Année **2024-2025** Session **1**

PROGRAMMATION FONCTIONNELLE
PG104

Filière : Informatique, 1ère Année

Date de l'examen : 22/05/2025 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

✎ En programmation fonctionnelle, qu'est-ce que l'on entend lorsqu'on parle de fonction *d'ordre supérieur* ? Donner un exemple de telle fonction, ainsi que son type en Typescript.

Quelle est le nom de la propriété, visible en général à travers leurs types, des fonctions d'ordre supérieur ?

✓

1. Les fonctions d'ordre supérieur sont des fonctions prenant en paramètre d'autres fonctions. Le cours contient un nombre important d'exemples, typiquement les fonctions comme `filter`, `map` ou `reduce` sur les listes. Dans l'exemple suivant, la fonction `filter` prend un paramètre `pred` qui est lui aussi une fonction :

```
function filter<T>(pred: (T) ⇒ boolean, arr: T[]) : T[] { /* ... */ }
```

2. Les fonctions d'ordre supérieur, de par leur tendance à déléguer les calculs à leurs paramètres, sont souvent (mais pas toujours) des fonctions *génériques*. Cela se remarque dans leur type par des variables de types comme le `T` de l'exemple ci-dessus.

Exercice 2: 5 points

La bibliothèque standard de Javascript contient une fonction `flatMap` sur les tableaux, décrite à l'adresse `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/flatMap`. Cette fonction applique une transformation à tous les éléments d'un tableau, cette transformation étant censée produire elle-même un tableau pour chaque élément. Le résultat est donc un tableau de tableaux, que la fonction concatène au final. Voici quelques exemples :

```
[].flatMap((num) ⇒ [num, num]);              // → [ ]
[1,2,1].flatMap((num) ⇒ [num, num]);         // → [ 1, 1, 2, 2, 1, 1 ]
[1,2].flatMap((num) ⇒ [num, num+1, num+2]); // → [ 1, 2, 3, 2, 3, 4 ]
```

Note : en Javascript, cette fonction est aussi capable de gérer le cas des transformations ne renvoyant pas des tableaux, mais dans cet exercice, nous considérons que la transformation doit *systématiquement* renvoyer un tableau.

Nous nous proposons d'implémenter cette fonction sur les listes chaînées du cours.

C'est à vous d'adapter les exemples précédents sur des listes chaînées.

Dans cet exercice, vous avez à disposition un ensemble de fonctions sur les listes vues en cours, implémentées dans un module `given/list.js`, ainsi que sur les arbres, dans un module `given/tree.js`.

2.1 ⌨ Écrire une fonction *récursive terminale* et *pure* `listReverse` qui prenne en paramètre une liste `aList` et renvoie la liste miroir, c'est-à-dire la liste de tous les éléments de `aList` dans l'ordre inverse.

2.2 ⌨ Écrire une fonction *récursive terminale* et *pure* `listFlatMap` prenant en paramètre une fonction `aFun`, et une liste `aList`, et renvoyant une liste définie comme suit :

- La fonction `aFun` prend un paramètre et renvoie une liste de valeurs.
- La fonction `aFun` est appliquée à tous les éléments de `aList`, et les listes résultantes sont concaténées.

Remarque : il est fortement conseillé d'écrire une fonction intermédiaire.

Une fonction qui s'écrit naturellement à partir de la fonction `listFlatMap` est la fonction qui réalise un parcours en profondeur d'un arbre.

```
import * as L from '#given/list.js';
import * as T from '#given/tree.js';

const aTree1 = T.node(1,
                      L.cons(T.leaf(2), L.cons(T.leaf(3), L.nil)));
// treeDepthTraversal(aTree1) should return the list (| 1, 2, 3 |)
const aTree2 = T.node(4,
                      L.cons(T.leaf(5), L.cons(T.leaf(6), L.cons(T.leaf(7), L.nil))));
// treeDepthTraversal(aTree2) should return the list (| 4, 5, 6, 7|)
const aTree3 = T.node(0,
                      L.cons(aTree1, L.cons(aTree2, L.nil)));
// treeDepthTraversal(aTree3) should return the list (| 0, 1, 2, 3, 4, 5, 6, 7|)
```

2.3 ⌨ Écrire une fonction *pure* `treeDepthTraversal` prenant en paramètre un arbre `aTree`, renvoyant une liste de ses noeuds dans l'ordre de parcours en profondeur.

