LIFAP5 – Programmation fonctionnelle pour le WEB
TD3/4 – programmation asynchrone en JavaScript

Licence informatique UCBL – Printemps 2021–2022

Exercice 1 : Fermetures

1. Dire quelle est la différence d'exécution entre ces deux programmes :

```
1 let tab = [];
2 for(var i = 0; i < 3; ++i) {
3
4     tab.push( () => i );
5 }
6 let t = tab.map( f => f() );
```

```
1 let tab = [];
2 for(var i = 0; i < 3; ++i) {
3     const j = i;
4     tab.push( () => j );
5 }
6 let t = tab.map( f => f() );
```

2. Réécrire la version de gauche utilisant `var i` avec une *Immediately invoked function expression* (IIFE) de telle façon qu'on obtienne le comportement de la version de droite, mais sans utiliser ni `let` ni `const`.

Exercice 2 : Décorateurs

Un décorateur est une fonction qui "modifie le comportement d'une autre fonction". Plus précisément, c'est une fonction d qui prend une fonction f unaire en argument, telle que $d(f)$ est la fonction f dont le comportement est modifié. Donner *le type et le code* pour les décorateurs suivants :

`maybe(f,v)` appelle `f` et si `f` renvoie `undefined`, alors renvoie `v` à la place.

`once(f)` au premier appel, renvoie le résultat renvoyé par `f` et renvoie ce même résultat, quels que soient les arguments passés en paramètres, pour les appels suivants (sans rappeller `f`).

`memoize(f)` si `f` a déjà été appelée avec le même argument, renvoie directement[1] la valeur retournée précédemment par `f` pour cet argument. On supposera que `f` ne prend qu'un seul argument qui peut être utilisé come clé d'un dictionnaire JavaScript. On rappelle que la méthode `hasOwnProperty` permet de tester si un objet possède un champ correspondant à une certaine clé.

`chain(n)(f)` enchaine la fonction `f` (supposée unaire) n fois, c'est-à-dire renvoie la fonction $x \mapsto f(f(\ldots(f(x))\ldots))$ où f est appellée n fois. Si l'entier n est nul alors la fonction renvoyée est $x \mapsto x$. On donnera une version *itérative* et une version *récursive* de `chain(n)(f)`.

1. *i.e.* sans rappeler `f`

Exercice 3 : Calcul asynchrone avec des promesses

```
function make_promise(str, time, ok=true){
  if(ok)
    return new Promise((resolve, reject) => {
      setTimeout(() => resolve(str), time);
    });
  else
    return new Promise((resolve, reject) => {
      setTimeout(() => reject(str), time);
    });
  }
let p1 = make_promise("P1", 1000, true);
let p2 = make_promise("P2",  500, true);
let p3 = make_promise("P3",  750, false);
// p1.then(console.log); p2.then(console.log); p3.catch(console.log);
```

1. Que fait la fonction `make_promise` ?
2. Qu'affiche le programme sur la console ?
3. Qu'affiche le programme sur la console si on lui ajoute à la fin `p1.then(console.log); p2.then(console.log); p3.catch(console.log);`
4. Une seconde après l'exécution précédente, on exécute `p1.then(console.log); p2.then(console.log); p3.catch(console.log);`. Que s'affiche-t-il ?
5. Qu'affiche `make_promise("PA", 100).then((x)=> make_promise("PB", 200)).then(console.log);`

Exercice 4 : Appel asynchrone et rendu

On souhaite créer une fonction utilitaire `chargeInsere(rendu)` pour faciliter l'insertion de contenu obtenu via un échange réseau JavaScript avec le serveur. La fonction `chargeInsere(rendu)` prend une fonction `rendu` en argument et *renvoit une fonction qui prend deux arguments* :

— une URL, paramètre nommé `url` ;
— l'identifiant d'un élément HTML de la page, paramètre nommé `id`.

La fonction `rendu` passée en paramètre est une fonction de rendu qui attend un objet JSON et renvoit une chaîne de caractère contenant du code HTML pour présenter les données de l'objet JSON. Un exemple typique de fonction de rendu est donné ci-après :

```
let mon_rendu = arr => arr.reduce(
    (acc,x) => (acc += x.titre.toUpperCase() + "</br>"), "");
```

La fonction renvoyée par `chargeInsere(rendu)` a le comportement suivant :

1. récupérer le contenu JSON (type `string`) situé à l'adresse `url` avec `fetch` puis le parser ;
2. appeler la fonction `rendu` sur l'objet JSON (de type `object`) ;
3. insérer le texte obtenu dans l'élément identifié par `id`.

On rappelle l'existence des fonctions / champs suivants :

— `fetch(url)` télécharge de manière asynchrone le contenu disponible à l'adresse `url` et renvoie une `Promise` de la réponse HTTP ;
— `document.getElementById(id)` retourne l'élément HTML identifié par `id` ;
— Si `elt` est un élément HTML, alors `elt.innerHTML` est un champ qui représente son contenu HTML sous forme d'une chaîne de caractères.

1. Expliquer ce que fait la fonction `mon_rendu` donnée en exemple. La réécrire avec `Array.prototype.map()` et `Array.prototype.join()`
2. Définir la fonction `chargeInsere(rendu)`.

Corrections

Solution de l'exercice 1

Objectif : réfléchir aux valeur utilisées dans les fonctions (fermetures). Pointer les problèmes inhérents au fonctionnement des fermetures en JavaScript. Ces questions ont déjà été posées en TP mais il faut enfoncer et faire la variante IIFE.

