

# comparación funcional - imperativo

Paradigmas de la Programación

FaMAF-UNC

2021

# declarativo vs. imperativo

- las construcciones más primitivas son imperativas:

*Traeme esa manzana*


`x: = 5`

- abstracción: las declarativas describen un hecho

*La tierra es redonda*

```
function f(int x) { return x+1; }
```

- las construcciones imperativas **cambian** un valor y las declarativas **crean** un nuevo valor

```
{ int x = 1;  
  x = x+1;  imperativa  
  { int y = x+1;  
    { int x = y+1;  
  }  
}
```

# asignación destructiva

- La asignación imperativa puede introducir *efectos secundarios*: puede destruir el valor anterior de una variable
- en programación funcional se la llama *asignación destructiva*

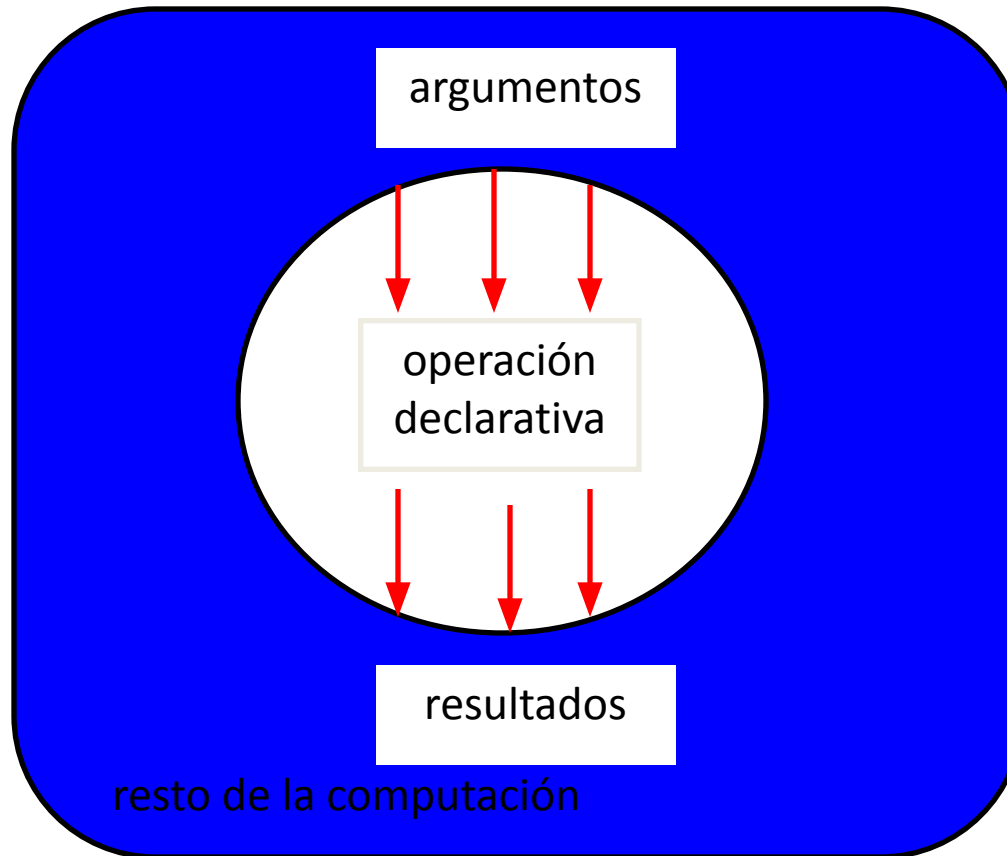
# operaciones declarativas

- una operación es declarativa si siempre que la llamamos con los mismos argumentos devuelve los mismos resultados, independientemente del estado de la computación
- la composición de dos operaciones declarativas es declarativa, por lo tanto, podemos crear grandes programas declarativos por composición de operaciones declarativas

# operaciones declarativas

- una operación declarativa es:
  - **independiente** (depende sólo de sus argumentos)
  - **sin estado** (no recuerda ningún estado entre llamados)
  - **determinística** (los llamados con los mismos argumentos siempre dan los mismos resultados)

# bondades de las operaciones declarativas



# ventajas de los componentes declarativos

- **Programación a pequeña escala:** es más fácil razonar sobre programas declarativos porque podemos usar técnicas algebraicas y lógicas
- **Programación a gran escala:** una componente declarativa se puede escribir, testear y verificar independientemente de otras componentes.
  - la complejidad de razonar sobre un programa compuesto de componentes no declarativas explota por la combinatoria de la interacción entre componentes

# ventajas de los componentes declarativos

- como las componentes declarativas son funciones matemáticas, se puede aplicar **razonamiento algebraico**, sustituyendo iguales por iguales
- se pueden escribir componentes declarativas en modelos que permiten tipos de datos con estado, pero perdemos las garantías de mantener declaratividad