✓ Le code suivant répondait à la question :

```
function listReverse(aList) {
    function listRevTR(l, r) {
        if (LA.isEmpty(l))
            return r;
        else
            return listRevTR(LA.tail(l), LA.cons(LA.head(l), r));
    }
    return listRevTR(aList, LA.nil);
}

function listFlatMapAux(aFun, aParamList, aConcatList, aResultList) {
    if (LA.isEmpty(aConcatList)) {
        if (LA.isEmpty(aParamList))
            return aResultList;
        else // aParamList is not empty
            return listFlatMapAux(aFun, LA.tail(aParamList),
                                  aFun(LA.head(aParamList)), aResultList);
    } else // aConcatList is not empty
        return listFlatMapAux(aFun, aParamList,
                              LA.tail(aConcatList),
                              LA.cons(LA.head(aConcatList), aResultList));
}

function listFlatMap(aFun, aList) {
    if (LA.isEmpty(aList))
        return LA.nil;
    else
        return listReverse(listFlatMapAux(aFun, aList, LA.nil, LA.nil));
}


function treeDepthTraversal(aTree) {
    return LA.cons(TA.val(aTree), listFlatMap(treeDepthTraversal, TA.children(aTree)));
}
```

Beaucoup de personnes ont réécrit une fonction `listConcat` (avec plus ou moins de bonheur) pour faire la fonction `listFlatMap`. Il était aussi accepté le code utilisant les fonctions d'ordre supérieur (`listMap` ...) s'il était fonctionnellement correct.

Exercice 3: 4 points

Considérons le code suivant tiré de la bibliothèque TerminalKit permettant d'afficher un dessin en ASCII art dans un terminal. Le code est fourni complètement dans le fichier `given/terminalkit.js` et il est possible de l'exécuter avec `node`.

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

var term;
var ScreenBuffer = T.ScreenBuffer;
var viewport, sprites = {}; // Buffers

function init( callback ) {
    T.getDetectedTerminal( function( error, detectedTerm ) {
        if ( error ) { throw new Error( 'Cannot␣detect␣terminal.' ); }
        term = detectedTerm;
        viewport = new ScreenBuffer({
            dst: term,
            width: Math.min( term.width ),
            height: Math.min( term.height - 1 ),
            y: 2 });
        createBackground();
        createSpaceship();
        term.hideCursor();
        term.grabInput();
        term.on( 'key', inputs );
        callback();
    });
}
```

