Recursión generativa y funciones acumulativas

Fundamentos de programación

Recursión como la conocemos hasta ahora

Factorial

```
//contrato factorial:
number->number
//propósito: calcula el factorial
de un número
//prototipo function factorial(n){}
//ejemplo factorial(1)=1
//ejemplo factorial(0)=1
//ejemplo factorial(5)=120
//ejemplo factorial(10)=3628800
function factorial(n){
 if(n<=1) return 1;
 else return n*factorial(n-1);
}

console.log (factorial(7))
```

Fibonacci

```
Contrato: fibonacci: number->number
Propósito: Hallar una fórmula, para calcular los
numero fibonacci.
function fibonacci (numero) {Cuerpo de la función}
fibonacci (2)->1
fibonacci (5)->5
*/
function fibonacci(numero){
 if(numero==0) return 1;
 else if(numero ==1) return 1;
 else return (fibonacci (numero-1))+(fibonacci
(numero-2));
 }
console.log (fibonacci (2))
```

Recursión como la conocemos hasta ahora

Sumar una Lista

```
function sum(l){
    if (empty(l)) {
        return 0;
    } else {
        return car(l) + sum(cdr(l));
    }
}
```

```
sum([1,2,3,4])
1 + sum([2,3,4])
1 + ( 2 + sum([3,4]) )
1 + ( 2 + ( 3 + sum([4]) ))
1 + ( 2 + ( 3 + ( 4 + sum([]) )))
1 + ( 2 + ( 3 + ( 4 + 0 ) ))
1 + ( 2 + ( 3 + 4 ) )
1 + ( 2 + 7 )
1 + 9
10
```

Sumar una Lista con functional-light

```
const {cons, append, first, rest, isEmpty, isList,
length, deepCopy} = require('functional-light');
// Suma los elementos de una lista
// @param {Array} number
// @returns {number}
// @example sum([1,2,3,4,5])=>15

function sum(list){
if (isEmpty(list)) {
        return 0;
    } else {
        return first(list) + sum(rest(list));
    }
}
sum([1,2,3,4,5])
```

Recursión estructural

La recursión estructural está guiada por los datos, como en el caso de la suma de una lista de nùmeros

```
const {cons, append, first, rest, isEmpty, isList,
length, deepCopy} = require('functional-light');
// Suma los elementos de una lista
// @param {Array} number
// @returns {number}
// @example sum([1,2,3,4,5])=>15

function sum(list){
if (isEmpty(list)) {
        return 0;
    } else {
        return first(list) + sum(rest(list));
    }
}
```

```
sum([1,2,3,4])
1 + sum([2,3,4])
1 + ( 2 + sum([3,4]) )
1 + ( 2 + ( 3 + sum([4]) ))
1 + ( 2 + ( 3 + ( 4 + sum([]) )))
1 + ( 2 + ( 3 + ( 4 + 0 ) ))
1 + ( 2 + ( 3 + 4 ) )
1 + ( 2 + 7 )
1 + 9
10
```

Recursión generativa.

- Un problema puede ser visto como una instancia de una gran clase de problemas y un algoritmo aplica para todos éstos problemas.
- Un algoritmo divide un problema en otros más pequeños y los resuelve.
- Por ejemplo, un algoritmo para planear un viaje de vacaciones requiere solucionar pequeños viajes, así el viaje de nuestro hogar al aeropuerto, un vuelo que nos lleve a un aeropuerto cercano a nuestro lugar de vacaciones, un recorrido de este aeropuerto hasta nuestro punto de estancia. El problema entero se resuelve combinando las soluciones para cada uno de estos problemas.

Recursión generativa.

- Diseñar un algoritmo recursivo distingue dos clases de problemas: los que son trivialmente solucionables y los que no lo son.
- Si un problema dado es trivialmente solucionable, un algoritmo produce la solución correspondiente. Por ejemplo, el problema de ir de nuestro hogar a un aeropuerto proximo, puede ser trivialmente solucionable. Podemos ir en carro, tomar un taxi, o pedirle a un amigo que nos lleve. Si no, el algoritmo genera un nuevo problema màs pequeño del mismo tipo, por ejemplo ir de nuestro hogar a la estación de MIO.

Recursión generativa.

Un viaje con varias escalas es un ejemplo de un problema que no es trivial y pueda ser solucionado generando nuevos problemas más pequeños.

Se llama recursión generativa dado que en alguna circunstancia computacional uno de los problemas más pequeños pertenece a menudo a la misma clase de problemas que el original.

