Conceptes Avançats de Programació

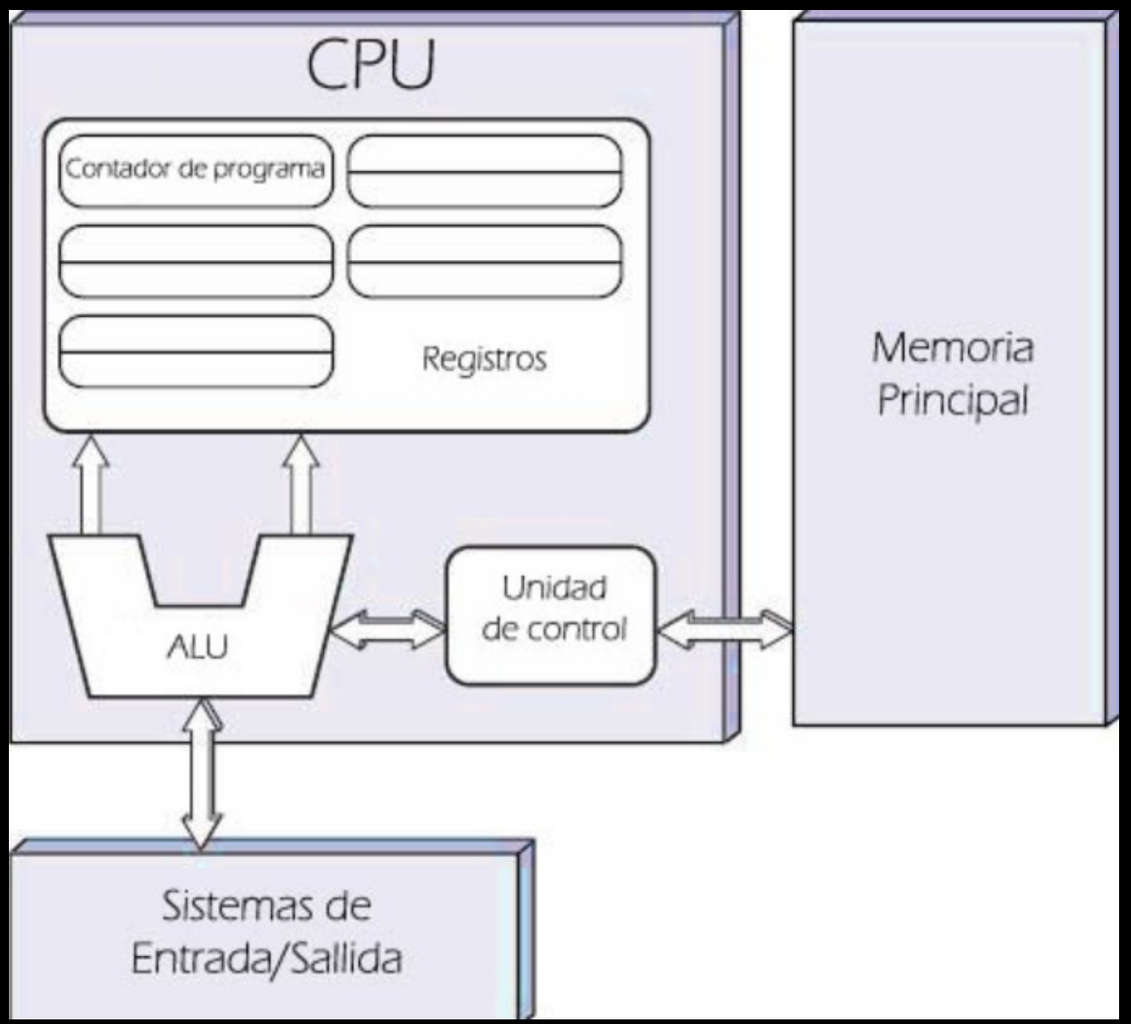
Estructures de Control, Continuacions

Scope & Closures (YDKJSY 2nd. ed., vol. 2)
Kyle Simpson
Leanpub 2020

JavaScript: The Definitive Guide
David Flanagan
O'Reilly 2020

CAP: Estructures de Control

Des dels primers ordinadors amb un sol processador i arquitectura von Neumann (*stored-program computers*), l'estructura de control *per defecte* ha estat la composició seqüèncial d'instruccions: Una instrucció darrera una altra, en l'ordre especificat en el programa.



CAP: Estructures de Control

Aquests ordinadors, però, no serien *Turing-complets* si no tinguessin alguna manera d'*alterar* aquesta seqüència. Els primers ordinadors disposaven ja de:

- **Salts condicionals**
- **Salts incondicionals**
- **Modificació dinàmica d'instruccions** (*self-modifying code*)

Tot i que l'aparició dels registres-índex i l'adreçament indirecte van eliminar la necessitat d'haver de modificar instruccions en temps d'execució (a més de ser una pràctica dubtosa des del punt de vista del disseny de programari).

CAP: Estructures de Control

L'aparició dels llenguatges d'alt nivell (finals dels '50: Fortran, Lisp, Cobol) va fer que s'utilitzessin aquestes instruccions bàsiques per construir estructures de control més abstractes. Es perdia flexibilitat, però es guanyava claredat i, sobre tot, abstracció:

- **Instruccions condicionals**
  (`if` / `case` / `switch` / etc.)
- **Instruccions iteratives**
  (`while...do` / `repeat...until` / `do...while` / etc.)
- **Crides a *subrutina***

CAP: Estructures de Control

Les subrutines, posteriorment anomenades funcions y/o procediments (*procedures*) **van introduir algunes abstraccions interessants:**

- **Pas de paràmetres** (*call by...*) **i retorn de valors**

Convention	Description	Common use
Call by value	Argument is evaluated and copy of the value is passed to subroutine	Default in most Algol-like languages after Algol 60, such as Pascal, Delphi, Simula, CPL, PL/M, Modula, Oberon, Ada, and many others. C, C++, Java (References to objects and arrays are also passed by value)
Call by reference	Reference to an argument, typically its address is passed	Selectable in most Algol-like languages after Algol 60, such as Algol 68, Pascal, Delphi, Simula, CPL, PL/M, Modula, Oberon, Ada, and many others. C++, Fortran, PL/I
Call by result	Parameter value is copied back to argument on return from the subroutine	Ada OUT parameters
Call by value-result	Parameter value is copied back on entry to the subroutine and again on return	Algol, Swift in-out parameters
Call by name	Like a macro – replace the parameters with the unevaluated argument expressions, then evaluate the argument in the context of the caller every time that the called routine uses the parameter.	Algol, Scala
Call by constant value	Like call by value except that the parameter is treated as a constant	PL/I NONASSIGNABLE parameters, Ada IN parameters