```
24 function terminate() {
25     term.hideCursor( false );
26     term.grabInput( false );
27     setTimeout(function() {
28                     term.moveTo( 1, term.height, '\n\n' );
29                     process.exit(); },
30                100);
31 }
32
33 function createBackground() {
34     sprites.background = new ScreenBuffer({ width: viewport.width * 4,
35                                             height: viewport.height, noFill: true });
36     sprites.background.fill({ attr: { color: 'white', bgColor: 'black' } });
37     var nStars = sprites.background.width * sprites.background.height * 0.004;
38     var i, x, y, c, char, stars = [ '*', '.', 'o', '+', '\'' ];
39     for ( i = 0; i < nStars; i ++ ) {
40         x = Math.floor( Math.random() * sprites.background.width );
41         y = Math.floor( Math.random() * sprites.background.height );
42         char = stars[ Math.floor( Math.random() * stars.length ) ];
43         c = Math.floor( Math.random() * 16 );
44         sprites.background.put({ x: x, y: y, attr: { color: c, bgColor: 'black' }}, char);
45     }
46 }
47
48 function createSpaceship() {
49     sprites.spaceship = new ScreenBuffer({ width: 1, height: 1 });
50     sprites.spaceship.put({ x: 0, y: 0, attr: { color: 'red', bgColor: 'white' } }, ">");
51     sprites.spaceship.x = 3;
52     sprites.spaceship.y = Math.floor(viewport.height / 2 - sprites.spaceship.height / 2);
53 }
54
55 function inputs( key ) {
56     switch ( key ) {
57         case 'UP' : sprites.spaceship.y --; break;
58         case 'DOWN' : sprites.spaceship.y ++; break;
59         case 'LEFT' : sprites.spaceship.x --; break;
60         case 'RIGHT' : sprites.spaceship.x ++; break;
61         case 'q': terminate(); break;
62     }
63 }
64
65 var frames = 0;
66
67 function draw() {
68     sprites.background.draw({ dst: viewport, tile: true });
69     sprites.spaceship.draw({ dst: viewport, blending: true, wrap: 'both' });
70     var stats = viewport.draw({ delta: true });
71     frames ++;
72 }
73
74 function animate() {
75     draw();
76     sprites.background.x --;
77     setTimeout(animate, 50);
78 }
79
80 init( function() { animate(); });
```

1. ✎ Que pouvez-vous dire de la portée des variables `term` (l. 3) et `detectedTerm` (l. 8) ? Où ces variables sont elles utilisées ?

2. ✎ En quoi est-ce que le choix de ces portées est-il critiquable ?
   Justifier la critique en proposant une utilisation du code menant à une erreur.
3. ✎ Quelles modifications du code seraient nécessaires pour faire que la fonction `terminate` (l. 25 à 32) prenne la variable `term` (l. 29) en paramètre ?
   Indice : une curryfication peut être nécessaire.
   Remarques : la réponse pourra contenir des extraits de code modifiés ; vous avez toute latitude pour faire des modifications avec le fichier `terminalkit.js`, mais celui-ci ne sera pas évalué pour la correction.
4. ✎ Quelles sont les propriétés de la programmation fonctionnelles mises en valeur dans ce code ? Justifier.

✓

1. La variable `detectedTerm` est une variable *locale* à la fonction anonyme définie l. 8. Sa portée est valide des lignes 8 à 22. Elle n'est utilisée en fait qu'à la ligne 10.

   La variable `term` est une variable *globale* définir l. 3. Sa portée est valide des lignes 3 à 80 (la fin du fichier). En fait, vu qu'elle est définie avec le mot-clé `var`, sa portée est l'ensemble du fichier, mais pour cette question, ceci est considéré comme un détail. Elle n'est utilisée que dans les fonctions `init` et `terminate`.

2. Le fait d'utiliser une variable globale est certainement critiquable, dans la mesure où l'entièreté du code a accès à cette variable. Il est donc possible de la manipuler de manière inconsidérée, en la modifiant (elle n'est pas constante) ou en y accédant avant sa définition (elle n'est pas initialisée).

   Un exemple d'utilisation incorrect est un simple appel à la fonction `terminate` avant l'appel à `init`, où l'accès se fera avant la définition du terminal.

3. Une solution pour diminuer la critique précédente est d'éliminer complètement cette variable, et de faire que la fonction `terminate` prenne en paramètre une variable `term`. Néanmoins, cela implique de modifier la fonction `inputs` qui fait un appel à `terminate`, et par conséquent la fonction `init`. Les modifications possibles sont les suivantes :

```
function init( callback ) {
    T.getDetectedTerminal( function( error, detectedTerm ) {
        if ( error ) { throw new Error( 'Cannot detect terminal.' ); }
        viewport = new ScreenBuffer({
            dst: detectedTerm,                                  // (*)
            width: Math.min( detectedTerm.width ),              // (*)
            height: Math.min( detectedTerm.height - 1 ),        // (*)
            y: 2 });
        createBackground();
        createSpaceship();
        detectedTerm.hideCursor();                              // (*)
        detectedTerm.grabInput();                               // (*)
        detectedTerm.on( 'key', inputs(detectedTerm) );         // (*)
        callback();
    });
}

