Listas de valores

Juan Álvarez Rubio

jalvarez@dcc.uchile.cl,

En la sección anterior se utilizaron datos compuestos por una cantidad pequeña y fija de valores. Por ejemplo, las fracciones están compuestas por dos valores (numerador y denominador) y las fechas están compuestas por tres valores (día, mes y año). Sin embargo, se necesita también manejar datos con una cantidad mayor y variable de valores. Por ejemplo, los nombres de los meses, las notas de los estudiantes de un curso, etc.

Un tipo para listas de números enteros

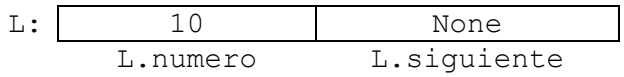
Una lista de una cantidad variable de números enteros se puede manejar definiendo el siguiente tipo de datos:

```
#lista: numero(int) siguiente(lista)
import estructura
estructura.crear("lista","numero siguiente")
```

De acuerdo a la definición del tipo lista, se puede definir una variable L como una lista que contiene el número entero 10 en la componente L.número y el valor None en la componente L.siguiente:

```
L=lista(10,None)
```

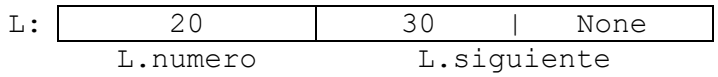
La palabra None se utiliza en Python para representar una lista vacía, es decir una lista que no tiene valores. La variable L se puede representar gráficamente de la siguiente manera:



Por otra parte, L se puede definir para representar una lista con 2 números:

```
L=lista(20, lista(30,None))
```

En este caso la componente L.numero contiene el número 20 y la componente L.siguiente contiene una lista con el valor 30. La representación de L se puede graficar:



Para operar con los dos números de la lista se deben referenciar apropiadamente las componentes. Por ejemplo, calcular y escribir la suma de los dos números de la lista L se puede lograr con las siguientes instrucciones:

```
L2 = L.siguiente
print("suma: ", L.numero + L2.numero)
```

El mismo efecto se podría lograr con una sola instrucción, considerando que L.siguiente es del tipo lista y por lo tanto sus componentes son L.siguiente.numero y L.siguiente.siguiente:

```
print("suma: ", L.numero + L.siguiente.numero)
```

Módulo lista

Para poder manejar listas que contengan una cantidad variable de valores (cero o más) de distintos tipos se necesita definir el tipo lista, y sus operaciones más frecuentes, en un módulo de modo que permita definir distintas listas:

```
from lista import *
nombres=lista("ana",lista("luis",lista("rosa",None)))
numeros=lista(12,lista(14,lista(12,lista(13,None))))
```

El tipo lista permite definir listas de tipos definidos por el usuario. Por ejemplo, para definir una lista de fracciones se podría escribir:

```
from fraccion import *
fracciones=lista(fraccion(1,2), lista(fraccion(3,4),None))
```

El módulo de nombre lista contiene la definición del tipo lista:

```
#lista: valor (any) siguiente (lista)
import estructura
estructura.crear("lista","valor siguiente")
```

De acuerdo a la definición, la componente valor puede ser de cualquier (any) tipo. Por su parte, la componente siguiente debe ser del tipo lista, lo que permite definir listas de una cantidad variable o arbitraria de valores.

Funciones para recorrer una lista

Para recorrer todos los valores de una lista se definen dos funciones: la función cabeza (head) devuelve el primer valor de la lista, y la función cola (tail) entrega el resto de la lista (desde el segundo valor en adelante).

```
#cabeza: lista -> any
#primer valor de una lista
#ej: cabeza(lista(1,lista(2,None)))->1
def cabeza(L):
    assert type(L)==lista
    return L.valor
assert cabeza(lista(1,lista(2,None)))==1

#cola: lista -> lista
#lista sin primer valor
#ej: cola(lista(1,lista(2,None)))->lista(2,None)
#ej: cola(lista(1,None))->None
def cola(L):
    assert type(L)==lista
    return L.siguiente
assert cola(lista(1,lista(2,None)))==lista(2,None)
assert cola(lista(1,None))==None
```

