Pràctica CAP Q1 curs 2022-23
Corutines en Rhino

- A realitzar en grups de **2 persones**.
- A entregar com a molt tard el **20 gener de 2023**.

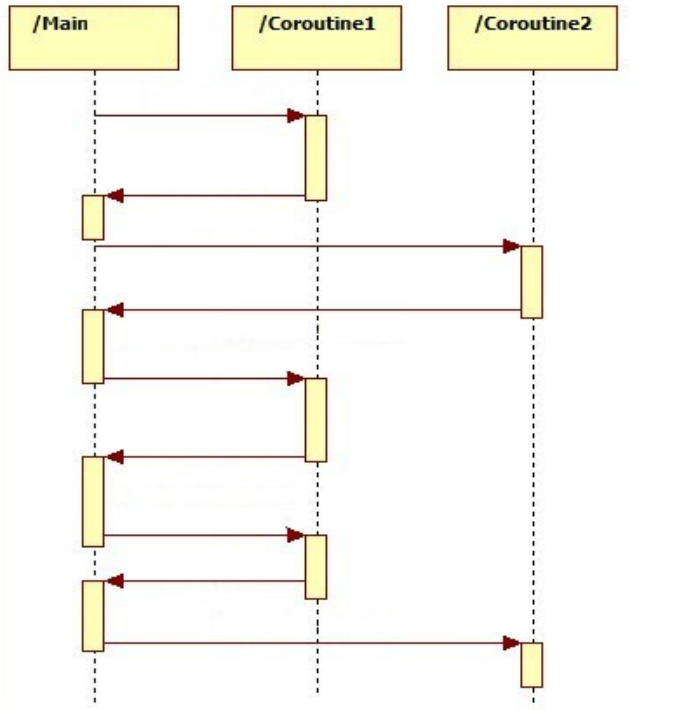
Descripció resumida:
tl;dr ⇒ Aquesta pràctica va de continuacions.
La pràctica de CAP d'enguany serà una investigació del concepte general d'estructura de control, aprofitant les capacitats d'introspecció i intercessió que ens proporcionen `Rhino`/`Javascript`. Estudiarem les conseqüències de poder guardar la pila d'execució (a la que tenim accés gràcies a la funció `Continuation`). El fet de poder guardar i restaurar la pila d'execució d'un programa ens permet implementar qualsevol estructura de control. També permeten 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*), una de les pràctiques recurrents de CAP. La novetat és que no l'hem fet mai amb `Rhino`.

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 via un fitxer .js.

Nota: Caldrà que feu servir un `Rhino` especial, el `.jar` del qual us passo amb l'enunciat de la pràctica. El `Rhino` a *github* té un *bug* molt emprenyador quan treballem amb continuacions. He modificat el codi font i he tornat a crear el `.jar` per no haver de tenir aquest problema. La qüestió és que ara el codi amb continuacions que he provat funciona bé, com cal, però no sé quin és l'abast real de la modificació que he fet, així que treballarem amb aquest `.jar` de manera, diguem, *experimental*. En principi, us estalviarà problemes a l'hora de fer la pràctica.

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 C_1 invoca una altra corutina C_2, s'atura i espera que se la torni a invocar. Si això passa, l'execució de C_1 es reprén just en el moment en que va invocar C_2; si ara C_1 torna a invocar C_2, l'execució de C_2 es reprén en el moment que va decidir invocar a un altre corutina. La idea es pot veure en aquest gràfic:



Hi ha diverses maneres de definir les corutines, en funció de diverses característiques i la literatura sobre el tema és molt nombrosa. Nosaltres ens limitarem a una definició senzilla, adaptada a Javascript, que ens permetrà jugar amb el concepte[1]. Això és el que us demano a l'enunciat, tot seguit.

Cal fer una funció: **`make_coroutine`**, tal que construeixi objectes que es comportin com a corutines (en aquest sentit, podriem pensar que són *com una mena de* funcions). Caldrà passar una funció com a argument, que és on tindrem el codi de la corutina. Aquesta funció tindrà dos paràmetres, que anomenem **`resume`** i **`value`** :

```
<corutina> ← make_coroutine(function(resume,value) { ... })
```

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

1 Es fa servir en el capítol 17 del llibre *Scheme and The Art Of Programming*
`https://www.cs.unm.edu/~williams/cs357/springer-friedman.pdf`

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

resume(<nom corutina>, <valor passat a la corutina invocada>)

ja que **resume** ha de ser *una funció amb dos paràmetres*: el primer és una referència a una altra corutina (així doncs, *no existiran corutines anònimes*), 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.

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. La funció que passarem com a paràmetre, **resume**, a la funció amb la que es construeix la corutina (el paràmetre de **make_coroutine**) *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 la funció que es passa com a paràmetre a **make_coroutine**, però després, quan s'invoca retornant d'una crida anterior (amb **resume(c,v)**), ha de fer una altra cosa (ha de fer **c(v)**, *però no només això!* aquesta és una de les coses que heu de pensar).

