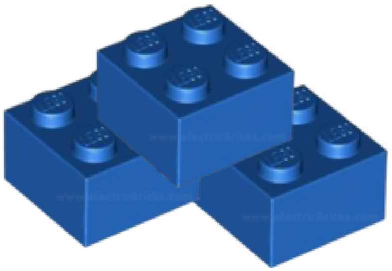
Bases Formales de la Programación Funcional

- Definición de Programación funcional
- Definición de función, valor, dominio, rango y evaluación
- Transparencia Referencial
- Funciones de Orden Superior
- Polimorfismo
- Estructuras de datos infinitas
- Lenguajes fuertemente tipados frente a lenguajes no tipados
- Paso por valor de argumentos.

Programación Funcional

- Características Generales
  - En cierto modo declarativa:
    - (P.L.: Lenguaje → Lógica, Elemento constructivo → Relación)
    - P.F.: Lenguaje → Matemático, Elemento constructivo → Función
  - Se centra en: *la Evaluación*
    - Deja de lado: *Resultado* y *Transferencia de datos*
  - No presenta asignaciones
    - No hay efectos laterales
    - Razonamiento sobre comportamiento y corrección mucho más sencillo
  - Las variables
    - Constantes con valor aún no fijado
  - Funciones fácilmente paralelizables
    - Transparencia Referencial



Programación Funcional

- Conceptos fundamentales

Datos Simbólicos
- Gran flexibilidad
- Listas y Átomos
- *List Processing (Técnica de programación)*

Composición Funcional
- Función como principal bloque constructivo
- Definición por composición de funciones
- Sin asignaciones (evaluación vs resultado/transferencia)

Recursión (en lugar de iteración)
- Diseños más elegantes, cortos y más fáciles de mantener

Funciones de Orden Superior
- Funciones como argumento y resultado de otras funciones
- Modificar y Parametrizar tanto datos como código

Lenguajes Funcionales

- LISP (*List Processing*)
  - Primer lenguaje funcional - John McCarthy (1958).
  - Lenguaje Híbrido (Funcional / Imperativo)
  - Nuevo *estilo de programación*
    - Los programas son más simples, elegantes, y cortos
    - Las transferencias de datos se reducen al mínimo
      - Máquina de Von Neumann no es la más adecuada
    - Gran variedad de tipos de datos
      - Las funciones son un tipo mas de datos

Expresiones Lambda
C++
(v. 11 – C++4.8/VS2012)
Java
(v. 8 - 2014)
C#
(v. 3.0 – 2007)
Javascript, Objective C, Python, Perl, PHP,…

Lenguajes Funcionales

- Scheme
  - Derivado de LISP
  - Características híbridas → se centra en las funcionales
  - Mucho más simple
  - Fácilmente implementable / empotrable / ampliable

Gimp
Macro Language:
Scheme

TinyScheme

- Lenguajes Funcionales Puros
  - No presentan características imperativas ni efectos laterales.
  - Los principales:

HOPE (1980) Miranda o LML (1984) Concurrent Clean (1991)

Haskell (1992)

Wiki:List_of_programming_languages_by_type→Functional_languages

Funciones y su Representación

- Definición de Función:

"Una <u>regla de correspondencia</u> que asocia a cada miembro de un <u>dominio</u> dado un único miembro en un <u>rango</u> dado"

- Tipos de Función:
  - **Parcial** sobre su dominio, si hay al menos un valor en su dominio para el cual el valor del rango correspondiente está indefinido.
  - **Total** sobre su dominio, si está definida para todo él.
  - **Estricta,** si la imagen de un valor indefinido del dominio es indefinido.
- Representación:

F::DomX1 × DomX2 ×…× DomXn → Rango

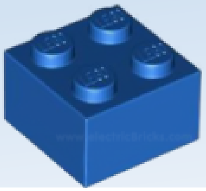
  - Regla de evaluación

$F(X_1, X_2,\ldots,X_n)$ ::= Expresión

Se aplican los valores del dominio a los argumentos y se calcula el valor del rango cuando se evalúa la función

