Pràctica CAP Q1 curs 2018-19
PROG

- A realitzar en grups de **2 persones**.
- A entregar com a molt tard el **20 gener de 2019**.

Descripció resumida:
tl;dr ⇒ Aquesta pràctica va de la pila de execució i la possibilitat de reificar-la.
La pràctica de CAP d'enguany serà una investigació del concepte general d'estructura de control, aprofitant les capacitats d'introspecció i intercessió que ens dóna Smalltalk. Estudiarem les conseqüències de poder inspeccionar la pila d'execució (a la que tenim accés gràcies a la pseudo-variable thisContext) per fer-la servir i/o manipular-la.

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, i el codi que m'heu d'entregar me'l podeu entregar de diverses maneres: via un fitxer .st obtingut d'un File Out, via un paquet .mcz o via un paquet a SmalltalkHub. Qualsevol d'aquestes maneres és vàlida. Utilitzareu Pharo 3.0 (pharo.org) per fer la pràctica.

Nota: Abans de començar, llegiu i estudieu amb atenció el capítol 14, *Blocks: A Detailed Analysis*, del llibre *Deep into Pharo* (deepintopharo.com). També repasseu el que vam explicar a classe amb l'exemple del lligam dinàmic (*dynamic binding*, exemple de l'article que ja teniu del P. Deutsch a la revista Byte d'agost de 1981), ja que ho haureu de fer servir.

Enunciat:

És possible que, enyorant temps més civilitzats, trobem a faltar la possibilitat de fer programes senzills, que consisteixin només en una llista d'instruccions que s'executa seqüencialment, llevat que no ens trobem algun *goto* (salt incondicional) d'aquells que són quasi bé un tabú avui dia.

Amb Pharo-Smalltalk tenim una mica complicat fer això, però ho intentarem. **Us demano fer una classe `PROG`, amb un mètode en el *class-side* anomenat `#withInit:do:`.** És clar que us caldrà fer més mètodes auxiliars però això ja es cosa vostra. L'utilitzarem de la següent manera:

`PROG withInit: { { #s`$_1$` . exp`$_1$` } . #s`$_2$` . { #s`$_3$` . exp`$_3$` } ... }  do:`

```
{
    { #label1 . bloc1 } .
    { #label2 . bloc2 } .
        ...
    { #labeln . blocn }
}
```

on el que es pretén és que s'executin els blocs **`bloc`$_1$**, **`bloc`$_2$**, etc. seqüencialment. Fixem-nos que els blocs representaran "*instruccions"* i la llista d'"*instruccions*":

```
{ #label1 . bloc1 }
{ #label2 . bloc2 }
    ...
{ #labeln . blocn }
```

va "*etiquetada*" amb les etiquetes **`#label`$_1$**, **`#label`$_2$**, etc. Això ho fem per poder fer salts incondicionals. Els simbols **`#s`$_1$**, **`#s`$_2$**, etc. juguen el paper de "*variables locals*" i es poden fer servir dins dels blocs **`bloc`$_1$**, **`bloc`$_2$**, etc. sense problemes. Ara bé, *no són variables de debó de Smalltalk*, són símbols, així que per utilitzar-los com si fossin variables caldrà fer:

- Consulta: **`#s`$_k$ `binding`** ens donarà "*el valor de*" **`#s`$_k$**
- Assignació: **`#s`$_k$ `changeBinding: v`** "*assignarà*" el valor **`v`** a **`#s`$_k$**.

Els blocs **`bloc`$_1$**, **`bloc`$_2$**, etc. s'executaran *seqüencialment*, però podem fer salts incondicionals entre blocs utilitzant les etiquetes **`#label`$_k$**. L'expressió **`#label`$_k$ `binding value`** dins de *qualsevol* dels blocs **`bloc`$_j$** equival a "**`goto #label`$_k$**".
Tindrem un simbol *reservat* anomenat **`#RETURN`**. Sempre podeu utilitzar **`#RETURN binding value: <valor a retornar>`** per a acabar el programa retornant **`<valor a retornar>`**. Si no el feu servir, el programa acabarà amb un **`#RETURN binding value: nil`** *implícit*.

