGames, currentGame, goToNextGame } = setupGames();
initGames();
console.log(currentGame());
goToNextGame();
console.log(currentGame());
```

Noter que cette technique est proche des techniques de programmation modulaire. Il serait aussi envisageable de rendre simplement les fonctions gênantes pures en leur faisant prendre la variable `games` en paramètre, mais en soi, cette solution n'atténue pas le fait que la variable reste globale.

Exercice 4: 4 points

Considérons des fonctions de type `number → number`, qui prenne des nombres en paramètres et renvoient des nombres comme résultats. On se limite ici aux fonctions définies sur toute entrée, et ne générant pas d'erreur. Par exemple la fonction suivante :

```
function sum(n) {
    if (n >= 1)
        return n*n + sum(n-1);
    else
        return 0;
}

[1,2,3,4,5].map(sum); // → [ 1, 5, 14, 30, 55 ]
```

L'exercice porte sur la possibilité d'ajouter un cache à ce type de fonction de manière générique, afin de mémoriser les calculs déjà effectués (technique de *mémoïsation*)[1].

1. ⌨ Écrire une fonction `memoize` qui prenne en paramètre une fonction `aFun` de la forme précédente, et renvoie un dictionnaire contenant deux entrées :
   - l'entrée `fun` est une fonction renvoyant les mêmes valeurs que la fonction `aFun` sur les mêmes entrées ;
   - l'entrée `cache` est un dictionnaire dont les clés sont des entrées pour `aFun` et les valeurs sont les sorties de `aFun` sur ces entrées.

   De plus, à chaque appel, la fonction `fun` doit stocker les calculs qu'elle réalise dans `cache` et les réutiliser autant que possible si ils s'y trouvent déjà.

   Exemple :

   ```
   let aDict = memoize(sum);
   aDict.fun(3); console.log(aDict.cache); // { '3': 14 }
   aDict.fun(5); console.log(aDict.cache); // { '3': 14, '5': 55 }
   aDict.fun(5); console.log(aDict.cache); // { '3': 14, '5': 55 } and no recomputation of sum(5)
   ```

2. ✎ Cette technique n'est pas très efficace pour gérer les fonctions récursives comme l'exemple `sum`. Pourquoi ? D'où vient le problème ?

Afin de rémédier au problème précédent, on propose une nouvelle manière d'écrire la fonction `sum`, en éliminant l'appel récursif et le remplaçant par un paramètre `next` (l'expression utilisée est "untying the recursive knot" ou "dénouer le noeud récursif") :

```
const sumOne = (next) ⇒
      (n) ⇒ {
          if (n >= 1)
              return n*n + next(n-1);
          else
              return 0;
      };
```

1. Ces questions sont inspirées du chapitre 8 du livre *Real World OCaml* de Y. Minsky et A. Madhavapeddy, portant sur la mémoïsation, https://dev.realworldocaml.org/imperative-programming.html#memoization-and-dynamic-programming

3. ✎ Quelle est la technique de programmation fonctionnelle utilisée dans l'écriture de cette fonction ?
4. ⌨ Écrire une fonction `sumTwo` qui utilise `sumOne`, et produit les mêmes résultats que `sum` sur les mêmes entrées.

Le code suivant permet de construire une version avec cache de la fonction `sum` :

```
let nf = undefined;
const sumThree = (n) ⇒ sumOne(nf)(n);
const anotherDict = memoize(sumThree);
nf = anotherDict.fun;
[5,4,3,2,1].map(anotherDict.fun);  // → [ 55, 30, 14, 5, 1 ]
```

5. ✎ Pourquoi cette fonction résout-elle le problème évoqué précédemment ? En quoi le fait de dénouer ce fameux noeud récursif a t'il aidé ?

✓

1. Une implémentation possible de `memoize` :

