Introducción a la programación funcional

Salvador Lucas Alba
Departamento de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia

http://www.dsic.upv.es/users/elp/slucas.html

Objetivos

- Relacionar la noción matemática de **función** y los lenguajes de programación funcionales
- Introducir los **recursos expresivos** básicos de los lenguajes funcionales
- Introducir el **modelo computacional** de los lenguajes funcionales

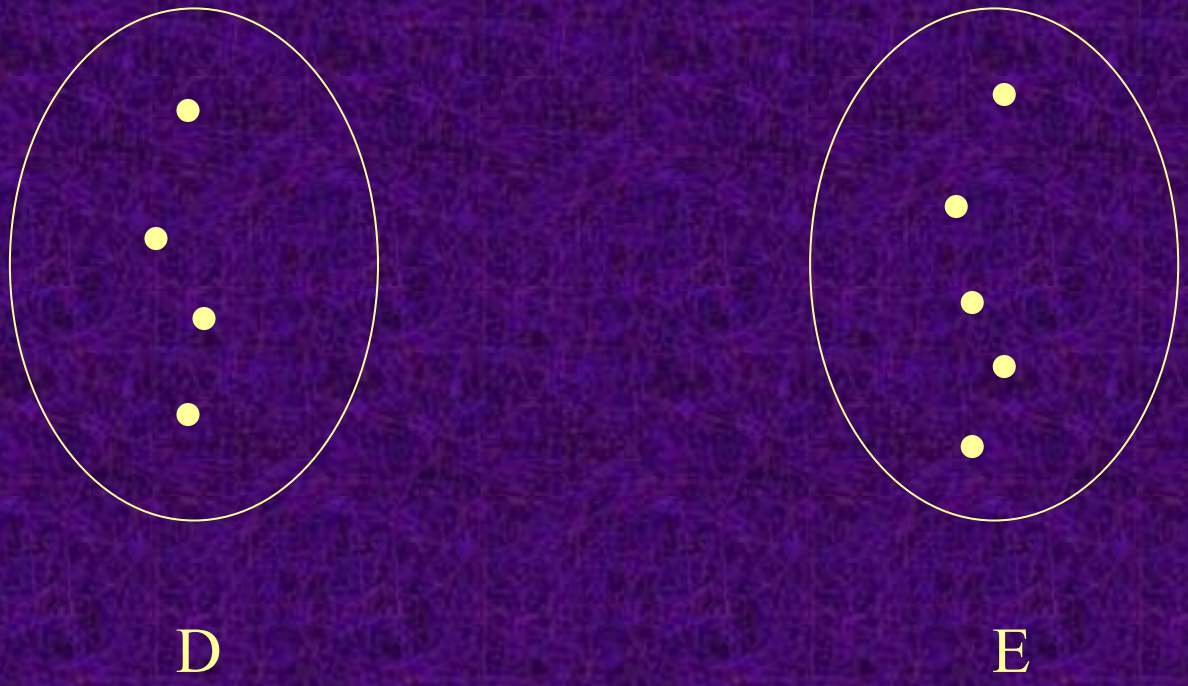
Desarrollo

1. Lenguajes funcionales como lenguajes de programación
2. Historia y evolución de los lenguajes funcionales
3. Ventajas e inconvenientes de los lenguajes funcionales
4. Aplicaciones

Desarrollo

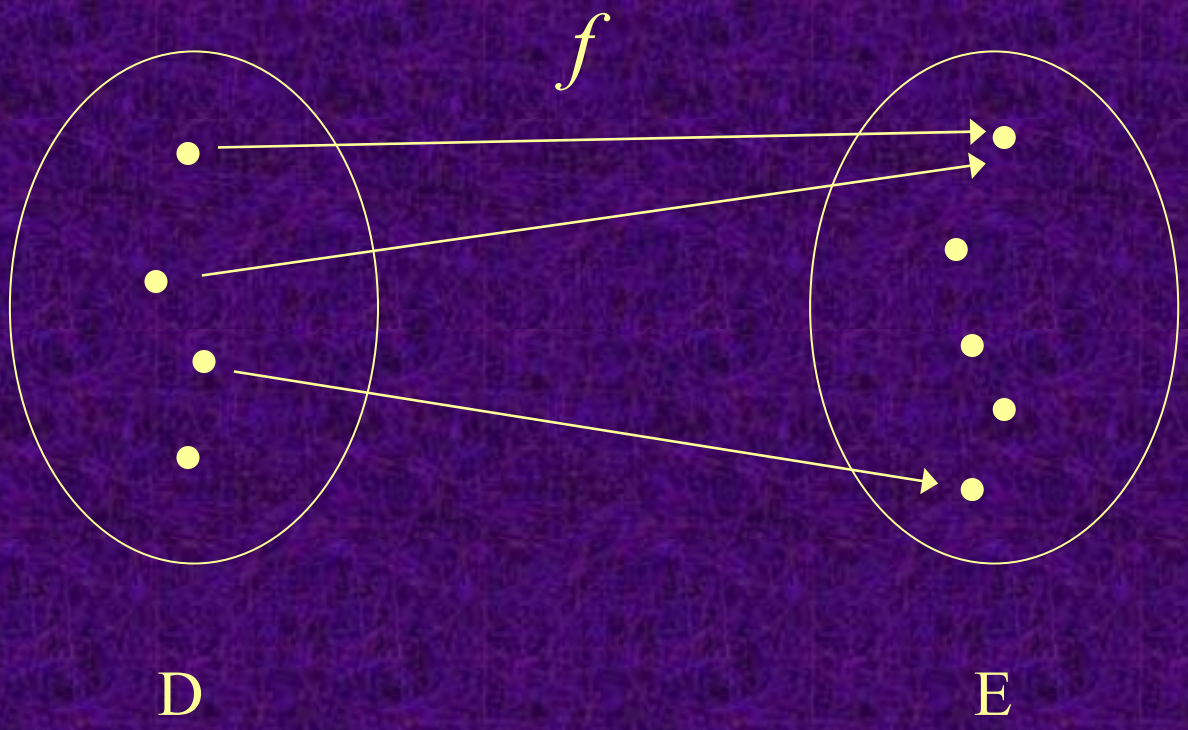
1. Lenguajes funcionales como lenguajes de programación
2. Historia y evolución de los lenguajes funcionales
3. Ventajas e inconvenientes de los lenguajes funcionales
4. Aplicaciones

