Année **2024-2025** Session **2**

PROGRAMMATION FONCTIONNELLE
PG104

Filière : Informatique, 1ère Année

Date de l'examen : 25/06/2025 Durée de l'examen : 2h00

Sujet de : D. RENAULT

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

Cet examen contient deux types de questions, avec un symbole ✎ ou ⌨ :

— ✎ **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 2.1

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

#### Reponse 2.1

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 horseWhiteColor(aKing) {
    //
    // Write code here
    //
}

export { horseWhiteColor };
```

Un fichier `README.md` fournit de l'aide quant à l'utilisation des outils. 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 indicatif. Les exercices sont *indépendants* les uns des autres.

Une fonction `anyToString` est fournie pour aider au débuggage, et convertit n'importe quelle valeur en chaîne de caractères pour affichage dans la console.

Exercice 1: 2 points

✎ Donner une définition de ce qu'est la *récursivité*, dans le cadre de la programmation fonctionnelle. Donner un exemple de quelque chose de récursif en programmation.

Expliquer en quoi la *récursivité* est une propriété qui se marie particulièrement bien avec la programmation fonctionnelle.

✓

1. Un problème ou un calcul sont dits récursifs lorsqu'ils peuvent être décomposés en un nombre fini de sous-problèmes ou de calculs *de même nature*, mais plus petits. En programmation fonctionnelle, une fonction est récursive si elle applique cette propriété en faisant que son code la rappelle elle-même pour compléter le calcul. Un exemple notoire de fonction récursive est celle réalisant le calcul de la factorielle :

   Le problème décrit de manière récursive :

   $$fact(n) = \begin{cases} 1 & \text{si } n \leq 1 \\ n \cdot fact(n-1) & \text{sinon} \end{cases}$$

   Et à droite la fonction qui l'implémente.

   ```
   function fact(n) {
     if (n <= 1)
       return 1;
     else
       return n*fact(n-1); // Recursive call
   }
   ```

2. La récursivité se marie bien avec la programmation fonctionnelle, parce que, en s'appuyant sur une description formelle des problèmes, elle permet plus facilement de décrire des calculs sans effets de bord, et donc des fonctions pures. A contrario, les problèmes/algorithmes décrits avec des boucles sont moins adaptés à la programmation fonctionnelle, voire invitent à faire des effets de bords.

Exercice 2: 5 points

Dans cet exercice, on désire écrire une fonction `treeTraversal` qui parcourt un arbre en réalisant un parcours *préfixe* (aussi appelé *en profondeur*).

Dans cet exercice, vous avez à disposition un ensemble de fonctions sur les listes dans le module `given/list.js` (appelé `L` dans le code), ainsi que sur les arbres dans le module `given/tree.js` (appelé `T` dans le code).

```
const aTree1 = T.node(1, L.cons(T.leaf(2), L.cons(T.leaf(3), L.nil)));
treeTraversal(aTree1); // → (|1, 2, 3|)

const aTree2 = T.node(1, L.cons(
    T.node(2, L.cons(T.leaf(3), L.cons(T.leaf(4), L.nil))), L.cons(
    T.node(5, L.cons(T.leaf(6), L.cons(T.leaf(7), L.nil))), L.nil)));
treeTraversal(aTree2); // → (|1, 2, 3, 4, 5, 6, 7|)

