

Paradigma Funcional

Caso de estudio: ML

Lenguajes de Programación 2018

Paradigma Funcional

FUNCIÓN:

Mapeo de un dominio en un rango
El mapeo se puede describir por medio
de una EXPRESIÓN

$$f(1) = 2$$

$$f(2) = 3$$

...

Enumeración

$$f(x) = 1 + x$$

Expresión

Paradigma Funcional

- ⦿ La **computación** se logra mediante la **evaluación de expresiones**
 - > *¡¡NO mediante **ejecución de sentencias**!!*
- ⦿ La **característica esencial** de la programación funcional es que los **cálculos** se ven como una **función matemática** que hace corresponder entradas con salidas.
- ⦿ A diferencia de la programación imperativa, **no hay una noción implícita de estado**,
 - > Por lo tanto, no hay instrucciones de asignación.

Paradigma Funcional

- Ejemplos:

- > Lisp (años 60 - sin tipos)
- > FP (años 70 – sin tipos)
- > ML, Scheme, Hope, Miranda (años 80 - tipados)
- > Haskell, Earlang (años 90 – tipados)

- El fundamento de la programación funcional es el **CALCULO LAMBDA** (Church [1941]):

$$(\lambda x. x * x)$$

Paradigma Funcional

Los Lenguajes Funcionales **se caracterizan por:**

- Conjunto de Objetos
- Conjunto de Funciones Primitivas
- Conjunto de Formas Funcionales
- Operación de Aplicación

Paradigma Funcional

⦿ Objetos

- Miembros del **Dominio** y **Rango** de las funciones.
- Depende de las **características** del lenguaje.

⦿ Funciones Primitivas

- Las **funciones** que provee el lenguaje como **predefinidas**
- Depende del **dominio de aplicación** del lenguaje.

Paradigma Funcional

- Formas Funcionales (Funciones de alto orden)
 - > Son las funciones cuyo **argumento o resultado es una función**.
 - > Tienen un fuerte impacto en la **expresividad del lenguaje**.
- Operación de Aplicación
 - > Es el principal **mecanismo de control**.
 - > Como las funciones se **llaman** entre si y como se **aplican sus argumentos**.

Paradigma Funcional

- ⦿ **Propiedades** de los lenguajes **funcionales puros**:
- ⦿ Semántica de valores
- ⦿ Transparencia referencial
- ⦿ Funciones como valores de primera clase
- ⦿ Currying
- ⦿ Evaluación perezosa

Paradigma Funcional

● Semántica de valores

Desaparece la noción de estado

Locación de memoria ~~↔~~ Variable

Modificación ~~↔~~ Asignación

Una **VARIABLE** se asocia a **UN VALOR**
(en favor de la noción **MATEMATICA!**)

Store

X	2
X	1
...	...

x=1

...

x=2

...

La información
del STORE no se
modifica, solo se
agrega nueva
información

Paradigma Funcional

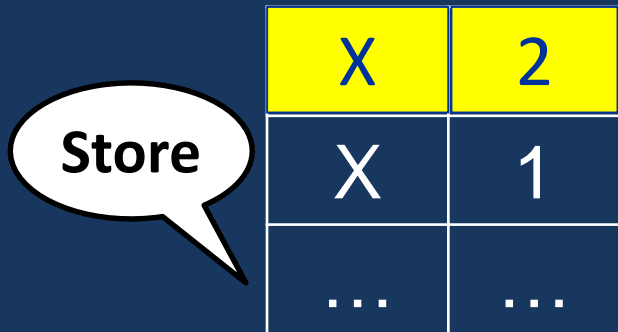
● Semántica de valores

Desaparece la noción de estado

Locación de memoria ~~↔~~ Variable

Modificación ~~↔~~ Asignación

Una **VARIABLE** se asocia a **UN VALOR**
(en favor de la noción **MATEMATICA!**)



X	2
X	1
...	...

x=1

fun f1 = x+2

x=2

fun f2 = x+2

...

f1

f2

3

4

Paradigma Funcional

Transparencia referencial

Lenguajes funcionales:

Una función computa diferentes resultados **únicamente** a partir de su argumento.