Concepto de función



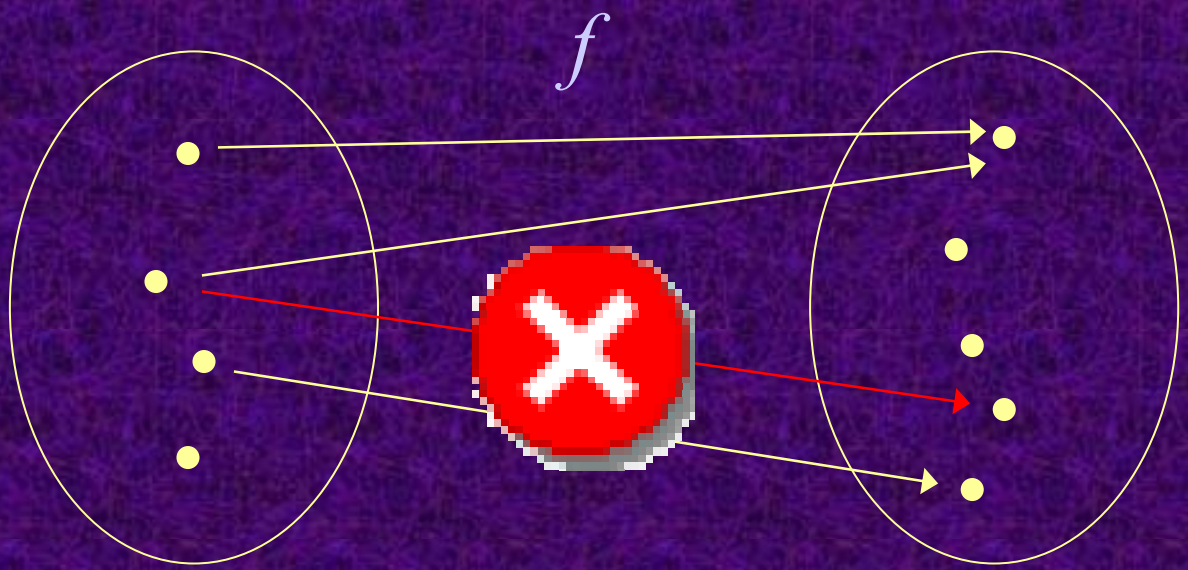
Concepto de función

Función parcial



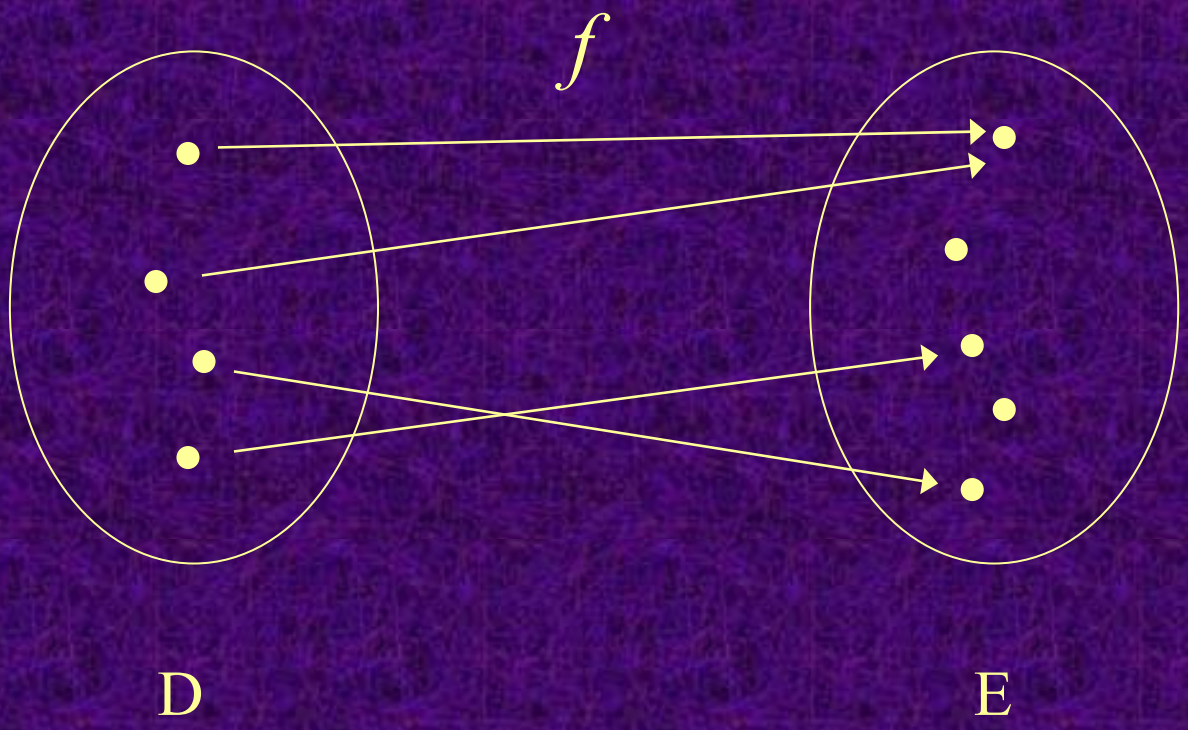
Concepto de función

Función parcial



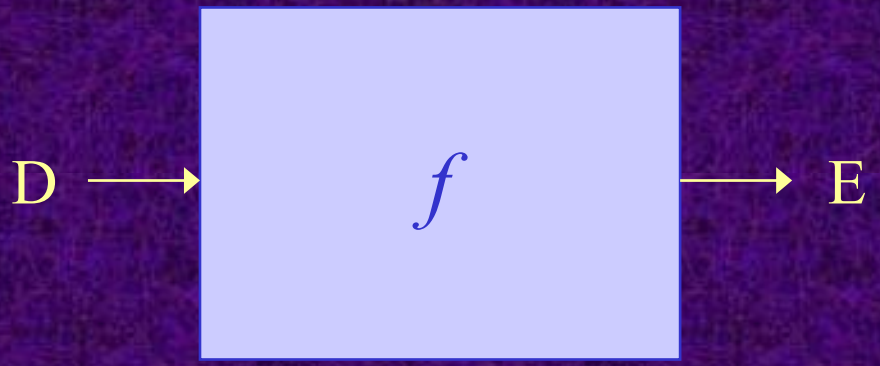
Concepto de función

Función total



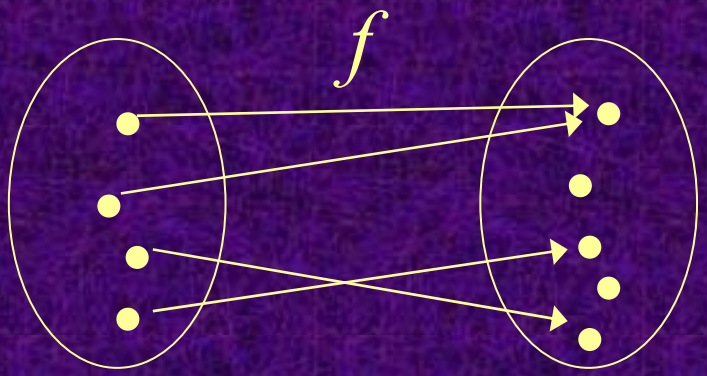
Concepto de función

Función como transformación

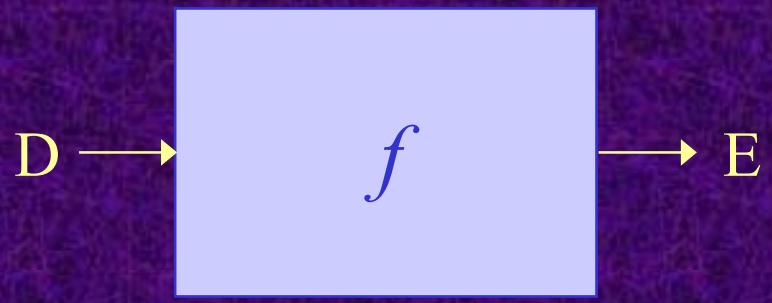


Descripción de una función

Extensional - Intensional



Grafo de la función



Regla de transformación

Descripción de una función

Extensional - Intensional

Ejemplo: sign:Int $\rightarrow$ {neg,cero,pos}

Descripción de una función

Extensional - Intensional

Ejemplo: sign:Int $\rightarrow$ {neg,cero,pos}

Descripción extensional:

{...,(-2,neg),(-1,neg),(0,cero),(1,pos),(2,pos),...}

Descripción de una función

Extensional - Intensional

Ejemplo: sign:Int $\rightarrow$ {neg,cero,pos}

Descripción extensional:

{...,(-2,neg),(-1,neg),(0,cero),(1,pos),(2,pos),...}

Descripción intensional:

$$\text{sign}(x) = \begin{cases} \text{neg} & \text{si } x<0 \\ \text{cero} & \text{si } x=0 \\ \text{pos} & \text{si } x>0 \end{cases}$$

Descripción de una función

Propiedades de interés computacional

- Determinismo
  - Dado un argumento de entrada, una función siempre devuelve el mismo resultado
- Dependencia de los argumentos
  - El resultado devuelto por una función *sólo* depende de sus argumentos de entrada
- Extensionalidad
  - Dos funciones son iguales si y sólo si responden igual ante los mismos argumentos

Descripción de una función

Propiedades de interés computacional

- Llamada a una función:

Función: $f: D \rightarrow E$

Elemento: $d \in D$

$\Big\}$ $f(d) \in E$

Llamada a la función

Hacia un lenguaje funcional

Planteamiento básico

- Resolver problemas de cómputo empleando:
  - definiciones de función para especificar problemas
  - llamadas a función para expresar el proceso de cómputo que resuelve el problema planteado