Las invocaciones cabeza(L) y cola(L) devuelven el contenido de las componentes L.valor y L.siguiente, proporcionando una notación funcional para el recorrido de las listas. Una aplicación de las funciones cabeza y cola se puede apreciar en una función recursiva que escribe los valores de una lista:

```
#escribir: lista ->
#escribe valores de L
#ej: escribir(lista(1,lista(2,None))) escribe 1 2
def escribir(L):
    if L!=None:
        print(cabeza(L))
        escribir(cola(L))
```

Otras funciones para listas

Las funciones cabeza y cola no pueden recibir una lista vacía, es decir, no acepta una lista sin valores. Al respecto, en el módulo se define la variable listaVacia como sinónimo de None (que en Python representa la ausencia de un valor):

```
listaVacia=None
```

De hecho, la variable listaVacia se puede usar al definir una lista. Por ejemplo:

```
nombres=lista("ana",lista("luis",lista("rosa",listaVacia)))
numeros=lista(12,lista(14,lista(12,lista(13,listaVacia))))
```

El módulo lista contiene también una función para comprobar si un argumento es una lista:

```
#esLista: any -> bool
#True si L es una lista
#ej: esLista(lista(1,lista(2,None))->True
#    esLista(listaVacia)->True
#    esLista(10)->False
def esLista(L):
    return L==listaVacia or type(L)==lista
assert esLista(lista(1,lista(2,None)))
assert esLista(listaVacia)
assert not esLista(10)
```

Una lista vacía o sin valores también se considera una lista. Como el valor None no tiene asociado un tipo, entonces se debe preguntar explícitamente si es una lista vacía antes de comprobar si es del tipo lista. Al respecto, se define una función para consultar si una lista es o no vacía:

```
#vacia: lista -> bool
#True si L es una lista vacia
#ej: vacia(lista(1,lista(2,None)))->False,
#    vacia(listaVacia)->True
def vacia(L):
    assert esLista(L)
    return L==listaVacia
assert not vacia(lista(1,lista(2,None)))
assert vacia(listaVacia)
```

Usando las funciones anteriores se puede implementar una función que permita determinar el largo, es decir, la cantidad de valores de una lista:

```
#largo: lista -> int
#numero de valores de lista L
#ej: largo(lista(10,lista(20,None)))->2, largo(listaVacia)->0
def largo(L):
  assert esLista(L)
  if vacia(L):
      return 0
  else:
      return 1 + largo(cola(L))
assert largo(lista(10,lista(20,None)))==2
assert largo(listaVacia)==0
```

El caso base es una lista vacía y debe devolver un 0. Si la lista no es vacía, entonces tiene al menos un valor, más la cantidad de valores que existan en la cola de la lista.

Uso del módulo

El módulo lista puede utilizarse para realizar operaciones con listas de valores de distintos tipos. Por ejemplo, la siguiente función suma los números de una lista:

```
from lista import *
#suma: lista (num) -> num
#suma números de L
#ej: suma(lista(1,lista(2,None)))->3
#ej: suma(listaVacia)->0
def suma(L):
    assert esLista(L)
    if vacia(L):
        return 0
    else:
        return cabeza(L) + suma(cola(L))
assert suma(lista(1,lista(2,None)))==3
assert suma(listaVacia)==0
```

El módulo también puede utilizarse con tipos definidos por el programador. Por ejemplo, la siguiente función suma una lista de fracciones:

```
from lista import *
from fraccion import *
fracciones = lista(fraccion(1,2),lista(fraccion(3,4),None))

#suma: lista (fraccion) -> fraccion
#suma fracciones de L
#ej: suma(fracciones)-> fraccion(10,8)
#ej: suma(listaVacia)->fraccion(0,1)
def suma(L):
    assert esLista(L)
    if vacia(L):
        return fraccion(0,1)
    else:
        return sumar(cabeza(L),suma(cola(L)))
assert suma(fracciones) == fraccion(10,8)
assert suma(listaVacia) == fraccion(0,1)
```

De acuerdo al contrato, la función devuelve una fracción como resultado. En el caso de una lista vacía devuelve una fracción con el valor cero. Si no es vacía, entonces devuelve la fracción que entrega por resultado la función sumar del módulo fraccion.