Se define como

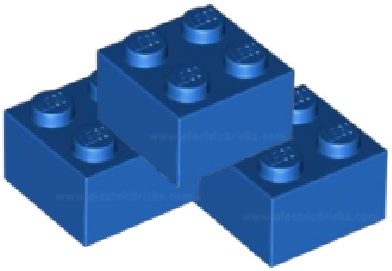
Funciones y su Representación



- Construcción por Composición Funcional

  Una expresión puede contener:

  - Nombres de otras funciones
  - El nombre de la función actual (recursión directa)
  - Expresiones condicionales del tipo SI-ENTONCES-SINO



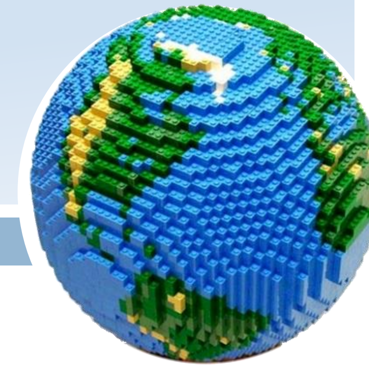
```
masuno(X)  ::= suma(X,1)

masdos(X)  ::= masuno(masuno(X))

factorial(X)  ::= si X = 0
                     entonces 1
                     sino X*factorial(X-1)
                   finsi
```

RECURSIÓN

Programas en Programación Funcional



La Función es un tipo de programa

Entrada: Valores del dominio que se aplican a los argumentos

Función f

Salida: Valor del rango que se retorna como resultado

Función f'

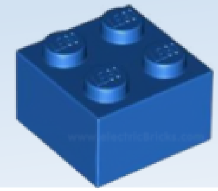
Función f''

Programa: regla que permite obtener el valor correspondiente del rango

No hay variables → No hay Asignación de resultados → Sólo Evaluación

Programa en P.F.: Un conjunto de definiciones funcionales

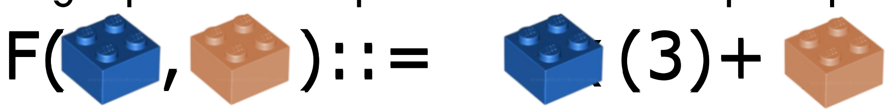
Programas en Programación Funcional



- Característica esencial: La Transparencia Referencial

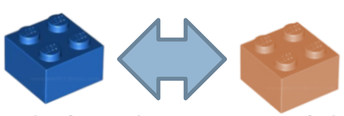
  "El valor de una función está determinado únicamente por el valor de sus argumentos"

- Toda expresión en un lenguaje funcional puro obedece este principio.



F(add1,5) => add1(3)+ 5 => 9

- Ventajas
  - MODULARIDAD: Cualquier función puede reemplazarse por otras que retornen el mismo conjunto de valores.
  - Las funciones pueden tratarse como otro objeto del lenguaje

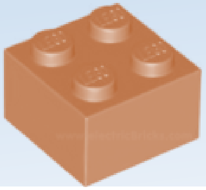


- Inconvenientes
  - La evaluación de una función supone SIEMPRE la generación de objetos nuevos.
  - Mayor tiempo de ejecución (Máq. Von Neumann "ineficiente").

Prog. Funcional vs Prog. Imperativa

Caract. / Progr.	Imperativo	Funcional
Tipo Datos	Numéricos	Simbólicos
Resolución	Secuencia de acciones	Función aplicada a sus argumentos
Definición	Iteración y asignación	Recursión y Func. de Orden Superior
Asignaciones	Gran número de transferencias de datos	Se reducen al mínimo las transferencias
Representación	Programas y datos sin nada en común	Misma representación (Scheme listas multinivel)

Semántica del Lenguaje (SCHEME)



- Datos simbólicos
  - Datos más genéricos que en Programación Imperativa
  - Datos Simples: Átomos
  - Datos (Simples y) Compuestos: S-Expresiones

- Átomos
  - Concatenación INDIVISIBLE de caracteres símbolo y alfanuméricos
  - Simbólicos: abcdef Hola hOLa op# ->> *<>*
    - Cadena alfanumérica
    - Operación comparación de igualdad
  - Numéricos: 356 –12 89.765
    - Cadena de dígitos, opcionalmente: precedido por el signo +o-, '.' decimal
    - Operaciones aritméticas y relacionales

Átomos booleanos (Scheme):

#t	true (valor de verdad)
#f	false (valor de falsedad)

S-expresiones y Listas

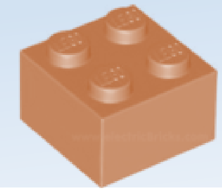
- S-expresión
  - Un átomo es una S-expresión
  - Si x e y son S-expresiones entonces el par (x . y) es una S-expresión s.

