Paradigmas de Programación (EIF-400) Principios de FP

CARLOS LORÍA-SÁENZ LORIACARLOS@GMAIL.COM

AGOSTO 2017

EIF/UNA

Objetivos

- Introducir el paradigma de FP en forma transicional: de OOP a FP
- Justificar su valor en la práctica
- Preparar su uso concreto en JS

Objetivos Específicos

- Función como Objeto Computacional
- Guías de uso
- Patrones de FP (Combinadores)
- Enfoque composicional
- Lambda, Clausura
- OOP y FP: Contraste y unificación
- Variantes estilo Python
- Ventajas/Desventajas enfoque unificado

Material

- Si incluyen en la presentación slides que son ejercicios a resolver en Java o JS según se indique
- Se adjuntan proyecto web de ejercicio

Esencia de lenguaje

Enfoques imperativos

- Imperativo plano (Assembler, `C` plano)
- *Datos primitivos* : enteros,(`int`, `uint`, `long`, `char`, `byte`), flotantes, punteros (`array`)
- *Datos de usuario*: `struct` (records)
- *Verbos datos*: aritmética (`+`, `*`, `-`, `/`, `<`), lógica (`&&`, `!`, `||`), bitwise (`<<`, `&`, `|`) , …
- *Verbos de cambio de* ***estado***: =, ++, --, +=
- *Verbos Control* : `if-then-else`, `switch`, `goto`,
- Imperativo estructurado: ~~`goto`~~, `while`, `for`, `try-catch`
- Procedimental: función/subrutina abstrae operaciones frecuentes (`call/return`)

Abstracción primitiva datos

Referencias informativas *clásicas*

- Teorema de la programación estructurada (Böhm-Jacopini)
- Debate : `goto` o no `goto`
- E. Dijkstra: `goto` es peligroso
  - Código espagetti
- D. Knuth: Programación estructurada con `goto`
  - Hay casos donde `goto` es aceptable y mejor
- En assembler es inevitable; no hay estructuras de control nativas para while, if-then-else
- Más moderno: "*Callback hell*" (Ejemplos AJAX requests, llamadas a MongoDB, …)

Imperativo estructurado

- Estatutos y expresiones
- Verbos imperativos (for, while, try, …) le pertenecen a la máquina, son órdenes de esta
- Programador le da órdenes

Ejercicio

- Llamemos fila a un arreglo de objetos. Llame matriz a un arreglo de filas.
- Se tiene una matriz `m` y se quiere contar cuántas veces aparece un objeto `x` dado en `m`.
- Escriba `enCuantasOcurre(m, x)` en JS usando programación imperativa estructurada
- Al final de esta charla: Reescriba en FP

Secuencia y Estado mutable

- Imperativo con estado implícito
- Hay una memoria `M` en el contexto que el programa afecta
- Hay un antes y después para `M`.
- Un programa consta de instrucciones individuales que se ejecutan en cierta secuencia
- Afectan (cambian) a `M`
- El ordenamiento temporal de las instrucciones es importante
- Se pierden propiedades matemáticas básicas: conmutativad, asociatividad, etc

Ejemplo: Problemas orden y estado

- Dos funciones son *equivalentes* si ante mismas entradas producen los mismos resultados
- Formalmente: $\forall z\ (f_1(z) = f_2(z))$
- ¿Son `f1` y `f2` equivalentes, en general? Justifique su respuesta

```
function f1(x){return x;}
function f2(x){return x;}

let z = 0;
function g(x){
    return z+=x, z;
}
// Si z = g(666) Es f1(z) = f2(z) ?
console.log(f1(g(666)) == f2(g(666)))
```

Enfoque Orientado a Objetos imperativo

- Se concentra en la parte declarativa (enfoques cognitivos: generalización, especialización
- Cercana a teoría de conjuntos
- Mejora la abstracción en datos: clases, objetos, herencia, encapsulamiento
- Casi siempre supedita el verbo al sustantivo
- Los verbos se sustantivan si es necesario
- Control se puede sustantivar: Ejemplo hilos
- En la parte operativa: imperativo estructurado con estado (= memoria del objeto)

Nueva Abstracción al nivel de datos

Misma Abstracción al nivel de Control

Enfoque Funcional (FP)