Veiem un exemple. Si executem la funció **exemple_senzill()**:

```
function exemple_senzill() {
    let a = make_coroutine( function(resume, value) {
        print("Ara estem a la corutina 'A'");
        print("Venim de", resume(b,'A'));
        print("Tornem a 'A'");
        print("Venim de", resume(c,'A'));
    });
    let b = make_coroutine( function(resume, value) {
        print("    Ara estem a la corutina 'B'");
        print("    Venim de", resume(c,'B'));
        print("    Tornem a 'B'");
        print("    Venim de", resume(a,'B'));
    });
    let c = make_coroutine( function(resume, value) {
        print("        Ara estem a la corutina 'C'");
        print("        Venim de", resume(a,'C'));
        print("        Tornem a 'C'");
        print("        Venim de", resume(b,'C'));
    });

    // amb aquest codi evitem "complicar-nos la vida" amb
    // problemes d'acabament quan cridem la corutina inicial
    if (typeof(a) === 'function') {
        a('*')  // el valor '*' que passem a 'a' és irrellevant
    }
}
```

El resultat és:

```
Ara estem a la corutina 'A'
    Ara estem a la corutina 'B'
        Ara estem a la corutina 'C'
Venim de C
Tornem a 'A'
        Venim de A
        Tornem a 'C'
    Venim de C
    Tornem a 'B'
Venim de B
```

Així doncs, el que us demano és:

a) [**6 punts**] Implementeu la funció **`make_coroutine`** i feu que l'exemple de l'enunciat funcioni correctament (si, a més, vosaltres penseu altres exemples, molt millor).

b) [**4 punts**] Direm que **dos arbres binaris tenen la *mateixa frontera* (*same fringe*) si tenen *exactament* les mateixes fulles, llegides d'esquerra a dreta**. Aleshores, **el problema *Samefringe*** consisteix en, **donats dos arbres binaris, respondre si tenen la mateixa frontera o no**.

En aquest apartat de la pràctica us demano que implementeu una solució al problema *Samefringe*, o de la *mateixa frontera*, fent servir corutines (les que heu implementat en l'apartat a).

Durant els anys 1976 i 1977, a les pàgines del *ACM SIGART Newsletter*[2], es va fer un debat sobre si *Samefringe* és el problema més simple que requereix multiprocessament o *corutines* per ser resolt fàcilment. El problema, en realitat, és escriure un programa per resoldre *Samefringe* que *no consumeixi molt d'emmagatzematge*. És a dir, no serveix un programa que genera una llista amb les fulles d'un arbre, una altra llista amb les fulles de l'altre arbre i compara les dues llistes. Hauríem d'acabar el procés tan aviat detectem dues fulles diferents, i no volem haver de generar les dues llistes completes per fer això.

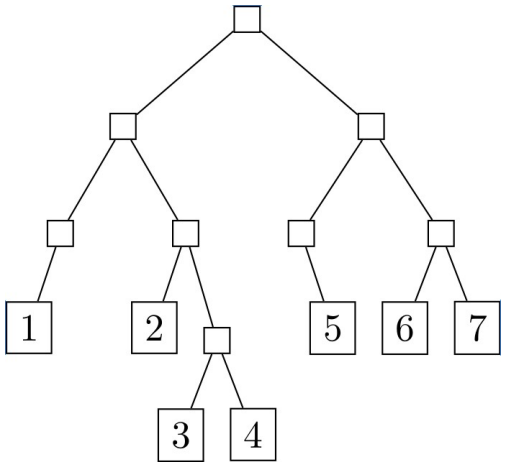
Com que només tenen interés les fulles de l'arbre, decidirem que, per a nosaltres, un arbre binari serà:

- Un arbre buit, representat per un *array* buit: **`[]`**
- Una fulla, representada per un *array* amb un sol element (un nombre): **`[x]`**
- Un arbre, representat per un *array* de 2 elements: **`[fill_esquerre, fill_dret]`**. Naturalment, **`fill_esquerre`** i **`fill_dret`** són arbres binaris.

2 Que podeu seguir a **`https://dl.acm.org/loi/intelligence/group/d1970.y1976`** prèvia autenticació UPC.

Per exemple, l'arbre de la figura es representarà amb l'*array*:

`[ [ [ [1], [] ], [ [2], [ [3], [4] ] ] ], [ [ [], [5] ], [ [6], [7] ] ] ]`



Altres exemples:

```
let a1 = [ [ [ [1], [] ], [ [2], [ [3], [4] ] ] ], [ [ [], [5] ], [ [6], [7] ] ] ]
let a2 = [ [ [ [1], [2] ],  [ [3], [4] ] ],  [ [ [5], [6] ], [ [7], [] ] ] ]
let a3 = [ [ [ [1], [2] ],  [ [3], [4] ] ],  [ [ [5], [9] ], [ [7], [] ] ] ]
let a4 = [ [ [ [1], [2] ],  [ [3], [4] ] ],  [ [ [5], [6] ], [ [7], [8] ] ] ]
```

Heu d'implementar una funció **`same_fringe(tree1, tree2)`**, *fent servir corutines*, tal que, per exemple:

`same_fringe(a1,a2)` ⇨ **`true`**
`same_fringe(a1,a4)` ⇨ **`false`**
`same_fringe(a4,a2)` ⇨ **`false`**
`same_fringe(a3,a4)` ⇨ **`false`**

i sigui eficient en espai, acabant el programa de seguida que trobi dues fulles diferents (és a dir, essencialment no cal guardar cap informació i només cal anar generant les fulles dels arbres i comparant-les).