

Informe Pràctica CAP - Quadrimestre Tardor 20/21

Integrants del grup: Eric González i Àlex Brugal

1. Introducció

Aquesta pràctica gira entorn el concepte d'estructures de control, gràcies a les capacitats d'introspecció i intercessió que ens proporciona Smalltalk, i que durem a terme mitjançant l'ús de les continuacions. Això ens permetrà guardar la pila d'execució amb la finalitat de poder accedir-hi o restaurar-la, i així podem manipular el flux de control del propi programa.

La pròpia pràctica està dividida en dues parts, encara que la segona requereix de l'ús de la classe implementada a la primera part per aconseguir l'objectiu desitjat. Així doncs, la primera secció tracta d'implementar una classe anomenada **BT** (*BackTracking*) mitjançant continuacions per poder utilitzar-la, en forma d'instància, a la segona part. Aquesta darrera part consisteix en la resolució del problema *NQueens* (classe **NQueen**), que busca sortejar aquest famós problema d'escacs i on nosaltres aplicarem l'algorisme tradicional que fa ús del *BackTracking*.

2. Classe BT

El primer que cal fer després d'haver importat l'arxiu amb extensió "st", que és la base de la classe **BT** i que hereda de la classe **Object**, és crear un mètode d'instància, anomenat **#try:**, ja que volem que sigui cridat per objectes de tipus **bt**, i no per la pròpia classe. Aquest mètode serà la base de la nostra pràctica, juntament amb **#next:**, ja que implementarà la funcionalitat principal.

```
Object subclass: #BT
  instanceVariableNames: 'continuationQueue'
  classVariableNames: ''
  package: 'Practica-CAP-2020'
```

2.1. Implementació del mètode **#try:**

L'objectiu del mètode o missatge **#try:** és rebre una col·lecció com a paràmetre, i anar assignant els valors, un a un, a la variable a la qual s'envia el missatge. Per tant l'estructura principal que busquem és la de recórrer una col·lecció, l'esquema de la qual es veu representada a grans trets en el codi següent, i que ens serveix per a tenir una idea molt senzilla però útil del que hauré d'aplicar a **#try:**.

```
recorrerColl: collection
| cont |
self value.
cont := Continuation callcc: [ :cc | cc ].
collection value
```

```
ifTrue: [ ^ nil ]  
iffalse: [ self value. cont value: cont ].
```

A partir de la idea anterior vam desenvolupar aquesta versió, la qual explicarem a continuació.

```
try: aCollection  
| aCollectionCopy cc |  
aCollectionCopy := aCollection asOrderedCollection.  
cc := Continuation callcc: [ :k | k ].  
aCollectionCopy isEmpty  
iffalse: [| aCollectionFirst |  
    aCollectionFirst := aCollectionCopy first. continuationQueue add: cc. aCollectionCopy  
removeFirst. ^ aCollectionFirst ]  
iftrue: [ |element| element := continuationQueue last. continuationQueue removeLast. element  
value: element].
```

En primer lloc declarem una variable local anomenada **aCollectionCopy**, a on copiarem la col·lecció que es passa com a paràmetre amb la finalitat d'ordenar-la. Seguidament es declara i s'assigna a la variable **cc** una instància de la classe **Continuation** a la qual s'envia el missatge **#callcc:** amb un bloc com a paràmetre o sender. La funció que té **#callcc:** és la de capturar el context d'execució actual i enviar-ho a la instància de **Continuation**, per després passar-ho al bloc que tenia com a paràmetre (en el nostre cas **[:k | k]**) amb la finalitat d'avaluar-ho.

La condició que hem de mirar per veure si cal seguir recorrent la col·lecció és evident que ha de ser sinònim de si aquesta està buida. D'aquí el **aCollectionCopy isEmpty**, on posteriorment hem d'avaluar la condició. Si la condició és falsa, cal extreure el primer element de la col·lecció, afegim a **continuationQueue** la variable **cc**, on recordem que guardem el context d'execució, borrem aquest primer element de la col·lecció, i finalment el retornem a través de la variable **aCollectionFirst** on havíem guardat el valor.

