Estructuras de Datos Recursivas (Cap. 9 - Leer)

- Las estructuras (structs) nos sirven para usar datos compuestos
- Sin embargo, en muchos casos, *no sabemos cuántas cosas queremos enumerar*:
    - entonces formamos una **lista**: puede tener un largo arbitrario, esto es, contiene una cantidad finita pero indeterminada de elementos

Listas

Para manejar listas ocuparemos el módulo `lista.py` (en material docente)

```
from lista import *
```

La definición de una lista es **recursiva**:

1. una lista puede ser **vacía**: no contiene ningún elemento (`listaVacia`).
2. una lista puede estar compuesta por elementos que contienen un **valor** y un enlace al resto de la lista.

Esto permite crear una lista larga concatenando listas, lo podemos entender como ir armando una cadena con dos campos en su estructura:

1. `valor`
2. la lista `siguiente`.

El campo **valor** puede ser de cualquier tipo (básico o compuesto), mientras que el campo **siguiente** es precisamente una lista

La definición de la estructura para listas está incluida en el módulo `lista.py`, note en particular en el contrato de la estructura su definición recursiva:

```
# Diseno de la estructura
# lista : valor (cualquier tipo) siguiente (lista)
estructura.crear("lista", "valor siguiente")
```

Para crear una lista nueva, el módulo provee la función `crearLista` que recibe dos parámetros: el `valor` del primer elemento de la lista y el `resto` de la lista.

```
from lista import *
L = crearLista(4, None)
```

```
L
```

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

```
L.valor
```

```
4
```

```
L.siguiente
```

Ejemplo: Lista con 2 valores

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

```
L
```

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

```
L.siguiente
```

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

Ejemplo:

Supongamos que queremos formar una lista con los planetas del sistema solar. Primero comenzamos con un eslabón de la lista que sólo contiene a Mercurio:

```
crearLista("Mercurio", listaVacia)
```

```
lista(valor='Mercurio', siguiente=None)
```

Luego viene el planeta Venus:

```
crearLista("Venus", crearLista("Mercurio", listaVacia))
```

```
lista(valor='Venus', siguiente=lista(valor='Mercurio', siguiente=None))
```

A continuación la Tierra, y así sucesivamente:

```
crearLista("Tierra", crearLista("Venus", crearLista("Mercurio", listaVacia)))
```

```
lista(valor='Tierra', siguiente=lista(valor='Venus', siguiente=lista(valor='Mercurio', siguier
```

En toda lista distinguimos dos elementos:

1. **cabeza**: valor que está al frente de la lista (primer valor disponible)
2. **cola**: todo lo que va encadenado a la cabeza

En nuestro último ejemplo, la **cabeza** de la lista es el string `"Tierra"`, mientras que la **cola** es la lista formada por el eslabón `(Venus, (Mercurio, (listaVacia)))`.

```
L = crearLista("Tierra", crearLista("Venus",crearLista("Mercurio", listaVacia)))
```

```
cabeza(L)
```

```
    'Tierra'
cola(L)
```

```
lista(valor='Venus', siguiente=lista(valor='Mercurio', siguiente=None))
```

Módulo `lista.py`

Revisemos qué contiene el módulo `lista.py`.

```
#modulo: lista.py
import estructura

# Diseno de la estructura
# lista : valor (any = cualquier tipo) siguiente (lista)
estructura.crear("lista", "valor siguiente")

# identificador para listas vacias
listaVacia = None

# crearLista: any lista -> lista
# devuelve una lista cuya cabeza es valor
# y la cola es resto
def crearLista(valor, resto):
        return lista(valor, resto)

# cabeza: lista -> any
# devuelve la cabeza de una lista (un valor)
def cabeza(lista):
        return lista.valor

# cola: lista -> lista
# devuelve la cola de una lista (una lista)
def cola(lista):
        return lista.siguiente

# vacia: lista -> bool
# devuelve True si la lista esta vacia
def vacia(lista):
        return lista == listaVacia


# Tests

test_lista = lista(1, lista(2, lista(3, listaVacia)))

assert cabeza(test_lista) == 1
assert cabeza(cola(test_lista)) == 2
assert cabeza(cola(cola(test_lista))) == 3
assert cola(cola(test_lista)) == lista(3, listaVacia)

assert vacia(listaVacia)
assert not vacia(test_lista)
assert vacia(cola(cola(cola(test_lista))))
```

```
from lista import * #todas las funciones
```

```
L=lista(4,lista(2,lista(3,None)))
cabeza(L)
```

4

```
cola(L)
```

lista(valor=2, siguiente=lista(valor=3, siguiente=None))