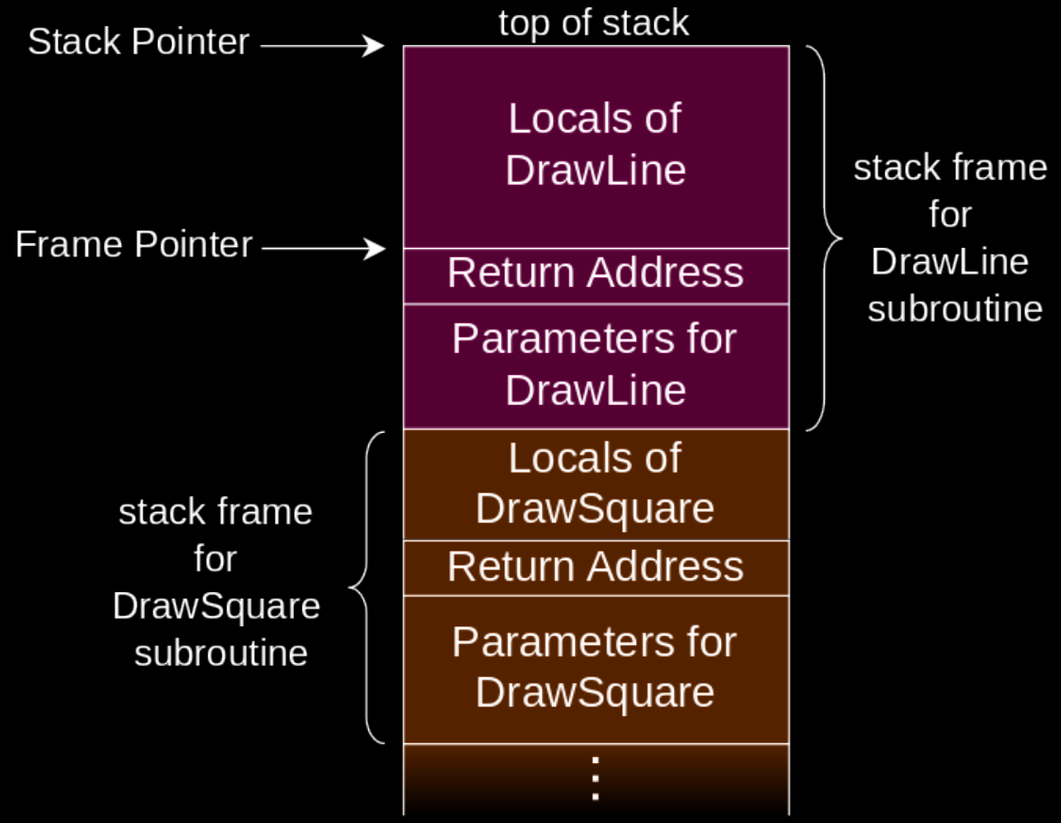
https://en.wikipedia.org/wiki/Subroutine#Main_concepts

CAP: Estructures de Control

Les subrutines, posteriorment anomenades funcions y/o procediments (*procedures*) **van introduir algunes abstraccions interessants:**

- **Pas de paràmetres** (*call by...*) **i retorn de valors**
- **La pila d'execució** (*call stack*)

La pila d'execució va començar sent una manera especialment interessant d'implementar les *subrutines*. De fet, fins i tot permetia que les *subrutines* fossin recursives.



https://en.wikipedia.org/wiki/Call_stack

CAP: Estructures de Control

A mida que va anar apareixent la necessitat de diferentes abstraccions, també van anar apareixent noves estructures de control:

Coroutines (*corutines*) - Contemporànies de les *subrutines* -- Assembladors (1958)

Multithreading - Concurrència a nivell del llenguatge (no del SO) -- Ada (1980), però avui dia molts llenguatges disposen de mecanismes associats: Java, Python, etc.

Backtracking - No com a tècnica de disseny algorísmic, sinò com a estructura de control -- Prolog (1972)

CAP: Estructures de Control

A mida que va anar apareixent la necessitat de diferentes abstraccions, també van anar apareixent noves estructures de control:

Generators (*semicoroutines*) - *subrutines* que generen valors d'un iterador cada cop que són cridades -- CLU (1975), Icon (1977)

Actors (pas de missatges) - Implementació del model teòric d'Actors, per entendre i dissenyar sistemes concurrents -- Erlang (1986)

Excepcions - Gestió d'errors a nivell del llenguatge -- PL/I (1964), però avui dia pràcticament tots els llenguatges "moderns" tenen algún mecanisme del tipus `try...catch` per gestionar-les.

... i moltes més:

A survey of control structures in programming languages
David A. Fisher
ACM SIGPLAN Notices, Volume 7, Issue 11, November 1972 pp 1–13
https://doi.org/10.1145/987361.987363

CAP: Estructures de Control

Les estructures de control també van ser objecte de força recerca acadèmica. Per exemple, P. J. Landin va qüestionar la rellevancia de l'estructura de control més bàsica, la composició seqüèncial, en el context dels llenguatges funcionals*.

Una descoberta interessant va ser la possibilitat de poder abstraure la pila d'execució (en direm *reificar* més endavant en el curs) per poder manipular-la des del programa a alt nivell. Aquesta abstracció s'anomena *continuació*, i és l'estructura de control *fonamental*, en el sentit que *qualsevol altra estructura de control es pot construir si disposem de la possibilitat de manipular continuacions*.**

* The Next 700 Programming Languages
Peter J. Landin
Communications of the ACM Volume 9, Issue 3, March 1966 pp. 157–166
https://doi.org/10.1145/365230.365257

** The Discoveries of Continuations
John C. Reynolds
Lisp and Symbolic Computation Volume 6, Issue 3/4, November 1993 pp. 233-247
https://doi.org/10.1007/BF01019459

CAP: Estructures de Control

Scheme (1975) va ser el primer llenguatge de programació d'alt nivell que va permetre la manipulació de continuacions. Ho feia utilitzant una forma especial pròpia del llenguatge anomenada:

`call-with-current-continuation (call/cc)`
`https://en.wikipedia.org/wiki/Call-with-current-continuation`

Des d'aleshores, diversos llenguatges de programació han incorporat aquesta estructura de control:

- **Ruby**
- **Standard ML**
- **Haskell** (*continuation monad*)
- **Racket**
- **Smalltalk**

CAP: Estructures de Control

Així doncs, quan diem que en aquest curs estudiarem *estructures de control*, volem dir que *estudiarem les continuacions i les farem servir per proporcionar diferents implementacions de diverses estructures de control*.