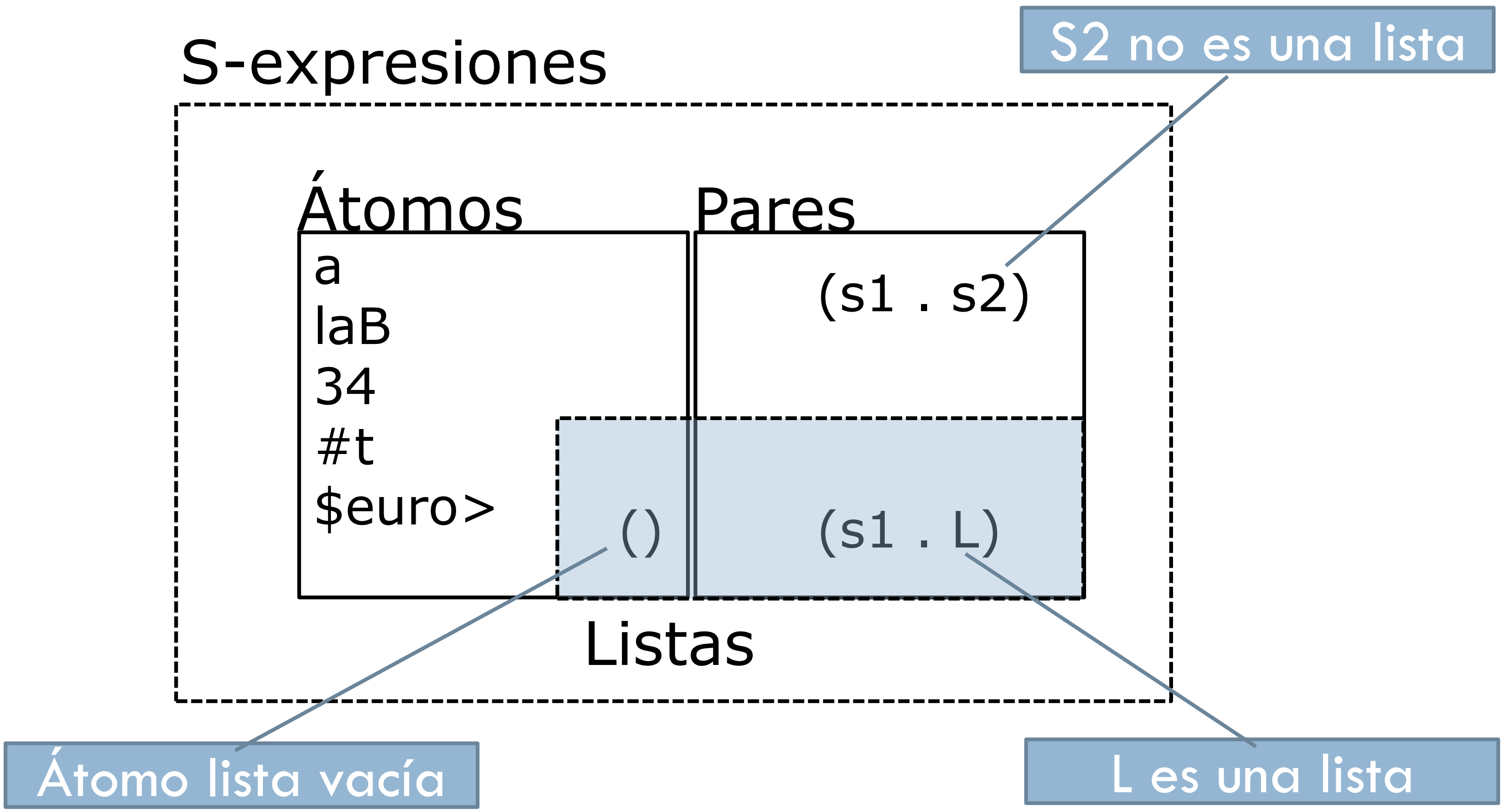
  s=(x . y) : x: *car de s* y: *cdr de s*

