v.7.7

Resolución práctica 5

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

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`. 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 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`.

Empezamos con el diseño de datos, la signatura y la declaración de propósito.

```
; 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.
```

Para armar los casos de prueba, debemos pensar en, al menos, dos casos: qué pasa cuando la lista que se toma es vacía, y qué pasa cuando la lista tiene al menos un elemento.

Supongamos que la lista es vacía, ¿podemos decir que todos sus elementos son verdaderos?. La respuesta es sí. En lógica, esto se considera una **cuantificación universal** sobre el **rango vacío**. La idea de cuantificación universal viene de una propiedad que debe cumplirse para todos los elementos (en este caso, de una lista). Decimos que esa propiedad la estamos aplicando sobre el rango vacío porque estamos queriendo ver si se cumple o no en el caso de tener una lista vacía (sin elementos). En general, si tenemos una propiedad que se debe cumplir para cada elemento de una lista, y la lista no tiene elementos, entonces dicha propiedad se cumple. Por lo tanto, ya tenemos nuestro primer caso de prueba.

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

Luego, pensamos casos de prueba para listas que no sean vacías, es decir, listas que contengan al menos un elemento.

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

Finalmente, nos queda escribir la definición. En el caso en que la lista sea vacía, sabemos que la respuesta debe ser `#true`. El caso para listas que contienen al menos un elemento es más trabajoso. La idea será hacer un `and` entre el primer elemento de la lista, y la llamada recursiva a la función `todos-verdaderos`.

```
(define (todos-verdaderos l)
  (cond [(empty? l) #t]
        [else (and (first l) (todos-verdaderos (rest l)))]))
```

Observar que utlizamos el operador `and`, ya que este da `#true` únicamente si todos los valores son verdaderos.

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 partado anterior. Por lo tanto, empezamos con la signatura y la declaración de propósito.

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

Nuevamente, para armar los casos de prueba, debemos pensar en, al menos, dos casos: qué pasa cuando la lista que se toma es vacía, y qué pasa cuando la lista tiene al menos un elemento.

Supongamos que la lista es vacía, ¿podemos decir que alguno de sus elementos es verdaderos?. En este caso, la respuesta es no. En lógica, esto se considera una **cuantificación existencial** sobre el **rango vacío**. La idea de cuantificación existencial viene de una propiedad que debe cumplirse para al menos uno de los elementos. Decimos que esa propiedad la estamos aplicando sobre el rango vacío porque estamos queriendo ver si se cumple o no en el caso de tener una lista vacía (sin elementos). En general, si tenemos una propiedad que se debe cumplir para algún elemento de una lista (es decir, debe existir un elemento que cumpla con una condición particular), y la lista no tiene elementos, entonces dicha propiedad no se cumple. Por lo tanto, ya tenemos nuestro primer caso de prueba.

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

Luego, pensamos casos de prueba para listas que no sean vacías, es decir, listas que contengan al menos un elemento.

```
(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)
```

Finalmente, nos queda escribir la definición. En el caso en que la lista sea vacía, sabemos que la respuesta debe ser `#false`. El caso para listas que contienen al menos un elemento es más trabajoso. La idea será hacer un `or` entre el primer elemento de la lista, y la llamada recursiva a la función `uno-verdadero`.

```
(define (uno-verdadero l)
  (cond [(empty? l) #f]
        [else (or (first l) (uno-verdadero (rest l)))]))
```

Observar que en este caso utlizamos el operador `or`, ya que este da `#true` cuando alguno de los valores es verdadero.

1.2 Ejercicio 17

En este ejercicio tomaremos una lista de números l y un número n, y eliminaremos todas las ocurrencias de n en la lista l.

Plasmaremos esta idea en el diseño de datos:

```
; ListNum es:
; - empty
; - (cons Number ListNum)
; interpretación: un objeto de ListNum es una lista cuyos objetos son números.

; eliminar : ListNum Number -> ListNum
; Dada una lista l y un número n,
; devuelve una lista eliminando todas las ocurrencias de n
```

Luego tenemos algunos casos que ilustran mejor el funcionamiento de la función. Los marcaremos en ejemplos de testeo, así la función que hagamos cumplirá por lo menos estos casos.

```
(check-expect (eliminar (list 1 2 3 2 7 6) 2)  (list 1 3 7 6))
(check-expect (eliminar (list -2 -2 -2 -2) -2) empty)
(check-expect (eliminar (list 1 2 3 2 7 6) 0)  (list 1 2 3 2 7 6))
```

Para eliminar las ocurrencias del número n en la lista l, una solución posible es **recorrer la lista**, avanzando elemento a elemento y comparando con n. `eliminar` devuelve una lista, por lo tanto necesitamos construir una nueva lista donde para cada elemento nos encontraremos con dos posibles situaciones.

En resumen, tenemos 3 posibles casos:

1. La lista es vacía, por lo tanto, no hay más elementos (caso base). Representa que ya recorrimos la lista.
2. La lista tiene elementos y el elemento en el que estoy es igual a n, entonces lo ignoraremos y seguiremos adelante (llamando a `eliminar` en el resto de la lista. Al no incluirlo, estaremos "eliminandolo" de la lista.
3. La lista tiene elementos y el primer elemento es distinto a n, lo agregaremos a la lista porque el resultado buscado es una lista con **todos los elementos que no sean n**

```
(define (eliminar l n)
  (cond [(empty? l) empty] ; Caso 1
        [else ( if (equal? (first l) n)
                   (eliminar (rest l) n) ; Caso 2
                   (cons (first l) (eliminar (rest l) n)))])) ; Caso 3
```

1.3 Ejercicio 28

Este ejercicio nos pide definir una función `maximo` que devuelve el número máximo de una lista de naturales. Para la lista vacía, devolvemos 0.

Como siempre, empezamos con el diseño de datos:

```
; ListaNumeros es:
; - empty
; - cons Number ListaNumeros
; interpretación: es una lista de números
```

Continuamos con declaración de propósito. Dado que tenemos que devolver el número máximo de una lista de naturales, esto resulta inmediato:

```
; maximo: ListaNumeros -> Number
```

A continuación, escribimos casos de prueba para la función del ejercicio. Tenemos uno a través del enunciado, que nos dice que en caso de que la lista sea vacía, la función devuelve `0`. Para el resto de los casos elegimos listas con un elemento, con valores negativos y con valores positivos:

```
(check-expect (maximo empty) 0)
(check-expect (maximo (list -1 -2 -3) -1))
(check-expect (maximo (list -1000) -1000))
(check-expect (maximo (list 1 2 3) 3))
```

Por último, implementamos la función: sabemos que si la lista pasada como argumento es vacía, debemos devolver 0. Adicionalmente, si la lista contiene un único elemento, entonces ese elemento es el máximo de la lista. Para listas de mayor tamaño, nos quedamos con el mayor entre el primer elemento de la lista y el máximo del resto de la lista:

```
(define (maximo lista)
  (cond
    [(empty? lista) 0]
    [(empty? (rest lista)) (first lista)]
    [else (max (first lista) (maximo (rest lista)))]))
```