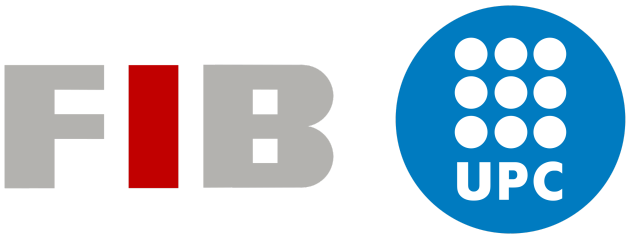
Pràctica CAP 22-23 - Q1:
Corutines en Rhino

Fet per Iván Risueño i Adam Lázaro



Índex

1. Introducció

En aquesta edició de CAP, la pràctica proposada ha estat la d'implementar una estructura de control, les corutines, amb una de les estructures de control que més hem treballat durant el curs i que és capaç d'implementar qualsevol altra estructura de control, sent aquesta les continuacions. No obstant, aquesta vegada, enlloc d'intentar-ho en Smalltalk, com que en aquesta edició s'ha començat per JavaScript, la pràctica també es farà per primera vegada en JavaScript.

Les corutines són una estructura de control que, a diferència de les subrutines normals i corrents, que quan són cridades, executen tot el codi que tenen dins, retornen i, si es tornen a cridar, el tornen a executar tot, aquestes, quan retornen, si tornen a ser cridades, poden continuar executant el codi des d'on ho van deixar l'última vegada que es va executar. A la *Figura 1* es troba la idea il·lustrada.

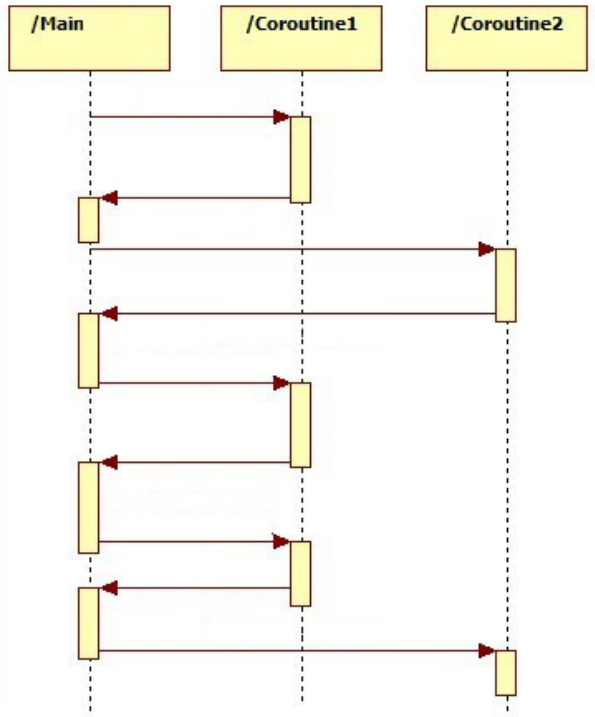


Figura 1: Diagrama de seqüència representant la idea de les corutines

La pràctica consisteix en crear una funció, *make_coroutine(func)*, per tal de crear una espècie d'objecte que representi la corutina i que pugui executar la funció designada, *func*, de la manera en la qual funciona una corutina, és a dir, que pugui continuar pel lloc on es va quedar l'últim cop que va ser cridada. Aquesta funció *func* té com a paràmetres *resume* i *value*. *resume* ha de ser una funció que, si es fa bé, ha de ser sempre la mateixa i que és la que ha de permetre cridar a altres corutines i que es torni en aquell punt del codi la pròxima vegada que es torni a aquella corutina, cosa que ho fa a través dels paràmetres *c*, representant la corutina a cridar i *v*, representant el valor amb el qual cridar-la. *value* també representa aquest valor, però especialment per la primera crida. El codi d'exemple donat és el mostrat a la *Figura 2*.

```
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
    }
}
```

Figura 2: Funció "exemple_senzill" per comprovar si les corutines funcionen

En els següents apartats es passarà a descriure, per cadascun dels apartats, quines han sigut les nostres peripècies per donar amb una solució mínimament satisfactòria, una descripció d'aquesta solució i una presentació dels pocs tests que hem fet per provar que veritablement funciona. Per últim, treurem unes conclusions sobre què hem après durant la realització de la pràctica.

