FIB

CONCEPTES AVANÇATS DE PROGRAMACIÓ
Temes 1 & 2
Col·lecció de Problemes

Recull per Jordi Delgado
(Dept. CS, UPC)

Grau d'EI, 2022-23
FIB (UPC)

Tema 1

Introducció: Abast i Closures:

1. Ja sabeu que a Javascript l'abast (*scope*) de les variables `var` és un abast de funció (*hoisting*). L'estàndar ECMAScript 6 va introduïr la possibilitat de declarar variables amb abast de bloc (utilitzant `let` en lloc de `var`), que són essencialment les que ja coneixeu i utilitzeu a Java o a C++. Expliqueu-ne la diferència i il·lustreu-ho amb un petit exemple de codi (us l'heu d'inventar). Expliqueu què és la *Temporal Dead Zone*.

2. 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())})
```

3. Executa aquest codi i *justifica* el resultat que obtens, és a dir, el valor de la variable `result`:

```
function misteri(n){
   let secret = 4;
   n += 2;
   function misteri2(mult) {
      mult *= n;
      return secret * mult;
   }
   return misteri2;
}
function misteri3(param){
   function misteri4(bonus){
      return param(6) + bonus;
   }
   return misteri4;
}
let h = misteri(3);
let j = misteri3(h);
let result = j(2);
```

4. Sabem que (quasi) tot objecte en Javascript té un prototipus (un altre objecte al que fa referència). I sabem que tot objecte-funció (objectes invocables) conté una propietat anomenada `prototype`. Aleshores, respón a aquestes qüestions:
   a) En general, el *prototipus* d'un objecte-funció i el `prototype` d'aquest objecte-funció són el mateix objecte?
   b) Hi ha cap excepció a la regla general?
   c) Per a què serveix el `prototype` d'un objecte-funció?

Orientació a objectes sense classes:

5. Vam veure a classe que, si volem simular les classes tradicionals en JavaScript, à la Java o Smalltalk, ho podiem fer amb certa facilitat. La simulació es complica una mica si a més volem tenir també herència. Una manera de fer-ho és via prototipus. Si volem fer que el prototipus dels objectes instància de B hereti del prototipus dels objectes instància d'A, cal fer:

   ```
   B.prototype = Object.create(A.prototype);
   B.prototype.constructor = B;
   ```

   Què passa amb aquesta solució si mètodes heretats per les instàncies de B *es volen fer servir*? *Com ho podem arreglar*?

   Aquest codi us pot servir d'exemple. Volem que la classe B sigui subclasse de la classe A. Fixeu-vos en la sortida: No és el que esperaríem...

   ```
   function A() {
      this.a = 0;
      this.b = 1;
   }
   A.prototype.retornaA = function() { return this.a }
   A.prototype.retornaB = function() { return this.b }
   // provem...
   let aa = new A();
   aa.a = aa.a + 1;
   aa.b = aa.b + 1;
   console.log(aa.retornaA());
   console.log(aa.retornaB());
   function B() {
     this.a = 100;
     this.c = 101;
   }
   B.prototype = Object.create(A.prototype);  // el que hem vist a classe
   B.prototype.constructor = B;
   B.prototype.retornaC = function() { return this.c }
   // provem...
   let bb = new B();
   console.log(bb.retornaA());
   console.log(bb.retornaB());
   console.log(bb.retornaC());
   ```

6. Suposem que tenim tres funcions constructores `A`, `B` i `C`. Volem que els objectes construïts per la funció `C` puguin utilitzar les funcionalitats que proporcionen les funcions constructores `A` i `B` (en un món OO amb classes i herència simple, diriem que `C` és una subclasse de `B` i que `B` és una subclasse d'`A`). Per exemple, si els objectes creats amb `A` tenen una propietat anomenada `propA` (de contingut inicial `a`), els objectes creats per `B` tenen una propietat anomenada `propB` (de contingut inicial `b`) i els objectes creats amb `C` tenen una propietat anomenada `propC` (de contingut inicial `c`), el resultat d'executar:

   ```
   let c = new C();
   console.log(c.propA);
   console.log(c.propB);
   console.log(c.propC);
   ```

   seria:

   ```
   a
   b
   c
   ```

7. Recordeu el fitxer `Singleton1wrong.js` que us vaig passar pel Racó. Per què diem que està *wrong*? Quin és el problema exactament? Fes una descripció del funcionament del programa i en quin punt falla.

8. Recordeu que el patró Decorador és útil per resoldre el problema d'un objecte que necessita ser modificat, afegint-li propietats (no confondre amb les "propietats" de Javascript) en temps d'execució. Ja vam veure un exemple de patró Decorador a classe, l'exemple amb la funció `Sale`. Ara, com que segur que sentiu nostàlgia de l'assignatura de PROP, us demano que utilitzeu el patró Decorador per implementar el famós exemple del Café, és a dir, el programa per a que el següent codi funcioni correctament:

   ```
   var c = new Cafe();
   c = c.decorate('llet');
   c = c.decorate('xocolata');
   c = c.decorate('crema');
   console.log(c.getPrice());
   console.log(c.getIngredients());
   ```

   que té com a sortida:

   ```
   2.75
   Cafe, llet, xocolata, crema
   ```

9. 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
```