const aTree3 = T.treeGenRandom({ generator: () ⇒ Math.floor(Math.random() * 10) });
treeTraversal(aTree3); // → (|9, 5, 4, 5, 6, 5, 0, 0, 5, 6|) (result varies)
```

L'idée est ici d'écrire cette fonction en quelques lignes, en utilisant les fonctions d'ordre supérieur que sont `listMap` et `listReduce`. L'exercice décompose les étapes.

> Bien noter que cette fonction renvoie des listes en non pas des tableaux.

L'algorithme se fait en trois étapes :

— Appliquer récursivement la fonction `treeTraversal` à tous les `children` de l'arbre. Ceci devrait produire une liste de listes.

— Concaténer toutes les listes de cette liste de listes dans l'ordre. Ceci devrait produire une liste de valeurs (ici des entiers).

— Ajouter en tête de liste la valeur au sommet de l'arbre.

Les deux premières étapes sont traitées dans les deux premières questions, la dernière consiste à produire l'algorithme complet.

2.1 ⌨ Écrire une fonction `applyTraversalToChildren` qui prend en paramètre un arbre `aTree` et une fonction `treeTraversalFun`, utilise la fonction `listMap`, et retourne la liste contenant le résultat de l'application de `treeTraversalFun` à tous les enfants de `aTree`.

```
applyTraversalToChildren(aTree1, treeTraversal); // → (| (| 2 |), (| 3 |) |)
applyTraversalToChildren(aTree2, treeTraversal); // → (| (| 2, 3, 4 |), (| 5, 6, 7 |) |)
```

Remarque : cette fonction ne peut évidemment pas être testée complètement sans la fonction finale. Néanmoins, on pourra la tester avec la fonction `(aTree) ⇒ T.val(aTree)` pour vérifier qu'elle n'engendre pas d'erreur (à défaut de fournir le résultat attendu).

2.2 ⌨ Écrire une fonction `concatenateListOfLists` qui prend en paramètre une liste de listes d'entiers `aListOfLists`, utilise la fonction `listReduce`, et produit la liste concaténée de toutes les listes de `aListOfLists`.

```
concatenateListOfLists((| 2 |), (| 3 |));                  // → (| 2, 3 |)
concatenateListOfLists((| (| 2, 3, 4 |), (| 5, 6, 7 |) |)); // → (| 2, 3, 4, 5, 6, 7 |)
```

2.3 ⌨ Écrire une fonction `treeTraversal` qui prend en paramètre un arbre `aTree` et retourne la liste des éléments de cet arbre produite dans l'ordre préfixe.

2.4 ⌨ Écrire une fonction `testTreeTraversal` qui génère un arbre aléatoirement avec la fonction `treeGenRandom`, l'affiche avec la fonction `treeNiceDisp`, calcule la liste de ses éléments avec `treeTraversal` et affiche son résultat.

Note : les fonctions `treeGenRandom` et `treeNiceDisp` sont fournies dans le module `tree.js`.

✓ Le code suivant répond aux consignes de l'énoncé :

```
function applyTraversalToChildren(aTree, treeTraversalFun) {
    return L.listMap(T.children(aTree), treeTraversalFun);
}

function concatenateListOfLists(aListOfLists) {
    return L.listReduce(aListOfLists,
                        (acc, val) ⇒ L.listAppend(acc, val),
                        L.nil);
}

function treeTraversal(aTree) {
    const children = applyTraversalToChildren(aTree, treeTraversal);
    const concatenatedChildren = concatenateListOfLists(children);
    return L.cons(T.val(aTree), concatenatedChildren);
}
```

Exercice 3: 5 points

Considérons le code suivant écrit à partir de la bibliothèque TerminalKit permettant de jouer au plus petit jeu de serpent du monde. Le code est fourni complètement dans le fichier `given/terminalkit.js` et il est possible de l'exécuter avec la commande `node given/terminalkit.js`. L'objectif de cet exercice est de réaliser une refactorisation de ce code.