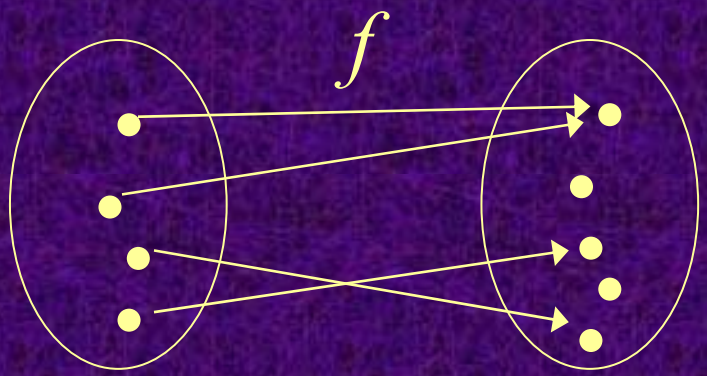
Hacia un lenguaje funcional

Planteamiento básico

- Resolver problemas de cómputo empleando:
  - **definiciones de función para especificar problemas**
  - llamadas a función para expresar el proceso de cómputo que resuelve el problema planteado

Hacia un lenguaje funcional

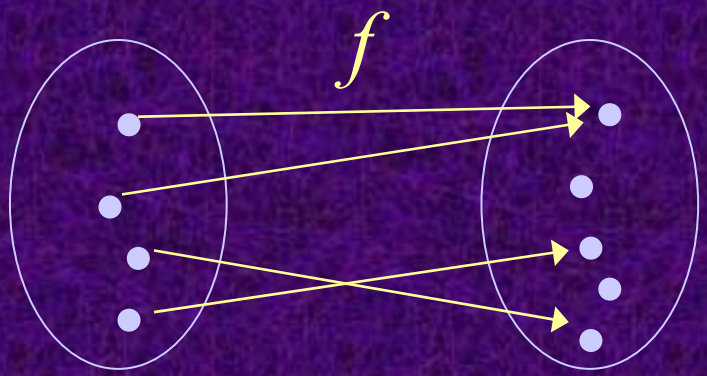
Planteamiento básico



El dominio y el codominio ...