Primer ho farem en el context de Javascript, fent servir la implementació de Javascript anomenada Rhino, una implementació feta en Java per programadors de Mozilla. Podeu trobar la darrera versió de Rhino (1.7.14, gener 2022) aquí:

`https://github.com/mozilla/rhino/releases/tag/Rhino1_7_14_Release`

Més tard, quan estudiem reflexió en llenguatges de programació dinàmics, tornarem a trobar les continuacions com a exemple de reflexió de comportament (*behavioral reflection*) en el context d'Smalltalk.

CAP: EC: Continuacions

A Javascript/Rhino podem capturar la continuació ***d'una crida a funció*** amb la funció especial **`Continuation()`**, que cal invocar amb **`new`**:

```
new Continuation()
```

Retorna *la continuació de la crida a la funció dins de la que s'ha invocat* **`new Continuation()`**.

CAP: EC: Continuacions

Comencem amb un exemple senzill:

```
function expressio_qualsevol() {
    print("pas 1 ");
    let x = (2 + 3 * 5);
    print("pas 2 ");
    let y = (4 - 3 * 3);
    return x*y
}
```

```
function expressio_qualsevol_amb_continuacions() {
    let k = new Continuation();
    print("pas 1 ");
    let x = (2 + k(3 * 5));
    print("pas 2 ");
    let y = (4 - 3 * 3);
    return x*y
}
```

```
js> expressio_qualsevol()
pas 1
pas 2
-85
js> expressio_qualsevol_amb_continuacions()
pas 1
15
```

En invocar la continuació **k**, hem retornat de la funció (on s'ha creat la continuació amb **new Continuation()**), *ignorant tot allò que quedava per fer*.

CAP: EC: Continuacions

Comencem amb un exemple senzill:

```
function expressio_qualsevol() {
    print("pas 1 ");
    let x = (2 + 3 * 5);
    print("pas 2 ");
    let y = (4 - 3 * 3);
    return x*y
}
```

```
function expressio_qualsevol_amb_continuacions() {
    let k = new Continuation();
    print("pas 1 ");
    let x = (2 + k(3 * 5));
    print("pas 2 ");
    let y = (4 - 3 * 3);
    return x*y
}
```

```
js> expressio_qualsevol()
pas 1
pas 2
-85
js> expressio_qualsevol_amb_continuacions()
pas 1
15
```

En invocar la continuació **k**, hem retornat de la funció (on s'ha creat la continuació amb **new Continuation()**), *ignorant tot allò que quedava per fer*.

Així doncs... és el mateix que fer **return**?

CAP: EC: Continuacions

Veiem un exemple per aclarir això...

```
function factorial(n) {

    function fact_aux(n,m) {
        if (n === 0) {
            return m
        } else {
            let res = fact_aux(n-1,m*n)
            print("pas ", n-1)
            return res
        }
    }

    return fact_aux(n,1)
}
```

```
function factorial_cont(n) {
    let kont = new Continuation()

    function fact_aux(n,m) {
        if (n === 0) {
            kont(m)
        } else {
            let res = fact_aux(n-1,m*n)
            print("pas ", n-1)
            return res
        }
    }

    return fact_aux(n,1)
}
```

```
js> factorial(5)
pas  0
pas  1
pas  2
pas  3
pas  4
120

js> factorial_cont(5)
120
```

CAP: EC: Continuacions

Veiem un altre exemple per aclarir això...

```
function producte_array(arr) {

    function prod_aux(a,i) {
        if (a.length === i) {
            return 1;
        } else {
            if (a[i] === 0) {
                return 0;
            } else {
                let res = a[i] * prod_aux(a,i+1);
                print("El resultat aquí és: ", res)
                return res;
            }
        }
    }

    let resfinal = prod_aux(arr,0);
    print("El resultat final és: ", resfinal);
}
```

```
function producte_array_amb_continuacions(arr) {
    let k = new Continuation();

    function prod_aux(a,i) {
        if (a.length === i) {
            return 1;
        } else {
            if (a[i] === 0) {
                k(0);
            } else {
                let res = a[i] * prod_aux(a,i+1);
                print("El resultat aquí és: ", res)
                return res;
            }
        }
    }

    let resfinal = prod_aux(arr,0);
    print("El resultat final és: ", resfinal);
}
```

CAP: EC: Continuacions

Veiem un altre exemple per aclarir això...

Si executem aquest codi en un array sense zeros:

```
js> producte_array([1,4,5,7,8])
El resultat aquí és:  8
El resultat aquí és:  56
El resultat aquí és:  280
El resultat aquí és:  1120
El resultat aquí és:  1120
El resultat final és:  1120
js> producte_array_amb_continuacions([1,4,5,7,8])
El resultat aquí és:  8
El resultat aquí és:  56
El resultat aquí és:  280
El resultat aquí és:  1120
El resultat aquí és:  1120
El resultat final és:  1120
```

Però, si l'executem en un array amb zeros:

```
js> producte_array([1,4,5,7,0,8])
El resultat aquí és:  0
El resultat aquí és:  0
El resultat aquí és:  0
El resultat final és:  0
js> producte_array_amb_continuacions([1,4,5,7,0,8])
0
```

Què ha passat?

Igual que en el cas anterior, invocar la continuació implica ignorar tot el que queda pendent de fer, en aquest cas, retornar de les crides recursives!

CAP: EC: Continuacions

Un altre exemple: Si executem el següent fragment de codi:

```
function someFunction()  {
  let kont  = new Continuation();
  print("captured: " + kont);
  return kont;
 }
let k = someFunction();
if (k instanceof Continuation) {
   print("k is a continuation");
   k(200);
} else {
   print("k is now a " + typeof(k));
}
print(k);
```

El resultat serà:

```
captured: [object Continuation]
k is a continuation
k is now a number
200
```

CAP: EC: Continuacions

Més exemples... iterant amb continuacions:

```
function current_continuation() {
    print("Agafem la continuacio");
    return new Continuation();
}

let value = 0,
    kont = current_continuation();

print(value);
if (value === 5)
    print("Ha arribat a 5 gracies a la continuacio");
else {
    value++;
    kont(kont);
}
```

tenim:

```
Agafem la continuacio
0
1
2
3
4
5
Ha arribat a 5 gracies a la continuacio
```