Una vegada s'hagin borrarat tots els elements de la col·lecció **aCollectionCopy**, aleshores saltam al **ifTrue**, on recorrem la col·lecció que conté els contextos d'execució. Per fer això, declarem una nova variable local en el context del bloc i que anomenarem **element**, amb la qual treballarem de forma similar, pel que fa referència a la manipulació dels elements de la col·lecció, a com hem fet al **iffalse**. Aquesta variable agafarà l'últim valor de la cua de continuacions **continuationQueue**, seguidament borrem aquest element dins la cua, i per últim cridem al mètode **Continuation>>#value:**, on el sender i el receiver són el mateix element. Aquest mètode, explicat de forma breu, fa el següent: s'abandona tot el que s'estava fent i es posa en el seu lloc la continuació. És a dir, com si el missatge de **#callcc:** hagués retornat **element**.

2.2. Testeig classe **BT**

Per fer el testing d'aquesta classe sobre els mètodes **#try:**, **#assert:**, i **#next** hem agafat les proves del mateix enunciat per veure si les respostes coincidien.

2.2.1. Test Inspect

The screenshot displays the Scala Playground interface, which is divided into three main sections: a code editor on the left, a variable inspector in the middle, and a transcript on the right.

Code Editor (Left): Contains the following Scala code:

```
| bt x |  
bt := BT new.  
bt inspect.  
x := bt try: { 1 . 3  
  . 5 . 7 . 9 . 11 }.  
x asString traceCr.
```

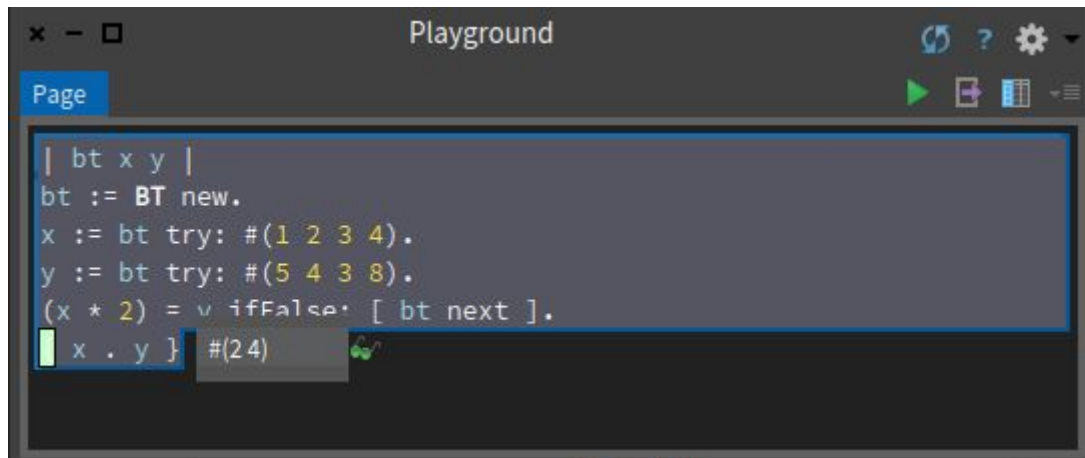
Variable Inspector (Middle): Titled "Inspector on a BT", it shows the state of the variable `a BT`. It has tabs for "Raw" and "Meta". The "Raw" tab is active, displaying a table of variables and their values:

Variable	Value
<code>self</code>	<code>a BT</code>
<code>{ } continuationQueue</code>	<code>an OrderedCollection [2 items] ([nil] a Continuation)</code>

Below the table, there is a text area showing the string representation of the object: `"a BT"`, followed by the command `self next`.

Transcript (Right): Displays the output of the code execution, showing a list of strings: `'1'`, `'3'`, `'5'`, `'7'`, `'9'`, and `'11'`.