Hacia un lenguaje funcional

Planteamiento básico

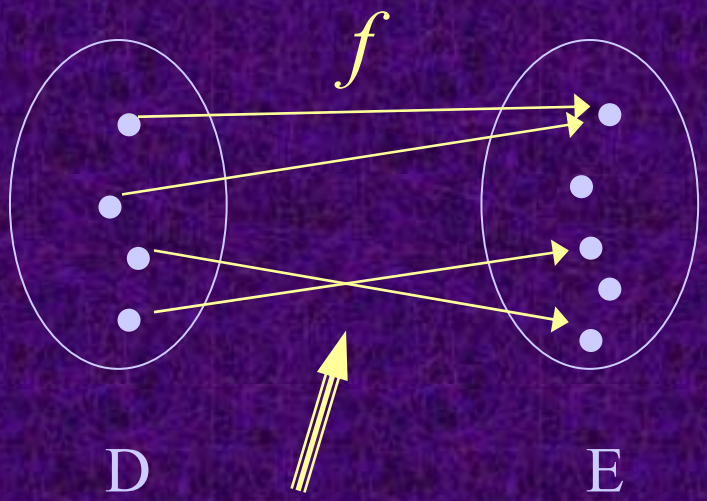


... se describen mediante tipos.

```
type D = ...
type E = ...
```

Hacia un lenguaje funcional

Planteamiento básico

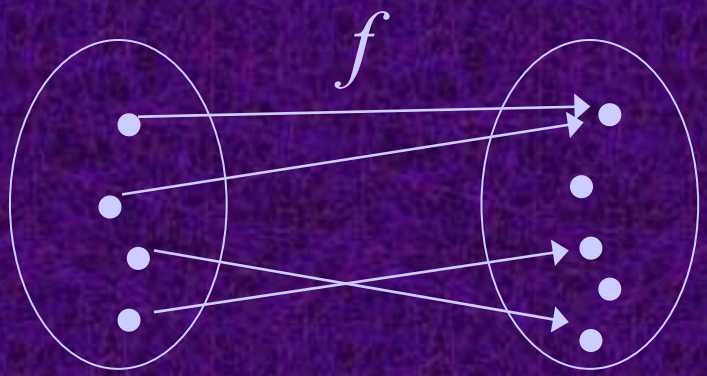


El grafo de la función ...

```
type D = ...
type E = ...
```

Hacia un lenguaje funcional

Planteamiento básico



... mediante ecuaciones.

```
type D = ...
type E = ...
```

```
f x = ...
```

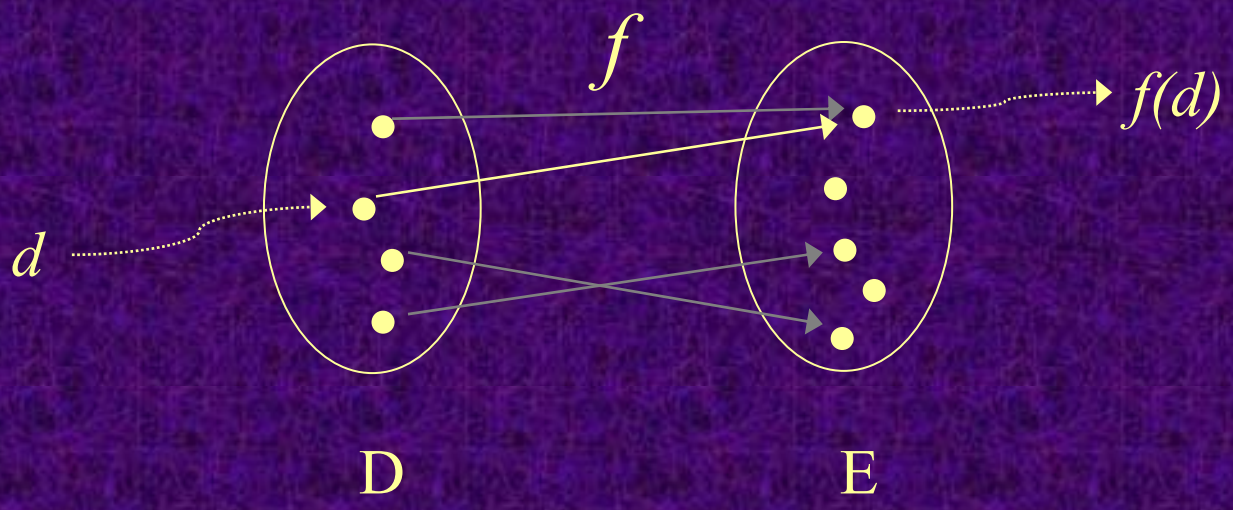
Hacia un lenguaje funcional

Planteamiento básico

- Resolver problemas de cómputo empleando:
  - definiciones de función para especificar problemas
  - **llamadas a función para expresar el proceso de cómputo que resuelve el problema planteado**

Hacia un lenguaje funcional

Planteamiento básico



Las llamadas a función requieren un mecanismo de cómputo

Hacia un lenguaje funcional

Planteamiento básico

- Funciones
  - Dominios y codominios
  - Descripción de una función
  - Llamada a una función
- Lenguajes funcionales
  - Tipos
  - Ecuaciones de definición de funciones
  - Expresiones sintácticas

    +

    Proceso de evaluación por reducción

Hacia un lenguaje funcional

Planteamiento básico

- **Lenguajes funcionales**
  - **Tipos**
  - **Ecuaciones de definición de funciones**
  - **Expresiones sintácticas**

    +

    Proceso de evaluación por reducción

Desarrollo

1. Lenguajes funcionales como lenguajes de programación
2. Historia y evolución de los lenguajes funcionales
3. Ventajas e inconvenientes de los lenguajes funcionales
4. Aplicaciones

El cálculo lambda

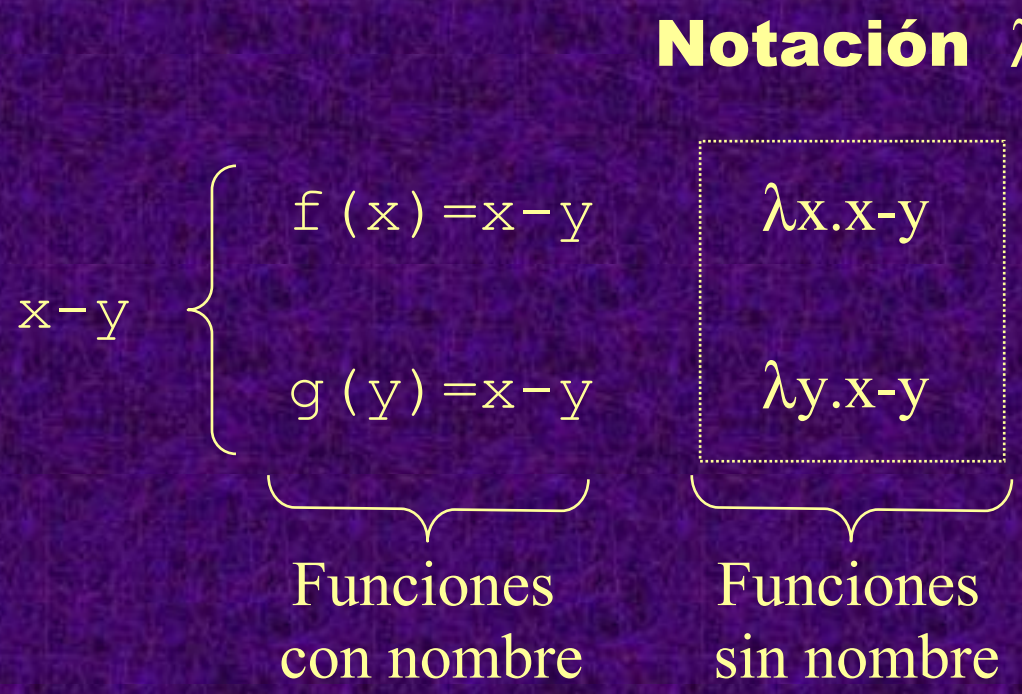
Motivación

- Introducido por A. Church entre 1930 y 1940
- Modelado de las características computacionales **fundamentales** de las **funciones**:
  - Una función *tiene* argumentos o parámetros formales (que *determinan* su resultado)
  - Una función se *aplica* sobre sus parámetros reales (*llamada a función*)
  - Al *evaluar* dicha llamada se obtiene el resultado