CAP: EC: Continuacions

Exercici: Què fa aquest programa? Per què?

```
let cont = 0;

function exercici() {
    cont = new Continuation()
    return false
}

let b = exercici();
for (let i = 1; i < 6; i++) {
    if (!b) {
        print(5-i)
    } else {
        print(i)
    }
}

if (!b) {
    cont(true)
}
```

CAP: EC: Continuacions

Com la continuació ens permet trencar amb el flux d'execució, podem sortir d'un bucle fàcilment...

```
function bucle_infinit(procediment) {
    function itera() {
        procediment();
        itera();
    }
    itera();
}

function comptar_fins_a_n(n) {
    function bucle(funcio_acabament) {
        let count=0

        bucle_infinit(function() {
            if (count === n) {
                funcio_acabament()
            } else {
                print("El comptador és",count);
                count++;
            }
        });
    }

    let k = new Continuation()
    bucle(k)
}
```

```
js> comptar_fins_a_n(10)
El comptador és 0
El comptador és 1
El comptador és 2
El comptador és 3
El comptador és 4
El comptador és 5
El comptador és 6
El comptador és 7
El comptador és 8
El comptador és 9
```

CAP: EC: Continuacions

Fins i tot podem construir un *pseudo-while*...

```
function current_continuation() {
    return new Continuation();
}

function while_cont(condicio_continuacio_bucle, cos_bucle) {
    let kont = current_continuation();
    if (condicio_continuacio_bucle()) {
        cos_bucle();
        kont(kont);
    }
    return undefined;
}

function comptar_n(n) {
    let count = 0;

    function condicio() { return (count < n) }

    function cos() {
       print("El comptador és",count);
       count++;
    }

    return while_cont(condicio,cos);
}
```

```
js> comptar_n(10)
El comptador és 0
El comptador és 1
El comptador és 2
El comptador és 3
El comptador és 4
El comptador és 5
El comptador és 6
El comptador és 7
El comptador és 8
El comptador és 9
```

CAP: EC: Continuacions

Exemple: No Determinisme (l'operador **`amb`**) - *backtracking*

```
function amb(choices) {
  let cc = current_continuation();
  if (choices && choices.length > 0) {
    let choice = choices.shift();
    fail_stack.push(cc);
    return choice;
  } else {
    return fail();
  }
}
```

En aquest exemple farem servir una *pila de continuacions* (**`fail_stack`**)

Veure *Teach Yourself Scheme in Fixnum Days*, (de Dorai Sitaram), cap. 14:
`https://ds26gte.github.io/tyscheme/index-Z-H-16.html#node_chap_14`

CAP: EC: Continuacions

Exemple: No Determinisme (l'operador **`amb`**) - *backtracking*

```
function fail() {
  if (fail_stack.length > 0) {
    let back_track_point = fail_stack.pop();
    back_track_point(back_track_point);
  } else {
    throw 'back-tracking stack exhausted!';
  }
}
```

```
function assert(condition) {
  if (condition) {
    return true;
  } else {
    fail();
  }
}
```

Veure *Teach Yourself Scheme in Fixnum Days*, (de Dorai Sitaram), cap. 14:
https://ds26gte.github.io/tyscheme/index-Z-H-16.html#node_chap_14

CAP: EC: Continuacions

Exemple: No Determinisme (l'operador **`amb`**) - *backtracking*

```
function current_continuation() {
   return new Continuation();
}

var { amb_reset, fail, amb, assert } =
   ( function () {
   let fail_stack = [];

   function amb_reset() { fail_stack = []; }

   function fail() { ... }

   function amb(choices) { ... }

   function assert(condition) { ... }

   return { amb_reset: amb_reset, fail: fail, amb: amb, assert: assert }
}());
```

Veure *Teach Yourself Scheme in Fixnum Days*, (de Dorai Sitaram), cap. 14:
https://ds26gte.github.io/tyscheme/index-Z-H-16.html#node_chap_14

CAP: EC: Continuacions

Exemple senzill:

```
var a = amb([1,2,3,4,5,6,7]);
var b = amb([1,2,3,4,5,6,7]);
var c = amb([1,2,3,4,5,6,7]);

assert( ((c*c) === (a*a + b*b)) );

print(a,' -- ',b,' -- ', c);

assert( (b < a) );

print(a,' -- ',b,' -- ', c);
```

```
$ java -cp ../rhino1.7.14/lib/rhino-1.7.14.jar
                org.mozilla.javascript.tools.shell.Main -opt -2 amb.js
3  --  4  --  5
4  --  3  --  5
4  --  3  --  5
```

CAP: EC: Continuacions

Exemple: La tribu des Kalotan:

Els Kalotan són una tribu desconeguda amb una característica peculiar: Els mascles sempre diuen la veritat. Les femelles no fan mai dues sentencies vertaderes consecutives, ni dues sentències falses consecutives (suposem l'existència de només dos gèneres).

Un antropòleg, anomenem-lo Worf, ha començat a estudiar els Kalotan, que parlen el llenguatge Kalotan. Un dia, es troba una parella (heterosexual) i el seu fill/filla Kibi. Worf pregunta en Kibi: "Ets un noi?" i Kibi respon en Kalotan, que l'antropòleg no entén.

Worf pregunta els pares (que entenien el català) que què ha dit en Kibi. Un dels pares respon: "Kibi ha dit: 'sóc un noi'". L'altre afegeix: "Kibi és noia. Kibi ha mentit”

Resol el gènere d'en Kibi i els seus pares.

CAP: EC: Continuacions

Exemple: La tribu des Kalotan:

```
var progenitor1     = amb(['m','f']);
var progenitor2     = amb(['m','f']);
var kibi            = amb(['m','f']);
var kibi_va_dir     = amb(['m','f']);
var kibi_va_mentir = amb([true,false]);

// els pares han de ser de sexe diferent
assert((progenitor1 != progenitor2));


// kibi és mascle => no va mentir
assert((       (kibi === 'm') ? (!kibi_va_mentir) : (true)));
```

CAP: EC: Continuacions

Exemple: La tribu des Kalotan:

```
// el primer progenitor és mascle => kibi va dir que era mascle i
// i com el segon progenitor és femella va mentir en una sentencia i va dir
// la veritat en l'altre
assert(((progenitor1 === 'm') ? (kibi_va_dir === 'm' &&
                                 XOR((kibi === 'f' && !kibi_va_mentir),
                                     (kibi === 'm' &&  kibi_va_mentir))) : (true)));
```

```
// el primer progenitor és femella => no sabem si el que va dir
// és cert o fals, però sabem que el segon progenitor és mascle i no va mentir.
assert(((progenitor1 === 'f') ? (kibi === 'f' && kibi_va_mentir) : (true)));
```

