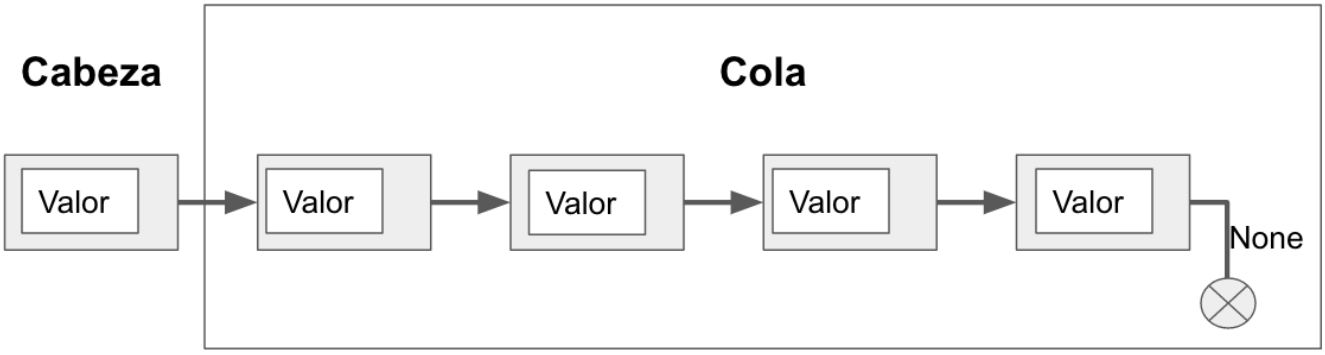
Mini-repaso

Listas



Ej:

```
# contiene : str lista(str) -> bool
# Determina si lista contiene el string s
# ejemplo contiene('auto', crearLista('auto', listaVacia)) retorna True
def contiene(s, unaLista):
    if vacia(unaLista):
        return False
    else:
        if cabeza(unaLista) == s:
            return True
        else:
            return contiene(s, cola(unaLista))
```

Abstracciones funcionales

1. Filtro: Filtrar algunos elementos de la lista y dejarlos en una nueva lista

```
# filtro: lista(any) (any any->bool) any -> lista(any)
# Devuelve lista con valores de L para los que comparacion con x es True
# ej:filtro(lista(5,lista(4,None)),menorQue,5)->lista(4,None)

def filtro(operador,unaLista,n):
...

>>> filtro(menorQue, L, 3)
```

2. Mapa: Hacer algo a cada elemento de la lista y ponerlos en una nueva lista

```
# mapa : (X -> Y) lista(X) -> lista(Y)
# devuelve lista con funcion aplicada a todos sus elementos

def mapa(f, unaLista):
```

```
...

>>> mapa(aString,L) # crea una nueva lista con strings de fracciones
```

3. Reducir (Fold): Convertir los elementos de una lista en un solo valor

```
# fold: (X X -> X) X lista(X) -> X
# procesa la lista con la funcion f y devuelve un unico valor
# el valor init se usa como valor inicial para procesar el primer
# valor de la lista y como acumulador para los resultados
# parciales
# pre-condicion: la lista debe tener al menos un valor

def fold(f, init, unaLista):
...

>>> fold(multiplicar, 1, unaLista)
```

Revisar Pauta Ejercicio 7

Funciones anónimas

Para evitar el tener que estar definiendo funciones auxiliares que luego son utilizadas sólo como parámetro de las funciones **`filtro`**, **`mapa`**, o **`fold`**, es posible definir **funciones anónimas**. Estas funciones tienen una declaración muy compacta en el código y son funciones "sin nombre", ya que están pensadas como funciones que se utilizan **una única vez**.

