Pràctica CAP Q1 curs 2016-17
Corutines

UPC - Facultat d'Informàtica de Barcelona
Conceptes Avançats de Programació (CAP)
QT 2016-2017

Daniel Limia Aspas
Álvaro Martínez Arroyo

Índex

1. Introducció

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 guardar la pila d'execució (a la que tenim accés gràcies a la pseudo-variable thisContext) per fer-la servir i/o manipular-la. El fet de poder guardar i restaurar la pila d'execució d'un programa ens permet implementar qualsevol estructura de control i implementar la versió més flexible i general de les construccions que manipulen el flux de control d'un programa: les continuacions. Utilitzant les continuacions implementarem una estructura de control anomenada Corutina (coroutine).

Enunciat:

La idea de l'estructura de control anomenada corutina és la següent: imaginem una funció (o procediment, o subrutina) tal i com ja les coneixem de C o C++. Les funcions s'invoquen, executen el seu cos, i quan acaben retornen. Si les tornem a invocar, torna a executar-se tot el cos de la funció, i quan acaba retorna. Les corutines funcionen diferent: Quan una corutina C1 invoca una altra corutina C2, s'atura i espera que se la torni a invocar. Si això passa, l'execució de C1 es reprén just en el moment en que va invocar C2; si ara C1 torna a invocar C2, l'execució de C2 es reprén en el moment que va decidir invocar a un altre corutina.

Hi ha diverses maneres de definir les corutines, en funció de diverses característiques i la literatura sobre el tema és nombrosa. Nosaltres ens limitarem a una definició senzilla, adaptada a Smalltalk i a la OOP, que ens permetrà jugar amb el concepte. Això és el que us demano a l'enunciat, tot seguit.

Cal fer una classe: **`Coroutine`**, tal que construeixi objectes que es comportin com a corutines (en aquest sentit, podriem pensar que són

com una mena de blocs). Per a això tenim un mètode **`Coroutine class >> #maker:`** que farem servir per instanciar aquesta classe. Caldrà passar un bloc com a paràmetre, que és on posarem el codi de la corutina. Aquest bloc serà un bloc de dos paràmetres:

```
Coroutine maker: [ :resume :value | . . . ]
```

En els punts suspensius és on posarem el codi de la corutina.

Aquesta corutina, via el paràmetre **`:resume`**, pot invocar altres corutines:

```
resume value: <nom corutina> value: <valor passat a la corutina invocada>
```

ja que **`resume`** ha de ser un bloc amb dos paràmetres: el primer és una referència a una altra corutina, el segon és el valor que se li passarà a aquesta corutina com a valor de retorn de la crida a corutina que va fer el darrer cop que es va executar.

La manera com s'invoca una corutina serà mitjançant un mètode de la classe **`Coroutine >> #value:`**. En un programa fet amb corutines només cal arrencar la primera corutina, a partir d'aquest moment les corutines es comencen a cridar entre elles. El bloc que passarem com a paràmetre **`:resume`** al bloc amb que es construeix la corutina (el paràmetre de **`#maker:`**) si el pensem bé, pot ser sempre el mateix i independent de qualsevol codi que posem dins les corutines. També cal que penseu que la corutina, quan s'invoca per primer cop, ha de posar en marxa el codi especificat en el bloc que es passa com a paràmetre a **`#maker:`**, però després, quan s'invoca retornant d'una crida anterior (amb **`resume value: c value: v`**), ha de fer una altra cosa (ha de fer **`c value: v`**, però no només això! aquesta és una de les coses que heu de pensar).

