Clase 11: Abstracción Funcional (Cap. 10)

Hay muchos programas que son muy similares algún valor o función

Ejemplo simple:

1. Una función que busca si aparece un string `"pelota"` en una lista

```
from lista import *

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

Ejecutamos la función:

```
hayPelotas(crearLista('pelota',listaVacia))
```

```
True
```

Es practicamente idéntica a:

2. Una función que busca si aparece un string "**auto**" en una lista

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

```
            return hayAutos(cola(unaLista))
```

```
hayAutos(crearLista('auto',listaVacia))
```

```
True
```

Ambas funciones consumen un string y lo buscan adentro de una lista de strings, la única diferencia es el nombre de las funciones y el string que buscan.

- Al **proceso de combinar dos o más funciones** en una sola más genérica se le denomina **abstracción funcional**
- La abstracción funcional es muy beneficiosa en programación
- Requiere agregar/modificar parámetros a las funciones para hacerlas genéricas

La abstracción funcional de `hayAutos` y `hayPelotas` queda como:

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

La única diferencia con las funciones anteriores es el parámetro `'elemento'`

```
contiene('auto',crearLista('auto',listaVacia))
```

```
True
```

```
contiene('pelota',crearLista('auto',listaVacia))
```

```
False
```

filtro: seleccionar elementos de una lista

Veamos un ejemplo más interesante:

```
# inferiores:  lista(num) num -> lista(num)
# Construye una lista de aquellos numeros
# de unaLista que sean inferiores a n
```

```
# ejemplo:  inferiores(crearLista(1,
# crearLista(2, listaVacia)), 2)
# devuelve (1, listaVacia)
def inferiores(unaLista, n):
    if vacia(unaLista):
        return listaVacia
    else:
        if cabeza(unaLista)<n:
            return crearLista(cabeza(unaLista),inferiores(cola(unaLista),n))
        else:
            return inferiores(cola(unaLista), n)
```

```
L = lista(4,lista(2,lista(3,lista(1,listaVacia))))
```

```
L
```

```
lista(valor=4, siguiente=lista(valor=2, siguiente=lista(valor=3, siguiente=lista
```

```
inferiores(L,3)
```

```
lista(valor=2, siguiente=lista(valor=1, siguiente=None))
```

```
# superiores:  lista(num) num -> lista(num)
# Construye una lista de aquellos numeros
# de unaLista que sean superiores a n
# ejemplo:  superiores(crearLista(2,
# crearLista(4, listaVacia)), 2)
# devuelve (4, listaVacia)
def superiores(unaLista, n):
    if vacia(unaLista):
        return listaVacia
    else:
        if cabeza(unaLista)>n:
            return crearLista(cabeza(unaLista),superiores(cola(unaLista),n))
        else:
            return superiores(cola(unaLista),n)
```

```
L = lista(4,lista(2,lista(3,lista(1,listaVacia))))
```

```
superiores(L,2)
```

```
lista(valor=4, siguiente=lista(valor=3, siguiente=None))
```

La función **`inferiores`** consume una lista de números y un número, y produce una lista de todos aquellos números de la lista que son inferiores a ese número; la función **`superiores`** produce una lista con todos aquellos números que están por encima de ese número.

- La diferencia entre ambas funciones es el **operador de la comparación**.
    - La primera ocupa **<** y
    - la segunda ocupa **>**.

¿Cómo podríamos hacer una función más general (abstracción) para ambas?

¡En Python una función puede recibir otra función como argumento!

Entonces, la abstracción funcional que utilizaremos es una llamada **filtro** o **función de selección**, ya que permite seleccionar algunos elementos de la 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):
    if vacia(unaLista):
        return listaVacia
    else:
        if operador(cabeza(unaLista),n):
            return crearLista(cabeza(unaLista),filtro(operador,cola(unaLista), n))
        else:
            return filtro(operador , cola(unaLista), n)
```

Según cómo definamos la función **`operador`** será los elementos que nuestra función filtro deje pasar a la nueva lista (por eso se llama filtro!)

```
# menorQue: num num -> bool
# devuelve True si el primer argumento es menor que el segundo
# y False en el caso contrario
# Ejemplo: menorQue(4,2) -> False
def menorQue(x,y):
    return x < y

assert not menorQue(4,2)
```

Con esta función ahora probamos nuestro filtro:

```
L #recordemos el valor de L
```

```
lista(valor=4, siguiente=lista(valor=2, siguiente=lista(valor=3, siguiente=lista
```