1. Dans la version de gauche, la variable `var i` est référencée dans les fermetures qui sont construites et ajoutées dans le tableau. Lorsque l'on appelle ces fermetures avec le `map` ligne 6, `i` est déréférencée et la valeur obtenue pour `t` est `[ 3, 3, 3 ]` car 3 est valeur de `i` en sortie de boucle, sans portée étant avec la déclaration `var` toute la fonction.
   Dans la version de droite, c'est la constante `j` (qui est recréée à chaque tour de boucle) qui est référencée. Elle n'est pas modifiée, on a donc $\texttt{t} = \texttt{[0, 1, 2]}$.
2. ES5 ne proposant pas de variables ou de constante à portée bloc (uniquement des `var`), la solution consistait avant ES6 à créer une fermeture avec une IIFE pour capture la variable `i` lors des tours de boucle :

```
1  let tab = [];
2  for(var i = 0; i < 3; ++i) {
3      (function (x) {
4        tab.push( () => x );
5      }) (i);
6  }
7  let t = tab.map( f => f() );
8
9  // in fine le plus élégant est peut être le suivant, avec le let/
       const dans la boucle for
10 let tab = [];
11 for(let i = 0; i < 3; ++i) {
12     tab.push( () => i );
13 }
14 let t = tab.map( f => f() );
```

   Notons enfin que ce programme simple a un réel intérêt en pratique : il démontrr comment contrôler avoir une forme *d'évaluation paresseuse* en JavaScript. En effet, le corps d'une fonction n'est évalué que lorsque celle-ci est effectivement appelée : en η-convertissant une expression, on peut contrôler le moment où elle sera évaluée. En un sens, le tableau `t` contient des « entiers paresseux ».

Solution de l'exercice 2

`maybe` : $(\alpha \to \alpha') \times \alpha' \to \alpha \to \alpha'$

```
1  // solution de base
2  function maybe(f,v) {
3    return function(x) {
4      const res = f(x);
5      // pour éviter de calculer 2 fois f(x)
6      if (res === undefined) {
7        return v;
8      } else {
9        return res;
10     }
11   };
12 }
13
```

```javascript
// avec une flèche et une if expression, mais on calcule 2 fois f(x)
const maybe = (f,v) => x => (f(x)===undefined)?v:f(x);

// une version **impure** (non fonctionnelle) où on utilise le fait qu'
// une affectation (x = v) a pour valeur celle de l'expression 'v'
// par exemple, console.log(x = 2) affiche 2
const maybe = (f,v) => x => ((res = f(x)) ===undefined)?v:res;

// avec une closure supplémentaire, comme dans l'exercice précédent une
// IIFE qui va calculer f(x) UNE SEULE fois car la variable x est
// **remplacée** à 2 endroits par la valeur calculée une seule fois
const maybe = (f,v) => x => (r => (r===undefined)?v:r)(f(x));

// Notons qu'on peut gérer le cas d'une fonction f variadique avec les
// rest paramaters et le spread operator :
// rest parameters permet de transformer des paramètres en Array
//    https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/rest_parameters
// le spread operator fait le contraire
//    https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_syntax
function maybe(f,v) {
  return function(...x) {
    const res = f(...x);
    if (res === undefined) {
      return v;
    } else {
      return res;
    }
  };
}

// version avec des flèches, rest/spred opérators et IIFE
const maybe = (f,v) => (...x) => (r => (r===undefined)?v:r)(f(...x));
```

once : $(\alpha \to \alpha') \to \alpha \to \alpha'$

```javascript
// solution de base
function once(f) {
  let has_run = false;
  // ce booléen est là pour gérer le cas où f renvoie undefined
  let previous = undefined;
  return function(x) {
    if (!has_run) {
      has_run = true;
      previous = f(x);
    }
    return previous;
  };
}

// En version variadique avec rest/spread
function once(f) {
  let has_run = false;
  let previous = undefined;
  return function(...x) {
    if (!has_run) {
      has_run = true;
      previous = f(...x);
```

```
    }
    return previous;
  };
}
```

memoïze : $(\alpha \to \alpha') \to \alpha \to \alpha'$ On reprend finalement la technique de once mais en stockant les retour de `f(x)` dans un objet utilisé comme un dictionnaire. On utilise la méthode `hasOwnProperty` pour éviter de tester `results[x] === undefined` qui ne fonctionne pas quand `f(x) === undefined`.

```
// Solution de base
function memoize(f) {
  let results = {}
  return function(x) {
    if ( ! results.hasOwnProperty(x) ) {
      results[x] = f(x);
    }
    return results[x];
  };
}

// La difficulté pour le codage variadique ici est que les clés de dictionnaire sont des chaînes de caractères.
// Les tableaux sont ainsi convertis automatiquement en {string}.
// Une possibilité consiste à encoder les paramètres avec {JSON.stringify}:
function memoize(f) {
  let results = {}
  return function(...x) {
    const k = JSON.stringify(x);
    if ( ! results.hasOwnProperty(k) ) {
      results[k] = f(...x);
    }
    return results[k];
  };
}
```

chain : number $\to (\alpha \to \alpha) \to (\alpha \to \alpha)$

```
// solution de base récursive
function chain(n) {
  return function(f) {
    return function(x) {
      if (n <= 0)
        return x;
      else{
        return chain(n-1)(f)(f(x));
        // ou
        // return f(chain(n-1)(f)(x));
      }
    }
  }
}

chain_r = (n) => (f) => (x) => (n <= 0)?(x):f(chain(n-1)(f)(x));
//ou
chain_l = (n) => (f) => (x) => (n <= 0)?(x):(chain(n-1)(f)(f(x)));

// solution itérative
function chain(n) {
```

```
  return function(f) {
    return function(x) {
      let ret = x;
      for (let i = 0; i < n; i += 1){
        ret = f(ret);
      }
      return ret;
    }
  }
}
```