Para el mismo valor del argumento siempre dará el mismo resultado.

No hay efectos colaterales.

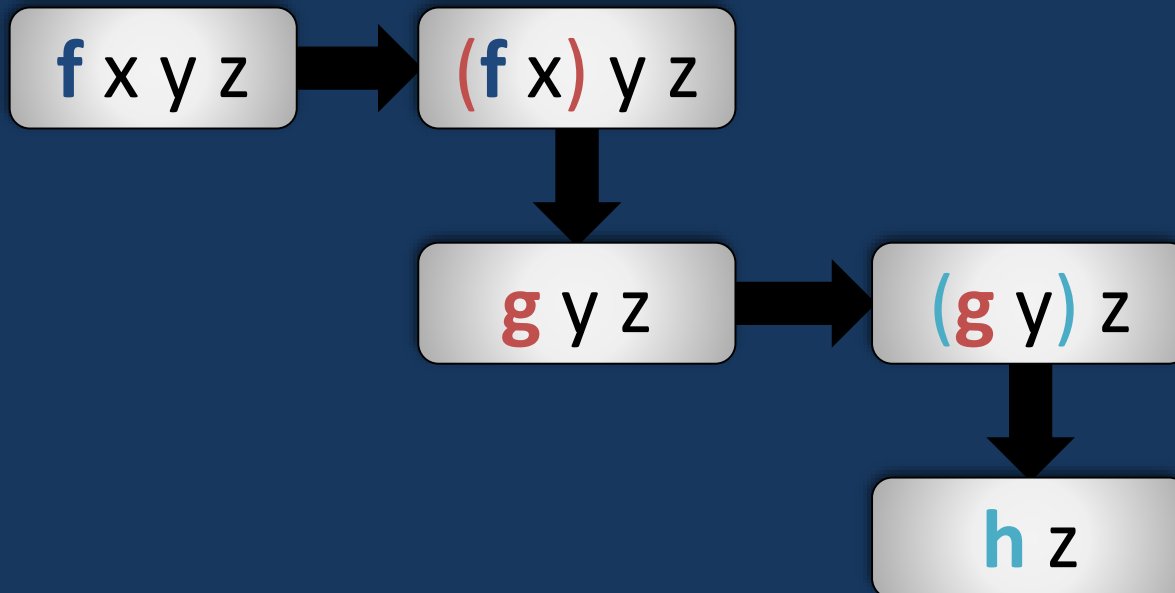
Paradigma Funcional

- ⦿ Funciones como valores de primera clase
 - > Pueden ser **pasadas como argumento**
 - > Pueden ser **retornadas** como resultado de otras funciones
 - > Pueden **combinarse para componer** nuevas funciones

Paradigma Funcional

- **Currying**

- Llamada a función que toma **múltiples argumentos** como una **cadena de llamadas** a funciones de **1 argumento**



Paradigma Funcional

Currying

```
fun add x y = x+y
```

El tipo de la función *curry* es:
 $\text{int} \rightarrow (\text{int} \rightarrow \text{int})$

add 3 5
(add 3) 5
→ g 5
8

- Permite definir nuevas funciones mediante la **aplicación parcial**.