Tema 2

Continuacions. `Continuation()` *a Rhino*:

10. El llenguatge de programació C disposa d'una llibreria que proveeix de dues funcions, `setjmp` i `longjmp` (*header* `setjmp.h`, si es vol fer servir). Veieu-ne dos exemples (compilar amb `gcc -o exemple exemple.c` i executar amb `./exemple`):

exemple1.c

```
#include <stdio.h>
#include <setjmp.h>

int main() {
   jmp_buf env;
   int i;

   i = setjmp(env);
   printf("i = %d\n", i);

   if (i != 0)
      return(0);

   longjmp(env, 2);
   printf("surt aixo?\n");
}
```

RESULTAT DE COMPILAR I EXECUTAR:

```
i = 0
i = 2
```

exemple2.c

```
#include <stdio.h>
#include <setjmp.h>
jmp_buf buf;

void segon(void) {
   printf("segon\n");
   longjmp(buf,1);
}

void primer(void) {
   segon();
   printf("primer\n");
}

int main() {
   if (!setjmp(buf))
      primer();
   else
      printf("main\n");
   return 0;
}
```

RESULTAT DE COMPILAR I EXECUTAR:

```
segon
main
```

Ja sé que no hem vist ni C (tot i que el coneixeu d'altres assignatures) ni la parella `setjmp`/`longjmp`, però a classe hem parlat molt de reificar la pila d'execució, de continuacions, de salts no locals, etc. Amb tot aquest coneixement i aquest dos exemples senzills us podeu fer una idea *aproximada* de què fan `setjmp`/`longjmp` (investigueu una mica).
Així doncs, expliqueu què fan `setjmp`/`longjmp` i, a partir del que heu deduit, feu una comparació amb la funció `Continuation()` de Javascript/Rhino.

11. Hem fet servir a diversos exemples de l'ús de `Continuation()` la funció `current_continuation()`, que definíem d'aquesta manera:

```
function current_continuation() {
    k = new Continuation();
    return k
}
```

Com podem aconsegur la *mateixa* funcionalitat fent servir només `callcc`?

12. Aquest programa és molt similar a un exemple que vam veure a classe i que teniu perquè us el vaig passar via Racó:

```
(function () {
      let value = 0;
      let kont = new Continuation();
      print(value);
      if (value === 5)
            print("Ha arribat a 5 gracies a la continuacio");
      else {
            value++;
            kont(kont);
      }
})();
```

    Però aquest programa no imprimeix tots els valors del 0 al 5. La seva sortida és senzillament 0. Argumenta per què aquest programa no fa el mateix que l'exemple que vam veure a classe, i per tant no funciona bé.

13. Quina diferència hi ha entre l'expressió

```
callcc( function (k) { return 2 + 4 * k(5) } )
```

    i l'expressió

```
2 + 4 * callcc( function (k) { return 5 } )
```

    Quin resultat obteniu en avaluar-les? Per què obteniu aquest resultat?

14. Utilitzant la definició que vam veure a classe de `callcc`, digueu amb detall què fa aquesta funció:

```
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 f = arg_fc();
f(true);
print(f(false));
```

15. Utilitzar dos cops `callcc` en una expressió fa que el codi resultant sigui força complicat, a part de tenir una certa importancia teòrica (que no podem discutir en aquesta assignatura). Veiem-ne un exemple. Primer definim `twicecc`:

```
function twicecc(coll) {
    return callcc(function (f) {
        function tmp1(n) {
```

```
            return f([n,coll[1]])
        }
        function tmp2() {
            return callcc(function(q) {
                return f([coll[0],q])
            })
        }
        return tmp1(tmp2())
    })
}
```

i després el fem servir en una expressió:

```
arr = twicecc([0, function(x) { return x } ])
print(arr[1](arr[0]+1))
```

Si avalueu aquest codi en el el resultat és **2**. Expliqueu detalladament l'avaluació de l'expressió i com s'arriba a aquest resultat.

Continuation Passing Style (CPS):

Considereu primitives (és a dir, no cal transformar-les a CPS) les funcions i mètodes pre-definides dins Javascript.