2. Apartat A: Corutines

2.1. Procés d'aprenentatge

A l'hora de començar, teníem clar que la manera més intuïtiva i fàcil d'implementar les corutines, amb la manera presentada, era a través d'un objecte Coroutine que tingués com a propietats, com a mínim, *body*, on es guardaria el cos de la funció que ha d'actuar com una corutina, i *cont*, on es guardaria la continuació de l'estat actual de la corutina present. Tot i així, a l'hora d'intentar fer una primera versió d'aquesta manera, vam veure alguns problemes, sent el primer i el més obvi que *make_coroutine* havia de retornar una funció enlloc d'un objecte.

Una vegada que ens vam adonar d'això, vam maniobrar per fer que *make_coroutine* retornés una funció. Al final, se'ns va ocórrer retornar una funció anònima que acceptés com a paràmetre *v*, el valor, i que retornés la crida a *body* amb *resume* i *v* assignats com a paràmetres, tot això a través del context lèxic capturat. Desgraciadament, al intentar accedir a *body*, ho intentàvem a través de *this*, però aparentment, dins de la funció anònima representava l'objecte global i vam haver de

fer-ho d'una altra manera, que aquesta va ser la de crear una Coroutine dins de *make_coroutine* i accedir a la propietat mitjançant context lèxic.

Pel que fa a *resume*, les primeres versions estaven lluny de ser la solució que presentem, però com a mínim en idea sí s'aproximaven: guardar-se sempre a *cont* la continuació de l'estat actual amb *new Continuation()* i comprovar el paràmetre *cont* de la corutina a cridar, si era *null*, s'havia de cridar a la corutina directament, altrament s'havia de cridar a *cont*. També vam trobar-nos amb el problema que, per algun motiu, *c* no ho detectava com una corutina i, per tant, *c.cont* sempre era *undefined*. Depenent de a on definíem *resume*, també teníem problemes amb el *this* o, directament, en passar la funció *resume* a la funció *body*.

Com que estàvem encallats en aquest punt, vam decidir fer una mica més de recerca sobre corutines i vam trobar que l'utilització d'una propietat booleana per comprovar si era la primera vegada que s'entrava en una corutina era més fiable, vam afegir la propietat *firstTime* per comprovar-ho. També ens vam adonar que les continuacions no retornaven bé ja que, en quan s'invoca la continuació, s'ha de comprovar si *cont* continua sent instància de Continuation o no, per tant, vam canviar l'assignació de la nova continuació per una crida a la funció *current_continuation()* i vam afegir les comprovacions. A partir d'aquests canvis, *resume* tenia molt bona pinta (i ja era casi el mateix *resume* que presentem).

Tots els problemes amb les instàncies desapareixent quan se'n tractava una van acabar derivant en una primera versió de la solució en la qual es feia un petit canvi a l'exemple: enlloc de cridar directament a les variables declarades, cridaven a una propietat de l'objecte anomenada *caller*, que contenia la funció anònima que retornàvem, per tant, les crides quedaven amb *a.caller('*')*, per exemple. *resume* estava també dins de l'objecte Coroutine com a variable, no com a propietat, per aprofitar-se del context lèxic. Aquí vam comprovar que l'exemple ens funcionava i que, en efecte, el problema estava en com i a on declaràvem les coses, és a dir, era més problema de JavaScript que de les continuacions.

Per últim, per suposat vam decidir de buscar la manera de fer-ho amb funcions i, després d'una mica de recerca, vam trobar la peça final i que vam oblidar: les funcions també són objectes i, per tant, poden tenir propietats si es declaren bé. Aquesta peça d'informació va ser la clau per acabar obtenint, finalment, la solució que presentem.

2.2. Solució proposada

A continuació, anirem presentant la solució proposada per la implementació de les corutines amb les continuacions de Rhino, que es mostra en la *Figura 3*.