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

lista(valor=2, siguiente=lista(valor=1, siguiente=None))

Podemos definir otra función operador:

```
# mayorQue: num num -> bool
# devuelve True si el primer argumento es mayor que el segundo
# y False en el caso contrario
# Ejemplo: mayorQue(4,2) -> True
def mayorQue(x,y):
    return x > y

assert mayorQue(4,2)
```

```
filtro(mayorQue, L, 3)
```

```
lista(valor=4, siguiente=None)
```

Otro ejemplo de filtro o función de selección

Seleccionar los números primos de una lista, para eso usamos como **operador** una función `esPrimo()` que indica si un número es primo o no (vista en la clase 8):

```
# esPrimo: int -> bool
# Función auxiliar, indica si un número es primo o no
# Ejemplo: esPrimo(13)-> True, esPrimo(14)-> False
def esPrimo(n,divisor=3):
    assert type(n) == int and n>=2
    if n==2:
        return True
    if n%2==0:  # es par
        return False
    if divisor**2 > n:
        return True
    if n % divisor == 0: # es divisible perfectamente por un número menor a n
        return False
    return esPrimo(n,divisor+2)

assert not esPrimo(12)
assert esPrimo(11)
```

Y utilizamos el operador `esPrimo()` en nuestro nuevo filtro `filtroPrimos`

```
from lista import *
#filtroPrimos: lista(int) -> lista(int)
#Devuelve una lista que contiene solo los numeros primos
#ej: filtroPrimos(lista(3,lista(6,None)) ->
```

```
#    lista(3,None)

def filtroPrimos(L):
    assert type(L)==lista or L == None

    if vacia(L): # L == None
        return listaVacia #return None
    else:
        if(esPrimo(cabeza(L))): #esPrimo es el operador que filtra
            return crearLista(cabeza(L),filtroPrimos(cola(L)))
        else:
            return filtroPrimos(cola(L))

assert filtroPrimos(lista(3,lista(6,None)))==lista(3,None)
```

En resúmen

Forma estándar de una función de tipo filtro:

```
# filtro: (X -> bool) lista(X) -> lista(X)
# devuelve lista con todos los valores donde operador devuelve True
def filtro(operador , unaLista):
    if vacia(unaLista):
        return listaVacia
    else:
        if operador(cabeza(unaLista)):
            return lista(cabeza(unaLista), filtro(operador, cola(unaLista)))
        else:
            return filtro(operador , cola(unaLista))
# Tests
## ...
```

Repeticiones dentro de una función

La repetición de código no sólo se da entre funciones relacionadas, sino que también puede darse dentro de una misma función. Observemos la función **`mayorLargo`**:

```
# largo: lista(any)->num
# Función auxiliar que devuelve el largo de una lista, si la lista es vacía devuelve 0
# Ejemplo largo(crearLista(4,crearLista(3,listaVacia))) -> 2
def largo(L):
    if L == None or type(L) != lista:
        return 0
    else:
```

```
    else:
        return 1+largo(cola(L))
assert largo(crearLista(4,crearLista(3,listaVacia))) == 2


# mayorLargo: lista(any) lista(any) -> num
# Devuelve el largo de la lista mas larga, si ambas son vacias
# devuelve -1
# Ejemplo: mayorLargo (crearLista(5, listaVacia), listaVacia) -> 1
def mayorLargo(x, y):
    if vacia(x) and vacia(y):
        return -1
    elif largo(x) > largo(y):
        return largo(x)
    else:
        return largo(y)
```

- En este ejemplo se observa que el largo de cada lista se calcula 2 veces
- Esto es poco eficiente
- Se resuelve utilizando una función auxiliar

```
# maximo: num num -> num
# Devuelve el maximo entre x e y
# ejemplo: maximo(4, 2) -> 4
def maximo(x, y):
    if x > y:
        return x
    else:
        return y

# listaMasLarga: lista lista -> numero
# Devuelve el largo de la lista mas larga, si ambas son vacias
# devuelve -1
# Ejemplo: listaMasLarga (crearLista(5, listaVacia), listaVacia) -> 1
def listaMasLarga(x, y):
    if vacia(x) and vacia(y):
        return -1
    else:
        return maximo(largo(x), largo(y))
```

```
L
```

```
lista(valor=4, siguiente=lista(valor=2, siguiente=lista(valor=3, siguiente=lista
```

```
L2 = lista(7,lista(4,None))
L2
```

```
lista(valor=7, siguiente=lista(valor=4, siguiente=None))
```

```
listaMasLarga(L,L2)
```

```
4
```

