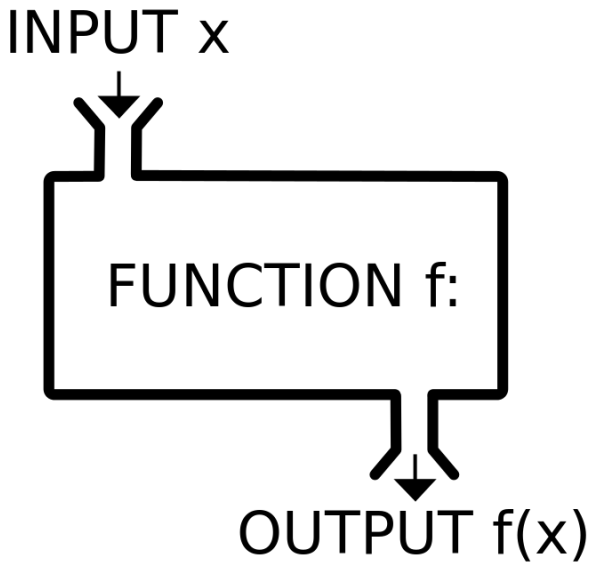
UNIVERSIDAD ADOLFO IBAÑEZ

Lenguajes y Paradigmas de Programación

INGENIERÍA

Paradigma Funcional

#TodosSomosFunciones



Paradigma Funcional

Idea: Variables no cambian de valor (inmutables)

→ No hay efectos colaterales en el llamado de funciones

misma entrada → misma salida siempre

Composición de Funciones

La salida de una función es la entrada de otra
→ no hay operaciones intermedias

g(f(x)) ← ¡de esta forma!

Lenguajes Funcionales

- Diseñados sobre el concepto de funciones matemáticas (condicionales + recursividad).

- La programación funcional admite funciones de orden superior (Tomar una o más funciones como entrada. Devolver una función como salida.)

Lenguajes Funcionales

- Los lenguajes de programación funcional no admiten controles de flujo. Utilizan directamente las funciones y llamadas funcionales.

- Al igual que OOP, los lenguajes de programación funcionales admiten conceptos populares como Abstracción, Encapsulación, Herencia y Polimorfismo.

Ventajas

- Código “libre” de errores: no hay efectos colaterales → no hay estados → más fácil de evitar errores.

- Escritos para programación paralela: una función puede paralelizarse en múltiples trabajadores (workers)

Ventajas

- Eficiencia: unidades independientes que se ejecutan concurrentemente.

- Soportan funciones anidadas: ¿alguien dijo recursión?

- Evaluación “floja” (lazy evaluation): son parte estructural de los lenguajes funcionales

Desventajas

La memoria a utilizar puede ser elevada, dado que requiere crear un objeto por cada asignación que se desee realizar (copias).

Clojure

Lenguaje de programación de propósito general dialecto de LISP. Clojure puede ser ejecutado sobre la Máquina Virtual de Java y la máquina virtual de la plataforma .NET, así como compilado a JavaScript.

- Eliminar la complejidad asociada a la programación concurrente.

Clojure

Típica manera de hacer las cosas en Clojure

(función argumento1 argumento2 ...)

Clojure

(+ 1 2)

→ Esto evalúa a 3

Phyton

```
def suma(number_1, number_2):
  suma = number_1 + number_2
  return suma

nro1 = 1
nro2 = 2
total = suma (nro1, nro2)
print(nro1, nro2, total)
```

Phyton

- Actividad: queremos hacer lo siguiente en Phyton

```
int x = 0;
x++;
printf("%d\n", x);
```

Phyton

¿Qué buscamos?

1. Definir una variable x con valor inicial 0
2. Incrementar el valor de la variable x en 3
3. Imprimir el resultado en pantalla

Usaremos repl.it para probar Phyton

Esto debe ser mediante una función.

Variables (Clojure)

- Identifiers: nombrar alguna cosa y poder darle un valor

  (**def** nombre valor)

Algunos valores posibles (Clojure)

```
1        ; integer
1N       ; arbitrary-precision integer
1.2      ; float/double/decimal
1.2M     ; arbitrary-precision decimal
1.2e4    ; scientific notation
1.2e4M   ; sci notation of arbitrary-precision decimal

0x3a     ; hex literal (58 in decimal)
1/3      ; Rational number
\a       ; The character "a".
"hi"     ; A string.
```

BTW ; es para comenzar un comentario… no es para terminar una línea :)

Variables (Clojure)

1. Definir una variable x con valor inicial 0

```
(def x 0)
```

Algunas cosas para jugar

Funciones varias

- Implementar las operaciones básicas como funciones en Phyton y ejecutarlas con parámetros de entrada.

Pero...

¿No dijimos que las variables eran inmutables en Clojure?

Variables

1. Definir una variable x con valor inicial 0

```
(def x (atom 0))
```

Función que define un átomo, que es una manera de manejar un estado.

Setear el valor de una variable

De un átomo en realidad

(swap! átomo función)

Setear el valor de una variable

2. Incremen 3

(swap! x (

swap! es la función que permite setear el valor de un átomo evaluando la función que se pasa como parámetro

Es una función que permite usar la función con menos parámetros

Imprimir en pantalla

Esto es lo más parecido a lo que saben :)

(println expresión)

Imprimir en pantalla

3. Imprimir el resultado en pantalla

```
(println x)
```

Imprimir en pantalla

3. Imprimir el resultado en pantalla

```
(println @x)
```

El @ es la forma de obtener el valor, no el “objeto”

Hay otras maneras

```
(let [
    a  0
    a + a 3
  ]
  (println a)
)
```

¿Qué hace este ejemplo?

```
(let [i (atom 0)]
  (defn generar-id-unico
    "Acá va una descripción :)"
    []
    (swap! i inc)))
```

Creando nuestras funciones

defn es la manera en que uno define funciones propias en Clojure.

Como vieron las funciones tienen:

- Nombre
- Descripción
- Parámetros
- Cuerpo
- Y SIEMPRE retornan un valor

Entonces podemos hacer esto

```
(defn suma3
    "Sumaremos 3 a un número"
    [i]
    (+ i 3))
```

Condicionales

Los if también son funciones:

(if (condicion) (funcion para true) (funcion para false))

Condicionales

```
(def a (atom 3))
(if (<= @a 5) (print-str @a) (println @a) )
```

-> llama a print-str @a dado que el valor de a es menor o igual a 5

Ciclos

Adivinen… while también es una función :)

(while (condicion) (funcion a ejecutar))

Ciclos

```
(def a (atom 3))
(while (<= @a 5) (do
    (println @a) (swap! a inc)
))
```

do es una función que permite agrupar varias llamadas para que parezcan una → útil para if, for, y otros

¿Y los ciclos fo

for es el keyword para ciclos. Lo primero que recibe es un bloque con las variables de ciclo.

```
(for [x (range 1000)
      :when (even? x)]
  (print x\ )
)
```

El cuerpo es una función que se evalúa cada vez.

Una gracia de Clojure: ciclos dobles

```
(for [x (range 6)
      y [1 2 3]]
  (print (* x y) \ )
)
```

¿Qué hace esto?

dotimes

Cuando lo que quiero es ejecutar algo un cierto número de veces entonces mejor uso dotimes en lugar de for:

```
(dotimes [i 10]
  (print i\ )
)
```

Ejercicio

Imprimir una pirámide en Clojure o Phyton

```
#
##
###
####
```

Una solución

https://repl.it/@DaniloBorquez/clojure-mario

¿Preguntas?

Esto se está complicando ^^