Programación Funcional

Marzo, 2018 - Semana 1

Agenda

- Que és?
- Declarativos vs Imperativos
- Programas como funciones
- Inmutabilidad
- Rol de la recursión
- Composición de funciones
- Funciones como componente de primera clase
- Recursos

Qué es?

Es un **paradigma de programación** que se basa en un **modelo matemático**. Se piensa a la ejecución de programas como la **evaluación de funciones matemáticas**.

Sus **principales ideas** son ser declarativo y usar funciones matemáticas.

Una **familia de lenguajes funcionales** muy utilizada se conoce con el nombre de **Lisp** (List Processing). Algunas **implementaciones** son: **Scheme**, **Common Lisp**, **Clojure**. Existen **otros** lenguajes como: **Haskell**, **R**, etc.

Declarativos vs Imperativos

- **Imperativos:** se describe paso a paso un conjunto de instrucciones que deben ejecutarse para variar el estado del programa y hallar la solución.

- **Declarativos:** se describe el problema que se quiere solucionar; se programa diciendo lo que se quiere resolver a nivel de usuario, pero no las instrucciones necesarias para solucionarlo.

Declarativos vs Imperativos

```
public class Persona {
    private String nombre;
    private int edad;

    public Persona(String nombre, int edad) {
        this.nombre = nombre;
        this.edad = edad;
    }

    public String getNombre() {
        return nombre;
    }

    public int getEdad() {
        return edad;
    }
}
```

Declarativos vs Imperativos

```
public class Main {
    public static void main(String[] args) {
        Persona p1 = new Persona("pepe",20);
        Persona p2 = new Persona("juan",12);
        Persona p3 = new Persona("angela",30);

        List<Persona> personas = Arrays.asList(p1, p2, p3); // Se tienen las personas creadas.

        System.out.println(calcularEdadPromedioDeMayores(personas));
    }
}
```

Imperativos

```
double calcularEdadPromedioDeMayores(List<Persona> personas) {
    int totalEdad = 0;
    int totalPersonas = 0;

    for (Persona persona: personas) {
        if (persona.getEdad() >= 18) {
            totalEdad += persona.getEdad();
            totalPersonas++;
        }
    }
    return totalEdad / totalPersonas;
}
```

Declarativos

```
double calcularEdadPromedioDeMayores(List<Persona> personas) {
    OptionalDouble resultado = personas.stream()
                .filter(persona -> persona.getEdad() >= 18)
                .mapToInt(persona -> persona.getEdad())
                .average();

    return resultado
          .orElse(/* Y si no habia personas? */);
}
```

Programas como funciones

Cada computación se piensa como la evaluación de una serie de funciones matemáticas.

- **Función Matemática:** Un mapeo que asigna una salida a cada entrada.
- **Computación:** Secuencia de acciones que puede parametrizarse con valores de entrada y generar valores de salida.

f(x) = 2x => 1 -> 2
2 -> 4
3 -> 6

g(x, y) = f(x) + f(y)
g(1, 2) = f(1) + f(2) = 2 + 4 = 6

Programas como funciones

El **elemento fundamental** del paradigma es la **función**. Cualquier programa se considera como la evaluación de un conjunto de funciones.
Estas respetan ciertas **restricciones**:

- **Predecible**: Siempre produce la misma salida para una determinada entrada.
- **Inmutable**: No pueden cambiar estado.

Programas como funciones

- Siempre produce la misma salida para una determinada entrada. (**Predictibilidad**).

1ra ejecución => **update() => x = 7**

2da ejecución=> **update() => x = 9**

¿Qué podemos esperar de una 3ra ejecución?

Inmutabilidad

- Las funciones no pueden cambiar estado.

```
Door.open = true
Door.open = false
```

```
Names = [“Jan”, “Kim”, “Sara”]
double_names():
    for (i, name) in Names
        Names[i] = name + name
```

Para cerrar la puerta es necesario cambiar su estado interno. La función double_names cambia estado “afuera” de sí misma (efecto de lado).

Inmutabilidad

- Problemas con el Estado:
  - Condiciones de carrera (orden de ejecución, concurrencia)
  - Complejidad (acceso)
  - No predecible (accion a distancia)
- Programación Funcional = Programación sin **Estado** (sin reasignación de variables)
- Las funciones no pueden cambiar su entrada y no pueden cambiar variables. (asignaciones, for).
- El concepto general es que **en lugar de actualizar** valores **se crean valores nuevos**.