- Paradigma que se concentra en la función como la unidad computacional (operativa) básica
- Raíces en la matemática: Cálculo Lambda (Church)
- Programar = especificar y componer funciones
- Computar = Evaluar funciones
- Control = Composición de funciones y recursión
- Deseable: Sin efectos-secundarios. Inmutabilidad.
- Principio: Función es **operación** y puede ser **dato** a la vez
- Existe independiente del objeto: *ciudadano de primera clase*
- "*Operadores*" para construir funciones a partir de funciones: *Combinadores* producen control
- Nosotros llamamos "*micro patrones*" de control. Pero no es término estándar

Función (morfismo)

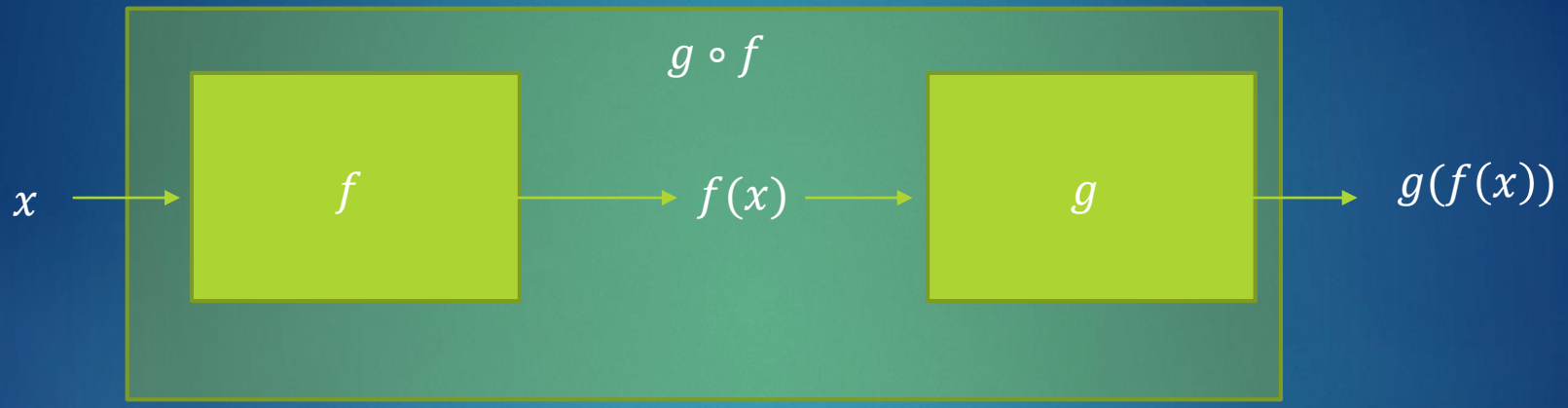
$$f: S \rightarrow T$$
$$f: x \rightarrow f(x)$$

Ejemplo: $f: int \rightarrow int$
$f: x \rightarrow x^2 - 2x + 1$

Categorías

f se dice ser un *morfismo* del objeto S en el objeto T

Composición: $g \circ f$ (g compuesto con f)



Restricción:
$codom(f) \subseteq dom(g)$

$\circ$: Toma dos morfismos y produce un morfismo

$g \circ f : dom(f) \to codom(g)$

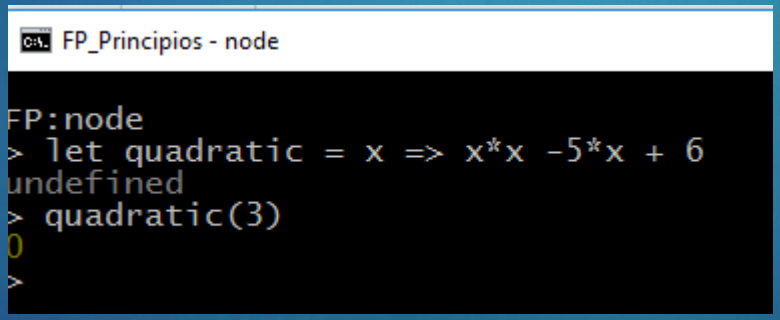
Ejemplo:

$$f : x \to 2x + 1$$
$$g : x \to x^2 + 2$$
$$g \circ f : x \to 4x^2 + 4x + 3$$

Ejercicio
Calcule $f \circ g$

Experimentamos en JS/ES6

- Estudiamos en JS/ES6 (leer)
- ES6: `arrow` como opción a `function`
- Sintaxis más estilo de FP
- Importante: `this` es léxico.
- Internamente sigue siendo `function`