Para definir funciones anónimas en Python se utiliza la instrucción **`lambda`**. La sintáxis es la siguiente:

```
lambda id1, id2, ...: expresion
```

Los identificadores **`id1, id, ...`** corresponde a los parámetros de la función anónima, y **`expresion`** es la expresión que evalúa la función (y devuelve el resultado). Por ejemplo, una función anónima que suma dos valores se implementa como:

```
lambda x,y: x + y
```

Una función anónima booleana que verifica si un número es menor que 5 se implementa como:

```
lambda x: x < 5
```

Recuerde que las funciones anónimas están pensadas como funciones que se utilizan una sola vez y luego se desechan. Si la función debe invocarse más de una vez, debe definirse de la manera usual siguiendo la receta de diseño **y no declarar dos veces la misma función anónima**, ya que esto correspondería a duplicación de código, lo que es una mala práctica de programación.

Ej:

```
from abstraccion import *
def sumarValoresLista(unaLista):
```

```
  return fold(lambda x,y: x + y, 0, unaLista)
```

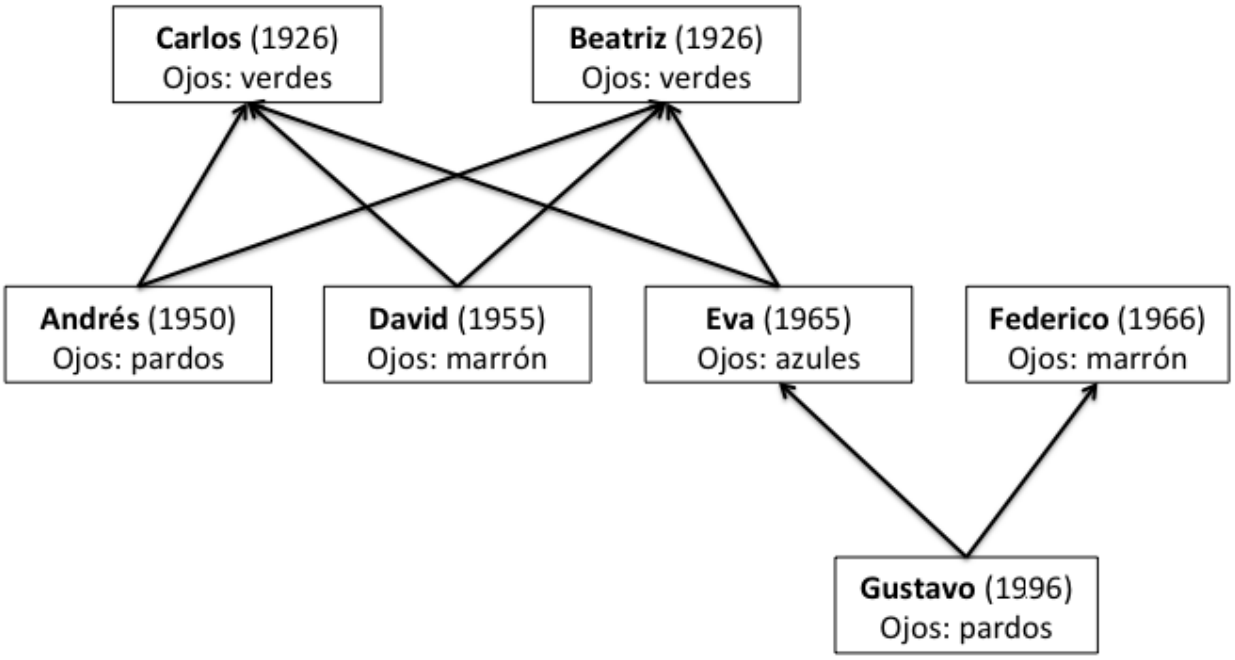
```
L=crearLista(4,crearLista(10,crearLista(1,listaVacia)))
```

```
sumarValoresLista(L)
```

```
15
```

Clase 11: Árboles (Capítulo 9.7)

Árbol de Ancestros




```
# persona: nombre(str) nacimiento(int) padre(persona) madre(persona)
estructura.crear("persona","nombre nacimiento padre madre")
```