Primerament, tenim la funció *current_continuation()*, que només retorna una nova instància de Continuation, però ho fem d'aquesta manera per tal que, al cridar la continuació, retorni en el moment de fer l'assignació d'aquesta a la propietat *cont* de la corutina en la funció *resume* (que veurem més endavant).

Ara sí, passem a descriure *make_coroutine(func)*. Primerament, crea una nova funció objecte (anònima) que serà la que retornarà *make_coroutine*. Aquesta funció anònima *cor* accepta com a paràmetre el valor *v* (tal i com exemplifica el codi d'exemple, per poder-la cridar amb *a(v)*, per exemple) i retorna l'execució de la funció cos de la corutina *body* (recordem, és una propietat) amb *resume* i *v* com a paràmetres, sent *resume* agafada per ser una closure.

Després, creem les propietats que aquesta funció objecte *cor* tindrà, sent aquestes *body*, que conté el codi de la funció que ha d'actuar com a corutina, *firstTime*, que indica si és la primera vegada que es passa per aquella corutina i *cont*, que és on es guarda la continuació del moment on es troba de l'execució de *body*.

Següentment, declarem com a una variable la funció *resume*, amb paràmetres *c* i *v*, sent aquests la corutina a la qual cridar i el valor amb el qual se l'ha de cridar, respectivament. *resume* és on es troba gran part de la gràcia de la solució, per tant, ara en parlarem més.

- El primer que fa és cridar a *current_continuation* per tal de guardar-se una continuació del retorn de la crida a *current_continuation* a la propietat *cont*. Aquest serà el punt al qual es retornarà quan un altre *resume* cridi a aquesta corutina amb un valor determinat.
- El següent és una comprovació que, en efecte, la propietat *cont* d'aquesta corutina sigui una continuació, com just s'acaba d'assignar. Si ho és, significa que el que ha de fer és executar la següent corutina, indicada per *c*, altrament, significa que és *v* i, per tant, és el valor que ha de retornar *resume*.
- En cas que sí sigui una continuació, primer es marca que ja no és la primera vegada que es passa per aquesta corutina, indicant així la propietat *firstTime* a 0.
- Es comprova si la propietat *firstTime* de la corutina *c* és certa, és a dir, si és la primera vegada per la qual es passarà per corutina *c*. En cas que així sigui, s'ha de cridar directament a *c* amb *v* com a paràmetre. En cas contrari, a qui s'ha de cridar és a la propietat *cont* de *c* amb el valor *v* com a paràmetre. Això farà que es torni a la l'assignació de la continuació a *cont* de *c*, però enlloc de la continuació, hi haurà el valor *v*.

```
function make_coroutine(func) {
    //Construcció de la funció objecte cor(v)
    let cor = function(v) {
        return cor.body(resume,v);
    }
    //Creació de les propietats de la funció
    cor.body = func;
    cor.firstTime = 1;
    cor.cont = null;

    let resume = function(c, v) {
        cor.cont = current_continuation(); //Al cridar a cont, retorna en aquest punt
        if (cor.cont instanceof Continuation) {
            cor.firstTime = 0;
            if (c.firstTime) {
                c(v);
            }
            else c.cont(v);
        }
        else return cor.cont; //Només pot passar si cont és el valor
    }
    return cor;
}
```

Figura 3: Funció "make_coroutine" amb la solució proposada i amb alguns comentaris

2.3. Tests

Ara presentem alguns dels tests més rellevants que hem fet per comprovar que l'execució de les corutines és correcta, per ordre de complexitat.

Reflexió

Primerament, hem volgut comprovar si una corutina es podria cridar a si mateixa. És més una prova de fortalesa que d'utilitat, però que és un punt més que poden tenir aquestes corutines a prova de comportaments inesperats. Per a fer-ho, només ens ha calgut fer unes modificacions a la funció *exemple_senzill* per tal que alguns dels *resume* es cridi a si mateix. Així doncs, tenim aquestes úniques variables mostrades a la *Figura 4*, amb la segona línia de cadascuna sent una crida a *resume* de si mateixes. Efectivament, retorna a si mateix amb el valor indicat i, per tant, es pot concloure que la funció *reflexive* retorna l'esperat.