Note: `let`

Ejercicio

- Implemente `comp(f, g)` en ES6 la composición de `g` con `f`.
- Asuma que son funciones *unarias* (sólo reciben un argumento) y que los dominios y rangos son compatibles
- Pruébela con el ejemplo siguiente

Ejemplo:

$$f: x \rightarrow 2x + 1$$
$$g: x \rightarrow x^2 + 2$$
$$g \circ f: x \rightarrow 4x^2 + 4x + 3$$

FP: puntos de partida

- La función como objeto computacional
- Recibe entradas genera salidas
- Programar es "combinar" funciones
- La salida de una función es la entrada de otra
- Correr es simplificar expresiones: transformar entradas en salidas, reemplazar "*igual-por-igual*"
- Aumentar: transparencia referencial
- Disminuir "estado mutable" (en especial si es compartido)

Ejemplo:

- ¿Qué calcula foo(a)? Asuma a es array.
- Pruebe por inducción su afirmación

```
function foo(a){
    let tail = (i, z) => (i < 0) ? z
                                 : tail(i - 1, z.concat( a[i] ))
    ;

    return tail(a.length - 1, []);
}
```

Lambda y clausura

- EN FP a la funciones objetos se les llama `lambdas`
- Origen del nombre: Cálculo lambda
- Una variable `x` que no es parámetro de una lambda `f` se dice ser una variable libre en `f`.
- Problema: "con qué objeto `z` se "liga" `x` y en qué momento. Eso es resolver x. A `z` se le llama el "binding" de `x`.
- Estrategias de resolución: ¿Cuál `z` usar?
- Lugar de definición: lugar donde se define `f` (alcance léxico)
- Lugar de aplicación: lugar donde se llama a `f` (aplica, ejecuta). (Alcance dinámico)
- Cerrar una lambda: ligar lógicamente las variables libres (resolver las variables libres)
- Enfoque más usado: cerrar léxicamente, es decir, usar resolución léxica
- Clausura: lambda y una estructura que tiene sus cierres

Ejemplo: ¿Cuánto imprime? ¿ 666 ó 1000?


```
function def(){
    let y = 999;
    return x => x + y;  //   lugar de creacion
}
function app(){
     let y = 665;
     return def()(1);     //   lugar de aplicacion
}

console.log(app());
```

