Tarea Independiente 21/08/2025

David Núñez Franco

August 24, 2025

Inventario de Conceptos Claves

- Evaluar: recorrer AST siguiendo estrategias
- Estrategias: en que orden se recorre el AST. Qué tan profundo bajar
  - Eager (aplicativa estricta): Full post-order
  - Lazy (por demanda, orden normal): Post-order solo hasta donde sea necesaria (ventajas/desventajas)
- Reducciones β y α: conceptual versus implementación (ingeniería) usando clausuras
- State: memoria a la que el código tiene acceso (Alias Scope, Context)
- Clausura: objeto que encapsula lambda + memoria local con acceso a la clausura padre y sirve como memoria de una lambda. NOTA: LO DESARROLLAREMOS LUEGO
- RESUMEN DE CLAUSURAS EN CUADERNO
- Lenguaje: faceta declarativa vs operativa
- FP: Programar en niveles altos de pirámide
- Historia:
  - Ensamblador: imperativa no estructurada ¡ 70
  - C/PASCAL: imperativa estructurada 70-80
  - C++: imperativa + OOP 90-actual
  - Programación multi-core (MCP) y multi-threaded (MTP)
- FP: Principio de inmutabilidad (Referencia Transparencial). No hay efectos secundarios
- FP Y PMT, PMC

Ejercicio 1

Vea este código, ejecute y explique las salidas. Este ejercicio tiene que ver con estrategias de evaluación. Nota: El caso Times/fibo dura un rato pues calcula fibonacci recursivamente que es MUY ineficiente.

```
        console.log("*** And ***")
        const foo = (msg, value) => (console.log(msg), value)

        const fibo_start = n =>  foo(`fibo(${n})`, fibo(n))
        const fibo = n => n <= 1 ? 1 : fibo(n - 1) + fibo(n - 2)

        console.log("Caso 1:", foo("A", true) && foo("B", false) )
        console.log("Caso 2:", foo("A", false) && foo("B", false) )

        console.log("\n*** Or ***")
        console.log("Caso 1:", foo("A", true) || foo("B", false) )
        console.log("Caso 2:", foo("A", false) || foo("B", false) )

        console.log("\n*** Times/fibo ***")
        console.log("Caso 1:", foo("A", fibo(10)) * foo("B", 0) )
        console.log("Caso 2:", foo("A", 0),  foo("B", fibo(10)) )

        console.log("\n*** Ternary (Warning!! it can take a big
           whiiile. Explain why) ***")
        console.log("Caso 1:", foo("A", fibo_start(50) < 0) ? foo("B
           ", fibo_start(100)) : foo("C", 0) )
```

¿Qué conclusiones se obtienen sobre tipos de estrategias de evaluación?

Solución

```
        console.log("*** And ***");

        const foo = (msg, value) => (console.log(msg), value);

        const fibo_start = (n) => foo(`fibo(${n})`, fibo(n));
        const fibo = (n) => (n <= 1 ? 1 : fibo(n - 1) + fibo(n - 2))
           ;

        // AND (&&) usa corto-circuito (estrategia lazy): si el
           primero es falso ya no evalua el segundo.
        console.log(
        "Caso 1:",
        foo("A", true) && // imprime "A", retorna true
        foo("B", false) // evalua porque el primero fue true,
           imprime "B", retorna false
        ); // Resultado final: false
```

```
14
15          console.log(
16          "Caso 2:",
17          foo("A", false) && // imprime "A", retorna false
18          foo("B", false) // NO se evalua por corto-circuito (
               estrategia lazy)
19          ); // Resultado final: false
20
21          console.log("\n*** Or ***");
22
23          // OR (||) tambien con corto-circuito (lazy): si el primero
               es true ya no evalua el segundo.
24          console.log(
25          "Caso 1:",
26          foo("A", true) || // imprime "A", retorna true
27          foo("B", false) // NO se evalua
28          ); // Resultado final: true
29
30          console.log(
31          "Caso 2:",
32          foo("A", false) || // imprime "A", retorna false
33          foo("B", false) // se evalua, imprime "B", retorna false
34          ); // Resultado final: false
35
36          console.log("\n*** Times/fibo ***");
37
38          // * fuerza evaluar ambos operandos siempre (estrategia
               aplicativa).
39          console.log(
40          "Caso 1:",
41          foo("A", fibo(10)) * // calcula fibo(10), imprime "A", ~55
42          foo("B", 0) // imprime "B", 0
43          ); // Resultado final: 55 * 0 = 0
44
45          // , (coma) evalua ambos, devuelve el ultimo (estrategia
               aplicativa).
46          console.log(
47          "Caso 2:",
48          foo("A", 0), // imprime "A", valor descartado
49          foo("B", fibo(10)) // imprime "B", calcula fibo(10)=55
50          ); // Resultado final: 55
51
52          console.log(
53          "\n*** Ternary (Warning!! it can take a big whiiile. Explain
                why) ***"
54          );
55
```

```
56          // ? : primero evalua condicion. Aqui fibo_start(50) es muy
               costoso (recursion exponencial).
57          console.log(
58          "Caso 1:",
59          foo("A", fibo_start(50) < 0) // imprime "A", calcula fibo
               (50) (muy lento), resultado false
60          ? foo("B", fibo_start(100)) // rama NO tomada (no se evalua)
61          : foo("C", 0) // imprime "C", retorna 0
62          ); // Resultado final: 0
63
64          // ====================
65          // Conclusion:
66          // JS usa evaluacion aplicativa (eager) de argumentos, pero
               algunos operadores (and, or, ?:)
67          // introducen evaluacion lazy (por demanda) mediante corto-
               circuito.
68          // Operadores aritmeticos y coma fuerzan evaluacion completa
                de sus operandos.
```