CAP: EC: Continuacions

Exemple: La tribu des Kalotan:

```
// kibi no va mentir => o bé va dir que era mascle i ho és, o va dir que era femella i ho és.
assert((    (!kibi_va_mentir) ? (XOR((kibi_va_dir === 'm' && kibi === 'm'),
                                     (kibi_va_dir === 'f' && kibi === 'f'))) : (true)));
```

```
// kibi va mentir => o bé va dir que era mascle i és femella o a l'inrevés
assert((    (kibi_va_mentir) ? (XOR((kibi_va_dir === 'm' && kibi === 'f'),
                                    (kibi_va_dir === 'f' && kibi === 'm'))) : (true)));
```

Finalment, només cal fer:

```
print('Progenitor1 -> ', progenitor1,
      ' Progenitor2 -> ', progenitor2,
      ' Kibi -> ', kibi);
```

CAP: EC: Continuacions

Exemple: No Determinisme (l'operador **`amb`**) - *backtracking*

Aquesta implementació no és 100% satisfactòria, ja que l'**`amb`**, tal i com l'hem implementat, no satisfà algunes propietats que hauria de tenir. Veure el capítol 14 de *Teach Yourself Scheme in Fixnum Days*, (de Dorai Sitaram), cap. 14:

`https://ds26gte.github.io/tyscheme/index-Z-H-16.html#node_chap_14`

CAP: EC: Continuacions

Hem explicat al començament que la primera implementació que es va fer del procés de *reificació* de les continuacions era en la forma d'una funció anomenada **`call-with-current-continuation`** (`call/cc`) en el llenguatge de programació **`Scheme`**.

Ara ja estem en condicions d'entendre aquesta manera de fer servir les continuacions: `call/cc` *té com a argument una funció d'un paràmetre, que s'executa passant-li com a argument la continuació de la crida a* `call/cc`.

CAP: EC: Continuacions

Hem explicat al començament que la primera implementació que es va fer del procés de *reificació* de les continuacions era en la forma d'una funció anomenada **`call-with-current-continuation`** (`call/cc`) en el llenguatge de programació **Scheme**.

Ara ja estem en condicions d'entendre aquesta manera de fer servir les continuacions: `call/cc` *té com a argument una funció d'un paràmetre, que s'executa passant-li com a argument la continuació de la crida a* `call/cc`.

Ho podem entendre millor implementant-la, ja que tenim la funció **`Continuation`** de Javascript/Rhino:

```
function callcc(f) {
    let kont = new Continuation();
    return f(kont);
}
```

CAP: EC: Continuacions

Exemple més complicat:
breakpoints

```
js> cc = [1,[2,[3,4]],[[9,7,[3]]]]
1,2,3,4,9,7,3
js> b = aplana_array_nombres(cc)
1
js> torna()
2
js> torna()
3
js> torna()
4
js> torna()
9
js> torna()
7
js> torna()
3
js> torna()
1,2,3,4,9,7,3
js> print_array(cc)
[1,[2,[3,4,],],[[9,7,[3,],],],]
js> print_array(b)
[1,2,3,4,9,7,3,]
```

```
let torna  = "qualsevol cosa"
let escapa = new Continuation()

function my_break(s) {
    function break_receiver(cont) {
        torna = function() { return cont(s) };
        escapa(s)
    }

    return callcc(break_receiver)
}
```

```
function aplana_array_nombres(arr) {
    if (arr.length === 0) {
        return []
    } else if (typeof(arr) === "number") {
        return [my_break(arr)]
    } else {
        let copia = arr.slice(0); // Copiem per no destruir
        let primer_element = copia.shift();
        let aplanat = aplana_array_nombres(primer_element);
        return aplanat.concat(aplana_array_nombres(copia));
    }
}
```

CAP: EC: Continuacions

Exemple: Fils cooperatius (*cooperative multithreading*)

Quan parlem de sistemes multi-fil (*multithread*) trobem dues possibilitats: O bé es gestiona de manera que cada fil cedeix el control a altres fils voluntariament (i així el fil s'executa fins que ell vol), o bé el sistema subjacent a l'execució multi-fil (la màquina virtual, per exemple) decideix quant de temps d'execució li pertoca a cada fil en funció, per exemple, d'un sistema de prioritats. En el primer cas parlarem de *cooperative* (o *non-preemptive*) *multithreading*, en el segon cas de *preemptive multithreading*. Tots dos models tenen els seus avantatges i inconvenients.

En aquest exemple farem servir continuacions per implementar un sistema que ens permeti executar els nostres programes de manera concurrent amb *cooperative multithreading*.

La idea és implementar una funció **`make_thread_system()`** que retorni un objecte amb quatre propietats, cadascuna d'elles és una funció de l'API amb la que podem utilitzar el multi-fil cooperatiu.

CAP: EC: Continuacions

Exemple: Fils cooperatius (*cooperative multithreading*)

Si fem `let mts = make_thread_system()`, aleshores `mts` és un objecte amb les funcions:

- `mts.spawn(thunk)`: Posa un *thread* nou (la funció que anomenem *thunk*, una funció *sense* paràmetres) a la cua de *threads.*
- `mts.quit()`: Atura el *thread* on s'executa i el treu de la cua de *threads.*
- `mts.relinquish()`: Cedeix (*yields*) el control del *thread* actual a un altre *thread*.
- `mts.start_threads()`: Comença a executar els *threads* de la cua de *threads*.

CAP: EC: Continuacions

Exemple: Fils cooperatius (*cooperative multithreading*)

```
// fitxer coop-threads.js

let counter = 10;

function make_thread_thunk(name, thread_system) {
    function loop() {
       if (counter < 0) {
          thread_system.quit();
       }
       print('in thread',name,'; counter =',counter);
       counter--;
       thread_system.relinquish();
       loop();
    };
    return loop;
}

let thread_sys =  make_thread_system();

thread_sys.spawn(make_thread_thunk('a', thread_sys));
thread_sys.spawn(make_thread_thunk('b', thread_sys));
thread_sys.spawn(make_thread_thunk('c', thread_sys));

thread_sys.start_threads();
```

```
$ rhino coop-threads.js
in thread a ; counter = 10
in thread b ; counter = 9
in thread c ; counter = 8
in thread a ; counter = 7
in thread b ; counter = 6
in thread c ; counter = 5
in thread a ; counter = 4
in thread b ; counter = 3
in thread c ; counter = 2
in thread a ; counter = 1
in thread b ; counter = 0
```