Rol de la recursión

Dado que no se puede utilizar asignación de variables los loops como **for** están descartados. Se utiliza la recursión como mecanismo de iteración.

for (**i=0**; i< 10; **i++**)

Rol de la recursión - Iterativo

```
def factorial(n):
        numero = 1
        while n >= 1:
                numero = numero * n
                n = n - 1
        return numero
```

Rol de la recursión - Recursivo

```
def factorial(numero):
    if numero <= 1:
        return 1
    else
        return numero * factorial(numero-1)
```

Rol de la recursión - Recursivo

```
(defn factorial [numero]
    (if (<= numero 1)
        1
        (* numero (factorial (- numero 1)))
    )
)
```

Funciones como componentes de primera clase

Las funciones se tratan como cualquier otro valor:

- Tienen expresiones literales

  (fn [x y] (+ x y))

- Pueden ser argumentos a funciones

  (some **(fn [name] (= "rob" name))** ["kyle" "siva" "rob" "celeste"])
  ;=> true

- Pueden ser retornadas por otras funciones

  ((**comp inc** *) 2 3 4)
  ;=> 25

Composición de funciones

- Las funciones se pueden componer para obtener nuevas funciones
- Funciones de **orden superior**: las funciones pueden recibir otras funciones como valores y devolver otras funciones como resultado.

```
(def bools [true true true false false])
(every? true? bools)
;=> false

(some (fn [p] (= "rob" p)) ["kyle" "siva" "rob" "celeste"])
;=> true
```

Composición de funciones

```
(def the-map {:a 1 :b 2 :c 3})

(the-map :b)
;=> 2
(:b the-map)
;=> 2
(:z the-map 26)
;=> 26 default value


(def users {
   :kyle {
      :date-joined "2009-01-01"
      :summary {
         :average {
            :monthly 1000
            :yearly 12000}}}
   ... })
```

```
(assoc-in map [key & more-keys] value)

(assoc-in users [:kyle :summary :average :monthly] 3000)
;=> {:kyle {:date-joined "2009-01-01",
            :summary {:average {:monthly 3000
                                :yearly 12000}}}
     ... }


(get-in users [:kyle :summary :average :monthly])
;=> 1000



(update-in map [key & more-keys] update-function & args)

(update-in users [:kyle :summary :average :monthly] +
500)
;=> {:kyle {:date-joined "2009-01-01"
            :summary {:average {:monthly 1500
                                :yearly 12000}}}
```

Funciones de orden superior

Son funciones que **operan sobre otras funciones** como argumento. Se usan para:

- Abstraer estructuras de control
- Describir la intención con la que se manipulan valores
- Tener componentes con propiedades conocidas

Funciones de orden superior

(some f coll), **(every? f coll)**, **(not-every? f coll)**, **(not-any? f coll)**: Usan un predicado **f** para probar los elementos de **coll** hasta determinar si se cumple (o no) su condición.

```
(some       even? (list 1 2 3)) => true
(every?     even? (list 1 2 3)) => false
(not-every? even? (list 1 2 3)) => true
(not-any?   even? (list 1 2 3)) => false
```

Funciones de orden superior

(map f coll): Crea una lista que contiene el resultado de aplicar **f** a cada elemento de **coll**.

```
(map inc [1 2 3]) => (2 3 4)
(map even? [1 2 3]) => (false true false)
```

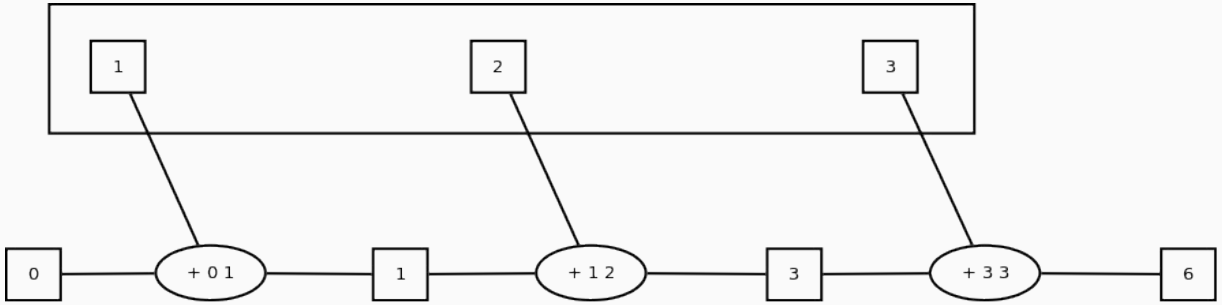
(filter f coll): Crea una lista que tiene solo los elementos de **coll** para los que **f** retorna verdadero.

```
(filter even? [1 2 3 4]) => (2 4)
```

Funciones de orden superior

(reduce f acc coll): Usa **f** para combinar un valor inicial **acc** con cada uno de los elementos de **coll**.