Listing 1: Explicacion de salidas en JS

SE CONCLUYE:

Como se explicó anteriormente, JS usa evaluación aplicativa (eager) de argumentos, pero algunos operadores (and, or, ?:) introducen evaluación lazy (por demanda) mediante corto-circuito. Operadores aritméticos y coma fuerzan evaluación completa de sus operandos.

Ejercicio 2

> Dos expresiones E1 y E2 se dicen Leibniz-equivalentes si para todo programa P que use E1 si cambiamos E1 por E2 en P y ejecutamos P los resultados de P antes y después del cambio en idénticas situaciones son los mismos. Es decir, P es invariante a cambios de E1 por E2 para cualquier P.
>
> El estudiante Carlisto :-)) afirma que dada la definición de myAdd, la expresión myAnd(x, y) es Leibniz-equivalente a la expresión x && y en JS para cualquier x y y ¿Es esa afirmación correcta? Justifique su respuesta formalmente.
>
> const myAnd = (x, y) =¿ x && y

Solución

Planteamiento:

Se afirma que, dada la definición:

```
const myAnd = (x, y) => x && y
```

la expresión `myAnd(x, y)` es Leibniz-equivalente a la expresión `x && y` en JavaScript, para cualquier `x` y `y`.

Definición de Leibniz-equivalencia:

Dos expresiones E_1 y E_2 son Leibniz'equivalentes si, para todo programa P que use E_1, al sustituir E_1 por E_2 en P y ejecutar en las mismas condiciones, los resultados son idénticos.

Análisis

- En JavaScript, las llamadas a funciones evalúan estrategia *eager* (estrictamente): antes de entrar al cuerpo de la función, se evalúan todos los argumentos.
- El operador lógico `&&` se evalúa con estrategia *lazy*: el segundo operando solo se evalúa si el primero es truthy.
- Esto implica que existen programas donde `x && y` y `myAnd(x, y)` producen efectos distintos.

Contraejemplo

```
1        // 1. Efecto observable:
2
3        const myAnd = (x, y) => x && y;
4
5        let s = 0;
6        const inc = () => (s++, true);
7
8        false && inc(); // Resultado: false, s == 0 (no se evalua
            inc, ya que evalua y solo si x es true)
9        myAnd(false, inc()); // Resultado: false, s == 1 (se evalua
            inc porque una funcion siempre evalua todos los
            argumentos)
```

Listing 2: Efecto observable en JS

Conclusión

La afirmación del estudiante es **incorrecta**. `myAnd(x, y)` no es Leibniz-equivalente a `x && y`, porque existen programas P donde la sustitución altera el comportamiento observable.

Ejercicio 3

Justifique o refute: en JS usar const en toda declaración siempre garantiza transparencia referencial.

Ejercicio 4

Dibuje ASTs y el grafo de memoria (state) del siguiente código y explique la salida. Recuerde que cada lambda se convierte en una nueva clausura como se explicó (ver arriba).

```
const y = 5
const x = y => x => x + y
console.log((y => x(y+1)(4))(y))
```