- Lista:
  - Definición: “Una secuencia de S-expresiones, que puede ser vacía, encerrada entre paréntesis, es una S-expresión llamada lista”
  - Subconjunto de las S-expresiones.
  - Definición Recursiva:
    - () : la lista vacía es una lista
    - (x . y) : el par es una lista si ‘y’ es una lista

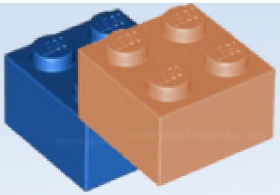
  (ejemplo de lista) (lista que contiene (esta otra lista)) ()

¿S-Expresiones, Átomos, Listas?

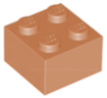




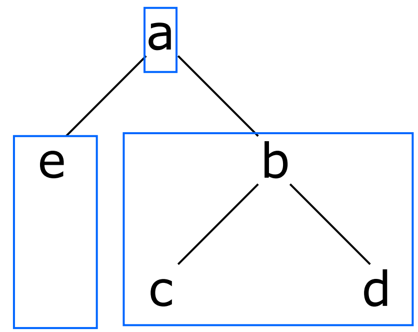
¿Qué podemos expresar con listas?



- Representan estructuras de datos complejas y heterogéneas
  - Árboles y Grafos



`(a (e () ()) (b (c () ()) (d () ())))`



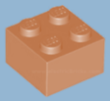
  - Expresiones algebraicas

`x+y ≡ (+ x y)`

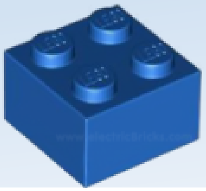
`2*x+3*y ≡ (+ (*2 x) (* 3 y))`

  - Funciones

`f(x,g(y,z),h(a)) ≡ (f x (g y z) (h a))`

(+ 5 7) ¿  o  ? (+ 5 7)→12 '(+ 5 7)→(5 7)

Funciones básicas sobre S-exprs/listas



- **`cons(x,y) ::=`** el par (x□.□y) □=espacio en blanco

  Función <u>Totalmente</u> Definida sobre las S-exp.
  (Inserta el elemento x al inicio de la lista y)

- **`car(x) ::=`** el primer elemento de la S-expresión x

  (el primer elemento de la lista x)

  Función <u>Parcialmente</u> Definida sobre las S-Expresiones

- **`cdr(x) ::=`** el segundo elemento de la S-expresión x

  (la lista x sin el primer elemento)

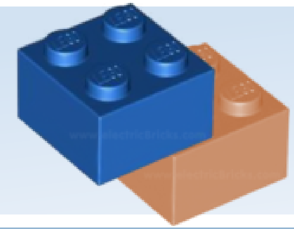
  Función <u>Parcialmente</u> Definida sobre las S-Expresiones

  Por tanto:

  `car(cons(x,y)) ≡ car((x . y)) ≡ x`

  `cdr(cons(x,y)) ≡ cdr((x . y)) ≡ y`

Construcción de Listas y S-expresiones



- Formas de expresar listas (lógicamente equivalentes)

Lista abstracta	**Construcción**	**Lista funcional**
vacía	átomo predefinido	()
[b]	cons(b , ()) ≡ (b . ())	(b)
[a, b]	cons(a , (b)) ≡ (a . (b . ()))	(a b)
	cons(a , cons(b , ()))) ≡ (a . (b . ()))	

- S-expresiones que NO son Listas
  - Cualquier átomo que no sea ()

    hola 5 #paz $amor$

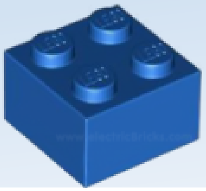
  - Todo par (X . **Y**) donde **Y** NO sea lista