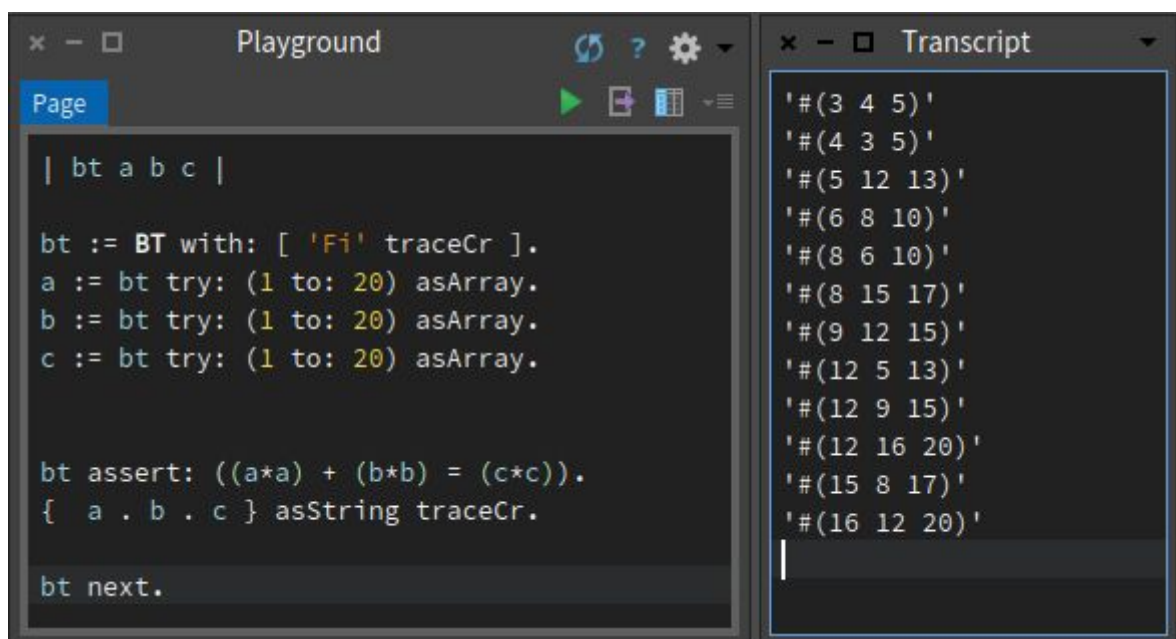
2.2.2. Test ($y = 2 \cdot x$)



The screenshot shows a 'Playground' window with a code editor. The code defines a block 'bt' with variables 'x' and 'y'. 'x' is assigned the value 1, and 'y' is assigned the value 2. The code then checks if $x \cdot 2 = y$. If false, it calls 'bt next'. The output shows the expression `x . y` evaluated to `#(24)`.

```
| bt x y |  
bt := BT new.  
x := bt try: #(1 2 3 4).  
y := bt try: #(5 4 3 8).  
(x * 2) = y ifFalse: [ bt next ].  
x . y } #(24)
```

2.2.3. Test Pitágoras



The screenshot shows a 'Playground' window and its 'Transcript' output. The code in the playground defines a block 'bt' with variables 'a', 'b', and 'c'. 'a', 'b', and 'c' are assigned values from 1 to 20. The code then checks if $(a^2 + b^2) = c^2$. If true, it prints the values of 'a', 'b', and 'c' as a string. The transcript shows the output of this test for various combinations of 'a', 'b', and 'c'.

```
| bt a b c |  
  
bt := BT with: [ 'Fi' traceCr ].  
a := bt try: (1 to: 20) asArray.  
b := bt try: (1 to: 20) asArray.  
c := bt try: (1 to: 20) asArray.  
  
bt assert: ((a*a) + (b*b) = (c*c)).  
{ a . b . c } asString traceCr.  
  
bt next.
```

Transcript output:

```
'#(3 4 5)'  
'#(4 3 5)'  
'#(5 12 13)'  
'#(6 8 10)'  
'#(8 6 10)'  
'#(8 15 17)'  
'#(9 12 15)'  
'#(12 5 13)'  
'#(12 9 15)'  
'#(12 16 20)'  
'#(15 8 17)'  
'#(16 12 20)'
```

3. Classe NQueens

El primer que cal fer evidentment és crear la classe, que hereda d'Object i conté una variable d'instància que hem anomenat **parametre**:

```
Object subclass: #NQueens
  instanceVariableNames: 'parametre'
  classVariableNames: ''
  package: 'Practica-CAP-2020'
```

A partir d'aquí hem implementat un mètode de classe i un d'instància per inicialitzar la variable d'instància **parametre**.