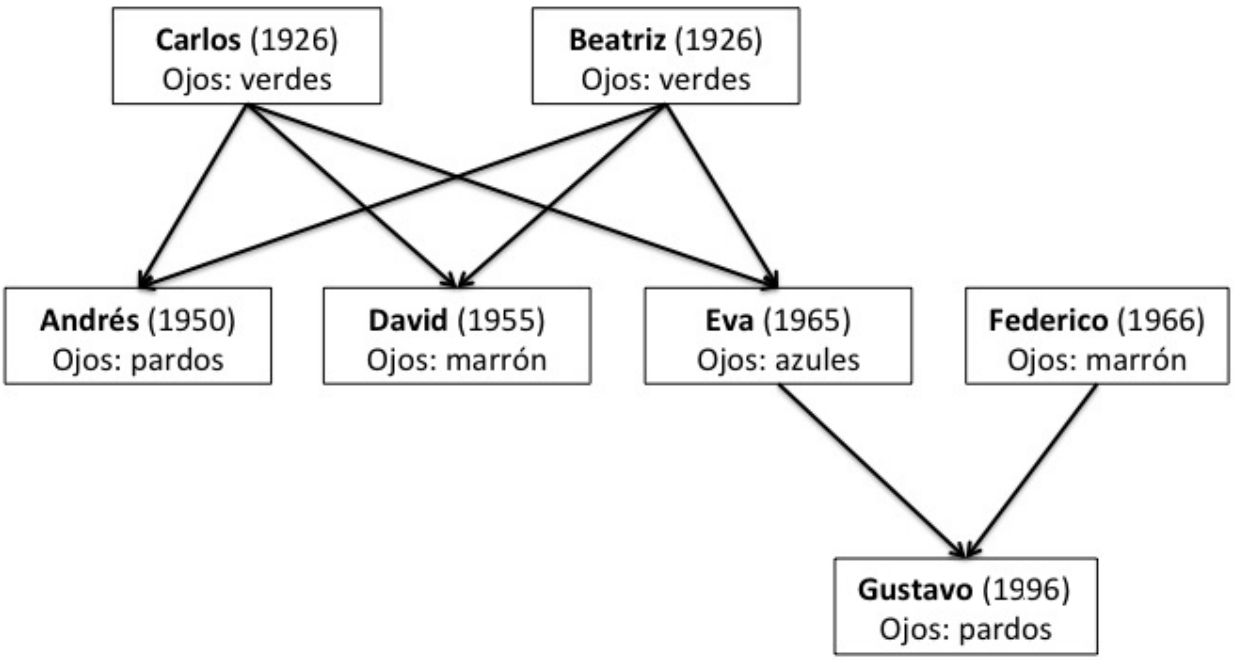
En otras palabras, una **`persona`** se construye a partir de cuatro elementos.

```
import estructura
#persona:nombre(str) nacimiento(int) padre(persona) madre(persona)
estructura.crear("persona","nombre nacimiento padre madre")

#primera generación (“abuelos”)
carlos=persona("carlos",1926,None,None)
beatriz=persona("beatriz",1926,None,None)

#segunda generacion (“hijos”)
andres=persona("andres",1950, carlos, beatriz)
david=persona("david",1955,carlos,beatriz)
eva=persona("eva",1965,carlos,beatriz)
federico=persona("federico",1966,None,None)
```

```
#tercera generacion (“nietos”)
gustavo=persona("gustavo",1996,federico,eva)
```

```
#esAncestro: persona persona -> bool
#True si persona x es ancestro de persona P
#ej: esAncestro(carlos,gustavo)->True
def esAncestro(x,P):
    if P==None:
        return False
    if P==x:
        return True
    #ancestro por parte de padre?
    if (esAncestro(x,P.padre)):
        return True
    #ancestro por parte de madre?
    elif(esAncestro(x,P.madre)):
        return True

    # no lo encontramos
    return False

assert esAncestro(beatriz,gustavo)
```

Árboles genealógicos descendentes




```
    # persona:nombre(str) nacimiento(int) hijos(lista(persona))
    estructura.crear("persona","nombre nacimiento hijos")
```