CAP: EC: Continuacions

Exemple: Fils cooperatius (*cooperative multithreading*)

```
// fitxer coop-threads-fibonacci.js

print("\nFIBONACCI(9)");  // podríem parametritzar-ho
const TAG = "[NFIB]";

let fibs = [];
function make_fib_thunk(n, thread_system) {
    // ... plana següent
}

let fib_thread_sys = make_thread_system();
fib_thread_sys.spawn(make_fib_thunk(9, fib_thread_sys));
fib_thread_sys.start_threads();
print(TAG, fibs);
```

CAP: EC: Continuacions

Exemple: Fils cooperatius (*cooperative multithreading*)

```
function make_fib_thunk(n, thread_system) {
    function nFib() {
        if (n <= 1) {
            print(TAG, "Base case:");
            print("         Fibonacci(0) = 0");
            print("         Fibonacci(1) = 1");
            fibs[0] = 0;
            fibs[1] = 1;

        } else {
            print(TAG, "No previous Fibonacci values, spawn Fibonacci("
                + (n - 1) + ") thunk");
            thread_system.spawn(make_fib_thunk(n - 1, thread_system));
            while (fibs[n - 1] === undefined || fibs[n - 2] === undefined) {
                thread_system.relinquish();
            }
            fibs[n] = fibs[n - 1] + fibs[n - 2];
            print(TAG, "n =", n, "| Fibonacci(" + n + ") =", fibs[n]);

        }
    };
    return nFib;
}
```

CAP: EC: Continuacions

Exemple: Fils cooperatius (*cooperative multithreading*)

```
$ rhino coop-threads-fibonacci.js

FIBONACCI(9)
[NFIB] No previous Fibonacci values, spawn Fibonacci(8) thunk
[NFIB] No previous Fibonacci values, spawn Fibonacci(7) thunk
[NFIB] No previous Fibonacci values, spawn Fibonacci(6) thunk
[NFIB] No previous Fibonacci values, spawn Fibonacci(5) thunk
[NFIB] No previous Fibonacci values, spawn Fibonacci(4) thunk
[NFIB] No previous Fibonacci values, spawn Fibonacci(3) thunk
[NFIB] No previous Fibonacci values, spawn Fibonacci(2) thunk
[NFIB] No previous Fibonacci values, spawn Fibonacci(1) thunk
[NFIB] Base case:
         Fibonacci(0) = 0
         Fibonacci(1) = 1
[NFIB] n = 2 | Fibonacci(2) = 1
[NFIB] n = 3 | Fibonacci(3) = 2
[NFIB] n = 4 | Fibonacci(4) = 3
[NFIB] n = 5 | Fibonacci(5) = 5
[NFIB] n = 6 | Fibonacci(6) = 8
[NFIB] n = 7 | Fibonacci(7) = 13
[NFIB] n = 8 | Fibonacci(8) = 21
[NFIB] n = 9 | Fibonacci(9) = 34
[NFIB] 0,1,1,2,3,5,8,13,21,34
```

CAP: EC: Continuacions

Exemple: Fils cooperatius (*cooperative multithreading*)

La funció demanada senzillament crearà quatre *closures* que **capturen** l'*array* **`thread_queue`** (que farà el paper de cua) i una variable **`halt`** a la que se li assignarà (a **`start_threads`**) una funció per poder sortir del sistema *gracefully*.

Un cop creades les *closures*, retornarà un objecte amb quatre propietats (amb el mateix nom)

```
function make_thread_system() {

    let thread_queue = [];
    let halt         = false;

    function spawn(thunk) { //... };

    function relinquish() { //... };

    function quit() { //... };

    function start_threads() { //... };

    return {
        spawn: spawn,
        relinquish: relinquish,
        quit: quit,
        start_threads: start_threads

    };

}
```

CAP: EC: Continuacions

Exemple: Fils cooperatius (*cooperative multithreading*)

Aquí teniu el codi de les funcions **`spawn`** i **`quit`** (les explicarem a classe)

```
function spawn(thunk) {
   let cc = current_continuation();
   if (cc instanceof Continuation) {
       thread_queue.push(cc);
   } else {
       thunk();
       quit();
   }
};
```

```
function quit() {
   if (thread_queue.length > 0) {
       let next_thread = thread_queue.shift();
       next_thread('resume'); // resume
   } else {
       halt();
   }
};
```

CAP: EC: Continuacions

Exemple: Fils cooperatius (*cooperative multithreading*)

Aquí teniu el codi de la funció **`relinquish`** (l'explicarem a classe)

```
function relinquish() {
   let cc = current_continuation();
   if ((cc instanceof Continuation) && (thread_queue.length > 0)) {
         let next_thread = thread_queue.shift();
         thread_queue.push(cc);
         next_thread('resume'); // resume
   }
};
```

CAP: EC: Continuacions

Exemple: Fils cooperatius (*cooperative multithreading*)

Aquí teniu el codi de la funció **`start_threads`**. Fixeu-vos en el mecanisme d'acabament definit a **`start_threads`**: Quan executem **`start_threads`** per primer cop, **`cc`** és *truthy*, i per tant s'executa l'**`if`**, assignant a **`halt`** una funció que invoca la continuació **`cc`** amb **`false`** com a argument. Aquest cop, invocar **`cc`** tornarà a l'assignació a **`cc`**. Com que aquest tindrà el valor **`false`** no farem res més, i acabarem la funció **`start_threads`**, invocada inicialment.

```
function start_threads() {
   let cc = current_continuation();
   if (cc) {
       halt = function () { cc(false) };
       if (thread_queue.length > 0) {
          let next_thread = thread_queue.shift();
          next_thread('resume');
       }
   }
};
```

CAP: EC: CPS

Veiem, doncs, la potència de les continuacions per poder definir estructures de control diverses.

Ara bé, hem parlat de llenguatges de programació que ofereixen la possibilitat de *reificar* les continuacions amb **`callcc`** (Scheme, Standard ML, Ruby, etc.) o amb la funció **`Continuation`** en el cas de Javascript/Rhino.

I si el nostre llenguatge de programació no ofereix aquesta possibilitat?

La qüestió és que, si disposem de closures i funcions d'ordre superior, podem fer explícita la pila d'execució.