Sabemos que este proceso (recursión generativa) es mucho más una actividad ad hoc que el diseño impulsado-por-datos.

Ordenamiento rápido o Quiksort

https://repl.it/@VictorBucheli/recurividadGenerativa

La función consume una lista de números y produce una lista que contiene los mismos números en orden ascendente.

La diferencia entre la función ordena y ordena-rapido es que ordena se basa en recursión estructural o recursión a partir de los datos y ordena-rápido se basa en recursión generativa.

Ordenamiento estructural [lista]

https://repl.it/@VictorBucheli/recurividadGenerativa

```
// inserta un elemento a la lista de acuerdo a si el nùmero es mayor que el primero de la lista
// @param {number, Array} array
// @returns {array}
// @example insert(4,[])->[4]
// @example insert(7,[5,6])->[7,5,6]
// @example insert(1,[2,3])->[2,3,1]
function insert(n, list) {
   if (isEmpty(list)) {
       return cons(n, []);
   }
   else {
       if (n >= first(list)) {
           return cons(n, list);
       }
       else {
           return cons(first(list), insert(n, rest(list)));
       }
   }
}
```

Ordenamiento estructural [lista]

https://repl.it/@VictorBucheli/recurividadGenerativa

```
// ordena una lista de mayor a menor
// @param {Array} array
// @returns {array}
// @example sortRecursionStructural([4,7,5,6])->[7,6,5,4]
// @example sortRecursionStructural([1,2,3])->[3,2,1]

function sortRecursionStructural(list) {

   if (isEmpty(list)) {
       return [];
   }
   else {
       return insert(first(list), sortRecursionStructural(rest(list)));
   }
}

sortRecursionStructural([1,3,4,5,5,5,10,6,7,9])
```

Ordenamiento rápido o Quiksort

https://repl.it/@VictorBucheli/recurividadGenerativa

La idea subyacente del paso generativo es una estrategia para resolver problemas: divide y vencerás.

Dividimos los casos no-triviales del problema en dos más pequeños, relacionando problemas, resolviendo esos problemas más pequeños, y combinando sus soluciones en una solución para el problema original.

Quicksort

En el caso de ordena-rápido, el objetivo intermedio es dividir la lista de números en dos listas:

1. una de ellas contiene todos los ítems estrictamente más pequeños que el primer ítem.
2. la otra contiene todos los ítems que son estrictamente más grandes que el primer ítem.

Después las dos listas más pequeñas son ordenadas usando el mismo procedimiento.

Una vez que las dos listas se ordenan, simplemente se yuxtaponen las piezas: de la forma menor, primero, mayor.

El programa termina en su caso màs trivial, este es ordenar una lista vacía.

Quicksort

(list 11 8 14 7)

El ítem primero es 11

1a. (list 8 7)

2a. (list 14)

con 1. (list 8 7) se repite la operación, El ítem primero es 8

1b. (list 7)

1b. (list 8)

2a. ya está ordenada OK

concatenar ((list 7 8) , 11 , (list 14))-***>(list 7,8,11,14)***

Tarea QuickSort

Simular los pasos de ordena-rápido para (list 11 9 2 18 12 14 4 1).

Primer elemento **11**

1a. (list 9,2,4,1) **9** (241) **2** 1 4

2a.(list 18,12,14)**18** (**12** 14)

.

.

.

QuickSort.JS

```
// ordena una lista de menor a mayor
// @param {Array} array
// @returns {array}
// @example quicksort([4,7,5,6])->[4,5,6,7]
// @example quicksort([1,2,3])->[1,2,3]
function quicksort(list) {
   if (isEmpty(list)) {
       return [];
   }
   else {
       return append(quicksort(intemsmenores(list, first(list))),
           append([first(list)], quicksort(itemsmayores(list, first(list)))));
   }

}
```

QuickSort.JS

```
// ordena una lista de acuerdo a un parámetro pivote, o primer elemento de la lista
// @param  {Array} ,number
// @returns {array}
// @example intemsmenores([4,7,5,6],4)->[]
// @example intemsmenores([3,2,1],3)->[2,1]
function intemsmenores(list, pivote) {
   if (isEmpty(list)) {
       return [];
   }
   else {
       if (first(list) < pivote) {
           return cons(first(list), intemsmenores(rest(list), pivote));
       } else {
           return intemsmenores(rest(list), pivote);
       }
   }
}
```