Donde
 $g\ y = 3+y$
 $\text{int} \rightarrow \text{int}$

Paradigma Funcional

- Evaluación perezosa
- Orden de *evaluación no estricto*
 - Se **evalúa** cada parte del **argumento** de una función **solo cuando es indispensable**.
- Donde además, se *recuerdan los valores de los argumentos ya calculados* para evitar su reevaluación (call by need)

ML

ML

*Creado por Robin Milner en los años 70.
Standard ML nació en 1983.*

- ❖ Soporta el **paradigma funcional**
- ❖ **Fuertemente tipado**
- ❖ Sistema de **tipos polimórfico**
- ❖ Tiene soporte para **definir TDAs**
- ❖ **Alcance estático**
- ❖ Mecanismos para el manejo de **excepciones**

Versión

Alice ML 1.4

<http://www.ps.uni-saarland.de/alice/>

>

> *val x=2*3;*

val x:int = 6

Tipos predefinidos

■ Simples

bool, int, string, real, char

true y false (andalso, orelse)

"la casa" ^ "grande"

#"a" #"j"

5, 100, ~2

5.0, 10.23

■ Estructurados (entre otros...)

Listas (secuencia)

Tuplas (producto cartesiano)

Listas

[1, 2]

Tipo: int list

Notacion

- [1,2,3,4]
- Cabeza::Cola
- nil o [] = lista vacia

Tuplas

(1, 2.5, [1,2])

Tipo: int*real*int list

Identificadores, Ligaduras y Declaraciones

Los **identificadores** antes de usarse **deben declararse**

Una variable se declara al **ligarla a un valor** usando la palabra reservada **val**

> **val** $x = 2 * 3$;

ML mantiene un **ambiente** con todas las **ligaduras creadas**

val en realidad liga un nombre a un valor resultante de una expresión

Una función **en general** se declara al **ligarla a su expresión** usando la palabra reservada **fun**

> **fun** $f\ x = x + 1$;

Inferencia de tipos

- ⦿ Los **tipos de las variables no** son (aunque pueden ser) **especificados explícitamente** en las declaraciones.
- ⦿ El **tipo** de una variable es **inferido** a partir del **tipo del valor** al que fue ligada **o por las operaciones** en las que la variable interviene.

*> val x=2*3;*

val x: int = 6

Inferencia de tipos

- El **tipo de una función** se determina de acuerdo al **tipo del argumento** y del **resultado**.

```
> fun f x = x+1;  
   val f : int -> int = fn
```

- El **tipo del resultado se infiere** según el tipo del rango de la **expresión** que lo computa.

Inferencia de tipos

ML tiene **operadores sobrecargados**

No tiene **coerción**

(no soporta expresiones mixtas)

Permite realizar **conversiones explícitas**

2+3.14 es una expresión débilmente tipada
No permitida por ML

Funciones

*> fun doble x = 2*x;*

val doble: int → int = fn

fun nombre argumento = expresion;

Evaluación
Ansiosa

- ⦿ Las **funciones** tienen **siempre UN único argumento**.
- ⦿ El **resultado** de la función es el resultado de **evaluar la expresión** que la define con el **argumento** dado.

Evaluación perezosa: expresión condicional, expresiones handle, andalso, oresle...

Funciones

Las funciones tienen **siempre UN único argumento**.

```
>fun f (x, y) = x+2*y;  
val f : (int * int) -> int = fn
```

Múltiples argumentos → TUPLAS o currying

Funciones curried

```
> fun producto x y = x*y;
```

```
val producto : int -> int -> int = fn
```

Aplicación parcial (principal utilidad)

```
> val porTres = producto 3;
```

```
val porTres : int -> int = fn
```

Son funciones que, por la manera en la que fueron definidas, permiten definir nuevas funciones mediante la aplicación parcial

Funciones

Expresión condicional

> fun f x =

if x < 10 then "menor que 10" else "mayor o igual que 10";
val f: int -> string = fn

Expresión condicional + recursividad

> fun factorial n =

if n = 0 then 1
else n * factorial(n-1);

val factorial: int -> int = fn

Aplicación
Recursiva

Expresión
Condicional

Restricciones de tipo en
las expresiones condicionales

Funciones definidas mediante cláusulas

```
> fun esCero 0 = true  
    | esCero n = false;  
val esCero : int -> bool = fn
```

Definición por **partes de acuerdo al argumento**

DEBEN ESTAR DEFINIDAS PARA TODOS LOS VALORES DEL
DOMINIO (totales).