```
function memoize(f) {
    let cache = {}
    function new_f(n) {
        if (!cache[n])
            cache[n] = f(n);
        return cache[n];
    }
    return { fun: new_f, cache: cache };
}
```

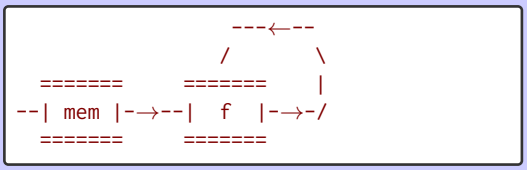
2. Cette technique n'est pas efficace pour mettre en cache les fonctions récursives comme `sum`. L'exemple montre en effet que les valeurs intermédiaires calculées dans `sum` ne sont pas mises en cache. Cela vient du fait que, même si on a mémoïsé la fonction `sum`, ses appels récursifs se font sur la version non mémoïsée, et donc ne passent pas par le cache.
3. Cette fonction est écrite de manière *curryfiée*. Le fait de rajouter un paramètre fonctionnel ici pourrait aussi être vu comme une généralisation fonctionnelle, mais c'est une généralisation biaisée, parce que la fonction `next` doit dans son calcul faire un appel récursif à `sumOne` pour compléter le calcul. Cela peut aussi être vu comme une forme de contrôle de l'évaluation, mais dans ce cas il faut justifier que c'est l'appel récursif qui est contrôlé (alors qu'usuellement on contrôle avec la curryfication le passage des paramètres).
4. Il suffit de "renouer" le noeud récursif :

```
const sumTwo = (n) ⇒ sumOne(sumTwo)(n);
```

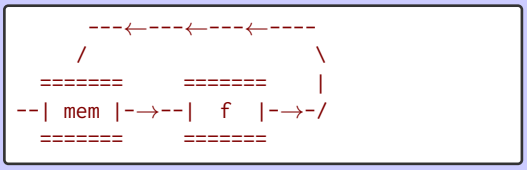
✓ 5. Une fonction récursive f s'appelle elle-même dans son code, ce qu'on peut représenter de la manière suivante :

On appelle cela un *noeud* (*knot* en anglais), même si c'est plus facilement visualisable comme une boucle. S'il est "fameux", c'est que c'est une représentation connue de la caractéristique essentielle d'une fonction récursive : s'appeler elle-même.

Si on veut appliquer une fonction *avant* tous les appels de `f` (comme pour la mémoïsation), celle-ci ne peut pas intercepter les appels récursifs internes. Cela correspond au dessin suivant (la mémoïsation est ici symbolisée par une fonction `mem` qui représente la mise en cache des appels) :

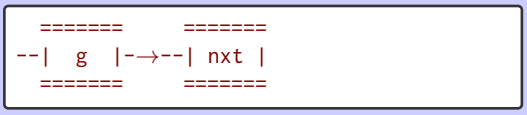


Et donc, dans l'exemple de la fonction `sum` fournie dans l'énoncé, on mémoïse bien le premier appel, mais pas les appels récursifs effectués par `sum`. Alors qu'en fait, ce qu'on voudrait, ce serait ceci :



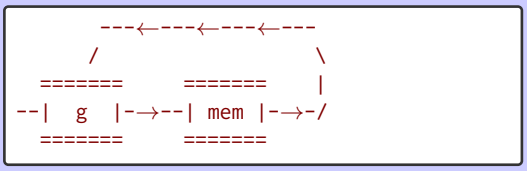
Mais cela n'est pas possible sans changer le code de `f`.

La transformation que l'on effectue dans l'exercice consiste à faire que l'appel récursif de la fonction (ici renommée `g`) soit transmis à une fonction `nxt` non spécifiée :



`g` n'est alors pas récursive, on a *défait le noeud*. Mais si on impose que `nxt` soit la fonction `g`, alors, on a le même comportement que le comportement initial.

Et comme on peut maintenant spécifier `nxt` comme on le désire, on peut faire que `nxt` appelle d'abord `mem`, *puis* la fonction `g` :



On a *dénoué* le noeud, pour permettre de le *renouer* ensuite. Et obtenu le résultat escompté en mémoïsant tous les appels à `g`.

Exercice 5: 5 points

Le code suivant en C++ permet de dessiner l'ensemble de Mandelbrot, en réalisant un calcul parallèle[2]. Chaque pixel de l'image possède une couleur, qui est liée au nombre d'itérations à l'intérieur d'une boucle `while`. Le parallèlisme est mis en oeuvre ici par la bibliothèque OpenMP à travers des directives `#pragma`. Cette directive va conduire à exécuter les lignes 12-14 en parallèle, dans des fils d'exécution distincts. Le résultat du calcul est mis dans une variable `result` contenant l'ensemble des pixels de l'image, variable qui est partagée par tous les fils d'exécution (cf. le mot-clé `shared`).

Remarque : il n'est pas nécessaire de comprendre l'entièreté du code pour réaliser cet exercice, simplement d'en dégager les idées générales.