QuickSort.JS

```
// ordena una lista de acuerdo a un paerametro pivote, o primer elemento de la lista
// @param  {Array} ,number
// @returns {array}
// @example itemsmayores([4,7,5,6],4)->[7,5,6]
// @example itemsmayores([3,2,1],3)->[]
function itemsmayores(list, pivote) {
   if (isEmpty(list)) {
       return [];
   }
   else {
       if (first(list) > pivote) {
           return cons(first(list), itemsmayores(rest(list), pivote));
       } else {
           return itemsmayores(rest(list), pivote);
       }
   }
}
```

Tarea QuickSort

1. Si la entrada para ordena-rápido contiene el mismo número varias veces, el algoritmo regresa una lista que es siempre más corta que la de entrada. ¿Por qué? Resuelva el problema para que la salida sea de la misma longitud que la entrada.

2. Utilice la función filter para definir ítems-menores e ítems-mayores.

Máximo común divisor (MCD)

Recursión estructural vs recursión generativa

El problema de encontrar el máximo común divisor de dos números naturales positivos. Todos estos números tienen al menos un divisor en común: 1.

El máximo común divisor (MCD) de dos o más números enteros, es el mayor número entero que los divide sin dejar residuo alguno.

MCD(6, 25) : 6 es divisible por ***1***, 2, 3, y 6 - 25 es divisible por ***1***, 5, y 25

MCD (18,24): 18 es divisible por 1, 2, 3, ***6***, 9, y 18; 24 es divisible por 1, 2, 3, 4, ***6***, 8, 12, y 24.

Máximo común divisor (MCD)

Recursión estructural vs recursión generativa

Ejercicio para resolver por recursión estructural y generativa. Explique cada uno de los algoritmos así como los resultados obtenidos, y la diferencia primordial entre la recursión estructural y generativa

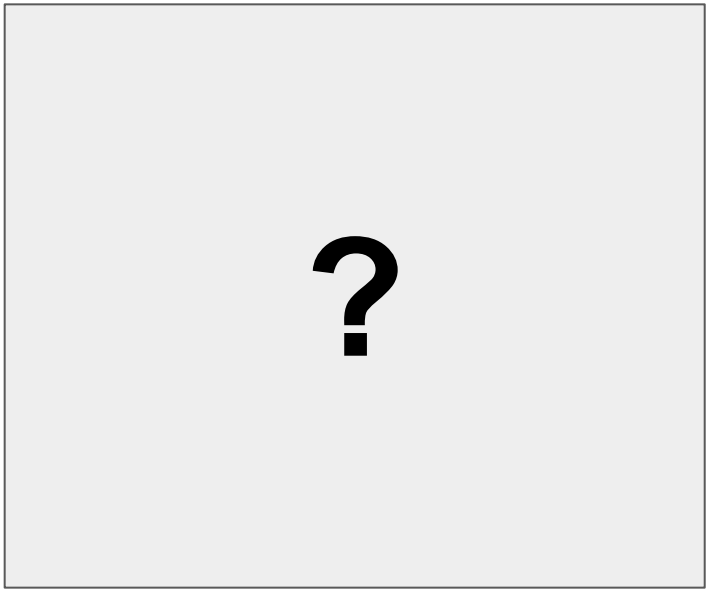
Tomar como base la explicación del libro:
https://htdp.org/2018-01-06/Book/part_five.html

mcd-estructural(18,24)
***mcd-estructural* (101135853, 45014640)**
***mcd-generativa* (101135853, 45014640)**
mcd-estructural
(101135743298743298749878539393935354874874387,499849898432434098432098432098845984598459845984598501464083838 38)
***mcd-generativa*(101135743298743298749878539393935354874874387,4998498984324340984320984320984320988459845984598459845985014640838383 8)**

MCD Estructural

```
function mcdStructural(n, m) {
    function primerdivisor(i) {
        if (i == 1) {
            return 1;
        }
        else {
            if (n % i == 0 && m % i == 0) {
                return i;
            }
            else {
                return primerdivisor(i - 1);
            }
        }
    }
    return primerdivisor(Math.min(n, m));
}
```

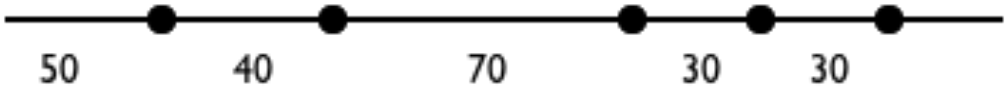
MCD Generativa



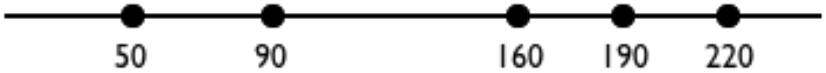
Pérdida de memoria

Se refieren a la pérdida de conocimiento que ocurre durante una evaluación recursiva. En la subsección se muestra cómo esta pérdida hace una función estructural recursiva más complicada y menos eficiente.

