Année 2020-2021 Session 2

PROGRAMMATION FONCTIONNELLE
PG104

Filière : Informatique, 1ère Année

Date de l'examen : 28/06/2021 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 programmation** Les réponses demandent d'écrire du code Ecmascript valide. La syntaxe ne sera pas respectée à la lettre, mais les standards de codage utilisés seront vérifiés. Il vaut toujours mieux justifier son code par des commentaires.

— **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. Toute forme d'imprécision sera sanctionnée.

Le barême est donné à titre purement indicatif.

Exercice 1: 4 points

Soit f une fonction définie sur les nombres réels et renvoyant des valeurs réelles.

1. Écrire un algorithme calculant la somme suivante pour $n \in \mathbb{N}$:

$$\sum_{i=0}^{2^n-1} f(i)$$

en utilisant une méthode récursive. La fonction obtenue ne doit pas faire d'effets de bords, ni dupliquer de code.

2. Expliquer comment utiliser la fonction précédente pour calculer la somme suivante :

$$\frac{1}{2^n} \sum_{i=0}^{2^n-1} f\left(a + (b-a)\frac{i}{2^n}\right) \qquad a, b \in \mathbb{R}$$

(qui se trouve être une somme de Riemann, et donc servir entre autres choses à l'approximation d'intégrales, mais cela n'a aucune influence sur la question)

✓

1. Il s'agit d'un problème récursif assez classique, dans lequel on peut découper l'intervalle en deux, et sommer récursivement sur les sous-intervalles.

```
function sumSimple(f, n) {
    function sumSimpleRec(f, [a, b], n) {
        if (n === 0)
            return f(a);
        else {
            const mid = (a+b)/2;
            return sumSimpleRec(f, [a, mid], n-1) +
                sumSimpleRec(f, [mid, b], n-1);
        }
    }
    return sumSimpleRec(f, [0, 2**n], n);
}
```

   Il était aussi possible de faire une somme avec un seul appel récursif.

2. Il s'agit d'une spécialisation de la fonction précédente :

```
function sumRiemann(f, [a, b], n) {
    const m = 2**n;
    const g = (x) ⇒ f(a + (b-a)*x/m);
    return sumSimple(g, n) / m;
  }
```

   Noter le calcul externe de la constante `2**n`, qui n'est pas recalculée à chaque appel de la fonction `g`. Un compilateur intelligent pourrait faire cette optimisation de lui-même, mais il est préférable de la faire au moment de l'écriture.

Exercice 2: 6 points

Dans cet exercice, on se propose de comparer l'utilisation de boucles `for` comme dans l'exemple suivant à des techniques utilisant la méthode `reduce`.

```
const anArray = [1, 2, 3];
var aResult = 10;
for (var i = 0; i < anArray.length; i++)
  aResult = aResult + anArray[i];
aResult; // → 16 = 10 + 1 + 2 + 3
```

A titre d'information, voici un extrait de la documentation de la méthode `reduce` sur les tableaux en Ecmascript [1] :

```
anArray.reduce(callbackFn)                                  // simple form
anArray.reduce(callbackFn, initialValue)                    // with en initial value
anArray.reduce((accumulator, currentValue) ⇒ {...}, initialValue) // same with the lambda expanded
```

1. Documentation simplifiée à partir de la page `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/Reduce`

`callbackFn` A function to execute on each element in the array.

It takes two arguments :

`accumulator` The accumulator accumulates `callbackFn`'s return values. It is the accumulated value previously returned in the last invocation of the callback — or `initialValue`, if it was supplied (see below).

`currentValue` The current element being processed in the array.

`initialValue` A value to use as the first argument to the first call of the `callbackFn`.

Un exemple d'utilisation de cette fonction pour produire le même résultat que pour la boucle `for` précédente :

```
[ 1, 2, 3 ].reduce(
  ( accumulator, currentValue ) ⇒ accumulator + currentValue,
  10
); // → 16 = 10 + 1 + 2 + 3
```

1. En quoi l'utilisation d'une méthode comme `reduce` d'une part et d'une boucle `for` parcourant les éléments d'un tableau d'autre part sont-ils comparables ?
2. En quoi l'utilisation de la méthode `reduce` peut-elle être qualifiée de "fonctionnelle" ? *Attention :* il y a potentiellement deux aspects remarquables ici.

Un programmeur a écrit la fonction suivante qui permet de compter les caractères présents dans une chaîne de caractères :

```
function stringToHash(aString) {
    const anArray = Array.from(aString); // converts aString into an array of chars
    const aHash = {};
    for (var i = 0; i < anArray.length; i++) {
        if (aHash[anArray[i]])
            aHash[anArray[i]]++;
        else
            aHash[anArray[i]] = 1;
    }
    return aHash;
}

stringToHash("abacaba");                // → { a: 4, b: 2, c: 1 }
stringToHash("abbaestungroupeenpoupe"); // → { a: 2, b: 2, e: 4, s: 1, t: 1, u: 3, n: 2, g: 1, r: 1, o: 2, p: 3 }
```

3. Écrire une version de cette fonction utilisant un `reduce` au lieu de la boucle `for`.
4. Est-ce que la fonction que vous avez écrite à la question précédente est pure ? Justifier.