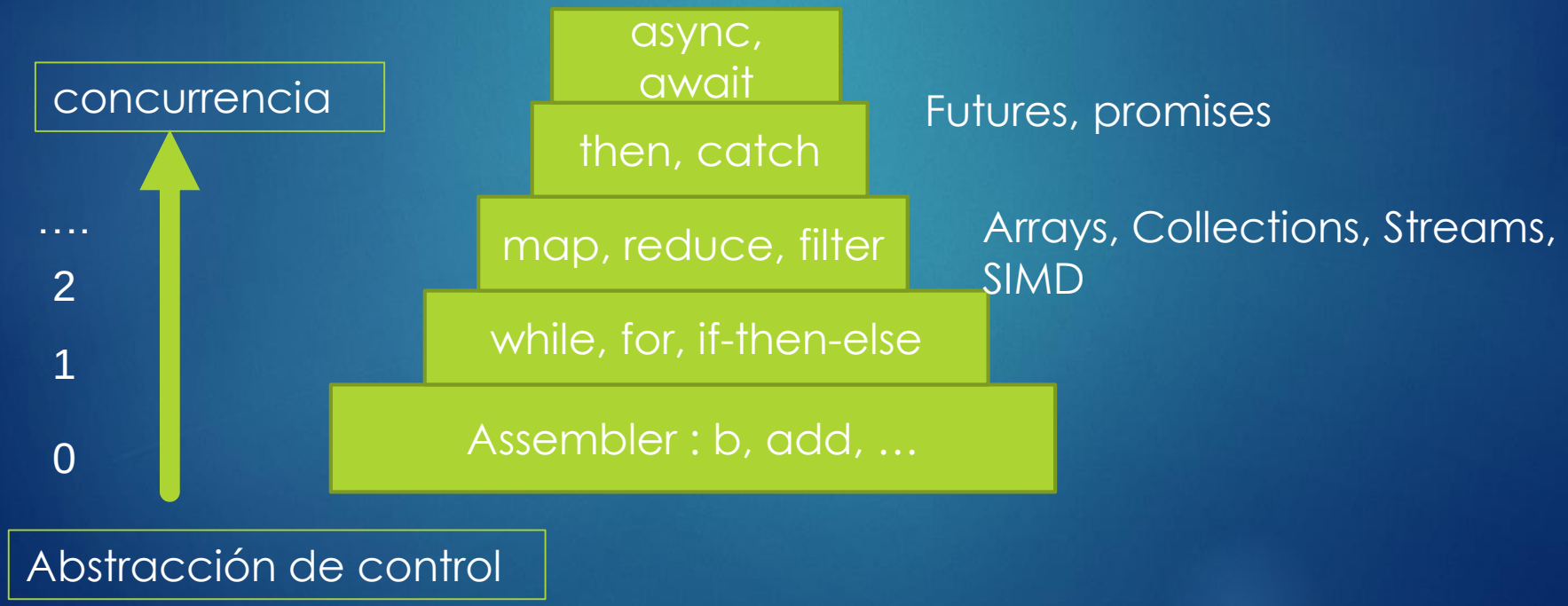
Redefiniendo this

Guías en combinación con OOP

- Control: operaciones sobre datos. Estos "llevan" el control
- Evitar `for`, `while`, = (mutable)
- Convertir en "combinadores" controlados por los objetos
- En FP "objetos" es llamado más generalmente "tipos" de datos

Combinadores comunes

- Los llamamos en este curso *micro-patrones*
- Expresan *unidades frecuentes de computación*
- Son más abstractos que los elementos de control imperativos estructurados



Combinadores básicos: ver Array

- `Map`
- `Reduce` (`fold left`)
- `Filter`
- `forEach`
- `flatten` (no está en ES6)
- `zip`, `unzip` (no están ES6)
- `some`, `every`, `find`

También se les dice *functores*: toman morfismos y los "extienden" a colecciones (categorías)

Map (versión no OOP)

No muta el arreglo de entrada
Crea uno nuevo

Filter

Reduce (fold left)

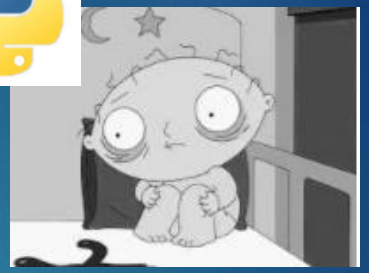
Recurrencia: $z_0 = z;\ z_i = f(z_{i-1}, x_{i-1})\ para\ i = 1, \dots, n$

`forEach`: el más imperativo

Operador coma

Es como `map`, pero no construye la lista de salida

Digresión: Python



- FP en Python es más general: no sólo en Array sino en iterables/generadores
- En Python se usan en mucho las listas por comprensión (sobre iterables)
- Populares del mundo Haskell (original SETL, Miranda)
- En Haskell son usadas para paralelismo
- Más detalles veremos luego
- Tutorial de Python anti-insonmio en dropbox

ES6 for-of

- Cercano a un for de Python
- Pero es estatuto no expresión
- Ver for-of

```
// nats.js
function * nats(n=0){
    do {
       yield n++;
    } while (true) yield

}

module.exports = {
    nats

}
```

```
FP:node
> var {nats} = require('./nats.js')
undefined
> NATS = nats()
{}
> for (let n of NATS) {if (n>10) break; console.log(n);}
0
1
2
3
4
5
6
7
8
9
10
undefined
> NATS = nats()
{}
> NATS.next()
{ value: 0, done: false }
> NATS.next()
{ value: 1, done: false }
>
```

Ejercicio

- Desarrolle un módulo fp en ES6 que tenga map, reduce y filter asumiendo que Array no los tiene
- Implemente ese módulo de manera imperativa
- Extienda para que funcione con Iterable

Ejercicio: back to the "Funda"

- Dado un arreglo `a` de números enteros, encontrar la suma de los elementos que ocurren en posiciones pares.
- Escriba una función `sumaEnPares(a)` en `JS` que realice esa tarea usando OOP imperativo
- Transforme en una versión FP