A més, caldrà tenir en compte que:

a) els **`#s`$_1$**, **`#s`$_2$**, **`#label`$_1$**, **`#label`$_2$**, etc. poden ser símbols qualsevol <u>*excepte*</u> **`#RETURN`**.

b) **`bloc`$_1$**, **`bloc`$_2$**, etc. són blocs *sense paràmetres*.

c) En aquest exemple utilitzo Arrays dinàmics **`{ a`$_1$ `. a`$_2$ `. ... . a`$_N$ `}`** per a tot (les

parelles variable-valor, la llista de variables, les parelles etiqueta-bloc i la llista d'instruccions), però crec que podeu utilitzar qualsevol col·lecció.

d) NO feu servir **`#label`$_k$ `changeBinding: v`** sota *cap* circumstància.

Per exemple:

```
PROG withInit: { #n } do:
{
{ #label1 . [ #n changeBinding: 0 ] } .
{ #label2 . [ #n changeBinding: (#n binding + 1) ] } .
{ #label3 . [ #n changeBinding: (#n binding + 1) ] } .
{ #label4 . [ #n changeBinding: (#n binding + 1) ] } .
{ #label5 . [ #n changeBinding: (#n binding + 1) ] } .
{ #label6 . [ #n changeBinding: (#n binding + 1) ] } .
{ #label7 . [ #RETURN binding value: #n binding  ] } .
}.
```

retorna 5.

Un exemple una miqueta més complicat:

```
PROG withInit: {{ #n . 10 } . { #coll . OrderedCollection new }}  do:
{
{ #label1 . [ #n binding == 0 ifTrue:
                    [ #RETURN binding value: #coll binding ]  ] } .
{ #label2 . [ #coll changeBinding:
                  ((#coll binding) add: #n binding; yourself) ] } .
{ #label3 . [ #n changeBinding: (#n binding - 1) ] } .
{ #label4 . [ #label1 binding value ] }
}.
```

retorna una **`OrderedCollection`** amb els nombres del 10 a l'1, en ordre decreixent.

ANNEX:

El mecanisme principal que farem servir per implementar el que es demana és el mecanisme de *lligam dinàmic* que ja vam veure a classe (recordeu l'article de Peter Deutsch a la revista *Byte*, plana 334, i la classe **`Binding`** dins el paquet **`DeutschByte`**. Tot això us ho vaig passar via Racó). Us caldrà, però, afegir el mètode **`#changeBinding:`**, per canviar el valor lligat al símbol, a la classe **`Symbol`** dins el protocol **`*DeutschByte`**, que és on ja podem trobar el mètode **`#binding`**. La implementació de **`#changeBinding:`** és molt semblant a la implementació de **`#binding`**.

En definitiva, el gruix de la feina en la implementació de **`PROG class >> #withInit:do:`** està en transformar l'expressió E_1:

```
PROG withInit: { { #s1 . exp1 } . #s2 . { #s3 . exp3 } ... }  do:
     {
          { #label1 . bloc1 } .
          { #label2 . bloc2 } .
               ...
          { #labeln . blocn }
     }
```

en l'expressió <u>equivalent</u> E_2:

```
#s1 bindTo: exp1 in: [
 #s2 bindTo: nil in: [
  #s3 bindTo: exp3 in: [
   ...
    #label1 bindTo: [ bloc1 value. #label2 binding value ] in: [
     #label2 bindTo: [ bloc2 value. #label3 binding value ] in: [
     ...
      #labeln bindTo: [blocn value. #RETURN binding value: nil] in: [
          [ #label1 binding value ]]...]]...]]]
```

Per tant, que quedi clar que un pas molt important és construir la transformació de E_1 a E_2. Cal, però, tenir en compte que el símbol reservat **`#RETURN`** representa una manera d'acabar la invocació a **`PROG withInit: ... do: ...`** . Al tanto, que haureu de donar valor a **`#RETURN`** *dins un context on sigui possible vincular el símbol* **`#RETURN`** *a una continuació* que ens permeti acabar immediatament, en ser invocada. Això ho farem, un cop més, amb lligam dinàmic:

`#RETURN bindTo: [ :val | ret value: val ] in: [` E_2 **`]`**

i **`ret`** és precisament la continuació capturada en el moment adeqüat de la invocació a **`PROG class >> #withInit:do:`**