Aquesta tècnica s'anomena:

Continuation-Passing Style (CPS)

CAP: EC: CPS

Continuation-Passing Style (CPS)

Tail Position

A function call is in <u>tail position</u> if the following criteria are met:

- *The calling function is in **strict** mode.*
- *The calling function is either a normal function or an arrow function.*
- *The calling function is **not a generator** function.*
- ***The return value of the called function is returned by the calling function**.*

https://webkit.org/blog/6240/ecmascript-6-proper-tail-calls-in-webkit/

CAP: EC: CPS

Continuation-Passing Style (CPS)

Tail Call Optimization

When a function call is in tail position, ECMAScript 6 mandates that such a call must reuse the stack space of its own frame instead of pushing another frame onto the call stack. To emphasize, ***ECMAScript 6 requires that a call in tail position will reuse the caller's stack space****. The calling function's frame is called a tail deleted frame as it is no longer on the stack once it makes a tail call.*

https://webkit.org/blog/6240/ecmascript-6-proper-tail-calls-in-webkit/

CAP: EC: CPS

Continuation-Passing Style (CPS)

Tail Call Optimization

La *Tail Call Optimization* (TCO) permet optimitzar les crides a funcions que estan en *tail position* no creant un nou *stack frame*.

La manera en que s'invoca una funció NO importa (recordem que n'hi ha 4 maneres en Javascript). Només importa si està en *tail position*.

CAP: EC: CPS

Continuation-Passing Style (CPS)

Tail Call Optimization

```
function factorial(x) {
    if (x <= 1) {
        return 1;
    }
    return x * factorial(x-1);
}
```

```
function factorial (x, acc) {
    if (x <= 1) {
        return acc;
    }
    return factorial(x-1, x*acc);
}
```

CAP: EC: CPS

Continuation-Passing Style (CPS)

Tail Call Optimization

```
function foo() {
    bar();
}
```

és equivalent a:

```
function foo() {
    bar();
    return undefined;
}
```

```
function foo() {
    return bar();
}
```

CAP: EC: CPS

Continuation-Passing Style (CPS)

Tail Call Optimization

```
function forEach(arr, body, start) {
  if (0 <= start && start < arr.length) {
    body(arr[start], start, arr);
    return forEach(arr, body, start+1);
  }
}
```

CAP: EC: CPS

Continuation-Passing Style (CPS)

Tail Call Optimization

Cal tenir tail call optimization per poder programar en CPS

TCO a node.js (*només a la versió 6!*):

```
node --harmony --use_strict
```

TCO a Rhino (*versió 1.7.14, gener 2022*):

```
java -cp rhino1.7.14/lib/rhino-1.7.14.jar
org.mozilla.javascript.tools.shell.Main -opt -2
```

CAP: EC: CPS

Continuation-Passing Style (CPS)

Tail Call Optimization

```
function forEach(arr, body, start) {
  if (0 <= start && start < arr.length) {
    body(arr[start], start, arr);
    forEach(arr, body, start+1);
  }
}
```

```
$ node --harmony --use_strict          [Provat amb node versió 6.17.1]
> forEach([...Array(100000).keys()],function(elem, i) {console.log(i, elem)}, 0)
. . .
12332 12332
12333 12333
RangeError: Maximum call stack size exceeded
```

CAP: EC: CPS

Continuation-Passing Style (CPS)

Tail Call Optimization

```
function forEach(arr, body, start) {
  if (0 <= start && start < arr.length) {
    body(arr[start], start, arr);
    return forEach(arr, body, start+1);
  }
}
```

```
$ node --harmony --use_strict          [Provat amb node versió 6.17.1]
> forEach([...Array(100000).keys()],function(elem, i) {console.log(i, elem)}, 0)
. . .
99997 99997
99998 99998
99999 99999
undefined
```

CAP: EC: CPS

Continuation-Passing Style (CPS)

Tail Call Optimization

```
function forEach(arr, body, start) {
  if (0 <= start && start < arr.length) {
    body(arr[start], start, arr);
    return forEach(arr, body, start+1);
  }
}
```

```
$ node --harmony --use_strict          [Provat amb node versió 16.16.0]
> forEach([...Array(100000).keys()],function(elem, i) {console.log(i, elem)}, 0)
. . .
7959 7959
7960 7960
Uncaught RangeError: Maximum call stack size exceeded
```

CAP: EC: CPS

Continuation-Passing Style (CPS)

- **Prohibit utilitzar cap expressió en un *return*: Es retorna sempre el resultat d'una crida a funció (en *tail position*), o una constant.**
- **El darrer paràmetre d'una funció és sempre la seva continuació** (*en la versió més senzilla*)**.**
- **Cada funció ha d'acabar cridant la seva continuació amb el resultat del seu càlcul.**

CAP: EC: CPS

Continuation-Passing Style (CPS)

Exemple: La funció identitat:

```
function id(x) {
  return x;
}
```

```
function id_cps(x,ret) {
  return ret(x);
}
```

CAP: EC: CPS

Continuation-Passing Style (CPS)

Exemple: La funció factorial:

```
function fact(n) {
  if (n <= 1) {  return 1; }
  else { return n * fact(n-1) }
}
```

↓

```
function fact_cps(n, ret) {
  if (n <= 1) { return ret(1); }
  else { return fact_cps (n-1, function (v) {
                                 return ret(v*n)  })}}
```

CAP: EC: CPS

Continuation-Passing Style (CPS)

Exemple: La funció coeficient binomial:

```
function binomial_coef (n,k) {
  return  fact(n) /  (fact(k) * fact(n-k)) ;
}
```

↓

```
function binomial_coef_cps (n,k,ret) {
  return fact_cps (n, function (factn) {
    return fact_cps (n-k, function (factnk) {
      return fact_cps (k, function (factk) {
        return ret(factn / (factnk * factk)) })})})}
```

CAP: EC: CPS

Continuation-Passing Style (CPS)

Exemple: La funció n-èssim nombre de Fibonacci:

```
function fib(n) {
  if (n < 2) { return 1 }
  else { return fib(n-1) + fib(n-2) }
}
```

↓

```
function fib_cps(n, ret) {
  if (n < 2) { return ret(1); }
  else { return fib_cps(n-1, function(fibn1) {
            return fib_cps(n-2, function(fibn2) {
              return ret(fibn1 + fibn2)})})}}
```

CAP: EC: CPS

Continuation-Passing Style (CPS)

Exemple: La funció eliminar un element **e** d'un *array*:

```
function remove(arr,e) {
  if (arr.length == 0) {
    return [];
  } else {
     let [car,...cdr] = arr;
     let rem = remove(cdr,e)
     if (car != e) {
       rem.unshift(car)
     }
     return rem;
  }
}
```

CAP: EC: CPS

Continuation-Passing Style (CPS)