16. 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).

17. Tenim la funció recursiva `my_reduce(f,arr,v)` , on `f` és una funció amb dos paràmetres `f ≡ f(x,total)` , de manera que
`my_reduce(f,[x1,x2,x3,...,xN],x0) = f(x1,f(x2,f(x3,...f(xN, x0)...)))`

La seva implementació és:

```
function my_reduce(f,arr,v) {
   if (arr.length === 0) {
      return v
   } else {
      let [car, ...cdr] = arr
```

```
      return f(car,my_reduce(f,cdr,v))
   }
}
```

Aleshores: `my_reduce((x,y) => x*y,[1,2,3,4,5,6,7,8,9],1)` retorna `362880` (que és 9!) o `my_reduce((x,y) => x+y,['h','o','l','a'],'')` retorna `'hola'`. Trobeu una funció equivalent recursiva final. Utilitzeu la transformació a *Continuation Passing Style* (CPS).

Trampolining

18. Imaginem que tenim una funció recursiva per sumar, de manera absurdament ineficient, dos nombres naturals:

```
function suma_ridicula(m,n) {
   if (n == 0) {
      return m;
   } else {
      return suma_ridicula(m+1,n-1)
    }
 }
```

Sabem que si fem servir Node.js, que no fa TCO (excepte la versió 6, però ara això ho ignorarem), tindrem problemes amb la pila. Si fem `suma_ridicula(2,16000)` obtindrem un error `RangeError: Maximum call stack size exceeded`. Apliqueu la técnica del *trampolining* per per obtenir una versió de suma_ridicula que no tingui problemes amb la mida de la pila.

19. Tenim una funció recursiva final per invertir un *array*. Si `arr` és l'array que volem invertir, s'ha de cridar `reverse(arr, [])`:

```
function reverse (arr, res) {
   if (arr.length === 0) {
      return res
   } else {
      let [car, ...cdr] = arr;
      res.unshift(car);
      return reverse(cdr,res);
    }
 }
```

Aleshores: `reverse ([1,2,3,4,5], [])` => `[5,4,3,2,1]`
Sabem que si fem servir Node.js, que no fa TCO, tindrem problemes amb la pila. Si fem `reverse([...Array(10000).keys()],[])` obtindrem un error `RangeError: Maximum call stack size exceeded`. Apliqueu la técnica del *trampolining* per obtenir una versió de `reverse` que no tingui problemes amb la mida de la pila.

20. 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;
      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 `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.

21. Tenim una funció recursiva final `my_filter_index(arr,f,res,i)`, que aplica un predicat `f` a tots els elements d'un *array*, i retorna un altre *array* amb l'índex d'aquells elements `e` tal que `f(e)` és `true` . Si `arr` és l'*array* al que volem aplicar `f` , s'ha de cridar `my_filter_index(arr,f,[],0)`:

```
function my_filter_index (arr, f, res, i) {
   if (arr.length === 0) {
      return res
   } else {
      let [car, ...cdr] = arr
      if (f(car)) {
         res.push(i)
      }
      return my_filter_index(cdr,f,res,i+1)
   }
}
```

Aleshores: `my_filter_index([1,2,3,4,5,6,7,8,9],x=>x%2===0,[],0)` retorna `[1,3,5,7]`.
Sabem que si fem servir Node.js, que no fa TCO, tindrem problemes amb la pila. Si fem , per exemple,
`my_filter_index([...Array(10000).keys()],x=>x%2===0,[],0)`
obtindrem un error `RangeError: Maximum call stack size exceeded`. Apliqueu la técnica del *trampolining* per obtenir una versió de `my_filter_index` que no tingui problemes amb la mida de la pila.

Combinacions de tècniques:

Considereu primitives (és a dir, no cal transformar-les a CPS) les funcions i mètodes pre-definides dins Javascript.

22. Tenim la funció `member(x,arr)` que retorna `true` si `x` pertany a l'*array* `arr`, i `false` en cas contrari. Suposem que no hi ha *arrays* dins d'`arr`:

```
function member(x,arr) {
    if (arr.length === 0) {
        return false
    } else {
        let [car, ...cdr] = arr
        return (x === car) || member(x,cdr)
    }
}
```

Primer convertiu aquesta funció en recursiva final fent servir CPS i després apliqueu la tècnica del trampolí per no tenir problemes amb la pila.