# transparencia referencial

- una expresión transparente referencialmente se puede sustituir por su valor sin cambiar la semántica del programa

```
{ int x = 1;  
  x = x+1;  
  { int y = x+1;  
    { int z = y+1;  
  }  
}
```

es lo mismo que...

```
{ int x = 1;  
  x = x+1;  
  { int y = x+1;  
    { int x = y+1;  
  }  
}
```

# transparencia referencial

- todas las componentes declarativas, independientemente de su estructura, se pueden usar como valores:
  - como argumentos de función, como
  - resultados de función
  - como partes de estructuras de datos

```
HayAlgunExceso xs n = foldr (  
    filter (>n) (  
        map convertirSistemaMetrico xs)  
    )  
    False xs
```

# traducción de imperativo a declarativo

$\{F \ X1 \ \dots \ Xn \ R\} \longrightarrow$  procedimiento  
(con efectos secundarios)

$R = \{F \ X1 \ \dots \ Xn\} \longrightarrow$  función  
(equivalente a un valor)

```
R = fun {F X1 ... Xn}  
  <sentencia>  
  <expresión>  
end
```

$\underbrace{\hspace{10em}}_{\langle \text{expresión} \rangle}$

$\underbrace{\hspace{15em}}_{\langle \text{sentencia} \rangle}$

```
proc {F X1 ... Xn R}  
  <sentencia>  
  R = <expresión>  
end
```

$\underbrace{\hspace{10em}}_{\langle \text{sentencia} \rangle}$

# anidamiento en estructuras de datos

$$Ys = \{F \ X\} \mid \{Map \ Yr \ F\}$$

se reescribe desanidado como:

```
local Y Yr in
  Ys = Y | Yr
  {F X Y}
  {Map Xr F Yr}
end
```

(el desanidado de las llamadas ocurre después de armada la estructura de datos)

# transparencia referencial y razonamiento sobre programas

```
globalValue = 0;
```

```
integer function rq(integer x)  
begin  
    globalValue = globalValue + 1;  
    return x + globalValue;  
end
```

```
integer function rt(integer x)  
begin  
    return x + 1;  
end
```

# transparencia referencial y razonamiento sobre programas

```
integer p = rq(x) + rq(y) * (rq(x) - rq(x));
```

# transparencia referencial y razonamiento sobre programas

```
integer p = rq(x) + rq(y) * (rq(x) - rq(x));
```

```
integer p = rq(x) + rq(y) * (0);
```

```
integer p = rq(x) + 0;
```

```
integer p = rq(x);
```

# transparencia referencial y razonamiento sobre programas

```
integer p = rq(x) + rq(y) * (rq(x) - rq(x)) ;
```

```
integer p = rq(x) + rq(y) * (0) ;
```

```
integer p = rq(x) + 0 ;
```

```
integer p = rq(x) ;
```

Pero cada ocurrencia de `rq()` evalúa a un valor distinto!!!!