Veiem un exemple. Si executem:

```
| a b c |
a := Coroutine maker: [ :resume :value |
      Transcript show: 'This is A'; cr.
      Transcript show: 'Came from ', (resume value: b value: 'A');cr.
      Transcript show: 'Back in A'; cr.
      Transcript show: 'Came from ', (resume value: c value: 'A');cr.].
b := Coroutine maker: [ :resume :value |
      Transcript show: '   This is B'; cr.
      Transcript show: '  Came from ', (resume value: c value: 'B');cr.
      Transcript show: '  Back in B'; cr.
      Transcript show: '  Came from ', (resume value: a value: 'B');cr.].
c := Coroutine maker: [ :resume :value |
      Transcript show: '    This is C'; cr.
      Transcript show: '    Came from ', (resume value: a value:'C');cr.
      Transcript show: '    Back in C'; cr.
      Transcript show: '    Came from ', (resume value: b value: 'C');cr.].

a value: nil. "en aquest exemple, el valor que li passem a a és irrellevant"
```

El resultat al Transcript és:

```
This is A
    This is B
        This is C
Came from C
Back in A
        Came from A
        Back in C
    Came from C
    Back in B
Came from B
```

a) Implementeu la classe **`Coroutine`** i feu que l'exemple de l'enunciat funcioni correctament (si a més vosaltres penseu altres exemples, millor).

b) Implementeu una solució al problema dels matrimonis estables (stable marriage problem) utilitzant corutines:

Donat un conjunt de N homes, N dones i una llista de preferències per a cada home i cada dóna, trobeu un aparellament estable pel grup. Un aparellament és inestable si hi ha un home i una dóna que no estan

aparellats però que es prefereixen l'un a l'altre (i l'altre a l'un) més que a la seva actual parella.

2. Implementació

Abans de començar hem d'aclarir certs aspectes per tal de que la pràctica es pugui provar exitosament:

- El mètode **callcc inicialment està implementat a ContinuationTest**. Nosaltres ho hem mogut de lloc a la classe Continuation. Per això és **imprescindible executar el Pharo amb la imatge i changes que adjuntem al ZIP**, o be, implementar el mètode callcc a Continuation. Per facilitar la feina del professor, només cal carregar la imatge.
- Encara que també adjuntem l'arxiu .st, **carregant la imatge i els changes que hi ha al ZIP al Pharo**, el professor podrà provar la pràctica més ràpidament ja que hi ha **dos workspaces preparats**, un per cada exercici, amb els jocs de prova adients.

La classe CoroutineCap té, tal i com s'especifica a l'enunciat, un mètode maker: aBlock que guarda el cos de la corutina. Per fer això hem creat el mètode al class side ja que es una constructora.

```
maker: aBlock
    "Creates a coroutine"
    | aCoroutine |
    aCoroutine := self new.
    aCoroutine aBlock: aBlock.
    aCoroutine aContext: nil.
    aCoroutine aResumeBlock: [ :param1 :param2|
                                    Continuation callcc:
                                    [:k |
                                        aCoroutine aContext: k.
                                        param1 value: param2.
                                    ].
                                ].
    ^ aCoroutine.
```

CoroutineCAP class>>#maker:

Com es pot veure a la imatge anterior, maker accepta un block. Els getters i setters s'han creat al instance side per tal de guardar aquest block com a una variable d'instància "aBlock".

Com que maker es una funció constructora, hem de crear una nova instància de la nostra classe amb "self new", guardar el block amb "aBlock:".

Ara arriba la part interessant de la pràctica: el block Resume.

Aquest block es el que resumirà , o iniciarà una corutina i la deixarà en pausa fins que torna a ser cridada.

Aquest resume block l'utilitzarem més endevant , per tant ho guardem a una variable d'instància.

Per executar una corutina desde 0, o internament al Resume Block utilitzem el mètode "value:":

```
value: aString
    "comment stating purpose of message"
    aContext notNil
        ifTrue: [ aContext value: aString ]
        ifFalse: [aBlock value: aResumeBlock value: aString].
```

CoroutinaCAP >>#value:

Com es pot veure el mètode value passa per paràmetre el resume block i un valor per tal que el resume block sigui executat dins del bloc que defineix la corutina (el que se li passa al maker).

```
a := CoroutineCAP maker: [ :resume :value |
Transcript show: 'This is A'; cr.
Transcript show: 'Came from ', (resume value: b value: 'A'); cr. Transcript show: 'Back in A'; cr.
Transcript show: 'Came from ', (resume value: c value: 'A'); cr. ].
```

Codi del exemple de l'enunciat

Al codi que es pot veure al workspace s'executa el mètode "value: nil" a la corutina A.

