Final CAP

Curs 2020-21 (13/1/2021) **Duració: 2 hores**

1.- (1.5 punts) Implementeu en Javascript/Rhino una funció **callcc(f)** que funcioni com l'estructura de control que ja coneixeu de Pharo, fent servir, naturalment, la funció **Continuation()** de Javascript/Rhino.

Solució:
Recordem com funciona **callcc**: Aquesta funció s'invoca amb una funció com a paràmetre, que s'invoca amb la continuació corresponent a la crida a **callcc** com a paràmetre:

```
function callcc(f) {
      var k = new Continuation();
      return f(k);
}
```

2.- (2 punts) Executa aquest codi i justifica el resultat que obtens:

```
let temp

function f(x) {
    let temp = x
    return function () { return temp }
}

function g(x) {
    temp = x
    return function () { return temp }
}

//[a,b,c,...].map(foo) aplica foo a cada element i retorna [foo(a),foo(b),foo(c),...]
let qf = [1,2,3,4,5].map(f)
let qg = [1,2,3,4,5].map(g)

//[a,b,c,...].forEach(foo) aplica foo a cada element però no retorna res (undefined)
qf.forEach(function (e) {console.log(e())})
console.log("----")
qg.forEach(function (e) {console.log(e())})
```

Solució:
En executar aquest codi obtenim el resultat:

```
1
2
3
4
5
----
5
5
5
5
5
```

La justificació és senzillament que la funció/*closure* que retorna **f** captura la variable local **temp**, per tant cada cop que s'invoca **f**, la variable capturada per la funció retornada és diferent. En canvi, la funció/*closure* que retorna **g** fa referència a una *única* variable **temp** *global*, que veu modificat el seu valor cada cop que invoquem **g**, i acaba valent 5.

3.- (2 punts) Donada la funció recursiva

```
function collatz(n) {
    if (n === 1) {
        return 0
    } else {
        let m = (n % 2 === 0) ? n/2 : 3*n+1 // operador ternari
        return 1 + collatz(m)
    }
}
```

trobeu una funció equivalent recursiva final. Utilitzeu la transformació a *Continuation Passing Style* (CPS).

Solució: Aplicant CPS, obtenim la funció equivalent:

```
function collatzCPS(n, ret) {
    if (n === 1) {
        return ret(0)
    } else {
        let m = (n % 2 === 0) ? n/2 : 3*n+1
        return collatzCPS(m, function (r) {
                                 return ret(1 + r)
                             })
    }
}
```

4.- (2 punts) Tenim una funció recursiva final **my_map**, que aplica una funció **f** a tots els elements d'un array. Si **arr** és l'array al que volem aplicar **f**, s'ha de cridar **my_map(arr,f,[])**:

```
function my_map (arr, f, res) {
    if (arr.length === 0) {
        return res
    } else {
        let [car, ...cdr] = arr // car és el primer element, cdr és la resta de l'array
        res.push(f(car))
        return my_map(cdr,f,res)
    }
}
```

Aleshores: **my_map([1,2,3,4,5], x => x+1, [])** retorna **[2,3,4,5,6]**

Sabem que si fem servir Node.js, que *no* fa TCO, tindrem problemes amb la pila. Si fem , per exemple, **my_map([...Array(10000).keys()], x => x+1, []),** obtindrem un error **RangeError: Maximum call stack size exceeded**. Apliqueu la técnica del *trampolining* per obtenir una versió de **my_map** que no tingui problemes amb la mida de la pila.

Solució: Apliquem l'esquema general que us vaig passar quan vam explicar el *trampolining*:

```
const my_map_tramp = (function () {
    function __f(a, f, r) {
        if (a.length > 0) {
            return function () {
                let [car, ...cdr] = a
                r.push(f(car))
                return __f(cdr,f,r)
            };
        }
        return r;
    };
    return function (arr, f, res) {
        return trampoline(__f(arr,f,res))
    }
})()
```

suposant que tenim la funció que ja coneixem:

```
function trampoline (fun) {
    while (typeof fun == 'function') {
        fun = fun();
    }
    return fun;
}
```

Així, si fem `my_map_tramp([...Array(10000).keys()], x => x+1, [])` obtenim el resultat esperat sense problemes.

5.- (2.5 punts) Feu servir la implementació en Javascript (sense classes) del patró **Factory** que vam veure a classe per implementar una fàbrica de consoles de joc. Cada consola ha de tenir dues propietats, la marca (que diu quina empresa la fabrica) i el màxim de fps (*frames per second*) que suporta (us podeu inventar el número, p.ex. 120). Aleshores, la sortida del codi:

```
let ps = FabricaConsoles.factory('ps5'),
    xb = FabricaConsoles.factory('xbox')
console.log(ps.marca());
console.log(ps.maxfps());
console.log(xb.marca());
console.log(xb.maxfps());
```

hauria de ser quelcom similar a:

```
Consola de joc: sony
Maxim d'fps 120
Consola de joc: microsoft
Maxim d'fps 120
```

Solució:
Una possible solució seria una de *molt* semblant a la que ja vam veure a classe del **`CarMaker`**. De fet, només us calia modificar molt lleugerament el codi mostrat a classe:

```
// constructor

function FabricaConsoles() {}

// Mètodes de la classe pare

FabricaConsoles.prototype.marca = function () {
    return "Consola de joc: " + this.fabricant
}

FabricaConsoles.prototype.maxfps = function () {
    return "Maxim d'fps " + this.maximfps
}

// el mètode estàtic de la fàbrica

FabricaConsoles.factory = function (type) {
    var constr = type,
        newcon;

    if (typeof FabricaConsoles[constr] !== "function") {
            throw { name: "Error",
                        message: constr + " doesn't exist"
                  };
    }

    if (typeof FabricaConsoles[constr].prototype.marca !== "function") {
         FabricaConsoles[constr].prototype =  new FabricaConsoles();
         FabricaConsoles[constr].prototype.constructor = FabricaConsoles[constr];
    }

    newcon = new FabricaConsoles[constr](); // crea una nova instància

    // aqui podriem fer més feina i en acabar retornem...

    return newcon;
}

// defineix objectes que pot fabricar, en forma de constructors associats a
// propietats de la fàbrica

FabricaConsoles.ps5 = function () {
    this.fabricant  = "sony"
    this.maximfps   = 120
};

FabricaConsoles.xbox = function () {
    this.fabricant  = "microsoft"
    this.maximfps   = 120
};
```