Année 2020-2021 Session 1

Programmation fonctionnelle
PG104

Filière : Informatique, 1ère Année

Date de l'examen : 27/05/2021 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: 2 points

Soit f une fonction continue à valeurs réelles, définie sur un intervalle $[a;b]$, et telle que $f(a) \times f(b) < 0$. Écrire l'algorithme calculant une racine de f sur l'intervalle $[a;b]$ avec une précision $eps > 0$, en utilisant la méthode dichotomique. La fonction obtenue ne doit pas faire d'effets de bords, ni dupliquer de code.

✓

```
function dicho(f, [a, b], eps) {
    const mid = (a+b)/2;
    if (b-a < eps)
        return mid;
    else {
        if (f(mid) === 0)
            return mid;
        else if (f(a)*f(mid) < 0)
            return dicho(f, [a, mid], eps);
        else
            return dicho(f, [mid, b], eps);
    }
}
```

La question invite naturellement à écrire une version récursive par opposition à la version impérative qui fait des effets de bords. Le calcul du point milieu ne doit pas être répété (même le `(a+b)/2`). La déclaration de cette valeur doit se faire avec `const`. Il n'était pas indispensable de regrouper les bornes de l'intervalle dans un tableau.

Exercice 2: 2 points
La fonction suivante est censée dupliquer les objets à l'intérieur d'un tableau, en ajoutant aux copies (et uniquement aux copies) un champ `isACopy` pour les identifier :

```
1 function duplicate(l) {
2     let lCopies = l.map((el) ⇒ {
3         let newel = el;
4         newel.isACopy = true; // assign a flag to copies
5         return newel;
6     });
7     return l.concat(lCopies);
8 }
```

```
let aList = [{ x: 1 }, { x: 2 }];
duplicate(aList); // → [{x:1, isACopy:true}, {x:2, isACopy:true}, {x:1, isACopy:true}, {x:2, isACopy:true}]
aList;            // → [{x:1}, {x:2}]
```

Identifier l'effet de bord effectué dans cette fonction. Expliquer pourquoi elle ne fonctionne pas correctement et proposer une solution.

✓ L'effet de bord est réalisé ligne 4, lorsqu'on modifie `newel` en lui ajoutant un attribut. Pour rappel, la méthode `concat` ne modifie pas les tableaux existants, et donc n'est pas la place d'un effet de bord (comme on peut le voir dans le test, `aList` n'est pas modifiée).
Le fonction `duplicate` souffre d'un problème d'*aliasing* : les deux variables `el` et `newel` sont des références vers le même objet. Une modification du premier entraîne une modification du second. Pour avoir un fonctionnement correct, il faut rendre les deux références indépendantes. Cela peut se faire en faisant une copie de l'objet avant de le modifier.

```
function duplicate(l) {
    return l.concat(l.map((el) ⇒ {
        let newel = { ...el }; // perform a shallow copy of 'el'
        newel.isACopy = true;
        return newel;
    }));
}
```

Exercice 3: 6 points
Le code suivant a été proposé pour servir de test pour l'implémentation d'une pile :

```
1  const { stackCreateEmpty, stackIsEmpty, stackPush, stackPeek } = require('./stack');
2
3  describe('a first test suite', () => {
4      let s;
5
6      test('empty stack should be empty', () => {
7          s = stackCreateEmpty();
8          return expect(stackIsEmpty(s)).toBe(true);
9      });
10
11     test('adding elements to stack should appear in reverse order', () => {
12         s = stackCreateEmpty();
13         s = stackPush(10, s); s = stackPush(20, s);
14         return expect(stackPeek(s)).toBe(20);
15     });
16 });
```

1. Proposer un type *précis* pour le 2nd argument de la fonction `test` utilisée ici.
   Donner le nom de la technique de programmation utilisée ici, et expliquer l'intérêt de l'utilisation de cette technique dans ce cas précis.
2. Décrire la portée de la variable `s` déclarée ligne 4.
   Donner le nom du mécanisme qui rend cette variable accessible au moment où les fonctions passées en paramètres à `test` sont exécutées.
3. Expliquer dans quelle mesure cette portée crée un problème de dépendance dans ce code. Proposer une solution.

✓

1. Le second argument de la fonction `test` est une fonction ne prenant pas de paramètres, et avec un type de retour indéterminé. Les fonctions utilisées servant de fonctions de test, il est raisonnable d'envisager que cette fonction renvoie un booléen, d'où le type `() ⇒ boolean`.