```
1  import { default as T } from 'terminal-kit';
2
3  let term; // The terminal inside which the game is played
4
5  T.getDetectedTerminal((error, detectedTerm) => {
6      if (error) throw new Error('Cannot␣detect␣terminal.');
7      term = detectedTerm;
8      term.hideCursor();
9      term.grabInput();
10     term.on('key', inputs);
11     term.arrowPosition = 1; // The column of the arrow, starting at 1
12     term.white(">\nPlay␣with␣'a'␣and␣'z',␣quit␣with␣'q'");
13 });
14
15 function inputs(key) {
16     switch (key) {
17         case 'a':
18             term.arrowPosition = Math.min(term.arrowPosition + 1, term.width);
19             term.up(1).eraseLine().column(term.arrowPosition).
20                 white(">").
21                 down(1).eraseLine().column(0).
22                 color("green")(`'a'␣detected!␣(pos=${term.arrowPosition})`);
23             break;
24         case 'z':
25             term.arrowPosition = Math.max(term.arrowPosition - 1, 1);
26             term.up(1).eraseLine().column(term.arrowPosition).
27                 white("<").
28                 down(1).eraseLine().column(0).
29                 color("red")(`'z'␣detected!␣(pos=${term.arrowPosition})`);
30             break;
31         case 'q':
32         case 'CTRL_C':
33             terminate();
34             break;
35     }
36 }
37
38 function terminate() {
39     term.hideCursor(false);
40     term.grabInput(false);
41     setTimeout(function () {
42         term.moveTo(1, term.height, '\n');
43         process.exit();
44     }, 100);
45 }
```

Manifestement, ce code pose plusieurs problèmes :

— la variable `term` (déclarée l. 3) est une variable globale ;

— la fonction `inputs` (l. 15 à 36) contient du code dupliqué.

Les questions de texte suivantes sont simplement là pour expliquer votre démarche et comptent pour environ 1/4 des points. Mais la majorité de l'exercice est située dans la dernière question, qui compte environ pour 3/4 des points.

3.1 ✎ Qu'est-ce qu'une *duplication* de code ?
Quel code est-il dupliqué à l'intérieur de la fonction `inputs` (indiquer les lignes) ?

3.2 ✎ En quoi l'écriture d'une fonction permettrait d'elle de supprimer la duplication évoquée précédemment ?

3.3 ✎ Expliquer rapidement la démarche pour supprimer la variable globale `term`.

3.4 ⌨ Proposer une réécriture du fichier `terminalkit.js` dans le fichier `ex3.4.js` qui vérifie les propriétés suivantes :

- le code fournit les mêmes fonctionnalités ;
- la code ne contient aucune variable globale (en particulier `term`) ;
- la duplication de code dans `inputs` a été éliminée au maximum.

✓

1. Une *duplication* de code est la présence dans un code donné de sous-parties qui sont soit identiques ou presque. Dans la fonction `inputs`, les deux mouvements possibles (cas `'a'`, l. 18 à 22, et `'z'`, l. 25 à 28 du `switch`) sont des exemples de duplication.
   A noter que la duplication ne demande pas à ce que le code dupliqué correspondent à deux blocs contigus identique. Dans le cas présent, les deux blocs varient par plusieurs points : le déplacement possible, le caractère affiché pour la flèche, la couleur utilisée et le caractère affiché pour le message.
2. Lorsque du code comme celui-ci est dupliqué, il est recommandé de construire une fonction qui factorise cette duplication. Concrètement, la fonction contient directement le bloc qui a été reconnu comme dupliqué. Et si ce bloc contient des variations, ces variations sont passées comme paramètres à ladite fonction.
3. Pour éliminer la variable globale `term`, il faut déjà comprendre qui l'utilise. Ici, elle est utilisé dans le bloc de `getDetectedTerminal` (où elle est initialisée), dans la fonction `inputs` et dans la fonction `terminate`.
   - Une première méthode consiste alors à confiner l'utilisation de `term` au bloc de `getDetectedTerminal`, en supprimant la l. 3, et en déplaçant les fonctions `inputs` et `terminate` dans ce bloc. Elle a le désavantage de forcer ces deux fonctions dans ce bloc, ce qui fait qu'on ne peut les déplacer dans un autre module ou les réutiliser ailleurs.
   - Une méthode plus raffinée consiste à laisser les deux fonctions en dehors du bloc, mais à leur ajouter un paramètre `term`. Cela pose problème pour `inputs`, car elle est déjà utilisée en tant que fonction prenant une `key` en paramètre. Pour arriver à nos fins, il suffit de curryfier `inputs` sur deux paramètres `term` et `key`, et de remplacer la l. 10 par :

     ```
     term.on('key', inputs(term));
     ```

✓ Le code suivant est une réponse possible à la dernière question :

