Parcial CAP

Curs 2019-20 (18/XI/2019) **Duració: 2 hores**

1.- (1 punt) Defineix *introspecció*, *intercessió* i digues què significa *reificar.* Quina relació ha d'existir entre allò reificat de l'entorn d'execució i l'entorn d'execució real per a que pugui existir la intercessió?

Solució: Les definicions són a les transparències, i la relació ha de ser una relació de ***connexió causal***, ja que sense aquesta els canvis que hom pot fer en l'estructura reificada no tindrien cap efecte sobre el *runtime* real, i per tant no es podria fer cap modificació sobre el programa ni sobre la seva execució.

2.- (1.5 punts) Si seleccionem aquest programa al **`Playground`** i fem **`Ctrl-d`**...

```
| comptador reset incrementa decrementa |
comptador := [ | valor | { [ valor := 0. valor traceCr. ] .
                          [ :n | valor := valor + n. valor traceCr. ] .
                          [ :n | valor := valor - n. valor traceCr. ] } ] value.
reset := comptador at: 1.
incrementa := comptador at: 2.
decrementa := comptador at: 3.
reset value.
incrementa value: 2.
incrementa value: 2.
decrementa value: 1.
```

apareixen al **`Transcript`** els números **`0 2 4 3`** (en aquest ordre).
En canvi, si seleccionem aquest programa al **`Playground`** i fem **`Ctrl-d`**...

```
| comptador reset incrementa decrementa |
comptador := [ { [ | valor | valor := 0. valor traceCr. ] .
                 [ :n | | valor | valor := valor + n. valor traceCr. ] .
                 [ :n | | valor | valor := valor - n. valor traceCr. ] } ] value.
reset := comptador at: 1.
incrementa := comptador at: 2.
decrementa := comptador at: 3.
reset value.
incrementa value: 2.
incrementa value: 2.
decrementa value: 1.
```

el programa no funciona. Apareix un error (**`#+ was sent to nil`**).

Explica i justifica la diferència en el resultat.

Solució: En el primer cas **`valor`** és una variable capturada pels tres blocs retornats i assignats a comptador (recordem que un bloc en Smalltalk és un *closure*). En el segon cas, **`valor`** és una variable local a cada un dels blocs retornats, i per tant independent en cada bloc. A més, no està inicialitzada en dos dels blocs, d'aquí l'error.

3.- (1.5 punts) Respon **cert** o **fals** a les següents afirmacions (cada una val 0.25 punts; respostes errònies no resten punts; no cal justificar la resposta):
a) El símbol **^** serveix per retornar valors d'un mètode -- **cert**
b) En Smalltalk el producte (*****) té més precedència que la suma (**+**) -- **fals**

c) Els mètodes d'una classe `C` que pertanyen al protocol **`private`** són privats i no es poden enviar missatges amb el corresponent selector a les instàncies de `C` -- **fals**
d) El símbol `^` serveix per retornar valors d'un bloc -- **fals**
e) En Smalltalk tot és un objecte, fins i tot les classes -- **cert**
f) `Class class class` és idèntic a `Class class class class` -- **fals**

4.- (1.5 punts) Explica i justifica la relació entre aquesta expressió:

```
Continuation callcc: [ :k | 2 + 4 * (k value: 5) ]
```

i aquesta:

```
2 + 4 * (Continuation callcc: [ :k | 5 ])
```

Suposem que totes dues s'avaluen al **`Playground`**. Explica i justifica també el resultat d'avaluar les dues expressions.

Solució: El resultat de la primera és 5, ja que l'avaluació de la continuació fa que desaparegui qualsevol context pendent d'avaluació, en canvi el resultat de la segona és 30 (recordem la precedència dels missatges binaris).

5.- (1.5 punts) Escriviu un programa en Java que, passant el nom d'una classe com a paràmetre de la línia de comandes, digui si aquesta classe és **`final`** (és a dir, no en podem fer subclasses).

Solució: Una possible solució seria:

```
import java.lang.reflect.*;

public class EsFinal {
    public static void main(String args[]) {
            if (resposta(Class.forName(args[0]))) {
                  System.out.println("SI, és final");
            } else {
                  System.out.println("NO, no és final");
            }
    }
    private static boolean resposta (Class c) {
      if (c == null) {
            return false;
      }
      return Modifier.isFinal(c.getModifiers());
    }
}
```

6.- (3 punts) 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 ni en aquest examen ni a l'assignatura). Veiem-ne un exemple. Primer definim **`twicecc`**:

```
| twicecc |
twicecc := [ :coll |
              Continuation callcc: [ :f |
                  [ :n | f value: { n . (coll at: 2) } ] value:
                        (Continuation callcc: [ :q |
                              f value: { (coll at: 1) . q } ] ) ] ].

[ :k | [ :arr | (arr at: 2) value: ((arr at: 1) + 1) ]
                          value: (twicecc value: { 0 . k } ) ] value: [ :x | x ]
```

i després el fem servir en una expressió. Si avalueu aquest codi en el el resultat és **2**.

Expliqueu detalladament l'avaluació de l'expressió i com s'arriba a aquest resultat.
(*Consell: Primer feu la resta de l'examen i poseu-vos a pensar aquest problema només si teniu temps*)

Solució:

El càlcul comença assignant a **`twicecc`** el bolc indicat (però aquest no s'avalua). Després d'aquesta assignació, avaluem l'expressió

```
[ :k | [ :arr | (arr at: 2) value: ((arr at: 1) + 1) ]
                        value: (twicecc value: { 0 . k } ) ] value: [ :x | x ]
```

on k pren per valor el bloc identitat `[ :x | x ]`. Així, la primera vegada que avaluem **`twicecc`** li passem com a paràmetre `{ 0 . [ :x | x ] }` (on aquesta avaluació de **`twicecc`** és resultat de l'avaluació normal de blocs, pel bloc `[ :arr | ... ] value: ...`).

Així doncs avaluem **`twicecc`** amb paràmetre `{ 0 . [ :x | x ] }`. La primera crida a **`Continuation callcc: [ :f | ... ]`** el que fa és posar en `f` el fet de retornar de **`twicecc`**:

```
[ :arr | (arr at: 2) value: ((arr at: 1) + 1) ] value: □
```

Aquest serà el valor de **`f`**, sempre tenint en compte que l'avaluació de la continuació elimina qualsevol context anterior.

El cos del bloc del primer **`callcc:`** fa que haguem d'avaluar

```
(Continuation callcc: [ :q |  f value: { (coll at: 1) . q } ] )
```

amb la qual cosa invoquem la primera continuació amb valor **`{ 0 . q }`**. El quid de la qüestió és veure que **`q`** és:

```
[ :n | f value: { n . (coll at: 2) } ] value: □
```

(on no hem escrit la resta de codi per avaluar, només el més "proper").

Per tant, ara hem d'avaluar

```
[ :arr | (arr at: 2) value: ((arr at: 1) + 1) ] value: { 0 . q }
≡
q value: 1
```

i per tant tornem a:

```
[ :n | f value: { n . (coll at: 2) } ] value: 1
```

i provoquem una segona avaluació de la primera continuació:

```
[ :arr | (arr at: 2) value: ((arr at: 1) + 1) ] value: { 1 . [ :x | x ] }
```

la qual cosa finalitza el procés amb `[ :x | x ] value: 2`

que finalment retorna **`2`**.