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

Figura 4: Variables modificades de l'exemple "exemple_senzill"

Seqüència de Fibonacci

El segon test que hem decidit implementar ha sigut un programa que calcula els nombres de la seqüència de Fibonacci. Concretament, aquest algorisme fa ús de dues corutines: una per a *Fib(n)* quan *n* és parell, i l'altra per a *Fib(n)* quan *n* és senar(l'input inicial és *Fib(0)* i *Fib(1)*, és a dir, 0 i 1 respectivament). Totes dues afegeixen els seus resultats a un array que va creixent amb els nombres, on $\forall i >= 0 \in array, array[i] = Fib(i)$.

```
function TestFibonacci() {
    print("Benvingut/da a la successió de Fibonacci!");
    print("A s'encarregarà de calcular fib(n) quan n sigui parell, i B ho farà quan n sigui senar.");
    print("Començant amb que fib(0) i fib(1) són 0 i 1 respectivament...");
    let a = make_coroutine( function(resume, value) {
        let currentFibValue;
        let newValues;
        for (let i = 2; i <= n; i += 2) {
            newValues = value;
            currentFibValue = newValues[i - 2] + newValues[i - 1];
            print("A: el resultat de fib(" + i + ") és: " + currentFibValue);
            newValues.push(currentFibValue);
            resume(b, newValues);
        }
    });
    let b = make_coroutine( function(resume, value) {
        let currentFibValue;
        let newValues;
        for (let i = 3; i <= n; i += 2) {
            newValues = value;
            currentFibValue = newValues[i - 2] + newValues[i - 1];
            print("B: el resultat de fib(" + i + ") és: " + currentFibValue);
            newValues.push(currentFibValue);
            resume(a, newValues);
        }
    });
    // amb aquest codi evitem "complicar-nos la vida" amb
```

Figura 5: Codi font del test TestFibonacci()

Ping-Pong

Per a finalitzar els tests de la nostra implementació de les corutines, hem escrit un codi que simula una partida de ping pong! La partida consta de dos jugadors, *A* i *B*, que són simulats per la màquina. També comptem amb un àrbitre, *C*, qui narra el partit i s'encarrega d'anunciar el marcador i qui guanya, perd, o si hi ha empat.

L'algorisme, el qual es pot veure a *Figura 6, Figura 7* i *Figura 8*, no és massa enrevessat. El partit comença amb *C* dient que treurà *A.* Per a cada torn, *A* i *B* es passen la pilota. Amb una probabilitat del 20%, un dels dos jugadors falla i resulta en un punt per a l'altre. El jugador que ha fallat, en comptes de tornar la bola per a seguir el joc, li dóna el control a *C* per a que anunciï el marcador en aquell moment i li doni la bola al jugador que ha obtingut el punt. Aquestes iteracions es repeteixen fins que acaba el joc a(de manera predeterminada) 50 torns. Quan s'ha acabat el partit, *C* anuncia el marcador i tot seguit qui ha guanyat, si és que no han empatat. Trobem que aquest és un bon test per a provar la correctesa de la nostra implementació de corutines, ja que l'execució va variant de manera indeterminada entre *A* i *B, A* i *C,* o *B* i *C.*

```
let a = make_coroutine( function(resume, value) {
    let r, newValues = value;
    while (tornActual < nTorns) {
        r = Math.floor(Math.random() * 100);
        if (r < failurePerc) {
            print("Torn [" + (tornActual<10 ? "0" : "") + tornActual + "] -> A falla! Punt per a B.");
            ++tornActual;
            newValues = resume(c, [newValues[0], newValues[1] + 1, 'b']);
        } else {
            print("Torn [" + (tornActual<10 ? "0" : "") + tornActual + "] -> A: ping!");
            ++tornActual;
            newValues = resume(b, newValues);
        }
    }
    resume(c, newValues);   // Si acabem aquí l'execució, C rebrà valors undefined i no podrem decidir qui ha guanyat
});
```