✓

1. Les deux constructions remplissent un même rôle algorithmique, à savoir parcourir les éléments d'un tableau un à un, et construire une valeur finale qui soit le résultat de la recontre de chacun de ces éléments en séquence. Les deux exemples fournis montrent cette équivalence, qui peut se généraliser à tout parcours. Il était possible de répondre à cette question en faisant un comparatif des différences entre les deux (ce qui risque d'être redondant avec la question suivante), mais ce n'est pas logique de le faire sans parler de leurs ressemblances.
2. La méthode `reduce` est fonctionnelle à au moins deux titres :

   Le premier titre consiste à remarquer qu'il s'agit d'une *fonction d'ordre supérieur*, prenant une autre fonction en paramètre, et donc utilisant les propriétés de *la première classe* des fonctions en Ecmascript.

   Le deuxième titre tient à l'élimination (au moins dans l'exemple) des effets de bords sur les variables comme `i` et `aResult`. L'accumulateur peut ainsi être une fonction *pure* et profiter des propriétés qui découlent de l'utilisation de techniques sans effets de bords (indépendance des calculs, mémoïsation ...)
3. La fonction suivante répond à la question :

   ```
   function stringToHash_reduce(s) {
       return Array.from(s).reduce((acc, el) => {
           if (acc[el])
               acc[el]++
           else
               acc[el] = 1;
           return acc;
       }, {});
   }
   ```
4. La réponse dépend forcément de la fonction écrite. Néanmoins, si l'on utilise un dictionnaire pour stocker les résultats comme dans la réponse ci-dessus, on effectue des effets de bords sur ce dictionnaire au long du parcours. La fonction n'est donc pas complètement pure. A titre d'explication supplémentaire, il s'agit d'un exemple d'utilisation de `reduce` dans un cadre impur. Il peut se justifier en argumentant que ces effets de bords sont restreints à l'intérieur de la fonction `stringToHash_reduce`, et donc n'ont pas d'influence sur le reste de l'exécution du programme.

   Toujours dans les explications supplémentaires, il est envsageable de proposer une implémentation pure de cette fonction en utilisant un dictionnaire immutable, comme le propose par exemple la bibliothèque Immutable.js.

Exercice 3: 6 points

La fonction `fiboNext` permet de calculer les nombres de la suite de Fibonacci :

```
1  const fiboNext = function () {
2      let aPair = [ 1, 1 ];
3      return function() {
4          let prev = aPair[0];
5          aPair[0] = aPair[1];
6          aPair[1] += prev;
7          return prev;
8      };
9  }();
10
11 for (let i = 0; i < 10; i++) // This outputs the 10 first numbers of the Fibonacci sequence
12     console.log(fiboNext()); // Output : 1 1 2 3 5 8 13 21 34 55
```

1. Donner un type précis pour `fiboNext`.
2. La variable `aPair` (l.2) est manifestement définie à l'extérieur de la fonction stockée dans `fiboNext` (l.3 à 8). Et pourtant, `fiboNext` peut continuer à s'exécuter sans problèmes (l.12). Quel est le nom du mécanisme permettant cela ?

La bibliothèque lazy.js (`https://danieltao.com/lazy.js`) est une bibliothèque de fonctions utilitaires en Javascript qui contient une fonction `generate` permettant de réaliser la chose suivante :

```
const L = require("lazy.js");
// Insert code of fiboNext (l.1 to 9 of the previous listing) here
const fiboLazy = L.generate(fiboNext);

fiboLazy.take(5).toArray(); // → [ 1,   1,   2,   3,   5 ]
fiboLazy.take(5).toArray(); // → [ 8,   13,  21,  34,  55 ]
fiboLazy.take(5).toArray(); // → [ 89, 144, 233, 377, 610 ]
```

La documentation de cette bibliothèque prétend qu'elle essaie de faire le minimum de travail nécessaire, tout en étant aussi flexible que possible. Le comportement ci-dessus est le comportement attendu dans cette bibliothèque et pas une erreur.

3. A quelle technique de programmation fonctionnelle se réfère le mot "*lazy*" ?
   A votre avis, qu'est-ce qui peut-être "*lazy*" dans le calcul précédent ?
4. Les 3 derniers appels de fonction sont identiques et pourtant ne donnent pas le même résultat. Pourriez-vous expliquer de quoi ce comportement est-il symptomatique ?