```
import { default as T } from 'terminal-kit';

T.getDetectedTerminal((error, term) => {
    if (error) throw new Error('Cannot detect terminal.');
    term.hideCursor();
    term.grabInput();
    term.on('key', inputs(term));
    term.arrowPosition = 1; // The column of the arrow, starting at 1
    term.white(">\nPlay with 'a' and 'z', quit with 'q'");
});

function moveArrow(term, move, arrowChar, color, key) {
    term.arrowPosition = Math.max(Math.min(term.arrowPosition + move, term.width), 0);
    term.up(1).eraseLine().column(term.arrowPosition).
        white(arrowChar).
        down(1).eraseLine().column(0).
        color(color)(`'${key}' detected! (pos=${term.arrowPosition})`);
}

function inputs(term) {                                    // Refactor : remove global
    return (key) => {                                      // Refactor : remove global
        switch (key) {
            case 'a':
                moveArrow(term, +1, '>', "green", 'a'); // Refactor : generalize
                break;
            case 'z':
                moveArrow(term, -1, '<', "red", 'z');   // Refactor : generalize
                break;
            case 'q':
            case 'CTRL_C':
                terminate(term);
                break;
        }
    };
}                                                          // Refactor : remove global

function terminate(term) {                                 // Refactor : remove global
    term.hideCursor(false);
    term.grabInput(false);
    setTimeout(function () {
        term.moveTo(1, term.height, '\n');
        process.exit();
    }, 100);
}
```

Ce code utilise une currification pour la fonction `inputs`, mais ce n'est pas la seule solution acceptable. Il était possible par exemple d'encapsuler les fonctions `inputs` et `terminate` à l'intérieur du bloc de `getDetectedTerminal`.
Pour la généralisation, après avoir créé une fonction (ici `moveArrow`), une des difficultés était de bien choisir les paramètres qui variaient dans les deux clauses du `switch`. Ne pas inclure la modification de `arrowPosition` dans la fonction `moveArrow` n'a pas été considéré comme une erreur.

Exercice 4: 5 points

Le but de cet exercice est d'implémenter un algorithme de tri fusion, aussi appelé *merge sort*, sur les tableaux standards de Javascript. L'algorithme pour ce tri, décrit sur la page `https://fr.wikipedia.org/wiki/Tri_fusion`, est un algorithme de type "diviser pour régner", qui s'écrit naturellement de façon récursive.

— Cas terminaux :
  — si le tableau est vide ou ne possède qu'un seul élément, il est déjà trié, ne rien faire et renvoyer le tableau.
— Appels récursifs :
  — diviser le tableau en deux sous-tableaux de tailles `Math.floor(n/2)` et `Math.ceil(n/2)` (utiliser pour cela la méthode `slice`) ;
  — trier récursivement les deux parties en utilisant le même algorithme ;
  — fusionner les deux tableaux triés en un seul tableau final trié.

La première question consiste à écrire la fonction de fusion, et la seconde question l'algorithme tout entier.

> Dans cet exercice, vous avez à disposition un ensemble de fonctions de comparaison dans le module `given/comparison.js`, appelé `C` dans le code.

4.1 ⌨ Écrire une fonction *récursive* `mergeSortedArrays` qui prenne en paramètre deux tableaux triés `anArray1` et `anArray2`, ainsi qu'une fonction de comparaison `cmpFun`, et qui renvoie un nouveau tableau contenant l'union des valeurs des deux tableaux en entrée, et trié selon la fonction `cmpFun`.

La fonction `cmpFun` prend en paramètres deux éléments `a` et `b`, et renvoie une valeur numérique < 0 si `a < b`, $= 0$ si `a === b` et > 0 si `a > b`.

Il est recommandé d'utiliser une fonction intermédiaire.