Figura 6: Funció que controla al jugador A

```
let b = make_coroutine( function(resume, value) {
    let r, newValues = value;
    while (tornActual < nTorns) {
        r = Math.floor(Math.random() * 100);
        if (r < failurePerc) {
            print("Torn [" + (tornActual<10 ? "0" : "") + tornActual + "] -> B falla! Punt per a A.");
            ++tornActual;
            newValues = resume(c, [newValues[0] + 1, newValues[1], 'a']);
        } else {
            print("Torn [" + (tornActual<10 ? "0" : "") + tornActual + "] -> B: pong!");
            ++tornActual;
            newValues = resume(a, newValues);
        }
    }
    resume(c, newValues);   // Si acabem aquí l'execució, C rebrà valors undefined i no podrem decidir qui ha guanyat
});
```

Figura 7: Funció que controla al jugador B

```
let c = make_coroutine( function(resume, value) {
    print("C: Comença el partit! Treurà A!");
    let newValues = value, guanyador;
    while (tornActual < nTorns) {
        newValues[2] == 'a' ? newValues = resume(a, newValues) : newValues = resume(b, newValues);
        print("C: El marcador és el següent: [" + newValues[0] + " - " + newValues[1] + "]");
        if (newValues[0] == newValues[1]) guanyador = "-";
        else if (newValues[0] < newValues[1]) guanyador = "B";
        else guanyador = "A";
    }
    print();
    print("C: Final del partit!");
    if (guanyador == "-") {
        print("C: Això sí que no m'ho esperava! Han empatat dues màquines al atzar!");
    } else print("C: El guanyador és " + guanyador + "!");
    print("C: Gràcies per jugar amb nosaltres! Quan vulguis jugar un altre cop, executan's!");
});
```

Figura 8: Funció que controla a l'àrbitre C

3. Apartat B: *Samefringe*

3.1. Procés d'aprenentatge

La primera tasca va ser entendre correctament el problema. Què se'ns donava, què se'ns demanava i com ens ho demanaven. El problema és el de *Same Fringe* o *Mateixa Frontera*. És un problema d'arbres binaris que consisteix en, donats dos arbres binaris, determinar si tenen les mateixes fulles llegides d'esquerra a dreta.

Sembla un problema simple. De fet ho és, però la qëustió era resoldre-ho fent ús de la nostra implementació de corutines. La primera versió que vam pensar tractava de fer ús de dues corutines, *a* i *b*. *a* era l'encarregada de buscar una fulla i, al trobar-la, resumir l'execució de *b* per tal de que aquesta trobi la seva primera fulla, se la passi a *a* i es comparin. Aquesta solució semblava simple, però a nivell algorísmic no ho era, ja que la corutina *a* feia massa feina. Amb tot plegat, la vam descartar.

Per últim, vam pensar en la proposta definitiva.

3.2. Solució proposada

Després de descartar la primera solució, vam arribar a la conclusió de que la feina "extra" que volíem que faci *a* la podia fer una corutina *c*. Així doncs, la nostra solució final consta de tres subrutines: *a, b* i *c.*

La crida comença a la corutina *c* amb dos paràmetres, els arbres binaris que es volen comparar. D'entrada es comprova que aquests arbres siguin correctes sintàcticament per evitar entrades errònies que desemboquin en comportaments estranys del programa(la qual cosa ens va ser molt útil per a afegir els nostres propis arbres). Els dos arbres de l'entrada s'envien a les corutines *a* i *b*, respectivament. Tot plegat, *c* és l'encarregada d'agafar valors d'*a* i de *b* i comparar-los.

