# Pràctica CAP Q1 curs 2023-24

## Implementar `catch/throw` de Common Lisp, en Rhino

- A realitzar en grups de **2 persones**.
- A entregar com a molt tard el **20 gener de 2024**.

**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 les utilitats de gestió d'errors anomenades `catch/throw`, inspirades en el seu equivalent en Common Lisp (però més senzilles, les hem simplificat).

**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.

**Observació**: 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. Si veieu *coses rares*, feu servir la versió 1.7.14 oficial.

## **Enunciat**:

El llenguatge de programació Common Lisp va néixer com un intent d'estandaritzar les diverses versions de Lisp que existien en aquella època (principalment MacLisp). El document ANSI definitiu que especifica Common Lisp és de l'any 1994, tot i que va començar a gestar-se deu anys abans, el 1984.

Aquest llenguatge té un mecanisme de control *dinàmic* anomenat `catch`/`throw`, que serveix per fer retorns dinàmics (amb `throw`) des d'una part del codi a una altra marcada amb `catch`. Una crida a `catch` inicia l'execució d'un cert fragment de codi etiquetat amb un objecte (el *tag*). La crida a `catch` retorna allò que retorni aquest fragment de codi. Des d'aquest codi que s'està executant (amb la crida a `catch` encara vigent, pendent de retornar) podem invocar la funció `throw` amb el mateix *tag* creat amb el `catch`, i retornarà al punt on es va cridar `catch` amb un valor de retorn determinat, especificat a la mateixa crida a `throw`. A grans trets, això és el que fan `catch`/`throw`[1].

Per a nosaltres, treballant en JavaScript/Rhino, els *tags* seran *strings*, i el codi a executar en la crida a `catch` serà una funció sense paràmetres (que anomenem, si recordeu el que vam explicar a classe, *thunk*). Anomenarem les funcions que volem implementar **`my_catch`** i **`my_throw`** (ja que JavaScript/Rhino ja té `catch` i `throw`, tot i que no fan el mateix que demanem aquí! No volem interferir amb les paraules clau de JavaScript/Rhino). Així doncs haurem de definir:

```
function my_catch (tag, fun) {
    // Pre: tag és una String; fun és un Thunk
    ...
}
```

i també

```
function my_throw (tag, valor) {
    // Pre: tag és una String; val és qualsevol valor (amb una excepció)
    ...
}
```

Aleshores, en invocar **`my_catch (tag, fun)`** s'executarà **`fun`**. Mentre estem executant **`fun`**, podem invocar **`my_throw (tag, valor)`**, amb el mateix **`tag`** amb el que hem invocat **`my_catch`**, i el valor de retorn de **`my_catch`** serà **`valor`**. Per raons tècniques, aquest **`valor`** no pot ser `instanceof Continuation`. Si, mentre executem **`fun`**, no s'invoca mai **`my_throw`** el valor de retorn de **`my_catch`** serà allò que retorni **`fun()`**.

Ara bé, això es pot complicar, perquè mentre executem **`fun`** podem haver cridat una o més vegades **`my_catch`** amb diferents **`tag`**s. I, si fem **`my_throw`** a un **`tag`** determinat, tots els **`tag`**s associats a **`my_catch`** posteriors al **`my_catch`** on estem retornant han de quedar invalidats. També podem fer **`my_throw`** a **`tag`**s que ja no existeixen (perquè ja hem acabat l'execució del *thunk* associat al **`my_catch`** que crea el **`tag`**) o que no s'han creat mai amb cap crida a **`my_catch`**. En aquest cas cal generar un error. Tampoc acceptarem **`tag`**s repetits.

1 Si teniu interés, veieu `https://www.cs.cmu.edu/Groups/AI/html/hyperspec/HyperSpec/Body/speope_catch.html#catch`

## Exemples:

Per aclarir una mica el que fan `my_catch` i `my_throw`, veiem-ne alguns exemples (suposem `my_catch` i `my_throw` ja definits d'alguna manera):

**a/** Exemples senzills:

```
print(my_catch('etiqueta',function () { return 2 + 3 * 100 }))
```

Aquest programa es limita a escriure el que retorna `my_catch`, que és el que retorna el *thunk* que li hem passat com a argument. És a dir, escriu `302`.