Se deben computar las distancias absolutas desde el origen. por ejemplo, puede ser una línea como ésta:



Lo que se requiere está en la figura siguiente, donde cada punto se anota con la distancia al punto extremo izquierdo:



https://repl.it/@VictorBucheli/ACCUM

Funciones recursivas y acumuladores

El tiempo de ejecución crece en la medida que el tamaño de la lista crece?

```
let dd = ((new Array(1000))).fill(2).map(x => {return Math.round(Math.random()* 1000)});
relativaAbsoluta(dd)
```

```
//agregatodo : número (listade número)
-> (listade número)
// para agregar n a cada número en uldn
//agregatodo(2,[1,2,3])->[ 3, 4, 5 ]
function agregatodo(n, list){
 if (isEmpty(list)) {
       return [];
   }
   else{
      return cons(n + first(list),
agregatodo(n, rest(list)));
   }
}
```

```
// relativaAbsoluta : (listade número) -> (listade número)
// para convertir una lista de distancias relativas a una lista de
distancias absolutas
// el primer ítem en la lista representa la distancia al origen
//relativaAbsoluta([50, 40, 70, 30, 30])->[ 50, 90, 160, 190, 220 ]
function relativaAbsoluta(list){
if (isEmpty(list)) {
       return [];
   }
   else{
     return
cons(first(list),agregatodo(first(list),relativaAbsoluta(rest(list))));
   }
}
```

Funciones recursivas y acumuladores

El tiempo de ejecución crece en la medida que el tamaño de la lista crece?

```
let dd = ((new Array(1000))).fill(2).map(x => {return Math.round(Math.random()* 1000)});
relativaAbsolutaAccum(dd,0)


   //relativaAbsolutaAccum : (listade número) -> (listade número)
   // para convertir una lista de distancias relativas a una lista de distancias
   //absolutas
   // el primer ítem en la lista representa la distancia al origen
   //relativaAbsolutaAccum([50, 40, 70, 30, 30],0)->[ 50, 90, 160, 190, 220 ]
   function relativaAbsolutaAccum(list, accum){
    if (isEmpty(list)) {
          return [];
      }
    else{
      return cons(first(list)+accum,relativaAbsolutaAccum(rest(list),first(list)+accum));
    }
   }
```

La función sumar lista como acumulador

```
const {cons, append, first, rest, isEmpty, isList,
length, deepCopy} = require('functional-light');
// Suma los elementos de una lista
// @param {Array} number
// @returns {number}
// @example sum([1,2,3,4,5])=>15

function sum(list){
if (isEmpty(list)) {
        return 0;
    } else {
        return first(list) + sum(rest(list));
    }
}
```

```
sum([1,2,3,4])
1 + sum([2,3,4])
1 + ( 2 + sum([3,4]) )
1 + ( 2 + ( 3 + sum([4]) ))
1 + ( 2 + ( 3 + ( 4 + sum([]) )))
1 + ( 2 + ( 3 + ( 4 + 0 ) ))
1 + ( 2 + ( 3 + 4 ) )
1 + ( 2 + 7 )
1 + 9
10
```

La función sumar lista como acumulador

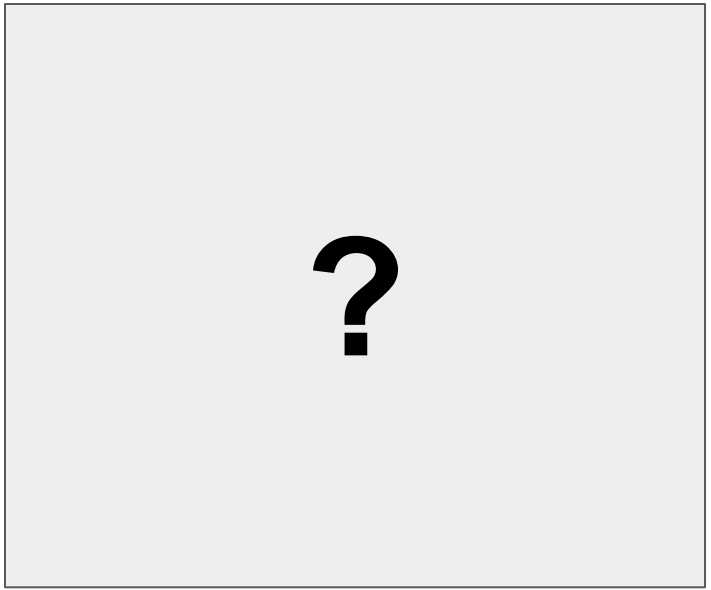
```
// sumacum los elementos de una lista como función
acumulación
// @param {Array,number} ->number
// @returns {number}
// @example sumacum([1,2,3,4,5],0)=>15
function sumacum(list,accum){
   if (isEmpty(list)) {
       return accum;
   }
   else{
    return sumacum (rest(list),accum+first(list));
   }
}
```

```
sum([1,2,3,4],0)
sum([2,3,4],1)
sum([3,4],3)
sum([4],6)
10
```