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

const inputs = (term) => (key) => {                             // (*)
    switch ( key ) {
        case 'UP' : sprites.spaceship.y --; break;
        case 'DOWN' : sprites.spaceship.y ++; break;
        case 'LEFT' : sprites.spaceship.x --; break;
        case 'RIGHT' : sprites.spaceship.x ++; break;
        case 'q': terminate(term);
}
```

4. Ce code met en valeur les *propriétés de première classe* de la programmation fonctionnelle, en particulier des fonctions *d'ordre supérieur*. Par exemple, la fonction `T.getDetectedTerminal` (l. 8) utilise un paramètre fonctionnel (ici un *callback*), et c'est aussi le cas de la fonction (ici une méthode) `term.on` (l. 20), et de la fonction `setTimeout` (l. 28). Des personnes ont cité la *composition de fonction*, ou même la *pureté* comme des propriétés de ce code, mais c'était difficilement justifiable.

▷ La façon dont l'exercice était posé, de nombreuses personnes ont curryfié la fonction `terminate`, ce qui n'était pas très utile. La seule présence d'une modification de la fonction `inputs` était bonifiée. Il est à noter que peu de personnes ont testé leurs propres modifications de code, ce qui aurait permis d'éviter de nombreuses propositions plus ou moins irréalistes ou mal fondées. En général, les propositions de modification sans code ont été ignorées.

Exercice 4: 5 points

La *médiane* d'un tableau $T = \{T[0], T[1], \dots T[n-1]\}$ de valeurs numériques de n éléments est une valeur $median(T)$ définie comme :

- si T est vide (i.e $n = 0$), alors $median(T) =$ `undefined` ;
- si T n'est pas vide, alors :
  - soit S le tableau contenant les valeurs de T triées dans l'ordre croissant ;
  - si n est impair, alors $median(T) = S[(n-1)/2]$
  - si n est pair, alors $median(T) = \frac{1}{2}(S[n/2-1] + S[n/2])$

Pour les besoins de l'exercice, on fournit la fonction suivante pour trier les tableaux :

```
// Return a copy of 'anArray' sorted in the increasing order
function sortArray(anArray) {
    const aCopy = [...anArray];
    aCopy.sort((a,b) => a-b);
    return aCopy;
}
```

4.1 ⌨ Écrire une fonction `findMedianBySort` qui, étant donné un tableau `anArray`, calcule sa valeur médiane en appliquant l'algorithme de l'énoncé et la fonction `sortArray`.

4.2 ✎ La fonction `findMedianBySort` ainsi écrite est-elle pure ? Est-elle référentiellement transparente ? Justifier.

4.3 ✎ En fonction de vos connaissances d'algorithmique, exprimez la complexité en temps de `findMedianBySort`.

Dans la suite de cet exercice, on se propose d'écrire une variante de l'algorithme précédent pour améliorer sa complexité [1]. Pour cela, on va écrire une fonction `findKthQuick` retournant le k-ème élément (s'il était trié) d'un tableau de valeurs numériques. Les exemples suivants montrent la manière d'appliquer cette fonction. Dans ces exemples, la fonction `randomPick` est une fonction expliquée plus tard.

```
findKthQuick([4,3,2,1],              0, randomPick); // → 1
findKthQuick([4,3,2,1],              3, randomPick); // → 4
findKthQuick([1,10,2,9,3,8,4,7,5,6], 4, randomPick); // → 5
findKthQuick([4,3,2,1],              0, randomPick); // → 1
findKthQuick([4,3,2,1],              5, randomPick); // → error
```

La fonction `findKthQuick` prend trois paramètres :

1. Cette question est inspirée de l'article `https://rcoh.me/posts/linear-time-median-finding` (et le résultat d'une suggestion de S. Lombardy)

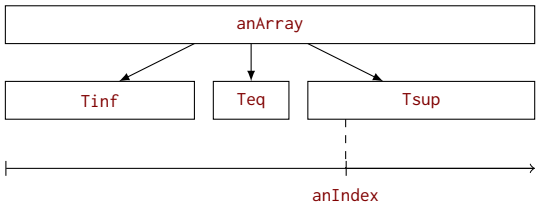
— un tableau `anArray`,

— un nombre `anIndex`,

— une fonction `pickFun` qui prend en paramètre un tableau et renvoie un élément de ce tableau censé servir de pivot.

L'algorithme de cette fonction est le suivant :

1. Sélectionner un pivot `aPivot` avec la fonction `pickFun` ;
2. Construire le tableau `Tinf` des éléments de `anArray` strictement plus petits que `aPivot`, le tableau `Teq` des éléments égaux à `aPivot` et le tableau $T_>$ des éléments strictement plus grands que `aPivot` ;
3. Rechercher récursivement l'élément `anIndex` dans un de ces trois tableaux, selon leur longueur, comme sur le schéma suivant :



Dans l'exemple du schéma, `anIndex` est plus grand que la somme des longueurs des tableaux `Tinf` et `Teq`, donc il faut rechercher récursivement dans le tableau `Tsup` l'élément d'indice `anIndex - Tinf.length - Teq.length`.

Évidemment, il ne faut pas calculer les tableaux dans lesquels on sait que l'on ne devra pas chercher. Par exemple si `anIndex < Tinf.length`, il ne faut pas calculer `Teq` ni `Tsup`. Dans les exemples, la fonction `randomPick` est une fonction tirant un élément au hasard du tableau.

4.4 ⌨ Écrire une fonction *récursive terminale* et *pure* `findKthQuick` prenant en paramètre un tableau `anArray`, un nombre `anIndex`, et une fonction `pickFun`, et renvoyant le `anIndex`-ème élément de `aSortedArray` (où `aSortedArray` est le résultat du tri dans l'ordre croissant de `anArray`) en utilisant l'algorithme de recherche rapide.

4.5 ✎ En considérant que la fonction de pivot permet systématiquement de découper le tableau en trois parties égales, exprimez la complexité en temps de `findKthQuick`.

Remarque insignifiante pour l'exercice : l'algorithme ainsi décrit dépend d'un pivot qu'il faut choisir intelligemment. Il est évidemment possible de choisir ce pivot au hasard, mais il existe aussi une technique déterministe pour trouver un tel pivot, rendant l'algorithme de recherche de médian systématiquement efficace.

1. L'implémentation suivante répond à la question :