En canvi, si fem servir el `my_throw`:

```
print(my_catch('etiqueta',function () { return 2 + 3 * my_throw('etiqueta',100) }))
```

Aquest programa es limita a escriure el que retorna `my_catch`, que, per culpa del `my_throw`, és `100`.

**b/** Cal anar amb compte, ja que no podem fer `my_throw` a etiquetes que no existeixen, o que ja no tenen sentit (perquè hem retornat del *thunk* corresponent al `my_catch`). Per exemple:

```
print(my_catch('etiq',function () { return 2 + 3 * my_throw('etiqueta',100) }))
```

Obtindrem un error -que podeu generar amb `throw` (el propi de JavaScript/Rhino!)-:

```
js: "catch_throw.js", line 70: exception from uncaught JavaScript throw: ==> my_throw a un
tag sense el my_catch corresponent <==
       at catch_throw.js:70 (_throw)
       (...)
```

Un altre exemple d'error és fer un `my_throw` després d'haver acabat l'execució del *thunk* que li hem passat al `my_catch`:

```
function test(x) {
    return 2 + 3 * (x === 0 ? my_throw('etiqueta',100) : 100)
}
print(my_catch('etiqueta', function () { return test(1) }))  // escriurà 302

print(my_catch('etiqueta', function () { return test(0) }))  // escriurà 100

print(test(0))            // generarà un error, el mateix que hem vist abans
```

**c/** Aquest exemple és una mica més sofisticat:

```
function getRandomInt(max) {
  return Math.floor(Math.random() * max);
}

function check(x) {
    if (x === 0) {
        my_throw('zero','zero')
    } else if (x === 1) {
        my_throw('one','one')
    }
```

```
    return x
}

print(my_catch('zero',function() {
    for (let i = 0; i < 10; i++) {
        print(my_catch('one', function () { return check(getRandomInt(5)) }) )}
    return 'finito'
}))
```

Aquest codi pot generar diversos *outputs* (ja que fem servir nombres aleatoris), per exemple (mireu d'entendre-ho bé tot):

```
2               3               2               one             zero
2               4               one             one
3               one             2               zero
4               2               one
4               3               3
3               3               4
4               2               2
2               3               zero
3               one
4               3
finito          finito
```

Fixeu-vos bé que un cop retornem per haver fet `my_throw('zero',...)` ja no podem fer `my_throw('one',...)`, perquè l'etiqueta `'one'`, *en haver estat creada dins del codi del que hem sortit amb el* `my_throw('zero',...)`, ja no té cap sentit. No podem fer un `my_throw('one',...)`. mai més (a no ser que *tornem* a fer un `my_catch('one',...)`).

## Esquema:

Un esquema que us pot ajudar a fer la pràctica pot ser:

```
function current_continuation() { // Sempre va bé tenir current_continuation a mà...
    return new Continuation()
}

// La definició de my_catch i my_throw

let { my_catch, my_throw } =

    ( function () {

        // ... les dades/atributs/estat necessaris (a decidir per vosaltres)

        // ... les funcions auxiliars que us facin falta (a decidir per vosaltres)

        function _catch(tag, fun) {
            // Pre: tag és una String; fun és un Thunk
            // ...
        }

        function _throw(tag, val) {
            // Pre: tag és una String; val és qualsevol valor (però no una Continuation!)
            // ...
        }

        return { my_catch: _catch, my_throw: _throw }

      }())
```

Aquest esquema és orientatiu. Si vosaltres penseu que és millor fer-ho d'una altra manera, endavant! Cap problema.

## Observacions:

Finalment, fixeu-vos que aquesta és una versió *molt senzilla* del `catch`/`throw` de Common Lisp. En particular és fàcil veure que, si en mig del codi on fem servir `my_catch` i/o `my_throw` també fem sortides no locals (*non-local exits*), per exemple fent servir excepcions o continuacions, tot l'esquema senzill que vull que implementeu deixa de funcionar. Podem tenir etiquetes presents al sistema que haurien d'haver-se eliminat i encara hi són, per exemple.

De cara a evitar això hauríem d'implementar un mecanisme similar al `finally` (del `try...catch...finally` de JavaScript), que s'encarrega d'executar un codi sense importar com sortim del bloc definit pel `try...catch`. Seguint amb la inspiració que ve de Common Lisp, hauríem d'implementar el que a Common Lisp s'anomena `unwind-protect`. Això, però, està fora de l'abast d'aquesta pràctica, i d'aquest curs de CAP.