```
S.mergeSortedArrays([1, 3], [2, 4], C.ltIntCmp); // → [1, 2, 3, 4]
S.mergeSortedArrays([],     [2, 4], C.ltIntCmp); // → [2, 4]
S.mergeSortedArrays([1, 3], [],     C.ltIntCmp); // → [1, 3]
S.mergeSortedArrays([3, 1], [4, 2], C.gtIntCmp); // → [4, 3, 2, 1]
```

4.2 ⌨ Écrire une fonction *récursive* `arrayMergeSortGen` qui prenne en paramètre un tableaux `anArray` et une fonction de comparaison `cmpFun`, et qui renvoie un nouveau tableau contenant les mêmes valeurs que `anArray`, et trié selon la fonction `cmpFun` avec l'algorithme du merge sort.

```
S.arrayMergeSortGen([1, 3, 2, 4], C.ltIntCmp); // → [1, 2, 3, 4]
S.arrayMergeSortGen([1, 3, 2, 4], C.gtIntCmp); // → [4, 3, 2, 1]
```

4.3 ✎ Exprimer la complexité en temps de chacun des deux algorithmes précédents.

✓ Le code suivant (qui a l'air long mais qui en fait est très répétitif) répond aux consignes de l'énoncé :

```
function mergeSortedArrays(anArray1, anArray2, aCmpFun) {
    const aResult = Array.from({length: anArray1.length + anArray2.length});
    function arrayInsert(aSource1, aSource2, aDestArray, anIndex1, anIndex2, aDestIndex) {
        if (anIndex1 >= aSource1.length) {
            if (anIndex2 >= aSource2.length) { // Both arrays merged
                return;
            } else { // Remains to merge anArray2
                aDestArray[aDestIndex] = anArray2[anIndex2];
                arrayInsert(aSource1, aSource2, aDestArray, anIndex1, anIndex2+1, aDestIndex+1);
            }
        } else {
            if (anIndex2 >= aSource2.length) { // Remains to merge anArray1
                aDestArray[aDestIndex] = anArray1[anIndex1];
                arrayInsert(aSource1, aSource2, aDestArray, anIndex1+1, anIndex2, aDestIndex+1);
            } else { // Both arrays can be inserted
                if (aCmpFun(anArray1[anIndex1], anArray2[anIndex2]) < 0) {
                    aDestArray[aDestIndex] = anArray1[anIndex1];
                    arrayInsert(aSource1, aSource2, aDestArray, anIndex1+1, anIndex2, aDestIndex+1);
                } else {
                    aDestArray[aDestIndex] = anArray2[anIndex2];
                    arrayInsert(aSource1, aSource2, aDestArray, anIndex1, anIndex2+1, aDestIndex+1);
                }
            }
        }
    }
    arrayInsert(anArray1, anArray2, aResult, 0, 0, 0);
    return aResult;
}

function arrayMergeSortGen(anArray, aCmpFun) {
    if (anArray.length <= 1)
        return anArray;
    else { // The length of anArray is >= 2
        const halflength = Math.floor(anArray.length / 2);
        const firsts = arrayMergeSortGen(anArray.slice(0, halflength), aCmpFun);
        const nexts = arrayMergeSortGen(anArray.slice(halflength, anArray.length), aCmpFun);
        return mergeSortedArrays(firsts, nexts, aCmpFun);
    }
}
```

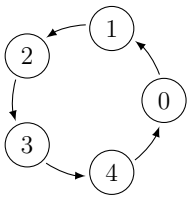
La complexité en temps de `mergeSortedArrays` est $\mathcal{O}(n_1 + n_2)$ où n_1 et n_2 sont les longueurs respectives des deux tableaux en entrée.
La complexité en temps de `arrayMergeSortGen` est $\mathcal{O}(n \log(n))$ où n est la longueur du tableau d'entrée.

Exercice 5: 4 points

Considérons des graphes dont les sommets sont choisis dans un ensemble V (par exemple, V pourrait être l'ensemble des valeurs de type `number` en Javascript). Une manière de représenter un graphe sur V consisterait à l'identifier à une fonction `aGraph` : $V \times V \to$ `boolean` qui, étant donné deux sommets v_1 et v_2, renvoie un booléen signifiant s'il existe une arête allant du sommet v_1 au sommet v_2.

Par exemple, le graphe de droite serait associé à la fonction suivante :

$$\mathsf{aGraph}(v_1, v_2) = \begin{cases} \mathsf{true} & \text{si } v_1 \in [0;4] \text{ et } v_2 = v1 + 1 \bmod 5 \\ \mathsf{false} & : \text{sinon} \end{cases}$$



Ou plus prosaïquement sa version en Javascript :