Mètode d'instància

```
with: finalBlock
  parametre := finalBlock
```

Mètode de classe

```
with: finalBlock
  ^ (self new) with: finalBlock
```

3.1 Implementació del mètode #solve

Per implementar aquest mètode ens vam inspirar en l'algorisme 6.1 dels apunts d'EDA que es troben adjunts a la pràctica i en el testing del problema de Pitàgores.

Primerament, inicialitzem la variable local **vec** com un **OrderedCollection** de size = **parametre** amb tots els seus elements com a **nil**.

També inicialitzem la variable **bt** com una instància de la classe **BT**, és a dir, la variable **continuationQueue** pren per valor un **OrderedCollection** amb un element **nil**.

Posteriorment cal assignar a cada posició de la variable local **vec** una instància de **bt try: (1 to parametre) asArray**. Amb això aconseguim que **vec** tingui totes les combinacions possibles de valors des de 1 fins a **parametre**. Cada combinació de valors representa el següent: en cada posició o columna on es mou una reina, el valor que es mostra és la posició de la fila a la qual es troba. Per exemple, si tenim el següent taulell per 4 reines, és a dir, **parametre** és igual a 4, el **OrderedCollection vec** guardarà els valors que trobem a continuació.

Variable vec:

Col 1	Col 2	Col 3	Col 4
2	4	1	3



Al guardar els possibles resultats així ens evitem la comprovació de dues reines en una mateixa columna.

A continuació, per comprovar que les posicions del **OrderedCollection** **vec** són legals hem utilitzat la funció de la classe **BT assert**. Aquestes tres instàncies utilitzades comproven que no hi ha dues reines en una mateixa fila i que no hi ha dues reines en una mateixa diagonal.

Com solament volem una única solució del problema de les NQueens, fem un **^ vec**, fent així que es pari l'execució de la funció i no es continuin executant les continuacions guardades de les instàncies **try**.

```
solve
| vec bt |
vec := OrderedCollection new: parametre withAll: nil.

bt := BT with: [ nil ].

"Inicialització del vector de trys"
1 to: parametre do: [ :i | vec at: i put: (bt try: (1 to: parametre ) asArray) ].

"Comprovació de les posicions legals"
1 to: parametre do: [ :i | 1 to: i do: [ :j |
    i ~= j
    ifTrue: [bt assert: (vec at: i) ~= (vec at: j).
               bt assert: ((vec at: i)-i) ~= ((vec at: j)-j).
               bt assert: ((vec at: i)+i) ~= ((vec at: j)+j).
            ]]].

^ vec.
```

