v.7.7

Resolución práctica 5, Segunda Parte

A continuación presentamos soluciones a ejercicios seleccionados de la Práctica 5, Segunda parte.

Como siempre, en el proceso de resolución de un problema, primero debemos dedicar un tiempo al análisis del mismo para asegurarnos de entender correctamente qué se pide. Luego, pasamos a la resolución propiamente dicha.

Recomendamos usar este documento como un elemento de ayuda y referencia. Es importante que no lean la solución sin antes haber leído y entendido el problema y sin haber dedicado un tiempo a pensar cómo resolverlo.

Por otro lado, no hay una única forma correcta de escribir una solución.

1 Listas

En esta práctica comenzaremos a trabajar con un tipo de datos autorreferencial muy importante: las listas. Para poder entender en profundidad cómo funcionan las listas, será necesario comprender sus constructores: `'()` y `cons`, y sus selectores: `first` y `rest`. Además, veremos tres funciones muy poderosas: `map`, `filter` y `fold`. Como siempre recordamos, para resolver ejercicios más complejos debe tenerse en cuenta todo lo aprendido anteriormente en cuanto a buenas prácticas de programación.

1.1 Ejercicio 3

1.1.1 Ejercicio 9

Este ejercicio consta de dos partes. La primera nos pide definir una función `todos-verdaderos`, que reciba como entrada una lista de valores booleanos y devuelva `#true` únicamente si todos los elementos de la lista son `#true`. Ahora debemos redefinirla utilizando `foldr`.

El diseño de datos, la signatura, la declaración de propósito y los casos de prueba ya los conocemos.

```
; ListaB es:
; - empty
; - cons Boolean ListaB
; interpretación: un elemento de ListaB es una lista
; cuyos elementos pertenecen al tipo Boolean.

; todos-verdaderos : ListaB -> Boolean
; recibe como entrada una lista de valores booleanos y
; devuelve #true únicamente si todos los elementos de la lista son #true.

(check-expect (todos-verdaderos empty) #t)
(check-expect (todos-verdaderos (list #t #t #t)) #t)
(check-expect (todos-verdaderos (list #t #f #t)) #f)
```

En el caso en que la lista sea vacía, sabemos que la respuesta debe ser `#true`. La idea para el caso de listas con al menos un elemento será hacer un `and` entre el primer elemento de la lista, y la llamada recursiva a la función `todos-verdaderos`. Por razones que no estudiaremos, no podremos utilizar directamente el operador `and` como primer argumento para el `foldr`. Para solucionarlo, definiremos

una función `f-and` que calcule el `and` entre dos booleanos. Finalmente, las definiciones nos quedan como se muestra a continuación.

```
(define (todos-verdaderos l) (foldr f-and #true l))

; f-and : Bool Bool -> Bool
; f-and toma dos booleanos, y calcula el and entre ellos

(check-expect (f-and #t #t) #t)
(check-expect (f-and #t #f) #f)
(check-expect (f-and #f #t) #f)
(check-expect (f-and #f #f) #f)

(define (f-and b1 b2) (and b1 b2))
```

La segunda parte del ejercicio nos pide definir una función `uno-verdadero`, que reciba como entrada una lista de valores booleanos y devuelva `#true` si al menos uno de los elementos de la lista es `#true`.

El diseño de datos será el mismo que el utilizado en el apartado anterior. La signatura, la declaración de propósito y los casos de prueba ya los conocemos.

```
; uno-verdadero : ListaB -> Boolean
; recibe como entrada una lista de valores booleanos y
; devuelve #true si algún elemento de la lista es #true.

(check-expect (uno-verdadero empty) #f)
(check-expect (uno-verdadero (list #f #t #t)) #t)
(check-expect (uno-verdadero (list #f #f #f)) #f)
(check-expect (uno-verdadero (list #t #t #t)) #t)
```

En el caso en que la lista sea vacía, sabemos que la respuesta debe ser `#false`. La idea para el caso de listas con al menos un elemento será hacer un `or` entre el primer elemento de la lista, y la llamada recursiva a la función `uno-verdadero`. Por razones que no estudiaremos, no podremos utilizar directamente el operador `or` como primer argumento para el `foldr`. Para solucionarlo, definiremos una función `f-or` que calcule el `or` entre dos booleanos. Finalmente, las definiciones nos quedan como se muestra a continuación.

```
(define (uno-verdadero l) (foldl f-or #f l))

; f-or : Bool Bool -> Bool
; f-or toma dos booleanos, y calcula el for entre ellos

(check-expect (f-or #t #t) #t)
(check-expect (f-or #t #f) #t)
(check-expect (f-or #f #f) #f)
(check-expect (f-or #f #t) #t)

(define (f-or b1 b2) (or b1 b2))
```

1.1.2 Ejercicio 27

Resolveremos el ejercicio 27 de la práctica anterior usando fold. Recordando, tenemos que diseñar una función que devuelva el máximo de una lista de naturales y para la lista vacía tenemos que devolver `0`.

