Final CAP

Curs 2018-19 (11/I/2019) Durada: 3 hores

1.- (2 punts) Recordeu el fitxer **`Singleton1wrong.js`** que us vaig passar pel Racó i que haurieu de tenir tots entre el material que us heu portat per a aquest examen? Per què diem que està *wrong*? Quin és el problema exactament? Fes una descripció del funcionament del programa i en quin punt falla.

Solució:
El problema és que en fer

```
Universe = function Universe () {
     return instance;
};
```

mantenim el *prototipus* de l'objecte resultat de la primera crida a **`new Universe()`** i per tant qualsevol propietat que s'hagi definit *després* de crear la primera (i única) instància de **`Universe`** es perdrà (no serà visible).

2.- (1 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);
}
```

3.- (1 punt) Sabem que en Javascript les funcions són objectes (especials, en el sentit que són *invocables*, però objectes al cap i a la fi). Sabem que tot objecte en Javascript té un altre objecte com a *prototipus* (excepte **`Object.prototype`**). Finalment, sabem que tot objecte-funció en Javascript té una propietat anomenada **`prototype`**. Aleshores, a les funcions Javascript: Quina relació hi ha entre el *prototipus* de la funció (en tant que objecte) i la seva propietat **`prototype`**?

Solució:
Cap ni una, excepte que a l'objecte **`Function`**, el seu *prototipus* és **`Function.prototype`**. Però aquest és un cas excepcional. En una funció, la seva propietat **`prototype`** referencia un objecte que servirà de *prototipus* als objectes nous creats en cas que la funció s'invoqui amb **`new`**. En una funció, el seu *prototipus* té el mateix rol que en qualsevol altre objecte.

4.- (1.5 punts) 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 obtenir una versió de **`suma_ridicula`** que no tingui problemes amb la mida de la pila.

Solució:

Si tenim

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

podem fer

```
const solucio = (function () {
    function __f(m, n) {
        if (n > 0) {
            return function () {
                return __f(m+1,n-1);
            };
        }
        return m;
    };

    return function (m, n) {
        return trampoline(__f(m,n));
    }
})();
```

5.- (2 punts) Aquest programa és molt similar a un exemple que vam veure a classe i que teniu perquè us el vaig passar via Racó (**`05-ExempleContinuacions_senzill.js`**).

```
(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.

Solució:

El problema aquí és que no estem utilitzant correctament **`Continuation()`**.
La versió que vam veure a classe utilitza

```
function current_continuation() {
    print("Agafem la continuacio");
    return new Continuation();
}
```

i fa l'assignació **`let kont = current_continuation();`**. En aquest cas la continuació assignada a **`kont`** és aquella que captura precisament el moment del programa en que s'assigna un valor a **`kont`** (per definició de **`Continuation()`**). En el programa de la pregunta, el problema és que la continuació captura el retorn al *prompt* principal, a l'entorn d'execució, així doncs invocar **`kont`** implica acabar el programa.

6.- (2.5 punts) Hi ha una propietat de les variables declarades amb **`let`** que no hem explicat a classe. Amb aquesta pregunta la idea és que vosaltres mateixos la trobeu. Fixeu-vos en el següent programa en C++:

```
#include <iostream>
using namespace std;

int variable = 0;

void senzilla() {
  cout << variable << endl;

  int variable = 3;
  cout << variable << endl;
}

int main ()
{
  senzilla();
  cout << variable << endl;
}
```

El resultat d'executar-lo és **`0 3 0`** (separats amb salts de línia). Des del punt de vista de variables l'abast (*scope*) de les quals és el bloc, aquest seria el comportament *esperable*. En canvi, aquest programa Javascript ens dona un **`ReferenceError`** allà on està assenyalat:

```
let variable = 0;

function senzilla() {
    console.log(variable); // <------

    let variable = 3;
    console.log(variable);
}

senzilla();
console.log(variable)
```

Aleshores:

- Quina creieu que és la diferència en el tractament de les variables amb abast de bloc (declarades amb **`let`**) a Javascript i les variables a C++ (on totes les variables tenen abast de bloc)?

- Quina propietat de les variables de Javascript (les originals del llenguatge, declarades amb **`var`**) sembla que es mantingui en les declarades amb **`let`** (i que és discutible si s'hauria de mantenir o no)?

Pista: Per tenir una idea del què passa, mireu de sustituir **`let`** per **`var`** a la declaració de la variable local i mireu què passa.

Solució:
Aquest és el fenòmen degut al que s'anomena *Temporal Dead Zone* (TDZ) i és que sembla que hi ha *hoisting* en les variables declarades amb **`let`**, tot i que diferent del *hoisting* que trobem amb les variables declarades amb **`var`**. Una variable declarada localment amb **`let`** enmascara *en **tot** el seu abast (scope)* qualsevol altre variable visible en aquell abast amb el mateix nom.