3.2 Implementació del mètode **#solveAll**

El mètode **#solveAll** es diferencia del **#solve** simplement en el fet que no retorna una única solució, si no que utilitza el Transcript per mostrar-les totes i anar iterant al Backtracking amb l'ús de **bt next**.

```

solveAll
| vec bt |
vec := OrderedCollection new: parametre withAll: nil.

bt := BT with: [ nil ].

"Inicialització del vector de try"
1 to: parametre do: [ :i | vec at: i put: (bt try: (1 to: parametre ) asArray) ].

"Comprovació de no repetició a la fila"
1 to: parametre do: [ :i | 1 to: i do: [ :j |
    i ~= j
    ifTrue: [bt assert: (vec at: i) ~= (vec at: j).
        bt assert: ((vec at: i)-i) ~= ((vec at: j)-j).
        bt assert: ((vec at: i)+i) ~= ((vec at: j)+j).
    ]]].

vec asString traceCr.

bt next.

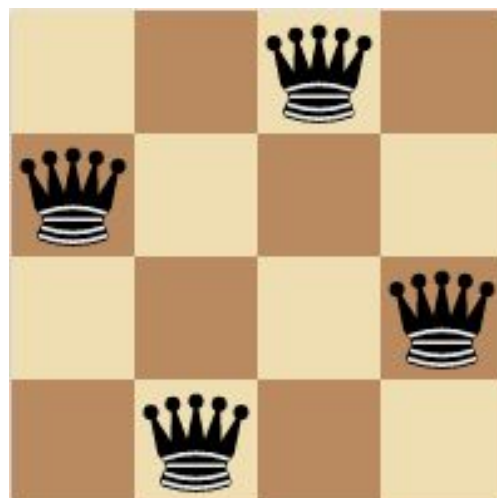
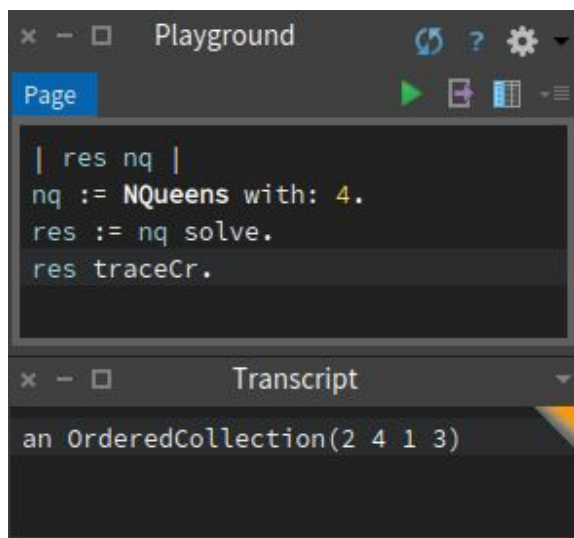
```

3.3 Testeig classe NQueens

Per realitzar el testing de la classe **NQueens** del mètode **solve** hem agafat diversos valors de **parametre** i hem introduït els nostres resultats en un editor d'escacs per veure si la solució era correcta.

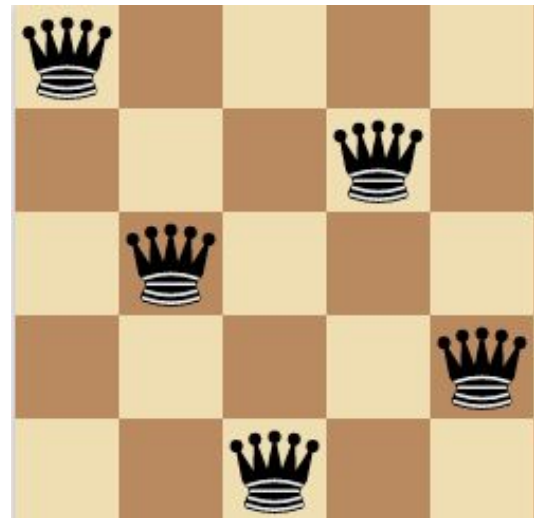
En el cas de comprovar els resultats per el **solveAll** ens hem fixat que el número de solucions que ens dona el nostre codi correspongui amb el número de solucions que ens diu la pàgina següent: <http://www.ic-net.or.jp/home/takaken/e/queen/>

3.3.1 Testeig mètode #solve



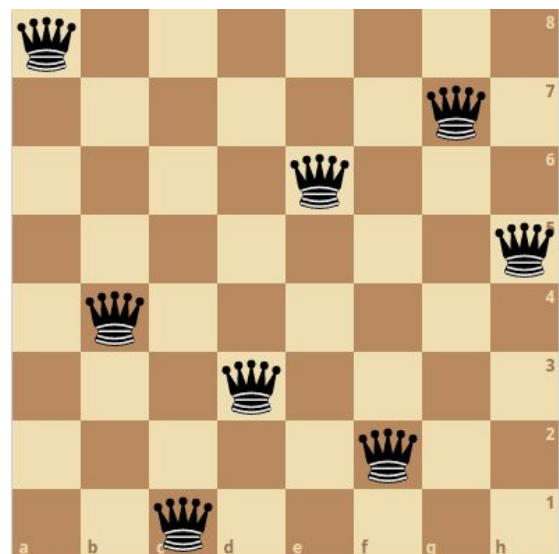

```
Playground
Page
| res nq |
nq := NQueens with: 5.
res := nq solve.
res traceCr.

Transcript
an OrderedCollection(1 3 5 2 4)
```



```
Playground
Page
| res nq |
nq := NQueens with: 8.
res := nq solve.
res traceCr.

Transcript
an OrderedCollection(1 5 8 6 3 7 2 4)
```



3.3.2 Testeig mètode #solveAll

The image displays two screenshots of a Scala Playground interface, demonstrating the `#solveAll` method of the `NQueens` class.