L'algorisme és molt senzill: mentre que no hàgim arribat al final de cap dels dos arbres, resumim l'execució d'*a* i de *b* (només una en un cert instant de temps, evidentment) fins que cadascuna trobi una fulla. Es comprova que els resultats retornats siguin efectivament dues fulles i que siguin iguals. Si ho són, es continua l'execució iterant exactament igual, i si no ho són, es retorna un codi que representa que els valors d'entrada no tenen la mateixa frontera. Si es surt del bucle, també comprovem que l'últim valor que ens retornen *a* i *b* són del mateix tipus(per a contemplar els casos on una corutina retorna la seva última fulla i l'altra ja no té més fulles, motiu pel qual retorna *undefined*). Si no ho són, es retorna el mateix codi esmentat a dalt.

Les corutines *a* i *b* funcionen idènticament. Si la seva entrada és buida, es retorna. Si, en canvi, l'entrada es tracta d'una fulla, s'envia a *c* i s'atura l'execució. Altrament, es recorren recursivament els fills esquerre i dret de l'entrada fins a acabar el recorregut de l'arbre.

```
function correctBinTree(BinTree) {
    if (BinTree.length == 0 || (BinTree.length == 1 && typeof BinTree[0] == 'number')) return true;
    else if (BinTree.length == 2) return correctBinTree(BinTree[0]) && correctBinTree(BinTree[1]);
    return false;
}

function same_fringe(tree1, tree2) {
    let c1 = correctBinTree(tree1), c2 = correctBinTree(tree2);
    if (!c1) print("The first tree provided is not a valid tree!");
    if (!c2) print("The second tree provided is not a valid tree!");
    if (!c1 || !c2) return;

    let a = make_coroutine( function(resume, value) {
        if (value.length == 0) return;
        else if (value.length == 1) {
            //print("A: li passo a c la fulla " + value[0]);
            resume(c, value[0]);
        } else {
            this(value[0]);
            this(value[1]);
        }
    });
    let b = make_coroutine( function(resume, value) {
        if (value.length == 0) return;
        else if (value.length == 1) {
            //print("B: li passo a c la fulla " + value[0]);
            resume(c, value[0]);
        } else {
            this(value[0]);
            this(value[1]);
        }
    });
    let c = make_coroutine( function(resume, value) {
       let t1 = 0, t2 = 0;
       while(typeof t1 != 'undefined' && typeof t2 != 'undefined') {    // Comprovem que no hàgim arribat al final
        t1 = resume(a, value[0]);
        t2 = resume(b, value[1]);
        if ((typeof t1 == 'number' && typeof t2 == 'number' && t1 != t2)) return 0;
       }
       if (typeof t1 != typeof t2) return 0;    // Comprovem que tinguin el mateix nombre de fulles, i.e. no hi ha cap pare
       return 1;
    });

    if (typeof(a) === 'function') {
        c([tree1, tree2]) ? print("Both trees have the same fringe!") : print("They doesn't have the same fringe :'(");
    }
}
```

Figura 9: Algorisme resolutiu del problema Same Fringe fent ús de la nostra implementació de corutines

3.3. Tests

Per provar el nostre algorisme em fet ús del *test case* que se'ns suggereix a l'enunciat, i hem afegit 6 nous arbres per fer proves. Els arbres en qüestió són:

```
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] ] ] ];
let a5 = [ [ [ [], [8] ], [ [7], [6] ] ], [ [ [5], [4] ], [ [3], [2] ] ] ];
let a6 = [ [ [ [8], [7] ], [ [6], [ [5], [4] ] ] ], [ [ [3], [] ], [ [2], [] ] ] ];
let a7 = [ [ [ [7], [8] ], [ [1], [ [], [] ] ] ], [ [ [], [] ], [ [], [] ] ] ];
let a8 = [ [ [ [], [] ], [ [], [ [], [] ] ] ], [ [ [], [7] ], [ [8], [1] ] ] ];
let a9 = [ [ [ [0], [] ], [ [], [ [], [] ] ] ], [ [ [], [] ], [ [], [] ] ] ];
let a10 = [ [ [ [], [] ], [ [], [ [0], [] ] ] ], [ [ [], [] ], [ [], [] ] ] ];
```

Figura 10: Arbres binaris utilitzats per a fer les proves de la nostra solució

Els arbres *a5, a6, a7* i *a8* són casos que considerem comuns, i els arbres *a9* i *a10* són casos extrems, ja que tenen només una sola fulla.