✓

1. `fiboNext` est une fonction qui ne prend pas de paramètres, et qui renvoie une valeur de type numérique `number`. La chose est évidente quand on lit son utilisation l.12.
2. Cette variable est définie l.2, et utilisée dans la fonction apparaissant entre les lignes 3 et 8, fonction qui est au final stockée dans `fiboNext`. Le mécanisme de *fermeture* fait que cette fonction peut encore accéder à son environnement, même s'il a été défini à l'extérieur à son code. Noter que la simple notion de portée lexicale ne répond pas à la question, `aPair` n'étant plus accessible (du point de vue de la portée) lors de l'utilisation de `fiboNext` l.12.
3. Le mot anglais *lazy* (paresseux) se réfère à une technique de programmation fonctionnelle *paresseuse*, une forme de contrôle de l'évaluation qui repousse les calculs jusqu'au moment où ils deviennent indispensables.

   Dans le calcul en question, on voit qu'on génère une suite de valeurs de la suite de Fibonacci dans un tableau, et que l'on peut les générer 10 par 10, au fur et à mesure qu'elles deviennent nécessaires. L'objet `fiboLazy` est donc bien paresseux.

   Noter que la fonction `fiboNext`, au vu de son type, pourrait être prise comme la congélation d'un calcul (et donc serait à ce titre paresseuse), mais ce n'est en fait pas le cas puisqu'elle renvoie un résultat différent à chaque fois (et donc fait un calcul différent à chaque fois).
4. Des appels de fonctions identiques, mais renvoyant des résultats différents, laissent à penser que lesdits appels sont impurs et réalisent des effets de bords. Mais ce n'est pas très étonnant, la fonction `fiboNext` modifiant son état interne (la variable `aPair`) à chaque appel.

Exercice 4: 5 points

Le terme de *véhicule* est utilisé pour représenter toute une gamme d'objets plus ou moins différents dont voici quelques exemples classiques représentés sous la formes d'icônes :

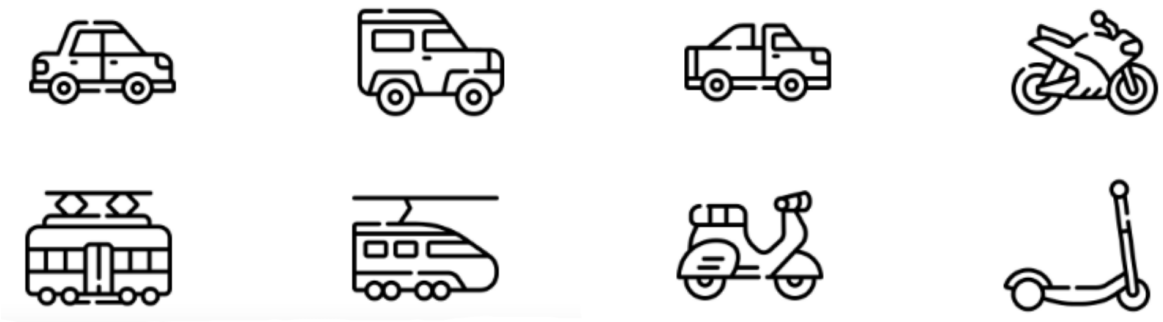

Image : flaticon.com

Remarquer que tous les véhicules ci-dessus possèdent des roues, en plus de leur qualité fondamentale de servir d' "appareil de transport"[2] pour passagers et marchandises.

Rappeler la définition de l'abstraction en programmation, ainsi que celles des types abstraits de données. En vous appuyant sur l'exemple des véhicules ci-dessus, illustrer et expliquer les différentes propriétés liées à ces définitions. Il est encouragé de s'appuyer sur

2. Il s'agit de la définition du CNRTL (cf. `https://www.cnrtl.fr/definition/v%C3%A9hicule`)

des exemples courts en pseudo-code. Toute manière correcte de relier le propos à de la programmation fonctionnelle sera bonifiée.

✓ En reprenant les définitions du cours, l'*abstraction* est une propriété logicielle selon laquelle les composants logiciels séparent leur part publique (interface) de leur part privée (implémentation). A ce titre, 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.
Au vu de l'exemple ci-dessus, un véhicule est naturellement un type abstrait de données. L'ensemble des valeurs associé est l'ensemble de tous les véhicules existant sur la planète, l'image fournie donnant une série d'exemples. Si on voit un véhicule comme un composant logiciel, on peut imaginer une interface comme un ensemble de prototypes de fonctions de la forme (en reprenant le style utilisé en PG116) :

```
type vehicle; // a type representing the possible vehicles

vehicle__load      : vehicle * data        → void // load some data into the vehicle
vehicle__move      : vehicle * destination → void // move the vehicle to some destination
vehicle__unload    : vehicle               → data // unload the data inside the vehicle
vehicle__nb_wheels : vehicle               → int  // return the number of wheels of the vehicle
```

Différentes implémentations existent des véhicules (vélo, voiture, avion . . .), mais s'ils remplissent tous l'interface précédente, cela signifie que l'on peut écrire des algorithmes (par exemple de routage de personnes ou de marchandises) sans se préoccuper de l'implémentation concrète du véhicule. En plus de cela, une implémentation de l'interface précédente peut se faire en fournissant un ensemble de fonctions, ensemble qui peut se stocker dans une structure de données, profitant des propriétés de 1ère classe des fonctions.