```
function findMedianBySort(anArray) {
    if (anArray.length === 0)
        return undefined;
    const aCopy = sortArray(anArray);
    if (aCopy.length % 2 === 1)
        return aCopy[(aCopy.length - 1) / 2];
    else
        return 0.5 * (aCopy[aCopy.length / 2 - 1] + aCopy[aCopy.length / 2]);
}
```

2. Ce n'est pas une fonction pure, dans la mesure où elle effectue, à travers la fonction `sortArray` et la méthode `sort`, un effet de bord en triant la copie du tableau. Néanmoins, cette modification est bornée à une copie locale, donc c'est bien une fonction référentiellement transparente.
3. Si n est la taille du tableau `anArray`, la fonction `findMedianBySort` est probablement de complexité $\mathcal{O}(n \log(n))$, si le tri de tableau est implémenté de manière efficace.
4. L'implémentation suivante répond à la question :

```
function findKthQuick(anArray, anIndex, aPickFun) {
    if ((anIndex >= anArray.length))
        throw new Error('findKthQuick : index over array length');
    else if (anArray.length === 1)
        return anArray[0];
    else {
        const aPivot = aPickFun(anArray);
        const lesserEls = anArray.filter((x) => x < aPivot);
        if (lesserEls.length > anIndex)
            return findKthQuick(lesserEls, anIndex, aPickFun);
        else {
            const middleEls = anArray.filter((x) => x === aPivot);
            if (middleEls.length + lesserEls.length > anIndex)
                return aPivot;
            else {
                const greaterEls = anArray.filter((x) => x > aPivot);
                return findKthQuick(greaterEls,
                                    anIndex - lesserEls.length - middleEls.length,
                                    aPickFun);
            }
        }
    }
}
```

5. Supposons que le tableau soit de taille n. La fonction fait au pire trois filtres du tableau pour calculer `Tinf`, `Teq` et `Tsup`. Ces filtres prennent un temps linéaire $\mathcal{O}(n)$. Ensuite, l'algorithme est appliqué récursivement à un de ces tableaux, dont la taille est $n/3$. Une application directe du Master Theorem permet de conclure, ou plus simplement une explication comme quoi le calcul prend au total un temps linéaire :

$$\mathcal{O}(n) + \mathcal{O}(\frac{n}{3}) + \mathcal{O}(\frac{n}{9}) \ldots = \frac{3}{2}\mathcal{O}(n)$$

▷

— Les réponses aux questions de complexité ont été particulièrement aléatoires. Pour la question 3, sur 84 copies, seules 33 ont parlé d'une complexité quasi-linéaire et 14 d'une complexité quadratique. Les autres propositions étaient plus ou moins fantaisistes. Noter que répondre "la complexité dépend de la complexité de l'algorithme de tri de Javascript, qui n'est pas spécifié" a été considéré comme une réponse vide, au sens où il était attendu une discussion plus précise du problème.

— Le code de la 4ème question valait deux fois plus de points que celui de la 1ère question. Il a été assez mal réussi en général, et surtout mal testé. L'une des difficultés consistait à produire une fonction `aPickFun`. Celle utilisée originellement dans les tests était :