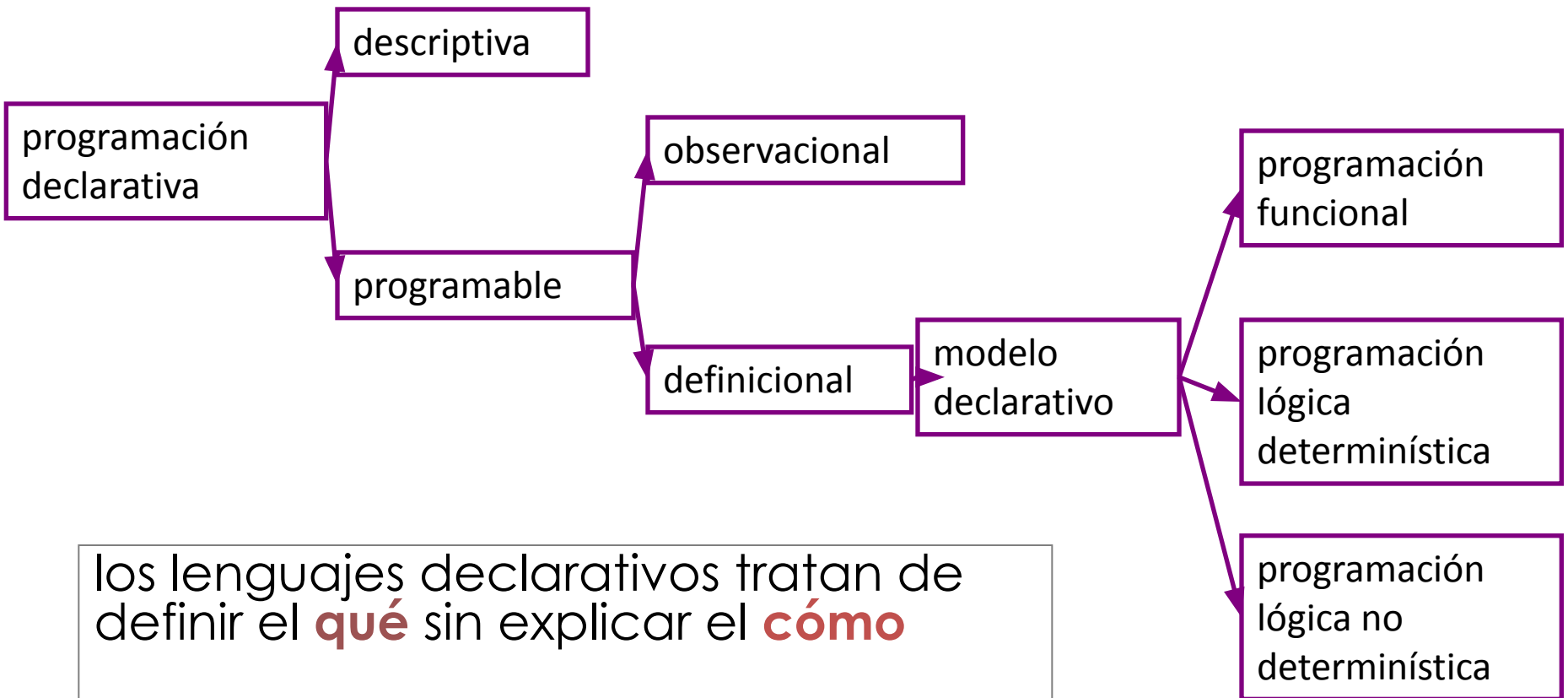
# transparencia referencial

- razonar sobre el código
- programas más robustos
- encontrar errores
- encontrar optimizaciones

# componentes declarativos vs. lenguajes declarativos

- en todos los lenguajes se pueden escribir componentes declarativos, que tendrán las propiedades mencionadas
- algunos lenguajes proveen sintaxis fuertemente declarativas y semántica más declarativa?
- de más declarativo a menos declarativo:  
Prolog puro, Haskell, OCaml, Scheme/Lisp, Python, Javascript, C--, Perl, PHP, C++, Pascal, C, Fortran, Assembly

# clasificación de lenguajes declarativos



entonces,  
para qué queremos el estado?

porque la realidad tiene estados

(la realidad del mundo y la realidad de la  
máquina)

# estado explícito

- el estado de la computación está siempre, también en un programa funcional (se puede diagramar la computación con los diferentes active records que se apilan y desapilan)
- los programas imperativos integran el estado de forma explícita: variables globales, resultados temporales
- es más adecuado hablar de componentes o programas imperativos y no lenguajes imperativos
- los lenguajes funcionales también incluyen formas de referirse al estado
  - mónadas
  - pasar el estado como parámetro

# cuándo queremos usar el estado explícito

- cuando queremos representar memoria
- cuando el entorno es determinante para el comportamiento de las componentes (agentes)

# cuándo queremos usar el estado explícito

- cuando la asignación destructiva convierte un problema intratable en tratable
- cuando queremos guardar resultados temporales (por ejemplo, en [programación dinámica](#))
  - encontrar el camino más corto (Dijkstra) (sabemos cuál es el camino más corto entre los puntos intermedios)
  - fibonacci
  - alineamiento de secuencias (distancia de edición)
  - torres de Hanoi
  - multiplicación de matrices



# fibonacci con estado explícito

- si usamos memoización, el tiempo de cálculo de fibonacci pasa de exponencial a lineal, con mayor uso de espacio

## **top-down, espacio $O(n)$**

```
var m := map(0 → 0, 1 → 1)
function fib(n)
    if key n is not in map m
        m[n] := fib(n - 1) + fib(n - 2)
    return m[n]
```

# fibonacci con estado explícito

- si usamos memoización, el tiempo de cálculo de fibonacci pasa de exponencial a lineal, con mayor uso de espacio

## bottom-up, espacio $O(1)$

```
function fib(n)
  if n = 0
    return 0
  else
    var previousFib := 0, currentFib := 1
    repeat n - 1 times // loop is skipped if n = 1
      var newFib := previousFib + currentFib
      previousFib := currentFib
      currentFib := newFib
    return currentFib
```

# mónadas

- muchos lenguajes funcionales “puros” proveen algún tipo de construcción lingüística para poder expresar instrucciones imperativas: las mónadas
- crear un alcance aislado del resto del programa
- se permiten ciertas operaciones con efectos secundarios: variables globales, asignación destructiva.
- “punto y coma programable”, que transportan datos entre unidades funcionales

# conurrencia declarativa

- paralelizar programas declarativos es trivial: las componentes declarativas se pueden ejecutar de forma concurrente sin que se den condiciones de carrera.
- algunas paralelizaciones son absurdas:
  - existe dependencia entre resultados
  - el overhead es demasiado alto para la ganancia obtenida

# algunas preguntas

- los lenguajes declarativos pierden mucha eficiencia con respecto a los imperativos?
- los lenguajes declarativos son más adecuados para representar los problemas?
- es siempre más fácil de razonar sobre un programa escrito en un lenguaje declarativo que en un lenguaje imperativo?