El cálculo lambda

Sintaxis

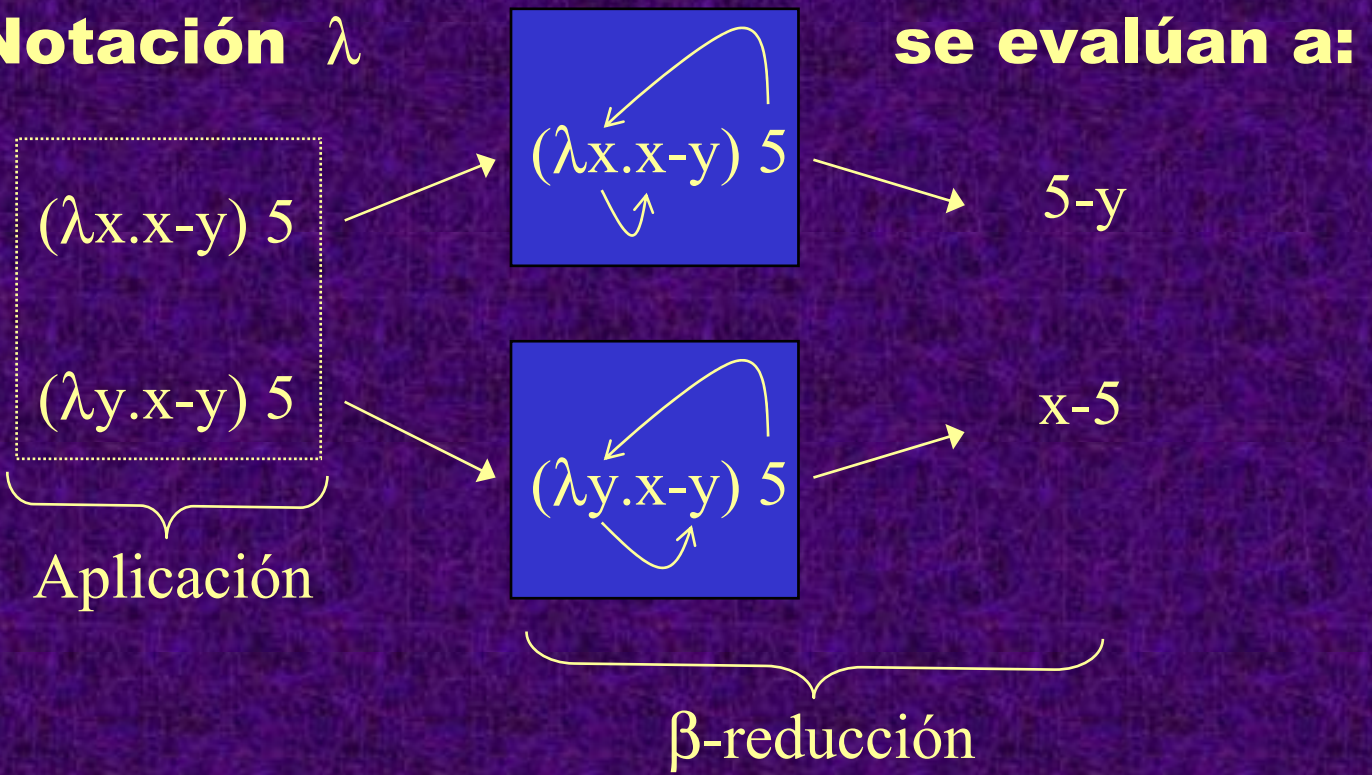
- Ejemplo:



El cálculo lambda

Sintaxis

- Ejemplo:



El cálculo lambda

- El cálculo λ proporciona un lenguaje funcional elemental

Lenguaje funcional = cálculo λ + `azúcar sintáctico´

Lisp

Motivación

- Introducido por J. McCarthy en 1960
- Aportaciones (I):
  - Programas como conjunto de **definiciones de función**
  - Ejecución de un programa como **evaluación** de una expresión inicial respecto a las definiciones
  - Utiliza la evaluación **impaciente** (*eager*) como mecanismo de cómputo

Lisp

Motivación

- Aportaciones (II):
  - Introdujo la **expresión condicional** (por contraposición a la *sentencia condicional*) fomentando su uso en definiciones recursivas
  - Utilización de **listas** y operaciones de **orden superior** sobre listas (*Lis*t *P*rocessing)
  - Introducción del operador de construcción de listas **cons** para representar éstas en el ordenador

Lisp

Sintaxis

- Ejemplo:

```
(define mapcar (fun lst)
    (if (null lst)
        nil
        (cons (fun (car lst))
              (mapcar fun (cdr lst))  )  )
)
```

Nombre

Argumentos

Expresión (condicional)

Constructor de listas

Recursividad

Iswim

Motivación

- Introducido por P. Landin en 1966 (Iswim: *I*f you *s*ee *w*hat *I m*ean)
- Aportaciones (I):
  - Abandono de la sintaxis prefija (para los operadores) a favor de la **infija**
  - Introducción de una norma de estructuración textual de los programas (**layout**) para agrupar componentes mediante sangrado

Iswim

Motivación

- Aportaciones (II):
  - Introducción de las cláusulas **let** y **where** junto con una noción de recursividad mutua y simultánea de las definiciones de función que permiten controlar el **alcance** de los identificadores definidos
  - Introducción de una máquina abstracta (la **SECD**) para compilar lenguajes funcionales

Iswim

Sintaxis

- Ejemplo:

```
e where f x = x
          a b = 1
```

o bien

```
e where {f x = x; a b = 1}
```

es distinto de

```
e where f x = x a
          b   = 1
```

Iswim

Sintaxis

- Ejemplo:

```
let f x = x
      a b = 1
in  e
```

o bien

```
let {f x = x; a b = 1} in e
```

FP

Motivación

- Introducido por P. Backus en 1978 (FP: *F*unctional *P*rogramming)
- Aportaciones:
  - Abogar por el estilo funcional como superador de las **deficiencias** del estilo imperativo
  - Proporcionar un núcleo sintáctico reducido con gran capacidad **combinatoria**

FP

Sintaxis

- Ejemplo:

$$\text{Def IP} = (/ +) \circ (\alpha \times) \circ \text{Trans}$$

es un programa FP que define el producto escalar de un vector. En el programa, / , ° y α son formas combinatorias (*insert*, *compose* y *apply to all*, respectivamente).