Amb aquests arbres podem veure clarament que les que tenen la mateixa frontera són les parelles *(a1, a2)*, *(a5, a6)*, *(a7, a8), (a9, a10)*. El nostre programa hauria d'indicar que aquestes parelles tenen la mateixa frontera, i les altres que hem provat (*(a1, a4)*, *(a4, a2)*, *(a3, a4)*) no.

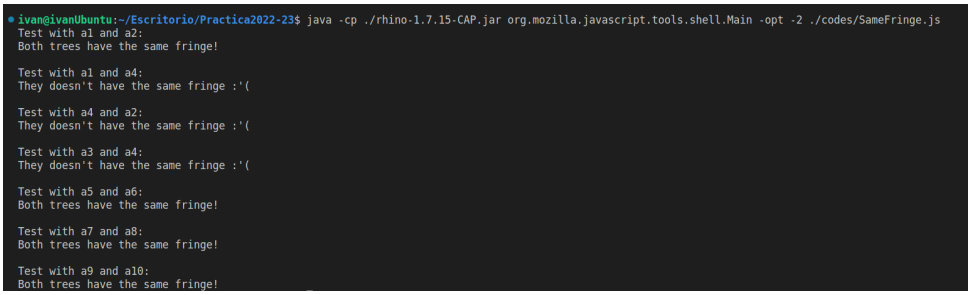


Figura 11: Sortida del test per a la solució de Same Fringe amb la nostra implementació de corutines

Com podem veure a la *Figura 11*, efectivament, la sortida del programa és l'esperada.

4. Conclusions

En aquest últim apartat farem un breu resum del que s'ha vist, explicarem què és el que hem après de fer aquesta pràctica i quines han estat les nostres sensacions al fer-la i després de fer-la.

Per una banda, hem vist com en la nostra solució, segons les condicions que hem posat a la funció de *resume*, les corutines s'anaven cridant les unes a les altres a través de les continuacions que tenien guardades i com aquestes, a mesura que eren cridades, s'anaven substituint pel valor enviat. Per comprovar-ho, hem vist tot un seguit de tests, des d'un simple que només provava la fortalesa de la solució, fins a un programa complet per jugar a ping-pong en un màquina contra màquina, sent el valor un array i provant també que no només serveix per a tipus bàsics.

Per l'altra banda, hem vist com, amb la nostra solució de les corutines i amb un algorisme relativament senzill, es pot aconseguir resoldre el problema de *SameFringe* amb una relativa simplesa, una prova més que la solució és mínimament satisfactòria. A més a més, també hem provat altres arbres, alguns essent casos extrems, com arbres on només hi ha fulles amb valor a un costat o arbres d'una sola fulla amb valor.

Durant la realització de la pràctica, podem dir que hem après bastant sobre el funcionament de les continuacions i el seu potencial per poder replicar altres estructures de control i, a més, poder navegar gairebé lliurement pel codi, utilitzant així les continuacions com si fossin instruccions *goto*. No només pel tema de les continuacions, sinó que també pel tema de les corutines també ha anat bé ja que, a través de les diferents execucions de prova en anteriors versions (i en l'actual) d'*exemple_senzill* i a través de la realització dels altres tests, com el del ping-pong, hem pogut comprendre més el seu funcionament i també notar

la utilitat i practicitat d'aquesta estructura de control i la facilitat amb la qual es poden obtenir resultats a l'hora de solucionar problemes que, d'altra manera, serien més complexos de resoldre.

Hem de reconèixer que hi ha hagut moments mentres fèiem la pràctica que ens sentíem una mica perduts ja que coses que ens pensàvem que sabíem o que crèiem que haurien de funcionar no eren així (encara que al final, sembla que el problema era més de JavaScript que de les continuacions o que del propi concepte de la corutina), però mitjançant la recerca i el nostre raonament crític, creiem que hem pogut entregar una solució mínimament satisfactòria i, tal i com ja hem dit, una amb la qual hem pogut aprendre uns quants coneixements.