    Ejemplo: PAR (a . b) ≠ LISTA (a b)

    car((a . b)) ≡ a   car((a b)) ≡ a

    cdr((a . b)) ≡ b   cdr((a b)) ≡ (b)

Funciones básicas (y II): predicados de tipo



- **`atom?(x)`** ::= cierto si x es un átomo y falso en otro caso. Función Total
- **`list?(x)`** ::= cierto si x es una lista y falso en otro caso. Función Total

```
list?(a) = falso
list?((a b)) = cierto
list?((a . b)) = falso
```

- **`pair?(x)`** ::= cierto si x es un par y falso en otro caso. Función Total

```
pair?(a) = falso
pair?((a . b)) = cierto
pair?( (a) ) = cierto
```

```
pair?((a b)) = cierto
pair?( () ) = falso
```

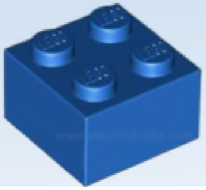
- **`eq?(x,y)`** ::= cierto si x e y son **átomos iguales** y falso en otro caso. Función Total
- **`null?(x)`** ::= cierto si x es la lista vacía y falso en otro caso. Función Total

- **Otras Funciones Básicas Predefinidas:** Operaciones aritméticas, lógicas y relacionales
- (+ …) (and …) (or …) (not …) (> …) (<= …) (<> …)

¿Cómo resolver problemas en Programación Funcional?

- Opción Básica: Recursión
  - La recursión como patrón de resolución
    - Análisis de casos: BASE y RECURRENCIA=Hipótesis+Tesis
  - Ventajas e Inconvenientes (vs Iteración)
    - Soluciones más compactas → más legibles
    - Sin necesidad de variables → menos posibilidad de errores
    - Inconveniente: curva de aprendizaje
    - Limitaciones de memoria de pila
      - Salvo optimización por recursión final
  - Cuando Utilizarla
    - No hay otra opción más simple/eficiente
      - ¿Composición funcional y/o F.O.S.?
    - Problemas de naturaleza recursiva [Listas y S-Expresiones]

Funciones recursivas sobre Listas



- Definición recurrente del dominio
  - Base: El átomo () es una lista
  - Recurrencia: Si S es una S-expresión y L es una lista, cons(S , L) es una nueva lista L2 tal que:

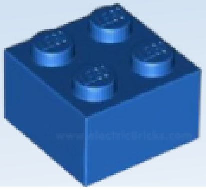
    car(L2) = S
    y
    cdr(L2) = L

- Definición recurrente de una función f(L)
  - Base: conocido f(L),
    donde L es ()
    ~~donde el car(L) o/y car(cdr(L)) o/y… cumplen una condición~~
  - Recurrencia:

    Hipótesis: conocido f(cdr(L)), L es una lista

    Tesis: f(L) como resultado de combinar car(L) y f(cdr(L))

Definición de longitud de una Lista



- **`length(x)  ::=`** la longitud de la Lista x

  Base: **`length( () )`** = 0

  Recurrencia:

CAR(X)	CDR(X)

  Hipótesis: conocido **`length(cdr(x))`**

  Tesis: **`length(x) = 1 + length(cdr(x))`**

Definición de longitud de una Lista (y II)


- ```
  length(x) ::=

  si list?(x)
        entonces aux-length(x)
  sino
        error
  fsi
  ```

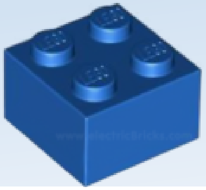
```
aux-length(x)::=

   si null?(x)
   entonces 0
   sino 1 + aux-length(cdr(x))
   fsi
```

```
(define (length x)
  (if (list? x) (length-aux x)
      (error "x no es una lista")))

(define (length-aux x)
  (if (null? x) 0 (+ 1 (length-aux (cdr x)))))
```
```

Definición de member( )



- **`member(x,y)`** **`::=`** la cola de la lista **`y`** comenzando por la primera ocurrencia del elemento **`x`**, o falso si **`x`** no pertenece a **`y`**

  Base: **`member(x,()) = #f`**

  Recurrencia:

CAR(Y)	CDR(Y)

  Hipótesis: **`member(x, cdr(y))`** es conocido

  Tesis:

  ```
  si car(y)=x
  entonces y
  sino member(x,cdr(y))
  ```

Implementación de member( )

- `member(x,y) ::=`

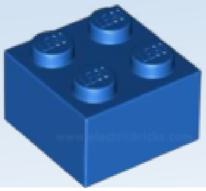
```
si list?(y)
      entonces aux-member(x,y)
sino
      error("argumentos erróneos")
fsi

aux-member(x,y) ::=

si null?(y)
      entonces #f
sino
      si equal?(x,car(y))
      entonces y
      sino aux-member(x,cdr(y))
      fsi
fsi
```

```
(define (aux-member x y)
  (if (null? y) #f
      (if (equal? x (car y)) y
          (aux-member x (cdr y)))))
```

Definición de reverse( )



- **`reverse(x)`** **`::=`** la lista invertida de x

  Base: **`reverse(()) = ()`**

  Recurrencia:

CAR(X)	CDR(X)

  Hipótesis: **`reverse(cdr(x))`** es conocido

  Tesis:

  `reverse(x)=`

REVERSE(CDR(X))	CAR(X)

Implementación de reverse( )


- ```
  reverse(x) ::=

  si list?(x)
      entonces aux-reverse(x)
  sino
      error("argumento erróneo")
  fsi
  ```

```
aux-reverse(x) ::=

  si null?(x)
      entonces x
  sino
      append(aux-reverse(cdr(x)), list(car(x)))
  fsi

list(x)::= cons(x,())
```

```
(define (reverse x)
  (if (null? x) x
      (append (reverse (cdr x)) (list(car x)))))
```
```

Funciones recursivas sobre S-Expresiones

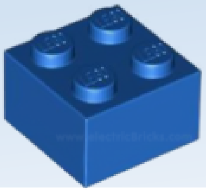
- Definición recurrente del dominio
  - Base: Un átomo es una S-expresión
  - Recurrencia: Si s1 y s2 son S-expresiones, cons(s1,s2) es una nueva S-expresión s tal que:

    car(s) = s1
    y
    cdr(s) = s2

- Definición recurrente de una función f(s)
  - Base: conocido f(s), donde s es un átomo.
  - Recurrencia: s es S-exp. no atómica s = cons(car(s) , cdr(s))
    Hipótesis: conocido f(car(s))
    y
    conocido f(cdr(s))

    Tesis: f(s) = resultado de combinar f(car(s)) y f(cdr(s))

Funciones recursivas



- **`equal?(x,y) ::=`** cierto si x e y son dos S-expresiones iguales

```
Base: Si atom?(x) ∧ atom?(y)
          Entonces eq?(x,y)
       Sino
          Si atom?(x) ∨ atom?(y)
          Entonces falso
```

Recurrencia:

CAR(X)	CDR(X)
CAR(Y)	CDR(Y)

Hipótesis: conocido **`equal?(car(x),car(y))`** y **`equal?(cdr(x),cdr(y))`**

Tesis: **`equal?(x,y)`** = **`equal?(car(x),car(y))`** ∧ **`equal?(cdr(x),cdr(y))`**

Funciones recursivas (y II)

- equal?(x,y) ::=

```
si atom?(x) ∧ atom?(y)
entonces
      eq?(x,y)
sino
      si atom?(x) ∨ atom?(y)
      entonces
                falso
      sino
                equal?(car(x),car(y)) ∧ equal?(cdr(x),cdr(y))
      fsi
fsi
```

```
(define (equal? x y)
  (if (and (atom? x) (atom? y)) (eq? x y)
      (if (or (atom? x) (atom? y)) #f
          (and (equal? (car x)(car y)) (equal? (cdr x)(cdr y))))))
```

Definición de palindromo( )

- **`palindromo(x) ::=`** cierto si la lista x es palíndromo, falso en otro caso

  `palindromo(x) ::= equal?(reverse(x),x)`