Année **2022-23** Session **2**

PROGRAMMATION FONCTIONNELLE
PG104

Filière : Informatique, 1ère année Année

Date de l'examen : 27/06/2023 Durée de l'examen : 2h00

Sujet de : D. Renault

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

Exercice 1: 3 points

✎ Dans un langage de programmation, qu'est-ce qu'on appelle une expression ? Donner un exemple. Expliquer en quoi les expressions se marient bien avec la programmation fonctionnelle.

✓ Reprenant l'une des définitions informelles du cours, une expression est la représentation arborescente d'un calcul. Formellement, il s'agit d'un arbre, dont les feuilles sont des valeurs (entiers, booléens ...) et dont les noeuds internes sont des opérations permettant de composes ces valeurs (usuellement des fonctions). Un exemple d'expression très simple :

qui représente le calcul `(1+2)`.

Les expressions constituent un élément central de la programmation fonctionnelle. En effet, elles mettent en valeur la notion de transformation ou de fonction, comme outil pour composer des calculs. Si on le compare à une autre forme de programmation, dans un style impératif, dans lequel les calculs sont décrits par des suites d'instruction, le style sous forme d'expression permet de mettre en valeur la façon dont les calculs dépendent les uns des autres (même si les fonctions impures peuvent contenir d'autres formes de dépendance).

Exercice 2: 6 points

Supposons vouloir écrire un programme capable de lire des expressions parenthésées (i.e ne contenant que les caractères `'('` et `')'`), afin de déterminer celles qui sont bien parenthésées des autres. Pour rappel, une chaîne est bien parenthésée si elle correspond à la chaîne d'une expression arithmétique valide dans laquelle on n'a gardé que les parenthèses. Par exemple, `()()` est bien parenthésée car elle correspond (par exemple) à l'expression `(7*3)+(2*5)`.

Une chaîne bien parenthésée, si elle est non vide, est censée commencer par une parenthèse ouvrante, sinon c'est une erreur. La *parenthèse de premier retour* d'une telle chaîne est définie comme la parenthèse fermante refermant la parenthèse ouvrante initiale. La chaîne vide n'a pas de parenthèse de premier retour. Par exemple :

```
()     # La parenthèse de premier retour est à l'index 1
(())   # La parenthèse de premier retour est à l'index 3
)      # Cette chaîne est mal parenthésée et n'a pas de premier retour
(()    # Cette chaîne est mal parenthésée et n'a pas de premier retour
())    # Cette chaîne est mal parenthesée mais elle a quand même
```

```
# une parenthèse de premier retour à l'index 1
```

1. ⌨ Écrire une fonction `getFirstReturn` de manière *récursive terminale* et *pure* qui, étant donnée une chaîne de caractères `aString` ne contenant que des parenthèses, renvoie l'index de la parenthèse de premier retour si elle existe, `-1` sinon.

   Exemple :

```
getFirstReturn("");      // -1
getFirstReturn("()");    // → 1
getFirstReturn("(())"); // → 3
getFirstReturn(")");     // → -1
getFirstReturn("())");   // → 1
```

Si on applique la fonction précédente de manière répétée, on peut alors découper notre chaîne de caractères en blocs représentant des sous-expressions. Sur l'exemple initial `()()`, l'idée consiste à renvoyer le tableau `[ "()", "()" ]`. Algorithmiquement, cela consiste à calculer la parenthèse de premier retour, extraire la sous-chaîne correspondante, et appliquer récursivement l'algorithme au reste de la chaîne.

2. ⌨ Écrire une fonction `splitParentheses` de manière *récursive terminale* et *pure* qui, étant donnée une chaîne de caractères `aString` renvoie un tableau de chaînes de caractères correspondant à l'application récursive de `getFirstReturn`. Toute forme d'erreur fait renvoyer le tableau vide `[]`.