ML

Motivación

- Introducido por R. Milner y otros autores en 1978 (ML: *M*eta *L*anguage)
- Aportaciones (I):
  - Incorpora un **sistema de tipos** muy potente e incluye **comprobación estática** de tipos
  - Utiliza **inferencia de tipos** para determinar el tipo de cada expresión, en lugar de requerir declaraciones de tipo explícitas

ML

Motivación

- Aportaciones (II):
  - Permite definir estructuras de datos y funciones **polimórficas**
- ML también utiliza la evaluación **impaciente**

ML

Sintaxis

- Ejemplo:

  – **val** pi = 3.1415927;

  indicaría a un intérprete ML la intención de definir un valor pi. El intérprete acepta la acción y responde

  > **val** pi = 3.1415927: real

  indicando el *tipo* que le corresponde.

ML

Sintaxis

- Ejemplo:

(1) – **fun** sum1to(n) = n*(n+1) div 2;

> **val** sum1to = **fn**: int -> int

(2) – **fun** area r = pi*r*r;

> **val** area = **fn**: real -> real

ML

Sintaxis

- Ejemplo:

(1) – **fun** triple x = 3*x;

> **val** triple = **fn**: int -> int

(2) – **val** triple = **fn** x => 3*x;

> **val** triple = **fn**: int -> int

ML

Sintaxis

- Ejemplo:

(1) – **fun** fst (x,y) = x;

> **val** fst = **fn**: ‘a * ‘b -> ‘a

(2) – **fun** snd (x,y) = y;

> **val** snd = **fn**: ‘a * ‘b -> ‘b

ML

Sintaxis

- Ejemplo: reducción *impaciente*

  sum1to(3*3)

  → sum1to(9)

  → 9*(9+1) div 2

  → 9*10 div 2

  → 90 div 2

  → 45

Hope

Motivación

- Introducido por R. Burstall y otros autores en 1980
- Aportaciones (I):
  - Permite la definición, por parte del usuario, de sus **propios** tipos y estructuras de datos

Hope

Motivación

- Aportaciones (II):
  - Utiliza **ajuste de patrones** para *acceder* a las estructuras de datos
  - Utiliza **ajuste de patrones** para *definir* las funciones del programa

Hope

Sintaxis

- Ejemplo: días laborables y colores

```
data DiasL == lun++mar++mie++jue++vie ;

data Colores == rojo++verde++azul ;
```

Hope

Sintaxis

- Ejemplo: listas de enteros y listas de caracteres

```
data NumList == nil++cons(num#NumList) ;

data CharList == nil++cons(char#CharList) ;
```

Hope

Sintaxis

- Ejemplo: listas de enteros y listas de caracteres

```
data NumList == nil++cons(num#NumList) ;

data CharList == nil++cons(char#CharList) ;
```

Tipos predefinidos

Hope

Sintaxis

- Ejemplo: listas de enteros y listas de caracteres

```
data NumList == nil++cons(num#NumList) ;

data CharList == nil++cons(char#CharList) ;
```

Constructores de tipo

Hope

Sintaxis

- Ejemplo: listas de enteros y listas de caracteres

```
data NumList == nil++cons(num#NumList) ;

data CharList == nil++cons(char#CharList) ;
```

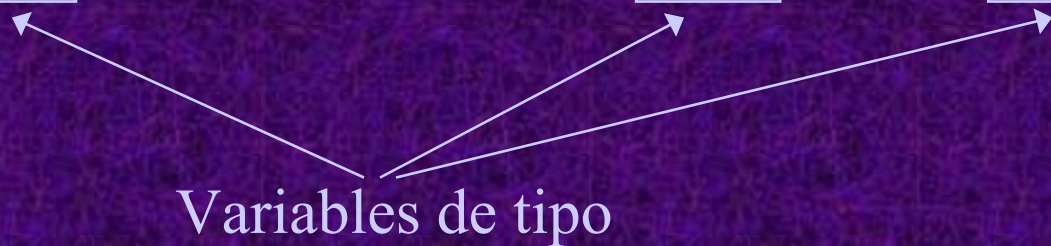
Constructores de datos

Hope

Sintaxis

- Ejemplo: listas *polimórficas*

```
typevar tipo ;
data List(tipo) == nil++cons(tipo#List(tipo))
```



Hope

Sintaxis

- Ejemplo: notación Hope para listas. Equivalente a:

```
infix :: : 7 ;
typevar tipo ;
data List(tipo) == nil++tipo::List(tipo) ;
```

Hope

Sintaxis

- Ejemplo: Las siguientes listas son válidas:

  1::2::nil ‘a’::‘b’::‘c’::nil lun::mie::nil

  pero no estas otras no:

  1::2::3 ‘a’::3::‘c’::nil jue::rojo::nil

Hope

Sintaxis

- Ejemplo: el ajuste de patrones permite acceder a subestructuras: el *patrón*

  1::x

  representa las listas que empiezan con 1.
  ¿Cuál de las siguientes listas *se ajusta* a él?

  2::3::nil 2::1::3::nil 1::nil 1::2::nil

Hope

Sintaxis

- Ejemplo: la variable x del *patrón*

  1::x

  queda ligada a nil y 2::nil, respectivamente, al ajustar las expresiones

  1::nil y 1::2::nil

  a dicho patrón

Hope

Sintaxis

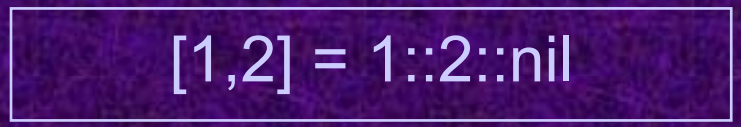
- Ejemplo: el ajuste de patrones puede emplearse para definir funciones:

```
dec Join : list(tipo) # list(tipo) -> list(tipo) ;
--- Join(nil,l) <= l ;
--- Join(x::y,l) <= x::Join(y,l) ;
```