Exemple d'interacció a Node:

```
> member(834,[...Array(1000).keys()])
true
> member_cps(834,[...Array(1000).keys()],x => x)
true
> member_cps_trampolined(834,[...Array(1000).keys()],x => x)
true
> member(10001,[...Array(1000).keys()])
false
> member_cps(10001,[...Array(1000).keys()],x => x)
false
> member_cps_trampolined(10001,[...Array(1000).keys()],x => x)
false
> member(10000,[...Array(10000).keys()])
Uncaught RangeError: Maximum call stack size exceeded
    at member ([eval]:4:16)
    at member ([eval]:9:31)
    (...)
> member_cps(10000,[...Array(10000).keys()], x => x)
Uncaught RangeError: Maximum call stack size exceeded
    at member_cps ([eval]:13:20)
    at member_cps ([eval]:18:16)
    (...)
> member_cps_trampolined(10000,[...Array(10000).keys()], x => x)
false
```

23. Tenim la funció `suma_array_nombres(arr)` que retorna la suma dels nombres que apareixen a l'*array* `arr`. Suposem que dins d'`arr` pot haver altres *arrays* de nombres, que cal tenir en compte per sumar-los també:

```
function suma_array_nombres(arr) {
    if (arr.length === 0) {
        return 0
    } else if (typeof(arr) === "number") {
        return arr
    } else {
        let copia = arr.slice(0);
        let primer_element = copia.shift();
        let sumat = suma_array_nombres(primer_element);
        return sumat + suma_array_nombres(copia);
    }
}
```

Primer convertiu aquesta funció en recursiva final fent servir CPS i després apliqueu la tècnica del trampolí per no tenir problemes amb la pila.

Exemple d'interacció a Node:

```
> let array_de_nombres = [[1,2,3,4],5,[6,[7,[8,9]]]]
undefined
> suma_array_nombres(array_de_nombres)
45
> suma_array_nombres_cps(array_de_nombres,x => x)
45
> suma_array_nombres_cps_tr(array_de_nombres,x => x)
45
>
> suma_array_nombres([...Array(1000).keys()])
499500
> suma_array_nombres_cps([...Array(1000).keys()], x => x)
499500
> suma_array_nombres_cps_tr([...Array(1000).keys()], x => x)
499500
>
> suma_array_nombres([...Array(10000).keys()])
Uncaught RangeError: Maximum call stack size exceeded
    at Array.slice (<anonymous>)
    at suma_array_nombres ([eval]:7:25)
    (...)
> suma_array_nombres_cps([...Array(10000).keys()], x => x)
Uncaught RangeError: Maximum call stack size exceeded
    at Array.slice (<anonymous>)
    at suma_array_nombres_cps ([eval]:27:25)
    (...)
> suma_array_nombres_cps_tr([...Array(10000).keys()], x => x)
49995000
```

24. Tenim la funció `producte_array_nombres(arr)` que retorna el producte dels nombres que apareixen a l'*array* `arr`. Suposem que dins d'`arr` pot haver altres *arrays* de nombres, que cal tenir en compte per multiplicar-los també:

```
function producte_array_nombres(arr) {
    if (arr.length === 0) {
        return 1
    } else if (typeof(arr) === "number") {
        return arr
    } else {
        let copia = arr.slice(0);
        let primer_element = copia.shift();
        let prod = producte_array_nombres(primer_element);
        return prod * producte_array_nombres(copia);
    }
}
```

Primer convertiu aquesta funció en recursiva final fent servir CPS i després apliqueu la tècnica del trampolí per no tenir problemes amb la pila. A més a més, voldríem retornar immediatament un `0` en cas de trobar-ne algún dins `arr`. Per a això farem servir el `callcc` que vam definir a classe per funcions que fan servir CPS.

Exemple d'interacció a Node:

```
> let a1 = [[1,2,3,4],5,[6,[7,[8,9]]]]
undefined
> let a2 = [[[[[1]]]],[1,2,3,4],5,[6,[7,[8,9]]]]
undefined
> let a3 = [[1,2,3,4],5,[6,[0,[8,9]]]]
undefined
> producte_array_nombres(a1)
362880
> producte_array_nombres(a2)
362880
> producte_array_nombres(a3)
0
> producte_array_nombres_cps(a1, x => x)
362880
> producte_array_nombres_cps(a2, x => x)
362880
> producte_array_nombres_cps(a3, x => x)
0
> producte_array_nombres_cps_tr(a1, x => x)
362880
> producte_array_nombres_cps_tr(a2, x => x)
362880
> producte_array_nombres_cps_tr(a3, x => x)
0
> producte_array_nombres_amb_escape(a1, x => x)
362880
```

```
> producte_array_nombres_amb_escape(a2, x => x)
362880
> // ara afegim un 'console.log' per veure que sortim de seguida
> producte_array_nombres_amb_escape(a3, x => x)
4 1
3 4
2 12
1 24
0
>
```