Agregamos una función para ver si algo es una lista:

```
#esLista: any -> bool
#True si L es una lista
#ej: esLista(lista(1,None))->True

def esLista(L):
    return type(L)==lista or L==None

assert esLista(lista(1,None))
```

```
from lista import *


# sumaTres: listaDeTresNumeros -> num
# suma los tres numeros en unaLista
# sumaTres(crearLista(2, crearLista(1, crearLista(0, listaVacia))))
# devuelve 3
# sumaTres(crearLista(0, crearLista(1, crearLista(0, listaVacia)))) # devuelve 1
def sumaTres(unaLista):
    # separamos los 3 valores
    valor1 = cabeza(unaLista)
    listaRestante = cola(unaLista)
    valor2 = cabeza(listaRestante)
    listaRestante = cola(listaRestante)
    valor3 = cabeza(listaRestante)

    return valor1 + valor2 + valor3

# Test
primerValor=crearLista(10,listaVacia)
segundoValor=crearLista(20,primerValor)
tercerValor=crearLista(30,segundoValor)
assert sumaTres(tercerValor)==60

# Ejercicio sumar una lista de tamanno arbitrario
```

Definición de datos para listas de largo arbitrario

Ejemplo: inventario de una juguetería

Supongamos que queremos representar el inventario de una juguetería que vende muñecas, sets de maquillaje, payasos, arcos, flechas y pelotas. Para construir el inventario el dueño debería tomar una hoja en blanco e irla llenando con los nombres de los jueguetes que tiene en los estantes.

Representar una lista de juguetes es simple, basta con crear una lista de strings, encadenando eslabones uno por uno:

Por ejemplo:

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

```
lista(valor='flecha', siguiente=lista(valor='pelota', siguiente=None))
```

```
eslabon1 = listaVacia
eslabon2 = crearLista('pelota', eslabon1)
eslabon3 = crearLista('flecha', eslabon2)

eslabon3
```

```
lista(valor='flecha', siguiente=lista(valor='pelota', siguiente=None))
```

- En una tienda real, esta lista contendrá muchos más elementos, y la lista crecerá y disminuirá a lo largo del tiempo
- Si quisiéramos desarrollar una función que consuma tales listas debemos poder procesar una lista de cualquier tamaño
- Necesitamos una definición de datos que precisamente describa la clase de listas que contenga una cantidad arbitraria de elementos (por ejemplo, strings).

Note que todos los ejemplos que hemos desarrollado hasta ahora siguen un patrón:

comenzamos con una lista vacía, y empezamos a encadenar elementos uno a continuación del otro

Podemos abstraer esta idea y plantear la siguiente definición de datos:

Una **lista de strings** es:

1. una lista vacía, `listaVacia`, o bien
2. `crearLista(X,Y)`, donde X es un string, e Y es una **lista de strings**.

La estructura lista se define en **términos de sí misma** en el segundo ítem.

Procesar una lista de largo arbitrario

Supongamos que queremos hacer una función que entrega `True` si es que hay pelotas en un inventario (lista) y `False` si es que no. ¿Cómo debe ser esta función?

```
# hayPelotas: lista(str) -> bool
# determinar si el string "pelota" esta en la lista unaLista
def hayPelotas(unaLista):
```

Siguiendo la receta de diseño vemos como funciona con unos ejemplos:

```
hayPelotas(listaVacia)
```

```
False
```

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

```
True
```

```
hayPelotas(crearLista("muneca", listaVacia))
```

```
False
```

Y finalmente, veamos el funcionamiento de la función en listas más generales (con más de un elemento):

```
hayPelotas(crearLista("arco", crearLista("flecha", \
        crearLista("muneca", listaVacia))))
```

```
False
```

```
hayPelotas(crearLista("soldadito", crearLista("pelota", \
        crearLista("oso", listaVacia))))
```

```
True
```

El paso siguiente es diseñar la plantilla de la función que se corresponda con la definición del tipo de datos. En este caso, dado que la definición de una lista de strings tiene dos cláusulas (lista vacía o una lista de strings), la plantilla se debe escribir usando un bloque condicional:

```
# hayPelotas: lista(str) -> bool
# determinar si el string "pelota" esta en la lista unaLista
# def hayPelotas(unaLista):
#        if vacia(unaLista):
#               ...
#       else:
#               ...  cabeza(unaLista)
#               ...  cola(unaLista) ...
```

```
from lista import *

# hayPelotas: lista(str) -> bool
# determinar si el string "pelota" esta en la lista unaLista
```

```
# def hayPelotas(unaLista):
#       if vacia(unaLista):
#               ...
#       else:
#               ... cabeza(unaLista)
#               ... cola(unaLista) ...

def hayPelotas(unaLista):
    if(vacia(unaLista)):
        return False
    else:
        if cabeza(unaLista)=="pelota":
            return True
        else:
            listaRestante = cola(unaLista)
            return hayPelotas(listaRestante)

# Test
juguetes = crearLista("soldadito", crearLista("pelota", crearLista("oso", listaVacia)))
assert hayPelotas(juguetes)
```

Si quisieramos contar juguetes?

```
# contarJuguetes: lista(str) -> int
# determinar cuantos juguetes hay en la lista unaLista
# def contarJuguetes(unaLista):
#       if vacia(unaLista):
#               ...
#       else:
#               ... cabeza(unaLista)
#               ... cola(unaLista) ...

def contarJuguetes(unaLista):
    if vacia(unaLista):
        return 0
    else:
        listaRestante = cola(unaLista)
        return 1 + contarJuguetes(listaRestante)
```