En fer això, com es pot veure en la foto anterior, aquest valor, que és nul es passat com a segon paràmetre al bloc principal (sempre que diem bloc principal ens referim al block definit al maker com a paràmetre) , a més del block resume, que serà, dins del bloc principal, executat amb "aString" com a paràmetre.

```
maker: aBlock
    "Creates a coroutine"
    | aCoroutine |
    aCoroutine := self new.
    aCoroutine aBlock: aBlock.
    aCoroutine aContext: nil.
    aCoroutine aResumeBlock: [ :param1 :param2|
                                    Continuation callcc:
                                    [:k |
                                        aCoroutine aContext: k.
                                        param1 value: param2.
                                    ].
                                ].
    ^ aCoroutine.
```

CoroutineCAP class>>#maker:

On està la gràcia del problema llavors? Que quan des de una corutina A volem resumir una corutina B, la corutina A es guarda a una variable d'instància el contexte actual, abans de cridar a resume de la corutina B.

Com es pot veure a la següent imatge, quan la corutina B es cridada amb el mètode "value:" mira si tenia guardat un contexte d'execució anterior. Si no ho té, simplement executa el bloc principal, si ho té, executa la continuació passant-li un valor.

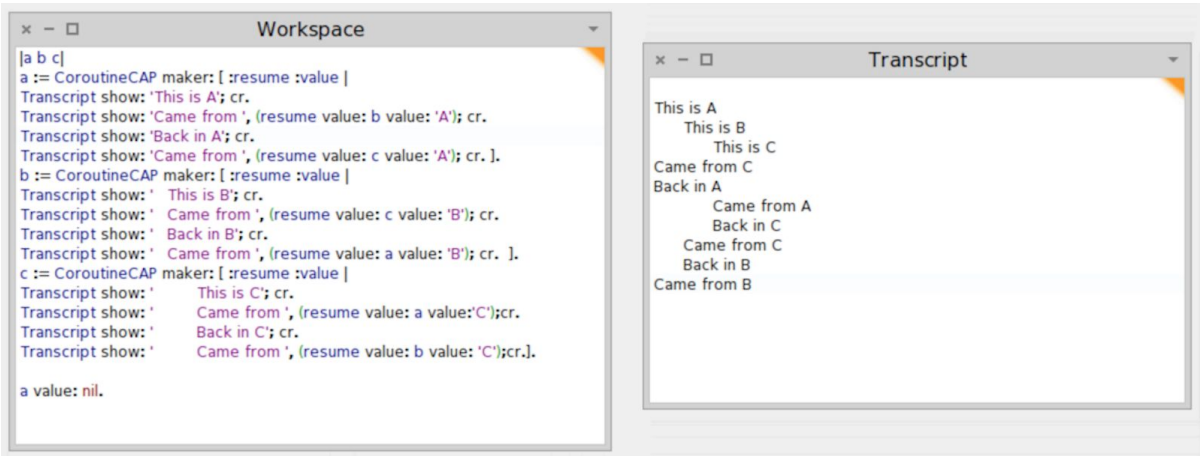
```
value: aString
    "comment stating purpose of message"
    aContext notNil
        ifTrue: [ aContext value: aString ]
        ifFalse: [aBlock value: aResumeBlock value: aString].
```

CoroutinaCAP >>#value:

En cridar a “value:” d’una continuació, es crida el contexte d’execució guardat en aquest, retornant el valor que se li passa per string (en el context d’execució que hi havia). Això és clau pel funcionament de la corutina, si no, el valor seria diferent.