```
1  int compute_color(complex c) {
2    complex z = 0;
3    int nb_iter = 0; // Number of iterations === Color of the pixel
4    while (norm(z) <= 4 && nb_iter < iter_limit) {
5      z = z*z + c; nb_iter++;
6    }
7    return nb_iter;
8  }
9
10 #pragma omp parallel shared(result)
11 for (int i = 0; i < x_resolution * y_resolution; i++) {
12   complex c = complex(x_begin + (i % x_resolution) * x_step,
13                       y_begin + (i / x_resolution) * y_step);
14   result[i] = compute_color(c);
15 }
```

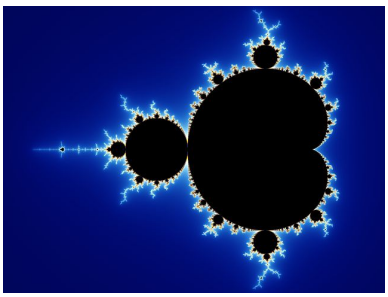


Image : Wikipedia

La documentation de la bibliothèque OpenMP affirme que pour pouvoir réaliser correctement un calcul parallèle :

> "Pour utiliser la directive `parallel`, le programme ne doit pas dépendre de l'ordre des évaluations des clauses, ou d'effets de bord dans les évaluations des clauses."[3]

✎ Rappeler les principes généraux de la programmation pure. En vous appuyant sur l'exemple ci-dessus, expliquer dans quelle mesure la programmation pure et le bon contrôle de l'indépendance des calculs sont des qualités majeures pour pouvoir écrire du code parallèle.

---

2. Il s'agit d'une version très simplifiée du code utilisé dans les TDs de DDRS sur les analyses de cycles de vie.

3. "A program must not depend on any ordering of the evaluations of the clauses of the parallel directive, or on any side effects of the evaluations of the clauses.", cf. https://www.openmp.org/spec-html/5.0/openmpse14.html

✓ Selon le cours, le principe général de la programmation pure est l'idée d'écrire du code en minimisant les dépendances entre les diverses entités, afin de mieux les contrôler. Cela peut se faire en utilisant des fonctions *pures*, ressemblant aux fonctions mathématiques : des fonctions qui produisent les mêmes résultats sur les mêmes entrées, sans effets de bord. Mais ce n'est pas indispensable.

Lorsqu'il s'agit d'écrire du code qui doit être exécuté en parallèle, la programmation pure joue un rôle fondamental : chaque calcul parallèle doit bien maîtriser ses dépendances afin de ne pas interférer avec les autres calculs. En effet, le côté "parallèle" empêche d'avoir une quelconque maîtrise sur l'ordre dans lequel ils sont réalisés.

C'est le cas pour la bibliothèque OpenMP, pour laquelle la documentation insiste sur le fait que le code parallèlisé ne doit pas dépendre de l'ordre d'évaluation de chaque calcul, ni d'effets de bords dans les calculs. Typiquement, si les fonctions parallèlisées sont pures, cette condition est automatiquement réalisée. Mais la condition est aussi valide si les calculs sont simplement indépendants.

L'exemple donné, pour l'ensemble de Mandelbrot en est une illustraction : le calcul de la couleur de chaque pixel (fait dans la fonction `compute_color`) est indépendant du calcul fait pour tous les autres. La boucle effectue des effets de bords à la ligne 5, mais ceux-ci ne touchent que des variables internes au calcul (ici `z` et `nbiter`). Ils sont donc bien parallèlisables. Noter que la fonction `compute_color` n'est *pas* pure car elle fait des effets de bords dans son calcul interne (e.g l.5). Par contre, elle *est* référentiellement transparente, i.e elle fournira toujours le même résultat sur les mêmes entrées.

Remarquer enfin que dans cet exemple, chacun des calculs parallèles stocke le résultat de son calcul dans une variable `result`, qui est partagée (cf. le mot-clé `shared` à la ligne 10). Les dépendances entre les calculs sont ici bien contrôlées, car seule l'écriture dans cette variable (l.14) est un frein au parallèlisme (par opposition au calcul de la valeur à écrire).