Primero, haremos el diseño de datos, signatura, declaración de propósito y ejemplos de uso:

> Es muy importante elegir buenos casos de uso para evitar errores y comportamientos inesperados. Para ello necesitamos analizar con mucho detenimiento los requerimientos y escribir los tests antes de comenzar a escribir código.

```
; ListN es:
; empty
; (cons Number ListN)
; interpretación: Un elemento de ListN es una lista de números

; maximo : ListN -> Number
; Dada una lista l,  devuelve el máximo de los elementos de l
; o 0 en el caso de la lista vacía.
(check-expect (maximofold (list))  0)
(check-expect (maximofold (list 0)) 0)
(check-expect (maximofold (list 2)) 2)
(check-expect (maximofold (list -1)) -1)
```

Claramente la función con la que se van a operar los elementos de la lista va a ser `max` ya que entre cada par de elementos que se operan se desea obtener el mayor de ellos. En cuanto al resultado esperado para la lista vacía, no podemos utilizar el valor `0` ya que para el caso de la lista con exactamente un valor numérico negativo estaríamos devolviendo `0`, lo cual es incorrecto según el requerimiento del ejercicio. Entonces, podemos elegir el primer elemento de la lista usando `(first l)` y así el fold devolverá el máximo de una lista no vacía (¿por qué?). Por lo tanto, deberemos devolver `0` cuando la lista es vacía o la aplicación de `foldr` con la función y el valor que vimos previamente cuando la lista es no vacía:

```
(define (maximo lista)
  (if (empty? lista) 0 (foldr max (first lista) lista)))
```

1.2 Ejercicio 9

En este ejercicio el objetivo es diseñar una función que tome una lista de números y calcule las raíces cuadradas de los números no negativos.

Este problema puede dividirse en dos partes.

Inicialmente, tenemos una lista donde debemos eliminar los números negativos.

Y luego, sobre los positivos, aplicar raíz cuadrada a cada uno de ellos.

Para comenzar, como siempre, haremos un diseño de la función a resolver, así como los tipos que debamos declarar, y algunos casos ejemplo.

```
; ListN es:
; empty
; (cons Number ListN)
; interpretación: Un elemento de ListN es una lista de números

; raices : ListN -> ListN
; dada una lista l,  devuelve una lista con las
; raíces cuadradas de los números no negativos de l.


(check-expect (raices (list -4 4 9)) (list 2 3))
(check-expect (raices (list -9 36 49 0)) (list 6 7 0))
(check-expect (raices (list -1 -36 -77 -25)) empty)
(check-expect (raices empty) empty)
```

Como siempre, es útil pensar **casos borde** como una lista llena de números negativos o una lista que está directamente vacía.

En este tipo de ejercicios, donde debemos resolver dos problemas que entre sí no tienen relación, es donde más se marca la necesidad de modularizar el código en funciones que resuelvan problemas separados. Esto no solo es práctico sino que también representa nuestra forma de encarar un problema similar.

Para separar los positivos de los negativos, resulta útil utilizar la función `filter`, para filtrar a los negativos. Para poder utilizarla, necesitamos una función para filtrar, que dé verdadero cuando un número es positivo y falso cuando sea negativo.

De esta forma diseñaremos una función llamada `noNeg?` para resolver este problema.

> Alternativamente, podríamos haber utilizado alguna función que racket nos brinda y no habría necesidad de definirla. ¿Se les ocurre cuál?

```
; noNeg? : Number -> Boolean
; dado un número, devuelve #true si es no negativo
(check-expect (noNeg? -4) #f)
(check-expect (noNeg? 4) #t)
(check-expect (noNeg? 0) #t)
(define (noNeg? x) (>= x 0))
```

Finalmente, con una función para filtrar ya diseñada y funcionando, y habiendo previamente diseñado la función `raices`, nos podemos enfocar en resolverla.

Por un lado debemos filtrar los números negativos de la lista, usando el predicado que definimos anteriormente, y luego, al resultado de este filtrado, solo debemos aplicar la raíz cuadrada a cada uno de sus elementos. Para resolver esto, utilizaremos la función `map` para aplicar una función a cada elemento, siendo esta función la raiz cuadrada.

```
(define (raices l)
  (map sqrt (filter noNeg? l)))
```