Mapa: Aplicar una misma función a cada elemento de una lista

Este es otro caso común que es cuando a cada elemento de una lista queremos aplicarle una función, por ejemplo, a una lista de estructuras de tipo **`fraccion`** (vistas en la clase 8 y usadas para almacenar fracciones), queremos simplificarlas una por una:

```
from lista import *
from fraccion import *
#simplificaListaFracciones: lista(fraccion) -> lista(fraccion)
#simplificar c/u de las fraccciones de una lista
#ej: simplificaListaFracciones(lista(fraccion(2,4),None))->
#        lista(fraccion(1,2),None)

def simplificaListaFracciones(L):
    assert type(L)==lista or L==listaVacia
    if vacia(L):
        return None
    else:
        return lista(simplificaFracciones(cabeza(L)),simplificaListaFracciones(cola(L)

assert simplificaListaFracciones(lista(fraccion(2,4),listaVacia))== \
                                 lista(fraccion(1,2),None)
```

```
simplificaListaFracciones(lista(fraccion(2,4),lista(fraccion(3,9),listaVacia)))
```

```
lista(valor=fraccion(numerador=1, denominador=2), siguiente=lista(valor=fraccion
```

La forma estándar de una función de tipo "mapa"

```
# mapa : (X -> Y) lista(X) -> lista(Y)
# devuelve lista con funcion aplicada a todos sus elementos
def mapa(funcion, unaLista):
    if vacia(unaLista):
        return listaVacia
    else:
        return lista(funcion(cabeza(unaLista)), mapa(funcion, cola(unaLista)))
# Tests
# ...
```

```
L = crearLista(fraccion(2,4), crearLista(fraccion(3,9),listaVacia))
L
```

```
lista(valor=fraccion(numerador=2, denominador=4), siguiente=lista(valor=fraccion
```

```
mapa(simplificaFracciones,L) # simplifica la lista L de tipo fraccion
```

```
lista(valor=fraccion(numerador=1, denominador=2), siguiente=lista(valor=fraccion
```

Fold: procesar una lista para obtener un único valor

Otros problemas relacionados con listas que se pueden abstraer en una única función son los siguientes:

- Sumar/multiplicar todos los valores de una lista.
- Concatenar todas las palabras de una lista.

Estos problemas implican procesar los elementos de la lista para obtener un único valor. Esto se puede abstraer a una función que llamaremos **fold** ("reducir"):

- recibe una lista, un valor inicial y una función de dos argumentos
- procesa los elementos de la lista y devuelve un único valor.
- la lista debe poseer al menos un valor para poder ser procesada.

Ejemplo: sumar valores de una lista

```
# Funcion de dos argumentos requerida
def sumar(x, y):
    return x + y

# sumarValoresLista: lista -> num
# suma los valores dentro de la lista y devuelve el resultado
# ejemplo: si unaLista = lista(10, lista(20, lista(30, listaVacia)))
# sumarValores(unaLista) devuelve 60
def sumarValoresLista(unaLista):
    return fold(sumar, 0, unaLista)
```

Ejemplo: multiplicar valores de una lista

```
# Funcion de dos argumentos requerida
def multiplicar(x, y):
    return x * y
```

```
#s umarValoresLista: lista -> num
# multiplica los valores dentro de la lista y devuelve el resultado
# ejemplo: si unaLista = lista(5, lista(3, lista(3, listaVacia)))
# multiplicarValores(unaLista) devuelve 45
def multiplicarValoresLista(unaLista):
    return fold(multiplicar, 1, unaLista)
```

Forma estándar de una función tipo "fold"

```
# fold: (X X -> X) X lista(X) -> X
# procesa la lista con funcion 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(funcion, init, unaLista):
    if vacia(cola(unaLista)): # un solo valor
        return funcion(init, cabeza(unaLista))
    else:
        return fold(funcion, funcion(init, cabeza(unaLista)), cola(unaLista))

# Tests
valores = lista(1, lista(2, lista(3, lista(4, listaVacia))))
assert fold(sumar, 0, valores) == 10
```

```
unaLista = lista(5, lista(3, lista(3, listaVacia)))
```

```
sumarValoresLista(unaLista)
```

```
11
```

```
multiplicarValoresLista(unaLista)
```

```
45
```