Exemple: La funció eliminar un element **e** d'un *array*:

```
function remove_cps(arr,e,ret) {
  if (arr.length === 0) {
    return ret([]);
  } else {
    let [car,...cdr] = arr;
    return remove_cps(cdr, e, function (rcdr) {
                      if (car != e) {
                          rcdr.unshift(car);
                      }
                      return ret(rcdr)})}}
```

CAP: EC: CPS

Continuation-Passing Style (CPS)

Exemple: La funció escriure els elements d'un *array*:

```
function escriuArray(arr) {
  for(let i=0; i < arr.length; i++) {
    console.log(arr[i]);
  }
  console.log("Done");
}
```

CAP: EC: CPS

Continuation-Passing Style (CPS)

Exemple: La funció escriure els elements d'un *array*:

```
function escriuArray(arr) {
    return forEachCps(arr,
                      function (elem, index, next) {
                         console.log(elem);
                         return next();
                      },
                      function () {
                        console.log("Done");
                      });
}
```

CAP: EC: CPS

Continuation-Passing Style (CPS)

on tenim la funció auxiliar:

```
function forEachCps(arr, visitor, done) {
  function forEachCpsRec(index, arr, visitor, done) {
    if (index < arr.length) {
      return visitor(arr[index],
                     index,
                     function () {
                        return forEachCpsRec(index+1, arr, visitor, done);
                     });
    } else {
      return done();
    }
  }

  return forEachCpsRec(0, arr, visitor, done)
}
```

CAP: EC: CPS

Continuation-Passing Style (CPS)

Aleshores... si no disposem de TCO, ***no podem fer-hi res?***

Estem condemnats a no guanyar res amb les crides a funció en tail position?

No necessàriament! Si disposem de closures i funcions d'ordre superior tenim la possibilitat de fer servir la tècnica del...

Trampolining

CAP: EC: *Trampolining*

Trampolining

La idea és amagar la crida en *tail position* en un *thunk*, i així executar-lo després de retornar de la crida prèvia, creant i destruint un nombre constant de *stack frames* per a cada crida, i per tant no fent crèixer la pila.

Aleshores fem servir el ***trampolí***:

```
function trampoline (fun) {
    while (typeof fun == 'function') {
        fun = fun();
    }
    return fun;
};
```

CAP: EC: *Trampolining*

Trampolining

És clar que cal transformar la funció original per adaptar-la a l'ús del trampolí. Per exemple, partim de la funció recursiva final:

```
function findIndex(arr, predicate, start = 0) {
    if (0 <= start && start < arr.length) {
        if (predicate(arr[start])) {
            return start;
        }
        return findIndex(arr, predicate, start+1); // tail call
    }
}
```

CAP: EC: *Trampolining*

Trampolining

Crearem una funció que construeixi una funció auxiliar que faci precisament això, retornar el *thunk* on s'*amaga* la crida recursiva...

```
function findIndexTrampoline (a, p, s=0) {
    function findIndexTR(arr, predicate, start = 0) {
        if (0 <= start && start < arr.length) {
            if (predicate(arr[start])) {
                return start;
            }
            return function () {  // funció que "amaga" la crida recursiva
                    return findIndexTR(arr, predicate, start+1);
                };

        }
    }

    return trampoline(findIndexTR(a,p,s));
}
```

CAP: EC: *Trampolining*

Trampolining

Per a funcions recursives finals prou senzilles, podem definir un ***esquema general***. Suposem que tenim una funció recursiva final **`f`**:

```
function f_trampoline (a1, a2, ..., aN) {
        function __f(aa1, aa2, ... , aaN) {
                if recursiveCase {
                        return function () {
                                        return __f(...);
                                };
                }

                return baseCase;
        };

        return trampoline(__f(a1, a2, ... , aN));
}
```

CAP: EC: *Trampolining*

Trampolining

Exercici: Apliqueu la técnica del trampolining per obtenir una versió de **`my_filter`** que no tingui problemes amb la mida de la pila

```
function my_filter (arr, f, res) {
   if (arr.length === 0) {
      return res
   } else {
      let [car, ...cdr] = arr
      if (f(car)) {
         res.push(car)
      }
      return my_filter(cdr,f,res)
   }
}
```

Sabem que si fem servir Node.js, que no fa TCO, tindrem problemes amb la pila. Si fem , per exemple, **`my_filter([...Array(10000).keys()], x => x % 2 === 0, [])`**, obtindrem un error **`RangeError: Maximum call stack size exceeded`**

CAP: EC: CPS

Continuation-Passing Style (CPS)

Imaginem que ***tot*** un programa està escrit en estil CPS.

Aleshores **callcc(f,c)** és molt fàcil d'implementar, ja que tenim la continuació explicitament en tot moment.

```
function callcc (f, cc) {
  return f( function (x, ret) { return cc(x) }, cc );
}
```

CAP: EC: CPS

Continuation-Passing Style (CPS)

Exemple d'ús de **`callcc(f,c)`** en CPS:

```
function fact_cps(n, ret) {
   if (n <= 1) {
      return callcc( function (cc, rt) {
                           kont = cc;
                           return rt(1);
                     }, ret);
   }
   else {
      return fact_cps (n-1, function (v) {
                              return ret(v*n)
                           })
   }
}
```

(on utilitzem la variable global **`kont`** i una funció **`function id(x) { console.log(x); return x; }`** com a continuació inicial)

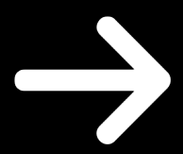
CAP: EC: Continuacions

Podem serialitzar i des-serialitzar continuacions:

```
function capture(filename) {
   var k = new Continuation();
   serialize(k, filename);
   java.lang.System.exit(0);
}

function foo(level) {
   var now = new java.util.Date();
   if(level > 5) {
       print("run the file foo.ser");
       capture("foo.ser");
   } else {
       print("next level");
       foo(level + 1);
   }
   print("restarted("+level+"): " + now)
}

foo(1);
```



```
next level
next level
next level
next level
next level
run the file foo.ser
```

CAP: EC: Continuacions

Podem serialitzar i des-serialitzar continuacions:

```
var k = deserialize("foo.ser");
k();
```

```
restarted(6): Wed Dec 06 12:05:23 CET 2017
restarted(5): Wed Dec 06 12:05:23 CET 2017
restarted(4): Wed Dec 06 12:05:23 CET 2017
restarted(3): Wed Dec 06 12:05:23 CET 2017
restarted(2): Wed Dec 06 12:05:23 CET 2017
restarted(1): Wed Dec 06 12:05:23 CET 2017
```