Hope

Sintaxis

- Ejemplo:



Join([1,2],[3])

→ 1::Join([2],[3])

→ 1::2::Join([],[3])

→ [1,2,3]

Hope

Sintaxis

- Ejemplo:

Join([1,2],[3])

→ 1::Join([2],[3])

→ 1::2::Join([],[3])

→ [1,2,3]

Join([1,2],[3]) se ajusta a Join(x::y,l) con x:=1, y:=[2], l:=[3]

SASL - Miranda

Motivación

- SASL y Miranda fueron introducidos por D. Turner en 1976 y 1985, respectivamente
- Aportaciones (I):
  - Utilizan evaluación **perezosa** (*lazy*)
  - Permiten definir funciones mediante ecuaciones con **guardas**
  - Permite utilizar la **currificación**, lo cual facilita la definición de funciones de orden superior

SASL - Miranda

Motivación

- Aportaciones (II):
  - En Miranda se introducen facilidades para la definición de listas: **listas ZF** (*comprehension lists*)
  - En Miranda se permite el uso de **secciones**

SASL - Miranda

Sintaxis

- Ejemplo:

```
sum1to::int -> int -> int
sum1to n = n*(n+1) div 2
```

SASL - Miranda

Sintaxis

- Ejemplo: reducción *perezosa*

  sum1to(3*3)

  → (3*3)*((3*3) +1) div 2

  → 9*((3*3) +1) div 2

  → 9*(9+1) div 2

  → 9*10 div 2

  → 90 div 2

  → 45

SASL - Miranda

Sintaxis

- Ejemplo: utilización de guardas

```
max:: num -> num -> num
max a b = a, if a>b
        = b, otherwise
```

SASL - Miranda

Sintaxis

- Ejemplo: currificación y orden superior

```
map f []       = []
map f (x:xs) = (f x):map f xs
```

SASL - Miranda

Sintaxis

- Ejemplo: listas ZF

  Lista de enteros y sus cuadrados:

  ```
  sqlist = [ (n,n*n) | n<- [1..] ]
  ```

  Lista de enteros y sus cubos:

  ```
  cubes = [ (x*y) | (x,y) <- sqlist ]
  ```

  Producto de enteros:

  ```
  [ (x,y) | x,y<-[1..10] ]
  ```

SASL - Miranda

Sintaxis

- Ejemplo: Secciones. Las declaraciones:

```
doble x = 2*x
doble   = (2 *)
doble   = (* 2)
```

definen la misma función

Haskell

- Haskell fue introducido por P. Hudak y P. Wadler en 1988
- Es un lenguaje perezoso que recoge lo mejor de las propuestas anteriores

Otros lenguajes

- Erlang es un lenguaje funcional paralelo desarrollado por *Ericsson*
- Clean es otro lenguaje funcional paralelo desarrollado por investigadores de la Universidad de Nimega (Holanda)

Desarrollo

1. Lenguajes funcionales como lenguajes de programación
2. Historia y evolución de los lenguajes funcionales
3. Ventajas e inconvenientes de los lenguajes funcionales
4. Aplicaciones

Ventajas de los LF

- Proporcionan muchos recursos expresivos ausentes en otros LP:
  1. Funciones como ‘ciudadanos de 1ª clase’
  2. Sistema de tipos: polimorfismo, inferencia
  3. Ausencia de efectos laterales
  4. Ajuste de patrones para acceder a datos
  5. Evaluación perezosa

Ventajas de los LF

- Los programas funcionales son pequeños y fáciles de mantener
- Los programas funcionales permiten utilizar técnicas de razonamiento formal
- La capacidad computacional de los lenguajes funcionales es equivalente a la de la máquina de Turing

Inconvenientes de los LF

- Eficiencia: la arquitectura von Neumann no es la más indicada para implementar cómputos funcionales
- Algunos algoritmos son de carácter imperativo y no se ajustan bien al estilo funcional

Desarrollo

1. Lenguajes funcionales como lenguajes de programación
2. Historia y evolución de los lenguajes funcionales
3. Ventajas e inconvenientes de los lenguajes funcionales
4. Aplicaciones

Aplicaciones

- Prototipado en Haskell [HJ93]
- *Erlang* y la programación de sistemas de telecomunicaciones y telefonía
- Especialización de programas Haskell para la asignación de tripulaciones a vuelos [Aug97]

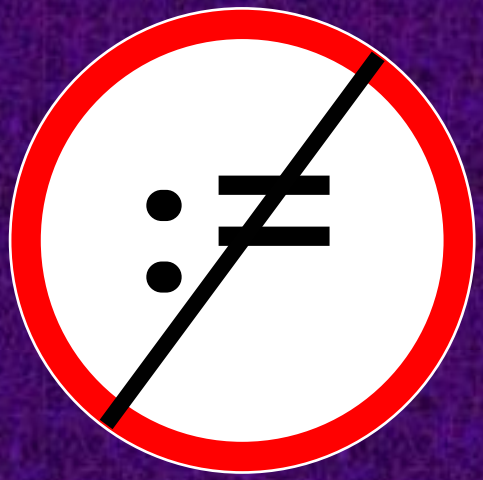
Bibliografía

[Hud89] P. Hudak. Conception, Evolution, and Application of Functional Programming Languages. *ACM Computing Surveys* 21(3):359-411, 1989

[PE93] R. Plasmeijer and M. van Eekelen. Functional Programming and Parallel Graph Rewriting. Addison-Wesley, Reading, MA, 1993

[Rea93] C. Reade. Elements of Functional Programming. Addison-Wesley, Reading, MA, 1993


λ


:=