Se resuelve el flujo de control
mediante “pattern matching”.

Funciones con referencias a *variables* “no locales”

> val c = 2.5;

val c : real = 2.5

> fun multx x = x*c;

val multx : real -> real = fn

Se comporta
como una
constante literal

LF puros: Una función computa distintos valores de acuerdo a su argumento (no olvidar la **semántica de valores y transparencia referencial**).

Misma entrada → misma salida

Funciones polimórficas

Función que se comporta de manera uniforme sobre un **conjunto de tipos**

POLITIPO: tipo que **contiene variables de tipo**

El tipo de una función polimórfica es un politipo:
POLIMORFISMO PARAMÉTRICO

```
> fun invertir(nil) = nil  
  | invertir(x::xs) = invertir xs @ [x];  
val invertir : 'a list -> 'a list = fn
```

Una gran ventaja de ML es que permite que las **funciones definidas por el usuario** puedan ser **polimorfas**. Otros lenguajes con tipos como C++ tienen **operadores polimórficos** predefinidos pero las **funciones definidas por el usuario no pueden serlo**.

Funciones de alto orden (Formas Funcionales)

Funciones que emplean otras funciones
como argumentos y/o resultado

Ejemplo: función *simpleMap*

Toma como **argumento** una función f y una lista L ,
y retorna una lista LR tal que cada elemento $LR_i = f(L_i)$

```
> fun simpleMap(F, nil) = nil  
  | simpleMap(F, h::t) = F(h)::simpleMap(F, t);  
val simpleMap : (('a -> 'b) * 'a list) -> 'b list = fn
```

Funciones de alto orden (Formas Funcionales)

Funciones que emplean otras funciones
como argumentos y/o resultado

Ejemplo: función *simpleMap*

Toma como **argumento** una función f y una lista L ,
y retorna una lista LR tal que cada elemento $LR_i = f(L_i)$

```
> fun simpleMap(F, nil) = nil  
  | simpleMap(F, h::t) = F(h)::simpleMap(F, t);  
val simpleMap : (('a -> 'b) * 'a list) -> 'b list = fn
```

```
MLWorks> simpleMap(par, [3, 5, 6, 10]);  
val it : bool list = [false, false, true, true]
```


Funciones de alto orden (Formas Funcionales)

Funciones que emplean otras funciones como argumentos y/o resultado

Ejemplo: función *simpleMap*

Toma como **argumento** una función f y una lista L , y retorna una lista LR tal que cada elemento $LR_i = f(L_i)$

```
> fun simpleMap(F, nil) = nil  
  | simpleMap(F, h::t) = F(h)::simpleMap(F, t);  
val simpleMap : (('a -> 'b) * 'a list) -> 'b list = fn
```

```
MLWorks> simpleMap(length, [], [1,2,3],[3]);  
val it : int list = [0, 3, 1]
```

Funciones de alto orden (Formas Funcionales)

Funciones que emplean otras funciones como argumentos y/o resultado

Ejemplo: función *simpleMap*

Toma como **argumento** una función f y una lista L , y retorna una lista LR tal que cada elemento $LR_i = f(L_i)$

```
> fun simpleMap(F, nil) = nil  
  | simpleMap(F, h::t) = F(h)::simpleMap(F, t);  
val simpleMap : (('a -> 'b) * 'a list) -> 'b list = fn
```

```
MLWorks> simpleMap(length, [["si"], [], ["hola", "chau"]]);  
val it : int list = [1, 0, 2]
```

Python

Características asociadas
al Paradigma Funcional

Python

- *Python es un lenguaje que solo **admite** el Paradigma Funcional*
- *Tiene herramientas que **facilitan la creación de programas** que siguen los lineamientos del paradigma*
- ***No** cuentan con todas las **características fundamentales** requeridas por el paradigma*
- *A continuación veremos algunas de las características que lo acercan al paradigma*

