Parcial CAP

Curs 2022-23 (2-XI-2022) Duració: 2 hores

1.- (2.5 punts) Una botiga d´arbres de Nadal ofereix l´opció de vendre els arbres ja adornats (tot i que es pot comprar l'arbre sense adornar). La botiga ofereix una sèrie d'opcions per adornar l'arbre i cada client escull les que desitja: boles, caramels, garlandes, penjolls, cintes, figures d'animals, etc. Imaginem que, per simplificar, només tenim *tres* opcions per guarnir l'arbre: boles, cintes i caramels. Cal considerar que els guarniments poden ser escollits pel client en qualsevol ordre i no necessàriament els ha de triar tots.

Exemples:

```
ad = new Arbre();
ad = ad.decorate('Cinta');
ad = ad.decorate('Bola');
ad = ad.decorate('Bola');
ad.print(); // retorna ==> 'Bola, Bola, Cinta, Arbre'

ad = new Arbre();
ad = ad.decorate('Bola');
ad = ad.decorate('Caramel');
ad = ad.decorate('Cinta');
ad.print(); // retorna ==> 'Cinta, Caramel, Bola, Arbre'
```

Solució:

```
function Arbre() {
    this.arbrenadal = 'Arbre';
}

Arbre.prototype.print = function () {
    return this.arbrenadal;
};


// No canvia respecte la que ja vam veure a classe...
Arbre.prototype.decorate = function (decorator) { ... }

// Els decoradors s'implementaran com a propietats d'una propietat del constructor

Arbre.decorators = {};

Arbre.decorators.Bola = {
    print: function () {
            let an = this.uber.print();
            an = 'Bola, ' + an;
            return an;
    }
};
```

```
Arbre.decorators.Cinta = {
    print: function () {
            let an = this.uber.print();
            an = 'Cinta, ' + an;
            return an;
    }
};

Arbre.decorators.Caramel = {
    print: function () {
            let an = this.uber.print();
            an = 'Caramel, ' + an;
            return an;
    }
};
```

2.- (1 punt) A Haskell tenim una funció `flip` definida: `flip f x y = f y x` (no cal saber Haskell per entendre aquesta expressió). Podríem definir un `flip` a Javascript tal que `flip(f)(x,y) = f(y,x)`. Fixeu-vos que `flip` té una funció (de dos paràmetres) com a paràmetre i retorna una funció (també de dos paràmetres). Aleshores, per exemple, `flip((x,y) => x - y)(5,3) = -2`
Escriu la funció `flip`.

Solució:

```
function flip(f) {
    function aux(x,y) {
        return f(y,x)
    }
    return aux
}
```

3.- (4 punts) Donada la funció recursiva `scanl(f,x0,arr)`, on `arr = [x1,x2,x3,...,xN]`, que retorna el següent array: `[x0, f(x0,x1), f(f(x0,x1),x2), f(f(f(x0,x1),x2),x3),...]`:

```
function scanl(f,x0,arr) {
    if (arr.length === 0) {
        return [x0]
    } else {
        let [car, ...cdr] = arr;
        return [x0].concat(scanl(f,f(x0,car),cdr))
    }
}
```

a/ (2 punts) Converteix-la a recursiva final fent servir *Continuation Passing Style* (on tot el que ve donat en Javascript no cal passar-ho a CPS, ho considerem operacions *primitives*).

b/ (2 punts) Transforma la funció obtinguda a l'apartat anterior en una funció que no tingui problemes amb la mida de la pila, fent servir la tècnica del *trampolining*.

Solució:

```
a/
function scanl_cps(f,x0,arr,cont) {
    if (arr.length === 0) {
        return cont([x0])
    } else {
        let [car, ...cdr] = arr;
        return scanl_cps(f,
                         f(x0,car),
                         cdr,
                         function(v) { return cont([x0].concat(v)) })
    }
}

b/ Fem servir la funció 'trampoline' que ja vam veure a classe

function scanl_cps_trampolined(func,x_inicial,llista,continuacio) {
    function __f(f,x0,arr,cont) {
        if (arr.length === 0) {
            return cont([x0])
        } else {
            let [car, ...cdr] = arr;
            return function() {
                return __f(f,
                           f(x0,car),
                           cdr,
                           function(v) { return cont([x0].concat(v)) })
            }
        }
    }
    return trampoline(__f(func,x_inicial,llista,continuacio))
}
```

4.- (2.5 punts) [Rhino] Digueu detalladament què fa aquesta funció (`callcc` és la funció que ja vam veure a classe).

```
function arg_fc() {
    return callcc(function(k) {
        k( function(x) {
            k( function(y) {
                return x;  })})});
};
```

Per exemple, mireu d'executar això (però podeu fer més proves per mirar d'esbrinar què fa aquest programa):

```
let g = arg_fc();
g(2);
print(g(1000));
```

Solució:

Aquest programa, en ser executat per primer cop, **`arg_fc()`**, retorna la funció:

```
function(x) {
  k( function(y) {
       return x;
     })
}
```

que és assignada a **g**. En fer **`g(2)`** s'executa aquesta funció, que torna al punt on vam retornar una funció que va ser assignada a **g**. Aquest cop, la funció assignada a **g** és:

```
function(y) {
   return x;
}
```

que, com la primera invocació ha estat amb paràmetre **2**, **x** té com a valor **2**. Aquesta funció *sempre* retornarà **2**, no importa quin sigui el seu argument.

Així doncs, bastant obvi a partir d'aquest experiment, hem aconseguit una funció que, a partir de la segona invocació, sempre retornarà l'argument de la primera invocació. Així, en fer **`print(g(1000))`** s'escriurà **2**.