   Exemple :

```
splitParentheses("");           // → []
splitParentheses("()");         // → [ "()" ]
splitParentheses("(())()");     // → [ "(())", "()" ]
splitParentheses("()()()()()"); // → [ "()", "()", "()", "()", "()" ]
splitParentheses(")");          // → []
```

1. Une implémentation possible de `getFirstReturn` :

```
function getFirstReturn(aString) {
    function getFRRec(anotherString, aLevel, anIndex) {
        if ((anotherString.length <= 0) || (aLevel < 0))
            return -1;
        else {
            const aHead = anotherString[0];
            const aTail = anotherString.slice(1);
            if (aHead === '(')
                return getFRRec(aTail, aLevel + 1, anIndex+1);
            else if (aHead === ')') {
                if (aLevel === 1)
                    return anIndex;
                else
                    return getFRRec(aTail, aLevel - 1, anIndex+1);
            } else
                return getFRRec(aTail, aLevel, anIndex+1);
        }
    }
    return getFRRec(aString, 0, 0);
}
```

2. Une implémentation possible de `splitParentheses` :

```
function splitParentheses(aString) {
    function splitRec(anotherString, anAcc) {
      if (anotherString.length === 0)
          return anAcc;
      else {
          const aHead = anotherString[0];
          if (aHead === ")")
              return [];
          else {
              const aSplit = getFirstReturn(anotherString);
              if (aSplit === -1)
                  return [];
              else {
                  const aFirst = anotherString.substring(0, aSplit+1);
                  const aNext  = anotherString.substring(aSplit+1);
                  return splitRec(aNext, anAcc.concat(aFirst));
              }
          }
      }
    }
    return splitRec(aString, []);
}
```

Exercice 3: 7 points

Supposons vouloir travailler sur les systèmes de fichiers, et vouloir en reconstruire une représentation arborescente. Pour cela, on fournit les fonctions suivantes dans un fichier `given/file.js`, fournissant des fonctions permettant de manipuler les systèmes de fichiers :

```
// Given a string 'aPath', returns a boolean telling if this string is
// a directory or not.
function isDirectory(aPath) { ... }

// Given a string 'aPath', and assuming it corresponds to a valid
// directory on the filesystem, return an array of strings
// corresponding to the files in that directory.
function lsDirectory(aPath) { ... }

// Given a string 'aPath', and assuming it corresponds to a valid path
// string, return the base name of the file associated to that string.
function baseName(aPath) { ... }

// Given a string 'aPath',and assuming it corresponds to a valid path
// string, return the file size in bytes of the file
function fileSize(aPath) { ... }
```

L'objectif de cet exercice est de voir comment manipuler ces arborescences avec des structures d'arbres. Il est donc fourni un fichier `given/tree.js` contenant un ensemble de fonctions habituelles sur les arbres (en particulier une fonction `treeDisp` affichant le contenu d'un arbre sur la console, cette fonction étant *curryfiée*) et un fichier `given/list.js` pour des fonctions sur les listes. Typiquement, on aimerait arriver à construire les arbres suivants :

Pour l'arborescence suivante :

```
.             1byte
|-- a        10bytes
\-- b         1byte
    \-- c    10bytes

2 directories, 2 files
```

. . . l'arbre associé serait :

```
{
  val: ".",
  children:
    cons({
      val: "a",
      children: nil
    },
    cons({
      val: "b",
      children:
        cons({
          val: "c",
          children: nil
        },
        nil),
    },
    nil)),
}
```

1. ⌨ Écrire une fonction `treeDu` de manière *récursive terminale* et *pure* qui, étant donné un chemin dans le système de fichiers, calcule la taille en bytes du sous-répertoire associé.

   Remarques : la taille renvoyée pour l'exemple initial est égale à 22 (bytes). Il est possible de tester ses résultats avec la commande `du -b`, tout en faisant attention de ne tester que sur une arborescence ne contenant que des fichiers normaux et des répertoires (et en particulier *pas* de liens symboliques).

2. ⌨ Écrire une fonction `treeLs` de manière *récursive terminale* et *pure* qui, étant donné un chemin dans le système de fichiers, construit l'arbre des fichiers en dessous de ce chemin.

   Remarque : l'exemple est exactement celui donné ci-dessus, et il est très facile de reconstruire cette arborescence sur son compte pour tester sa fonction.