Uso de listas para resolver un problema

Las funciones del módulo pueden utilizarse para resolver el problema de leer una lista de strings y escribirlos en orden inverso. El programa correspondiente lee strings, que terminan con un punto, y los escribe a la inversa en la forma indicada en el siguiente diálogo:

```
string?primero
string?segundo
string?tercero
string?.
tercero
segundo
primero
```

El programa puede implementarse simplemente con las instrucciones:

```
from lista import *
escribir(invertir(leerLista()))
```

En forma equivalente se puede escribir:

```
from lista import *
L=leerLista()
L=invertir(L)
escribir(L)
```

Las funciones leer e invertir entregan una lista como resultado. Por ejemplo, la función leerLista entrega una lista con los strings que se leen desde el teclado y que terminan con un valor especial para indicar el fin de los datos:

```
from lista import *
#leerLista: str str -> lista(str)
#lista con valores que se leen (hasta valor fin)
#ej: leerLista() -> lista('hola',lista('chao',None))
#    si lee hola, chao, .
#ej: leerLista() -> listaVacia si lee .
def leerLista(pregunta='string?',fin='.'):
  assert type(pregunta)==str
  assert type(fin)==str
  valor=input(pregunta)
  if valor==fin:
    return listaVacia
  else:
    return lista(valor, leerLista(pregunta,fin))
```

```
#assert leerLista()==lista('hola',lista('chao',None))
#assert leerLista()==listaVacia
```

La función invertir debe recibir una lista y entregar una lista con los valores en orden inverso. Para lograrlo debe agregar el primer valor (la cabeza) al final de la lista que resulta de invertir la cola:

```
from lista import *
#invertir: lista -> lista
#lista con valores de L a la inversa
#ej: invertir(lista(1,lista(2,None)))->lista(2,lista(1,None))
def invertir(L):
    assert esLista(L)
    def agregarAlFinal(x,L):
        …
    if vacia(L):
        return listaVacia
    else:
        return agregarAlFinal(cabeza(L),invertir(cola(L)))
assert invertir(lista(1,lista(2,None)))==lista(2,lista(1,None))
assert invertir(listaVacia)==listaVacia
```

La función interna agregarAlFinal recibe un valor y una lista y devuelve una nueva lista, igual a la anterior, pero con el valor al final. Si la lista está vacía debe devolver una lista con un solo valor:

```
    #agregarAlFinal: any lista -> lista
    def agregarAlFinal(x,L):
        if vacia(L):
            return lista(x,None)
        else:
            return lista(cabeza(L),agregarAlFinal(x,cola(L)))
    assert agregarAlFinal(2,lista(1,None))==lista(1,lista(2,None))
    assert agregarAlfinal(1,listaVacia)==lista(1,None)
```

Por tratarse de una función interna, agregarAlFinal no necesita ceñirse a la receta de diseño. Se ha incluido el contrato y un par de pruebas con el propósito de entender mejor su objetivo.

Esquema general de las funciones para listas

En resumen, toda función que recibe y procesa una lista tiene la siguiente estructura general:

```
from lista import *
#nombre: lista … -> …
#objetivo
#ejemplo(s)
```

```
def nombre(L,…):
    #precondiciones
    assert esLista(L)
    …
    #caso base
    if vacia(L):
        return …
    #caso recursivo
    else:
        #procesar valor en la cabeza de la lista
        …cabeza(L)…
        #invocación recursiva
        return … nombre(cola(L),…) …
#prueba(s)
assert nombre(L,..)==…
assert nombre(listaVacia,…)==…
```