Python – Funciones como Objetos

- *En Python las funciones son un objeto del lenguaje*
 - Pueden ser el **Argumento** de una Función

¿Es aplicar una
función de alto
orden?

```
>def por2 (X) :  
    return X*2  
  
>def por3 (X) :  
    return X*3  
  
>def aplicar (Funcion, Arg) :  
    return Funcion (Arg)  
  
>aplicar (por2, 4)  
8  
  
>aplicar (por3, 4)  
12
```

Python – Funciones como Objetos

- *En Python las funciones son un objeto del lenguaje*
 - > Pueden ser el **Argumento** de una Función
 - > Pueden ser parte de **Estructuras de Datos**

```
>def por2 (X) :  
    return X*2  
  
>def por3 (X) :  
    return X*3  
  
>Lista = [por2, por3]  
  
>Lista[0] (4)  
8
```

Python – Funciones como Objetos

● *En Python las funciones son un objeto del lenguaje*

- Pueden ser el **Argumento** de una Función
- Pueden ser parte de **Estructuras de Datos**
- Pueden ser la **Salida** de una Función

¿Es darNesimo
una función de
alto orden?

```
>def por2(X):  
    return X*2  
  
>def por3(X):  
    return X*3  
  
>Lista = [por2, por3]  
  
>def darNesimo(Lista, N):  
    return Lista[N]  
  
>darNesimo(Lista, 0) (4)  
8
```

Python – Aplicación Parcial

- En Python como en varios lenguajes funcionales **no** hay **Currying**
- Aun así, el lenguaje cuenta con un mecanismo de **aplicación parcial**
- Para esto es necesario utilizar la función **partial** de la librería **functools**

```
>import functools
>def suma(X,Y):
    return X+Y
>suma3 = functools.partial(suma, Y=3)

>suma3(5)
8
```

Con X esto no funciona
¿Por qué creen que es así?

Python – Conjuntos por Comprensión

- Una noción clásica en matemática es la de definir **conjuntos por comprensión**:

$$S = \{ 2 \cdot x \mid x \in \mathbb{N}, x^2 > 3 \}$$

- Python al igual que varios lenguajes Funcionales permite crear este tipo de estructuras mediante:
- Listas por Comprensión**
- Generadores**

ML no tiene este tipo de constructores!

Python – Listas por Comprensión

- Las **Listas por comprensión** permiten crear **listas finitas** utilizando una notación similar a la de conjuntos por comprensión

$$S = \{ 2 \cdot x \mid x \in \mathbb{N}, x^2 > 3 \}$$

```
> S = [ 2*X for X in range(5) if (X*X > 3) ]  
> S  
[4, 6, 8]
```

¿Por qué tiene que ser Finito?

Python – Expresiones Generadoras

- Python tiene otro mecanismo especial para modelar **estructuras infinitas**, conocidos como **expresiones generadoras**.

$$S = \{ 2 \cdot x \mid x \in \mathbb{N}, x^2 > 3 \}$$

```
> S = ( 2*X for X in Naturales() if (X*X > 3) )
```

¿Que queda almacenado en S?

Basicamente un iterador 😊

¿Como Implementarian Naturales()?

También con un iterador! 😊

Python – Función Generadora

- Las funciones como Naturales() son llamadas **Funciones Generadoras** y permiten crear un iterador básico
- Para esto se valen de la instrucción **yield**
- Es una instrucción de retorno, que a diferencia del return, la **función mantiene el ambiente** (vars locales y punto de ejecución)

```
> def Naturales () :  
    X = 0  
    while (True) :  
        X = X+1  
        yield X
```

Esta Función genera un iterador que permite recorrer todos los números naturales

Referencias:

- ✓ “Apuntes sobre paradigmas de programación”,
Ignacio Ponzoni y Jessica Carballido