Pràctica CAP Q1 curs 2017-18
Fils cooperatius

- A realitzar en grups de **2 persones**.
- A entregar com a molt tard el **30 de gener de 2018**.

Descripció resumida:
tl;dr ⇒ Aquesta pràctica va de continuacions.
La pràctica de CAP d'enguany serà una investigació del concepte general d'estructura de control, aprofitant que Javascript té tant *closures* com continuacions[1]. Estudiarem les conseqüències de poder guardar la continuació d'una funció, a la que tenim accés gràcies a la funció **`Continuation()`**, per fer-la servir i/o manipular-la. El fet de poder guardar i restaurar la continuació d'una funció ens permet implementar qualsevol estructura de control. Així doncs, per un costat implementarem aquesta estructura de control de Javascript en Pharo i **`#callcc:`** de Pharo en Javascript, per un altre costat, utilitzant les continuacions de Javascript, implementarem una estructura de control anomenada fils cooperatius (*cooperative threads*).

Material a entregar:
tl;dr ⇒ Amb l'entrega del codi que resol el problema que us poso a la pràctica NO n'hi ha prou. Cal entregar un informe i els tests que hagueu fet.
Haureu d'implementar el que us demano i entregar-me finalment un **informe** on m'explicareu, què heu aprés, i *com* ho heu arribat a aprendre (és a dir, m'interessa especialment el codi lligat a les proves que heu fet per saber si la vostra pràctica és correcte). Les vostres respostes seran la demostració de que heu entés el que espero que entengueu. El format de l'informe és lliure. El codi, com és habitual a les pràctiques de CAP, costa més de pensar que d'escriure. Us demano la definició de **`continuation`** en Pharo i la definició de **`callcc(f)`** en Javascript (vegeu enunciat, apartat a) i el fitxer .js amb la resposta a l'apartat b.

1 Bé, rigorosament parlant només la implementació de Mozilla, anomenada *Rhino*, de Javascript disposa de la funció **`Continuation`**, per tant aquesta implementació és la que farem servir. Un cop instal·lat *Rhino* (darrera versió 1.7.7.2), caldrà utilitzar-lo per executar el fitxer **`foo.js`** de la següent manera:

```
$ java -cp rhino1.7.7.2/lib/rhino-1.7.7.2.jar org.mozilla.javascript.tools.shell.Main -opt -2 foo.js
```

Si no passeu cap fitxer us apareix un repl de Javascript. Podeu trobar *Rhino* a:
`developer.mozilla.org/es/docs/Rhino` i a **`github.com/mozilla/rhino`**

Enunciat:
Aquest enunciat té dues parts diferents i independents:

a.- Hem vist a classe que Javascript té la funció **`Continuation()`** que ens permet crear i guardar continuacions. Treballant amb Pharo vam estar utilitzant una construcció molt més comuna, **`Continuation class >> callcc:`**, ja que la podem trobar a llenguatges com Standard ML, Scheme o Ruby. Una de les peculiaritats de **`Continuation()`** és que, com ja vam veure, **NO** funciona igual que **`#callcc:`**. És clar que un dels avantatges que tenim a Smalltalk és que disposem d'un enllaç directe a (una reificació de) la pila d'execució: **`thisContext`**, per tant podem implementar pràcticament qualsevol estructura de control. Així doncs, aquest primer apartat demana:

a.1.- Implementeu en Pharo un mètode unari **`Continuation class >> continuation`**, tal que funcioni exactament igual que el **`Continuation()`** de Javascript. El fariem servir directament així: **`Continuation continuation`**, i el retorn seria una instància de **`Continuation`**. Teniu l'opció de fer servir el mètode **`Continuation class >> callcc:`** que ja coneixeu, però una implementació directa (molt semblant a la implementació de **`#callcc:`**) és molt més senzilla.

a.2.- Implementeu en Javascript una funció **`callcc(f)`** que funcioni com l'estructura de control que ja coneixeu, fent servir, naturalment, la funció **`Continuation()`** de Javascript.

b.- Quan parlem de sistemes multi-fil (*multithread*) trobem dues possibilitats: O bé es gestiona de manera que cada fil cedeix el control a altres fils *voluntariament* (i així el fil s'executa fins que ell vol), o bé el sistema subjacent a l'execució multi-fil (la màquina virtual, per exemple) decideix quant de temps d'execució li pertoca a cada fil en funció, per exemple, d'un sistema de prioritats. En el primer cas parlarem de *cooperative* (o *non-preemptive*) *multithreading*, en el segon cas de *preemptive multithreading*. Tots dos models tenen els seus avantatges i inconvenients.

En aquesta pràctica farem servir continuacions per implementar un sistema que ens permeti executar els nostres programes de manera concurrent amb *cooperative multithreading*.

La idea és implementar una funció **`make_thread_system()`** que retorni un objecte amb *quatre* propietats, cadascuna d'elles és una funció de l'API amb la que puc utilitzar el multi-fil cooperatiu. Si fem **`var mts = make_thread_system()`**, aleshores **`mts`** és un objecte amb les funcions:

- **`mts.spawn(thunk):`** Posa un *thread* nou (la funció que anomenem *thunk*) a la cua de *threads*
- **`mts.quit():`** Atura el *thread* on s'executa i el treu de la cua de *threads*
- **`mts.relinquish():`** Cedeix (*yields*) el control del *thread* actual a un altre *thread*.
- **`mts.start_threads():`** Comença a executar els *threads* de la cua de *threads*

que em permeten gestionar una cua de *threads*. És molt possible que us faci falta una funció auxiliar **`halt()`** (que no té per què formar part de les funcions públiques de l'objecte creat per **`make_thread_system()`**, jugarà un paper auxiliar) per aturar el funcionament iniciat amb **`start_threads()`**.

Un example del que us demano és el següent programa on fem la funció **make_thread_thunk** que retorna un *thunk* (un *thunk* és com anomenem a una funció sense paràmetres) que després farem servir com a *thread* cooperatiu:

```
var counter = 10;

function make_thread_thunk(name, thread_system) {
    function loop() {
      if (counter < 0) {
         thread_system.quit();
      }
      print('in thread',name,'; counter =',counter);
      counter--;
      thread_system.relinquish();
      loop();
    };

    return loop;
}
var thread_sys =  make_thread_system();
thread_sys.spawn(make_thread_thunk('a', thread_sys));
thread_sys.spawn(make_thread_thunk('b', thread_sys));
thread_sys.spawn(make_thread_thunk('c', thread_sys));
thread_sys.start_threads();
```

Després d'executar aquest programa (amb nom, per exemple, **exempleCAP.js**) fent:

```
$ java -cp rhino1.7.7.2/lib/rhino-1.7.7.2.jar
       org.mozilla.javascript.tools.shell.Main -opt -2 exempleCAP.js
```

el resultat ha de ser:

```
in thread a ; counter = 10
in thread b ; counter = 9
in thread c ; counter = 8
in thread a ; counter = 7
in thread b ; counter = 6
in thread c ; counter = 5
in thread a ; counter = 4
in thread b ; counter = 3
in thread c ; counter = 2
in thread a ; counter = 1
in thread b ; counter = 0
```

Implementeu la funció **make_thread_system** i, com a mínim, feu que l'exemple de l'enunciat funcioni correctament (si a més vosaltres penseu altres exemples, es valorarà molt positivament).