   La fonction `test` utilise un paramètre fonctionnel, pour pouvoir exécuter le test au moment où elle le désire. Il s'agit d'une technique de *contrôle de l'évaluation*. Elle utilise une des propriétés de 1ère classe des fonctions, qui n'est pas une technique.

   Le contrôle de l'évaluation permet ici par exemple de repousser le moment où le test est effectué, d'effectuer les tests dans un ordre quelconque ou en parallèle. Elle permet aussi d'envisager des tests de charge en répétant une fonction de test un nombre donné de fois.

2. La variable `s` est déclarée avec le mot-clé `let`, et donc a une portée de bloc. Elle est accessible dans tout le bloc à partir de sa définition, soit des lignes 4 à 14.

   Les deux fonctions `test` prennent en paramètre des fonctions anonymes faisant usage de la variable `s`. Au moment où ces fonctions seront utilisées, elles auront aussi accès à `s` à travers le mécanisme de *fermeture* qui assure qu'une fonction embarque avec elle son environnement.

3. Ces fermetures induisent un problème de dépendance, dans la mesure où les deux fonctions anonymes dépendent de `s`, en faisant des accès en lecture et en écriture dessus. Ainsi, ces deux fonctions ne doivent surtout pas s'exécuter en parallèle, comme proposé dans la réponse précédente, sous peine de comportement incohérent (aussi appelé *race condition*).

   La variable `s` n'a pas à être globale au bloc, et il suffit pour solutionner le problème de déclarer dans chaque fonction anonyme une variable indépendante (potentiellement appelée `s` aussi).

Exercice 4: 5 points

Une *lentille* (*lens* en anglais) est une valeur représentant un "moyen d'accéder et de modifier un attribut d'un objet". Elle est constituée de deux parties : `view` (qui permet d'accéder) et `set` (qui permet de modifier). Le code suivant (adapté d'un code de Eric Eliott `https://medium.com/javascript-scene/lenses-b85976cb0534`) explique comment construire et utiliser une lentille associée à l'attribut `aField` d'un objet :

```
const aFieldLens = {
  view: store ⇒ store["aField"],
  set: (value, store) ⇒ ({
    ...store,                    // Copy keys/values from 'store'
    ["aField"]: value            // Update the field 'aField'
  })
};
```

La lentille s'utilise alors de la manière suivante pour accéder à l'attribut `aField` :

```
const anObject = { aField: 7, aDistinctField: '3cha3' };
anObject;                   // → { aField: 7, aDistinctField: '3cha3' }
aFieldLens.view(anObject);  // → 7
```

Et elle permet de modifier l'attribut `aField` de la manière suivante :

```
anObject;                   // → { aField: 7, aDistinctField: '3cha3' }
const newObject = aFieldLens.set(11, anObject);
newObject;                  // → { aField: 11, aDistinctField: '3cha3' }
aFieldLens.view(newObject); // → 11
```

1. En quoi la façon de faire pour construire `aFieldLens` peut-elle être qualifiée de "fonctionnelle" ?

   Attention : il y a potentiellement deux aspects remarquables ici.

2. Proposer une généralisation de `aFieldLens` sous la forme d'une fonction `lensProp` qui s'appliquerait à n'importe quel nom d'attribut.

Considérons l'idée de *composer* les lentilles afin de pouvoir traiter par exemple les attributs des attributs d'un objet donné. Proposons nous de construire une fonction `lensCompose` prenant en paramètre deux lentilles associées à deux attributs et renvoyant une nouvelle lentille qui accéderait à l'attribut de l'attribut. Par exemple :

```
// `aFieldLens`        is a lens attached to "aField"
// `anotherFieldLens` is a lens attached to "anotherField"
// `lensCompose`      is a lens resulting from the composition of both
const aComposedLens = lensCompose(aFieldLens, anotherFieldLens);

// Example using `aComposedLens`
const aDeepObject = { aField: { anotherField: "4boo4" } };
aDeepObject;                        // → { aField: { anotherField: '4boo4' } }
aComposedLens.view(aDeepObject);    // → '4boo4'
const anotherDeepObject = aComposedLens.set("5nut5", aDeepObject);
anotherDeepObject;                  // → { aField: { anotherField: '5nut5' } }
```

Concrètement, une fonction `lensCompose` prendrait en paramètre deux lentilles `lens1` et `lens2` et produirait une lentille `lensF = lensCompose(lens1, lens2)`. Pour faciliter l'écriture de `lensCompose`, posons les définitions suivantes :

- `view1 = lens1.view` et `set1 = lens1.set` ;
- `view2 = lens2.view` et `set2 = lens2.set` ;
- `viewF = lensF.view` et `setF = lensF.set`.