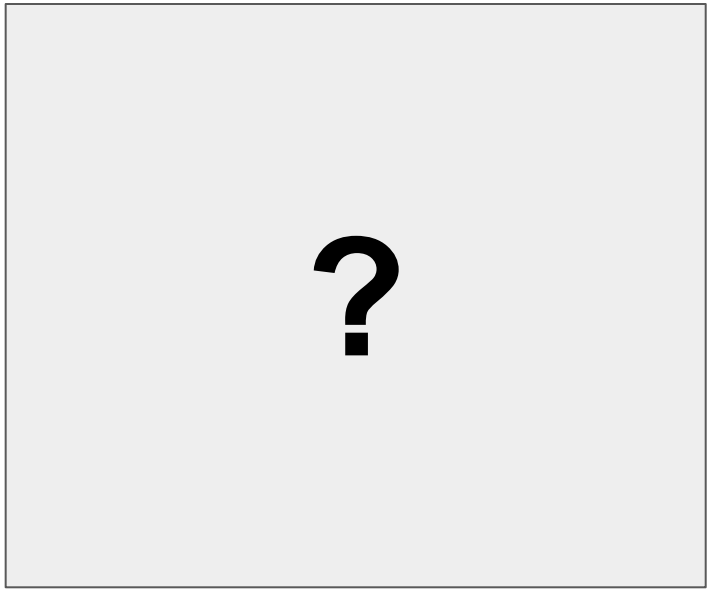
La función factorial como acumulador

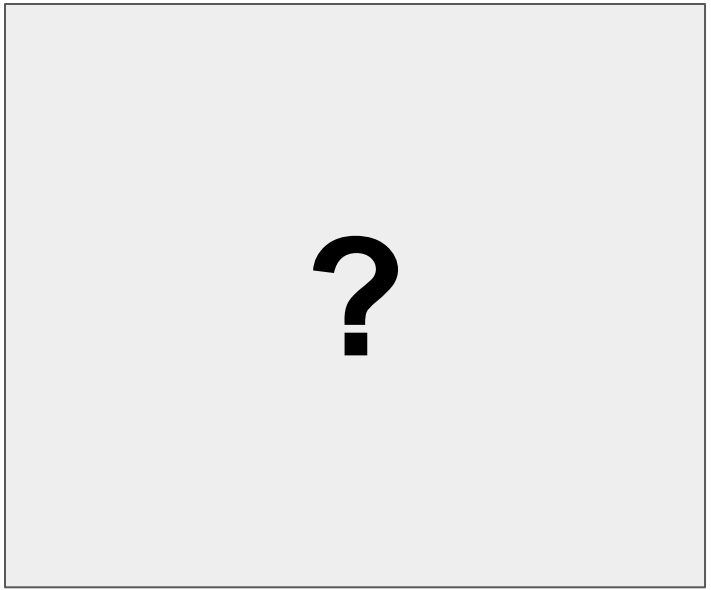
```
//contrato factorial: number->number
//propósito: calcula el factorial de un número
//prototipo function factorial(n){}
//ejemplo factorial(1)=1
//ejemplo factorial(0)=1
//ejemplo factorial(5)=120
//ejemplo factorial(10)=3628800
function factorial(n){
 if(n<=1) return 1;
 else return n*factorial(n-1);
}

console.log (factorial(7))
```



La función factorial como acumulador





La función producto de una lista como acumulador

Desarrollar una versión estilo-acumulador de producto, la función que computa el producto de una lista de números. Mostrar las etapas que expliquen lo que representa el acumulador.

Gracias