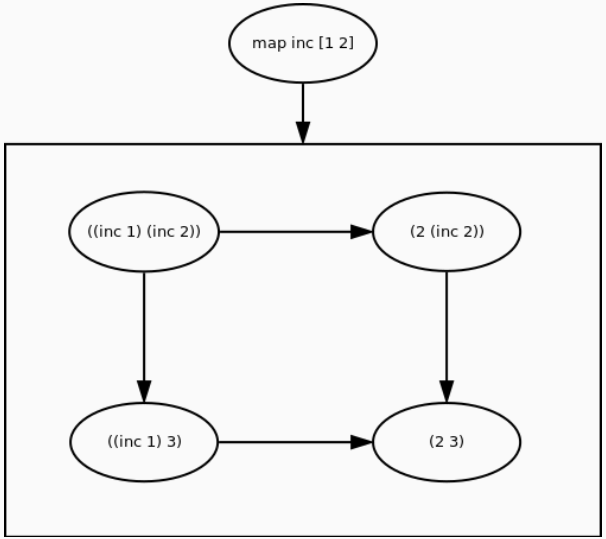
```
(reduce + 0 [1 2 3])
```



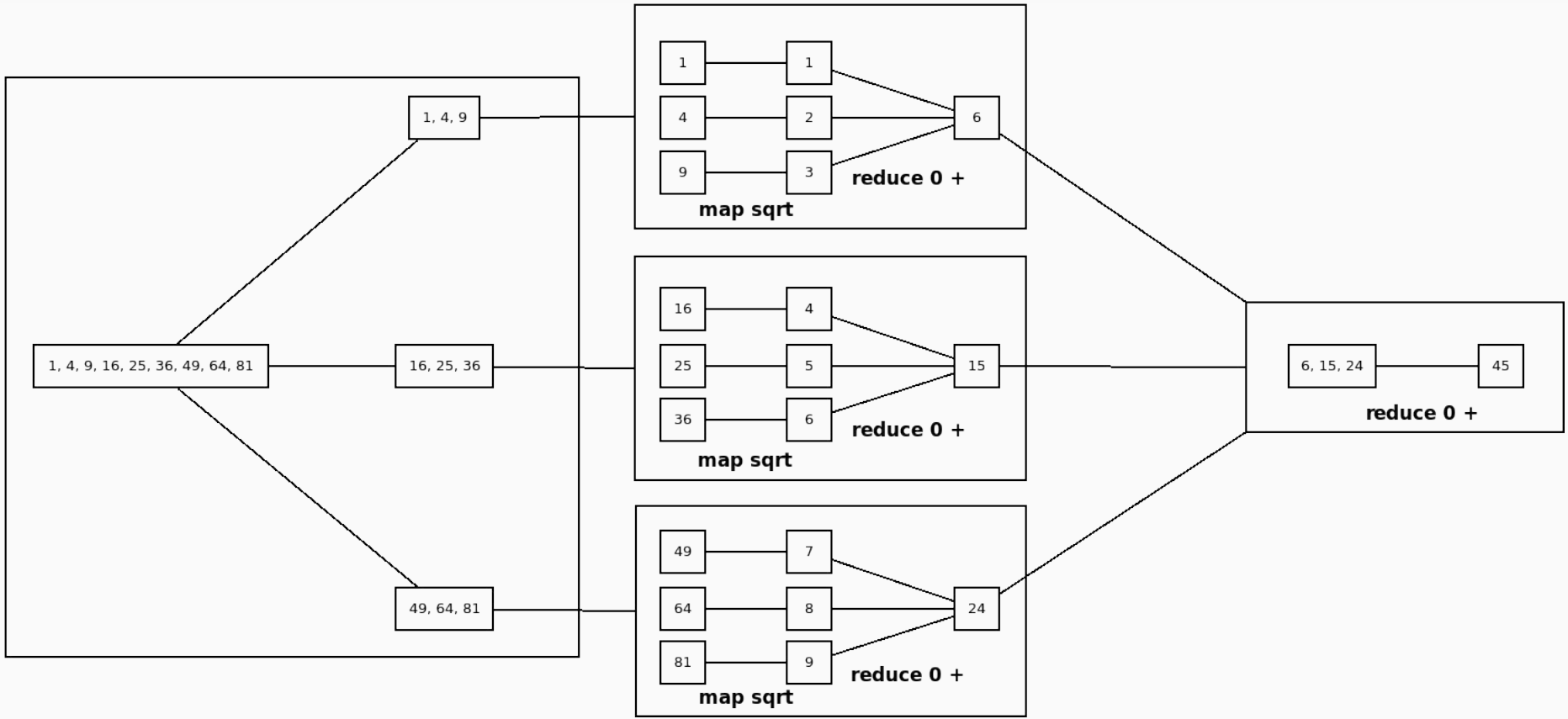
Funciones de orden superior

Las funciones de orden superior usuales son comunes en programación funcional por tener propiedades útiles al usarlas con funciones matemáticas:

map	Los elementos del resultado pueden calcularse en cualquier orden y momento (nunca, si no se usan).
map, filter	El resultado de una sublista es una sublista del resultado completo.
reduce	`f` asociativa ⇒ Se pueden hacer las combinaciones en cualquier orden.



Funciones de orden superior



Recursos

- Repl
- Try Clojure
- Clojure in Action
- Structure and Interpretation of Computer Programs
- Functional Thinking
- Ejercicios
- Más Ejercicios