3. Écrire `viewF` comme une composition d'appels à `view1` et `view2`.
4. Écrire `setF` comme une composition d'appels à `set1`, `set2`, et `view1`.

Pour ces deux questions, il est recommandé de ne pas écrire le résultat en une seule ligne et d'utiliser des `let`. Toute réponse injustifiée sera pénalisée.

✓

1. Il y a deux manières de répondre à cette question, la totalité des points n'est attribuée que si les deux manières sont expliquées.

   La première manière consiste à voir la construction entre les lignes 3 et 6 comme une fonction anonyme. Il s'agit d'un style fonctionnel, dans la mesure où l'on utilise la propriété de la première classe consistant à stocker des fonctions dans des structures de données.

   Le deuxième manière consiste à voir les points de suspension '`...`' comme une reconnaissance de motif. Dans le cas présent, la fonction `set` construit une nouvelle instance de l'objet passé en paramètre, et ne fait donc pas d'effet de bord. Il s'agit donc aussi d'un style fonctionnel dans la mesure où les fonctions utilisées sont pures.

2. La question porte sur une généralisation fonctionnelle, même si elle est originale par rapport au cours au sens où elle ne part pas d'une fonction. Le résultat est le suivant :

```
function lensProp(prop) {
    return {
        view: store ⇒ store[prop],
        set: (value, store) ⇒ ({
            ...store,
            [prop]: value
        })
    };
}
```

3. Le code de la fonction `compose`, pour laquelle la difficulté consiste à bien trouver l'ordre dans lequel composer les `view` et les `set` :

```
function compose(lens1, lens2) {
    return {
        view: store ⇒ lens2.view(lens1.view(store)),
        set: (value, store) ⇒
        lens1.set(lens2.set(value, lens1.view(store)), store)
    };
}
```

Exercice 5: 5 points

Un jeu de construction connu (des ensembles de briques élémentaires qui s'emboîtent les unes dans les autres pour construire des objets plus grands) vous est présenté comme étant "modulaire". Il contient en particulier des boîtes de briques différentes permettant au total de construire un chateau : une enceinte extérieure, une enceinte intérieure, des tourelles à ajouter sur les enceintes, un donjon central, et évidemment un dragon en bonus se nichant sur le toit du donjon.

Rappeler la définition de la modularité en programmation, ainsi que celles des modules. Pour chacune des propriétés associées à ces définitions, expliquer de quelle manière elle peut ou ne peut pas s'appliquer à l'exemple précédent.

✓ Les boîtes de jeu de construction peuvent être vus comme des modules offrant un certain nombre de fonctionnalités. Chacun des modules peut être considéré comme *cohésif* en cela que les pièces d'un même module se composent ensemble (e.g la tête et les ailes du dragon se collent sur le corps du dragon, et pas au fond des fondations). Les modules peuvent aussi être considérés comme *couplés*, et il est simple d'imaginer des dépendances entre eux (e.g pour avoir besoin de construire le donjon, il peut être nécessaire d'avoir la boîte fournissant les fondations).
Les deux notions propres aux modules sont sujettes à interprétation. Selon la façon de penser, la notion d'*interface* peut être appropriée ou pas : il est envisageable que la connection entre les fondations et les étages supérieurs partagent une interface (par exemple simplement une surface plate entres les fondations et les tourelles). On peut aussi argumenter que si les boîtes sont indépendantes (un autre argument de vente !), de telles interfaces n'existent simplement pas. La notion d'*encapsulation*, elle aussi, est sujette à interprétation. Il est possible de considérer que certaines pièces des modules soient complètement internes (par exemple, le corps du dragon ne peut s'attacher qu'aux membres du dragon), mais il est aussi possible de considérer qu'un jeu de construction est suffisamment flexible pour rendre l'ensemble de ses pièces compatibles avec toutes les autres.
Plusieurs personnes interprètent à tort les modules comme devant être des entités complètement indépendantes. Mais cela n'a pas de sens, puisqu'ils partagent par définition des relations de dépendance (par exemple les tourelles sur les enceintes). Cela a amené à des discours confus. Un dessin d'un graphe de dépendance pour expliquer son argumentation était évidemment bien reçu.