Jocs de prova a)

El primer exemple que hem comprovat que funciona ha sigut el de l’enunciat, on si seleccionem tot el codi del *Workspace* i li donem a “*Do it*” podem veure que la sortida al *Transcript* és la correcta, com es pot comprovar en la següent imatge:



Exemple de l’enunciat

Altres exemples que hem pensat nosaltres per confirmar que la classe funciona correctament:

Si executem:

```
|a b c d|
a := CoroutineCAP maker: [ :resume :value |
Transcript show: 'This is A'; cr.
Transcript show: 'Came from ', (resume value: d value: 'A'); cr.
Transcript show: 'Back in A'; cr.
Transcript show: 'Came from ', (resume value: c value: 'A'); cr. ].
b := CoroutineCAP maker: [ :resume :value |
Transcript show: '  This is B'; cr.
Transcript show: '  Came from ', (resume value: c value: 'B'); cr.
Transcript show: '  Back in B'; cr.
Transcript show: '  Came from ', (resume value: a value: 'B'); cr. ].
```

```
c := CoroutineCAP maker: [ :resume :value |
Transcript show: '    This is C'; cr.
Transcript show: '    Came from ', (resume value: d value:'C');cr.
Transcript show: '    Back in C'; cr.
Transcript show: '    Came from ', (resume value: b value: 'C');cr.].
d := CoroutineCAP maker: [ :resume :value |
Transcript show: '      This is D'; cr.
Transcript show: '      Came from ', (resume value: b value:'D');cr.
Transcript show: '      Back in D'; cr.
Transcript show: '      Came from ', (resume value: a value: 'D');cr.].

a value: nil.
```

El resultat al Transcript és:

```
This is A
                  This is D
      This is B
            This is C
                  Came from C
                  Back in D
Came from D
Back in A
            Came from A
            Back in C
      Came from C
      Back in B
Came from B
```

Si executem:

```
|a b c d|
a := CoroutineCAP maker: [ :resume :value |
Transcript show: 'This is A'; cr.
Transcript show: 'Came from ', (resume value: b value: 'A'); cr.
Transcript show: 'Back in A'; cr.
Transcript show: 'Came from ', (resume value: d value: 'A'); cr. ].
b := CoroutineCAP maker: [ :resume :value |
Transcript show: '   This is B'; cr.
Transcript show: '   Came from ', (resume value: c value: 'B'); cr.
Transcript show: '   Back in B'; cr.
Transcript show: '   Came from ', (resume value: a value: 'B'); cr.  ].
c := CoroutineCAP maker: [ :resume :value |
Transcript show: '    This is C'; cr.
Transcript show: '    Came from ', (resume value: a value:'C');cr.
Transcript show: '    Back in C'; cr.
Transcript show: '    Came from ', (resume value: b value: 'C');cr.].
d := CoroutineCAP maker: [ :resume :value |
Transcript show: '      This is D'; cr.
```

```
Transcript show: '      Came from ', (resume value: b value:'D');cr.
Transcript show: '      Back in D'; cr.
Transcript show: '      Came from ', (resume value: a value: 'D');cr.].

a value: nil.
```

El resultat al Transcript és:

```
This is A
      This is B
            This is C
Came from C
Back in A
                  This is D
      Came from D
      Back in B
Came from B
```

En aquests exemples voliem jugar amb varies corutines que no acabin i comprovar que, efectivament, el context es guarda correctament.

b) A l'hora d'implementar una solució al problema dels matrimonis estables, hem realitzat un codi a partir dels dos exemples que hi han a l'article Lloyd Allison de 1983.