```
const aPickFun = (anArray) ⇒ anArray[Math.random()*anArray.length];
```

Néanmoins, un nombre non négligeable de codes étaient plus ou moins corrects avec la fonction :

```
const aPickFun = (anArray) ⇒ anArray[0];
```

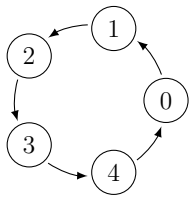
Le code a donc été testé avec les deux fonctions.

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 fournir une fonction `edge` : $V \times V \to$ `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 :

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



✎ 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) ?

✓ Selon le cours, la *représentation des données par des fonctions* est une technique de programmation fonctionnelle dans laquelle les données sont directement des fonctions. L'exemple donné en cours est celui des fonctions caractéristiques sur des ensembles, mais l'idée proposée dans cet exercice est du même ordre : représenter l'ensemble des arêtes d'un graphe par une fonction.
Une type abstrait de données pour de tels graphes pourrait prendre la forme suivante :

```
// The definition of a type for the graphs
type graph<T> = (v1:T, v2:T) ⇒ boolean

// Returns a graph with no edges
function graphCreateEmpty<T>(): graph<T>;
// Given a graph `g`, returns a new graph with the same edges as `g`
// and an additional oriented edge from `v1` to `v2`
function graphAddEdge<T>(g: graph<T>, v1: T, v2: T): graph<T>;
// Given a graph `g`, returns a new graph with the same edges as `g`
// but without edge from `v1` to `v2`
function graphRemoveEdge<T>(g: graph<T>, v1: T, v2: T): graph<T>;
// Given a graph `g`, returns a new graph where there is an edge from
// `v1` to `v2` if and only if there was an edge from `v2` to `v1` in `g`.
function graphRevertOrientation<T>(g: graph<T>): graph<T>
```

Il n'était pas attendu d'utiliser des variables de type dans la réponse, remplacer la variable par `any` était tout à fait acceptable. Par contre, il était attendu une définition de type concrète, ainsi qu'une interface ne faisant pas d'effets de bords (*i.e* pas de `void`) et documentée.
Sur une telle représentation des graphes, il est facile de voir qu'une fonction comme `graphRevertOrientation` n'est pas très difficile à implémenter :

```
function graphRevertOrientation<T>(g: graph<T>): graph<T> {
    return (v1: T, v2: T) ⇒ g(v2, v1); // revert order
}
```

Plus généralement, toutes les fonctions réalisant des opérations logiques (union, intersection, complémentaire ...) et des compositions de fonctions se feront efficacement.
Par contre, on peut imaginer que d'autres opérations comme par exemple lister les sommets du graphe, chercher un chemin entre deux sommets ou calculer une distance ne seront pas aisées à réaliser sans informations supplémentaires sur le graphe. Il pourrait par exemple y avoir du sens à rajouter dans le type de données la liste des sommets du graphe. D'un autre côté, cette représentation est adaptée pour représenter des graphes infinis.

▷ Comme souvent avec les questions de réflexion, cet exercice a été assez mal réussi.

— La notion de représentation des données par des fonctions a trop souvent été confondu avec la notion plus floue de “représentation d’un type de données avec un ensemble de fonctions”, qui s’apparentait à la définition d’un type abstrait de données ou d’interface. Mais ce n’était pas la réponse attendue.

— Ces questions sont trop souvent l’occasion de raconter tout un tas de choses apprises sur la programmation fonctionnelle, de manière plus ou moins aléatoire. La technique paie un peu, puisque la correction favorise les personnes ayant appris leur cours, mais jamais plus de la moitié des points dédiés à la fonction (et moins si ce qui est raconté contient des erreurs).

— Beaucoup de personnes ont presque complètement ignoré l’exemple fourni, même dans les TADs qu’ils ont proposés. Beaucoup de personnes ont ignoré les questions de l’exercice en fait. Fournir des exemples de code, fournir un type pour les graphes, un exemple d’implémentation, et discuter concrètement des opérations faciles et difficiles étaient des manières simples d’engranger des points.

La question a été évaluée sur 9 points (capée à 5, un point de plus qu’indiqué par le barême original), mais la moyenne finale est de 1.14 sur 84 copies.

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.