Top Screenshot:

- Code Editor:** Contains the following Scala code:

```
| res nq |  
nq := NQueens with: 4.  
res := nq solveAll.
```
- Transcript:** Shows the output of the code execution:

```
'an OrderedCollection(2 4 1 3)'  
'an OrderedCollection(3 1 4 2)'
```

Bottom Screenshot:

- Code Editor:** Contains the following Scala code:

```
| res nq |  
nq := NQueens with: 6.  
res := nq solveAll.
```
- Transcript:** Shows the output of the code execution, listing four solutions:

```
'an OrderedCollection(2 4 6 1 3 5)'  
'an OrderedCollection(3 6 2 5 1 4)'  
'an OrderedCollection(4 1 5 2 6 3)'  
'an OrderedCollection(5 3 1 6 4 2)'
```

× − □ Playground ↺ ? ⚙

Page ▶ 📄 📋 ☰

```
| res nq |  
nq := NQueens with: 7.  
res := nq solveAll.
```

× − □ Transcript ▼

'an OrderedCollection(1 3 5 7 2 4 6)'
'an OrderedCollection(1 4 7 3 6 2 5)'
'an OrderedCollection(1 5 2 6 3 7 4)'
'an OrderedCollection(1 6 4 2 7 5 3)'
'an OrderedCollection(2 4 1 7 5 3 6)'
'an OrderedCollection(2 4 6 1 3 5 7)'
'an OrderedCollection(2 5 1 4 7 3 6)'
'an OrderedCollection(2 5 3 1 7 4 6)'
'an OrderedCollection(2 5 7 4 1 3 6)'
'an OrderedCollection(2 6 3 7 4 1 5)'
'an OrderedCollection(2 7 5 3 1 6 4)'
'an OrderedCollection(3 1 6 2 5 7 4)'
'an OrderedCollection(3 1 6 4 2 7 5)'
'an OrderedCollection(3 5 7 2 4 6 1)'
'an OrderedCollection(3 6 2 5 1 4 7)'
'an OrderedCollection(3 7 2 4 6 1 5)'
'an OrderedCollection(3 7 4 1 5 2 6)'
'an OrderedCollection(4 1 3 6 2 7 5)'
'an OrderedCollection(4 1 5 2 6 3 7)'
'an OrderedCollection(4 2 7 5 3 1 6)'
'an OrderedCollection(4 6 1 3 5 7 2)'
'an OrderedCollection(4 7 3 6 2 5 1)'
'an OrderedCollection(4 7 5 2 6 1 3)'
'an OrderedCollection(5 1 4 7 3 6 2)'
'an OrderedCollection(5 1 6 4 2 7 3)'
'an OrderedCollection(5 2 6 3 7 4 1)'
'an OrderedCollection(5 3 1 6 4 2 7)'
'an OrderedCollection(5 7 2 4 6 1 3)'
'an OrderedCollection(5 7 2 6 3 1 4)'
'an OrderedCollection(6 1 3 5 7 2 4)'
'an OrderedCollection(6 2 5 1 4 7 3)'
'an OrderedCollection(6 3 1 4 7 5 2)'
'an OrderedCollection(6 3 5 7 1 4 2)'
'an OrderedCollection(6 3 7 4 1 5 2)'
'an OrderedCollection(6 4 2 7 5 3 1)'
'an OrderedCollection(6 4 7 1 3 5 2)'
'an OrderedCollection(7 2 4 6 1 3 5)'
'an OrderedCollection(7 3 6 2 5 1 4)'
'an OrderedCollection(7 4 1 5 2 6 3)'
'an OrderedCollection(7 5 3 1 6 4 2)'