```
const aGraph = (v1, v2) => {
    return (v1 >= 0) && (v1 <= 4) && (v2 === v1 + 1 % 5);
}
```

✎ Rappeler ce que signifie en programmation fonctionnelle la *représentation des données par des fonctions*.

Proposer un type abstrait de données représentant les graphes comme des structures de données fonctionnelles *en utilisant l'idée précédente*. L'interface du TAD devra permettre *au minimum* de créer des graphes sans arêtes, d'ajouter une arête, et de retourner le sens de toutes les arêtes d'un graphe donné. Il est possible d'utiliser les types Typescript pour mieux décrire les fonctions de l'interface.

Discuter les opérations efficaces sur cette structure. Quelles autres opérations pouvez-vous imaginer n'entrant pas dans cette catégorie (i.e *moins que* efficace) ?

✓ Il s'agit du même exercice que celui donné en 1ère session, avec quelques modifications cosmétiques. Se référer à la correction déjà donnée dans cet examen.

Recommandations globales

Cette section ne contient pas de questions pour l'examen, mais un certain nombre de recommandations et de consignes générales.

Outils disponibles

— Tous les outils utilisables usuellement en TD sont disponibles pour programmer. Cela contient en particulier `node`, `emacs`, `code`, `firefox` et `evince`.

— Un certain nombre d'outils standard pour le développement en Javascript sont aussi installés, ainsi que les dépendances dans le répertoire `node_modules` : `tsc`, `eslint` et `jest`.

— Des fichiers de configuration pour les outils précédents sont disponibles dans le répertoire `exam`, et les scripts suivants utilisables avec `npm run` :

  — `npm run tsc` pour le compilateur Typescript,

  — `npm run eslint` pour le linter ESLint et

  — `npm run jest` pour la bibliothèque de tests Jest.

Questions de texte

— Il est demandé de répondre avec précision aux questions. Toute forme d'imprécision sera considérée comme un malus potentiel.

— Si vous estimez que vous êtes imprécis, pensez à prendre un exemple pour étoffer votre explication.

— En cas d'utilisation d'un exemple, pensez à relier cet exemple aux définitions vues en cours.

Questions de code

— Tout fichier syntaxiquement incorrect, ou dont l'utilisation directe avec `node` ne termine pas sans erreur ne sera pas considéré pour la correction. Le script `verif.sh` ne fait qu'une vérification syntaxique (avec `node --check`).

— Aucun autre fichier que ceux demandés dans les exercices ne sera lu ni considéré pour la correction. Écrire des tests dans un autre fichier (avec `jest` ou toute autre forme de test) peut aider pour développer, mais ils ne seront pas considérés comme des rendus de l'examen, sauf s'ils sont insérés dans les fichiers réponses.

— Aucun import n'est nécessaire dans le code fourni, hormis ceux déjà présents dans les fichiers fournis. Si jamais des fonctions dépendaient l'une de l'autre, elles seraient présentes dans le même fichier. Modifier les imports existant ou ajouter ses propres imports, c'est prendre le risque que le code final ne soit pas considéré.