Ces fonctions ne sont pas évidentes à tester, vu qu'elles dépendent du système de fichiers sur lequel on les exécute. Une technique habituelle en test consiste à remplacer les fonctions utilisées par des *faussaires* qui renvoient des valeurs prévues à l'avance. Par exemple :

```
1  import F from './given/file.js';
2
3  function treeDu(aPath) {
4    ... some code that uses F.fileSize ...
5  }
6
7  function myFileSize(aPath) {
8    if (aPath === "./b")
9      return 100;
10   else
11     return 1;
12 }
13
14 treeDu(".");              // Returns 22  = 1+10+1+10 on the initial example
15 F.fileSize = myFileSize; // Replace fileSize by our function
16 treeDu(".");              // Returns 103 = 1+1+100+1 on the initial example
```

3. ✎ Expliquer ce qu'est la valeur `F` dans le code précédent, d'une part comme valeur Javascript (quel est son type), et ensuite comme élément d'un programme en général (typiquement une notion apparaissant dans le cours).

4. ✎ Quelle(s) est(sont) la(les) propriété(s) de la programmation fonctionnelle utilisée(s) pour appliquer une telle technique ?

5. ✎ Manifestement, l'exécution de `treeDu` ne donne pas le même résultat entre les lignes 14 et 16. Qu'est-ce qui fait que cela fonctionne en Javascript ? Comment ferait-on si on voulait appliquer la même technique en C ?

6. ✎ Expliquer, en détaillant les fonctions utilisées, comment écrire un test utilisant la technique précédente, qui ferait que l'appel `treeDu(".")` utilise l'arborescence donnée dans l'exemple initial, et les tailles fournies par la fonction `myFileSize`.

1. Une implémentation possible de `treeDu` :

```
import F from './utils/file.js';

function treeDu(aPath) {
    const files = F.lsDirectory(aPath);
    return files.map((aFile) => {
        const aFilePath = aPath + "/" + aFile;
        if (F.isDirectory(aFilePath))
            return treeDu(aFilePath);
        else
            return F.fileSize(aFilePath);
    }).reduce( (acc, val) => acc + val, F.fileSize(aPath));
}
```

2. Une implémentation possible de `treeLs` :

```
import F from './utils/file.js';
import * as L from './utils/list.js';
import * as T from './utils/tree.js';

function treeLs(aPath) {
    const files = F.lsDirectory(aPath);
    const children = files.map((aFile) => {
        const aFilePath = aPath + "/" + aFile;
        if (F.isDirectory(aFilePath))
            return treeLs(aFilePath);
        else
            return T.leaf(F.baseName(aFilePath));
    });
    return T.node(F.baseName(aPath), L.arrayToList(children));
}
```

3. Tel quel, `F` est un *objet* Javascript, plus précisément un dictionnaire contenant les fonctions exportées par `file.js`. En programmation, `F` est simplement un *module*.

✓

3. Le module `F` est un objet contenant des fonctions, et on remplace lesdites fonctions par d'autres fonctions. On utilise donc la propriété de première classe, typiquement celle permettant de stocker des fonctions dans des structures de données et de les modifier. La notion de pureté n'est pas utilisée ici.
4. Les fonctions en Javascript sont liées *dynamiquement*. Ainsi, le code de `treeDu`, en faisant un appel à la fonction `F.fileSize`, se réfère au pointeur de fonction stocké dans l'objet `F`. Si ce pointeur est modifié, les exécutions de `treeDu` vont elles aussi être modifiées. C'est à mettre en regard du C, où les fonctions sont liées *statiquement*.