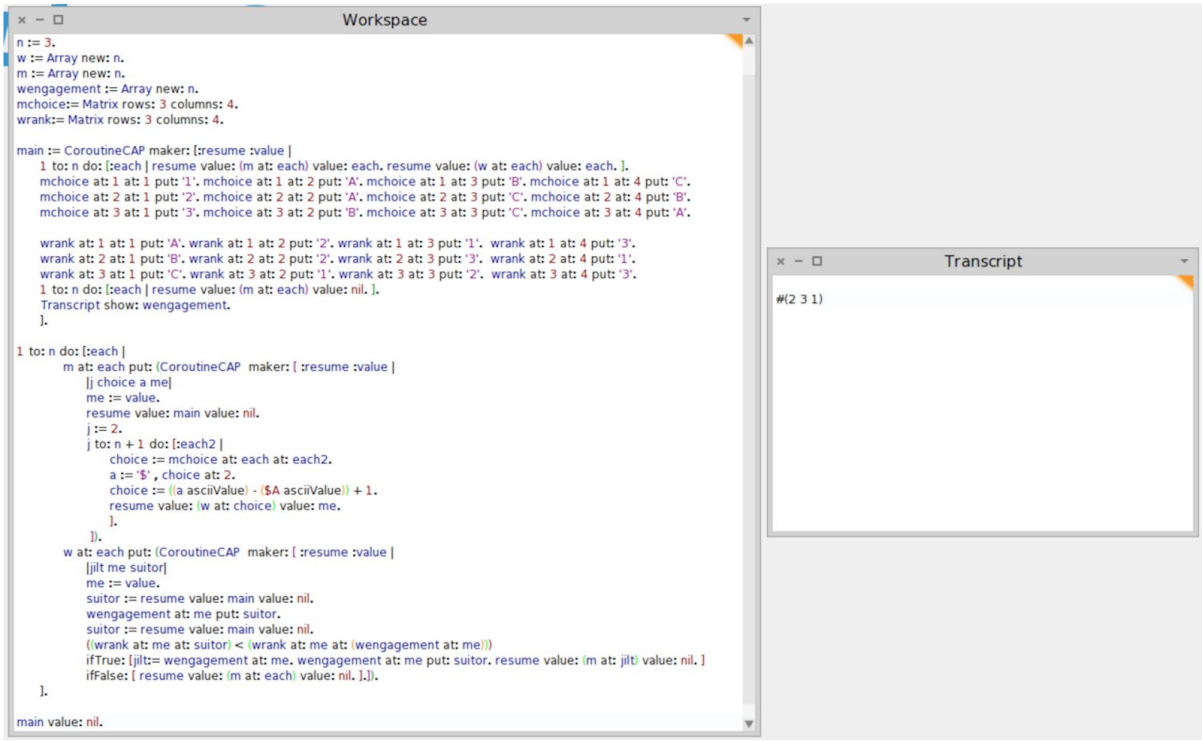
Per establir la estructura general del codi ha sigut més important el codi que està en SL5-like ja que té una notació més semblant a altres llenguatges de programació que ja coneixem i és més fàcil detectar les funcions, els bucles etc. Però a l'hora de jugar amb les corutines i amb el pas de paràmetres ha sigut clau la versió que està en Modula-2 ja que el codi està explicat més detalladament i les crides a les corutines queden més clares.

Jocs de prova b)

Tot el codi ho hem realitzat en el Workspace i si seleccionem tot el codi i li donem a "Do it", amb l'exemple de l'article, podem veure que la sortida al Transcript és la correcta, ja que en l'article podem veure que la sortida ha de ser:

The stable marriage is then (1, C) (2, A) (3, B).

com podem veure a la següent imatge, l'ordre de sortida és A,B,C , per tant, podem veure com surt el vector 2,3,1:



Exemple de l'article de l'enunciat

Com el codi no és veu amb claredat, ho adjuntem a continuació i també ho adjuntarem a l'entrega en un fitxer a part.

```
|mchoice wrank n w m wengagement main|

n := 3.
w := Array new: n.
m := Array new: n.
wengagement := Array new: n.
mchoice:= Matrix rows: 3 columns: 4.
wrank:= Matrix rows: 3 columns: 4.

main := CoroutineCAP maker: [:resume :value |
      1 to: n do: [:each | resume value: (m at: each) value: each. resume
value: (w at: each) value: each. ].
      mchoice at: 1 at: 1 put: '1'. mchoice at: 1 at: 2 put: 'A'. mchoice
at: 1 at: 3 put: 'B'. mchoice at: 1 at: 4 put: 'C'.
```

```
      mchoice at: 2 at: 1 put: '2'. mchoice at: 2 at: 2 put: 'A'. mchoice
at: 2 at: 3 put: 'C'. mchoice at: 2 at: 4 put: 'B'.
      mchoice at: 3 at: 1 put: '3'. mchoice at: 3 at: 2 put: 'B'. mchoice
at: 3 at: 3 put: 'C'. mchoice at: 3 at: 4 put: 'A'.

      wrank at: 1 at: 1 put: 'A'. wrank at: 1 at: 2 put: '2'. wrank at: 1
at: 3 put: '1'.  wrank at: 1 at: 4 put: '3'.
      wrank at: 2 at: 1 put: 'B'. wrank at: 2 at: 2 put: '2'. wrank at: 2
at: 3 put: '3'.  wrank at: 2 at: 4 put: '1'.
      wrank at: 3 at: 1 put: 'C'. wrank at: 3 at: 2 put: '1'. wrank at: 3
at: 3 put: '2'.  wrank at: 3 at: 4 put: '3'.
      1 to: n do: [:each | resume value: (m at: each) value: nil. ].
      Transcript show: wengagement.
      ].

1 to: n do: [:each |
            m at: each put: (CoroutineCAP  maker: [ :resume :value |
                  |j choice a me|
                  me := value.
                  resume value: main value: nil.
                  j := 2.
                  j to: n + 1 do: [:each2 |
                        choice := mchoice at: each at: each2.
                        a := '$' , choice at: 2.
                        choice := ((a asciiValue) - ($A asciiValue)) + 1.
                        resume value: (w at: choice) value: me.
                        ].
                   ]).
            w at: each put: (CoroutineCAP  maker: [ :resume :value |
                  |jilt me suitor|
                  me := value.
                  suitor := resume value: main value: nil.
                  wengagement at: me put: suitor.
                  suitor := resume value: main value: nil.
                  ((wrank at: me at: suitor) < (wrank at: me at:
(wengagement at: me)))
                  ifTrue: [jilt:= wengagement at: me. wengagement at: me
put: suitor. resume value: (m at: jilt) value: nil. ]
                  ifFalse: [ resume value: (m at: each) value: nil. ].]).
      ].

main value: nil.
```

3. Entrega

En el ZIP de l'entrega podem trobar:

- **CAPQ116-17.pdf** (aquest mateix pdf)
- **Pharo3.0.image** (La imatge, que com hem explicat anteriorment, es necessària per executar la pràctica amb èxit)
- **Pharo3.0.changes**
- **CAP.st** (Codi de la classe CoroutineCAP obtingut a partir d'un File out)
- **CodiApartatB).txt** (Codi del apartat b) per si no queda clar en aquest document)
- **CodiApartatA).txt** (Codi de l'enunciat del apartat a))
- **AltresExemplesA).txt** (Altres exemples que hem pensat nosaltres per l'apartat a))

4. Bibliografia

- Capítol 14, Blocks: A Detailed Analysis, del llibre Deep into Pharo (deepintopharo.com)

- Apunts continuacions (CAP QT 16-17)

- L'article Stable Marriages by Coroutines, de Lloyd Allison.

- Callcc, continuations, coroutines. https://ds26gte.github.io/tyscheme/index-Z-H-15.html

- call-with-current-continuation. ftp://ftp.cs.utexas.edu/pub/garbage/cs345/schintro-v14/schintro_141.html