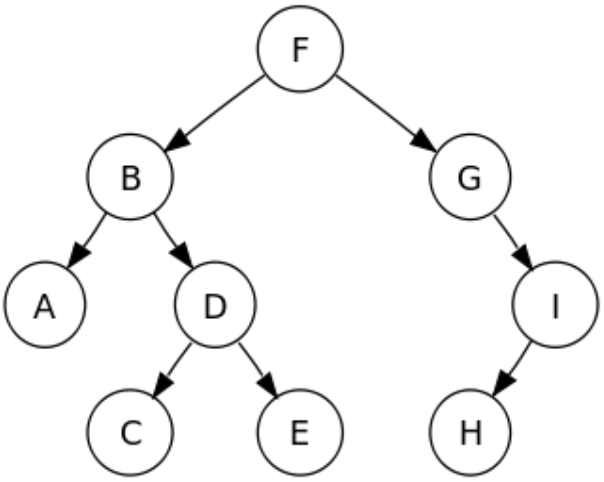
```
from lista import *
```

```
#persona:nombre(str) nacimiento(int) hijos(lista(persona))
estructura.crear("persona","nombre nacimiento hijos")

#nietos
gustavo=persona("gustavo",1996,None)

#padres
andres=persona("andres",1950, None)
david=persona("david",1955,None)
eva=persona("eva",1965,lista(gustavo,None))
federico=persona("federico",1966,lista(gustavo,None))

#abuelos
hijosCarlosBeatriz=lista(andres,lista(david,lista(eva,None)))
carlos=persona("carlos",1926,hijosCarlosBeatriz)
beatriz=persona("beatriz",1926,hijosCarlosBeatriz)
```

```
#esDescendiente: persona persona -> bool
#True si persona x es descendiente de persona P
#ej: esDescendiente(gustavo,carlos)->True
def esDescendiente(x,P):
    if P==None:
        return False
    if P==x:
        return True
    else:
        return esDescendienteHijos(x,P.hijos)
# Test
assert esDescendiente(gustavo,carlos)
```

```
#esDescendienteHijos: persona lista(persona) -> bool
#True si persona x es descendiente de listaHijos
#ej: esDescendienteHijos(gustavo,hijosCarlosBeatriz) -> True
def esDescendienteHijos(x, listaHijos):
    if vacia(listaHijos):
        return False
    else:
        if cabeza(listaHijos)==x:
            return True
        elif esDescendiente(x,cabeza(listaHijos)):
            return True
        else:
            return esDescendienteHijos(x,cola(listaHijos))

# Test
assert esDescendienteHijos(gustavo,hijosCarlosBeatriz)
```

Árboles binarios (AB)



Cumplen alguna de las siguientes condiciones:

- Un AB es vacío, o
- Tiene un valor, un AB a la izquierda, y otro AB a la derecha

```
import estructura
#Arbol Binario
#AB: valor(any), izq(AB), der(AB)
estructura.crear("AB","valor izq der")

ab=AB("F",\
    AB("B",\
        AB("A",None,None),\
        AB("D",\
            AB("C",None,None),\
            AB("E",None,None))),\
    AB("G",\
        None,\
        AB("I",\
            AB("H", None, None),\
            None)))
```

```
#valores: AB -> int
#nº de valores de arbol A
#ej: valores(ab) -> 9

def valores(A):
    assert A==None or type(A)==AB
    if A==None:
        return 0
    return 1 + valores(A.izq) + valores(A.der)

assert valores(ab)==9
```

```
#altura: AB -> int
#nº de niveles de valores de arbol A
#ej: altura(ab) -> 4

def altura(A):
```

```
    assert A==None or type(A)==AB
    if A==None:
        return 0
    return 1 + max(altura(A.izq),altura(A.der))

assert altura(ab)==4
```

Ejemplo: Contar las hojas de un AB

Las hojas son nodos que no tienen AB izquiero ni AB derecho (es decir, nodos sin hijos). En la figura son A, C, E, H.

```
#hojas: AB -> int
#nº de valores sin "hijos"
#ej: hojas(ab)->4

def hojas(A):
    assert A==None or type(A)==AB
    if A==None:
        return 0
    if A.izq==None and A.der==None:
        return 1
    return hojas(A.izq) + hojas(A.der)

assert hojas(ab)==4
```