   Si on avait voulu appliquer la même technique en C, une solution aurait été d'utiliser un pointeur de fonction dans le code de `treeDu`, pointeur que l'on aurait modifié de l'extérieur. Les réponses à base de recompilation du code étaient aussi acceptées.
5. Il suffit de remplacer les trois fonctions `F.lsDirectory`, `F.isDirectory` et `F.fileSize`, en mettant des faussaires appropriés. Les fonctions seraient de la forme :

```
F.isDirectory = (aPath) ⇒ aPath === "b";
F.lsDirectory = (aPath) ⇒ {
  switch (aPath) {
    case ".":   return [ "./a", "./b" ];
    case "./b": return [ "./b/c" ];
    default:    return [];
  }
};
```

Exercice 4: 4 points

La bibliothèque Ramda (cf. `https://ramdajs.com`) se présente comme une bibliothèque de programmation fonctionnelle pratique pour les programmeurs Javascript[1]. Dans l'exemple suivant, cette bibliothèque est utilisée pour écrire une requête manipulant les éléments d'un tableau.

Remarques : le site `https://ramdajs.com` est accessible et contient la documentation de toutes les fonctions utilisées ici. Il n'est pas nécessaire de comprendre l'intégralité du fonctionnement de ce code pour répondre à la question. Le code est suivant est donné dans un fichier `given/ramda.js` exécutable et modifiable pour tester ses effets, mais qui ne fait pas partie du code évalué.

[1] "A practical functional library" sur le site `https://ramdajs.com`.

```
import * as R from 'ramda';

const fetchData = function() {
    return new Promise((resolve) => resolve(
        { tasks : [
            { id: 1, username: 'bob', complete: false, dueDate: 1981, title: "Work" },
            { id: 2, username: 'amy', complete: false, dueDate: 1961, title: "Work" },
            { id: 3, username: 'bob', complete: true,  dueDate: 2005, title: "Sleep" },
            { id: 4, username: 'sue', complete: true,  dueDate: 2023, title: "Work" },
            { id: 5, username: 'bob', complete: false, dueDate: 2012, title: "Eat" },
        ] }
    ));
}

const Rget = R.curry((name, obj) => obj[name]);

const getIncompleteTaskSummaries = function(membername) {
    return fetchData()
        .then(Rget('tasks'))
        .then(R.filter(R.propEq(membername, 'username')))
        .then(R.reject(R.propEq(true, 'complete')))
        .then(R.map(R.pick(['id', 'dueDate', 'title'])))
        .then(R.sortBy(Rget('dueDate')));
};

const result = await getIncompleteTaskSummaries('bob');
console.log(result);
```

L'exécution de ce code produit le résultat suivant :

```
[
  { id: 1, dueDate: 1981, title: 'Work' },
  { id: 5, dueDate: 2012, title: 'Eat' }
]
```

✎ Rappeler les principes généraux de la première classe en programmation fonctionnelle.
Illustrer ces principes en vous appuyant sur l'exemple ci-dessus.
Expliquer en quoi un tel code est intéressant (i.e quelles sont ses qualités).

✓ 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 (cf. l.17 l'affectaction dans `getIncompleteTaskSummaries`) ;
- être créée à la demande dans n'importe quel contexte (cf. l.15 dans l'appel à `curry`);
- être passée comme argument à une fonction (cf. l.19 à 23 dans tous les appels à `then`);
- être retournée comme résultat d'une fonction (cf. l.15 dans l'appel à `curry`, ce qui se justifie en se référant à la documentation de Ramda, qui renvoie un équivalent currifié de la fonction en paramètre);
- apparaître dans n'importe quelle structure de données (plus difficile à justifier dans ce code, mais `R` est un dictionnaire contenant des fonctions)

Le fait que le code soit court n'est pas sa qualité première, même si on peut arguer du fait que cela le rendre plus lisible (et même cela est discutable, "court" n'implique pas forcément "compréhensible").
Principalement, le code fourni est intéressant dans la mesure où il illustre la notion de *composition* de fonctions. Principalement, le fait d'enchaîner les appels à `then` dans les lignes 19 à 23, que l'on peut tester en les commentant, montre qu'on compose des transformations sur la liste initiale fournie par `fetchData`.
Le deuxième point intéressant est l'illustration de l'utilisation de fonctions *curryfiées* pour faciliter leur *réutilisation*. Une fonction comme `Rget` écrite à la ligne 15 est mise sous forme curryfiée, et réutilisée deux fois aux lignes 19 et 23. C'est en fait aussi le cas de toutes les fonctions de la blbiothèque Ramda utilisées dans ce code.