v.7.7

Resolución práctica 6

A continuación presentamos soluciones a ejercicios seleccionados de la Práctica 6.

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 Números Naturales

En esta práctica trabajaremos con números naturales. Para hacerlo, será necesario comprender el funcionamiento de sus constructores `0` y `add1`, su selector `sub1` y predicados correspondientes `zero?` y `positive?`.

1.1 Ejercicio 4

En este ejercicio se nos pide diseñar una función `Sigma`, que dados un número natural `n` y una función `f`, devuelva la sumatoria de los valores entre `f (0)` y `f (n)`.

Por ejemplo, si quisiéramos calcular `(Sigma 4 sqr)`, deberíamos hacer `(+ (sqr 4) (sqr 3) (sqr 2) (sqr 1) (sqr 0))`.

Empezamos con la declaración de propósito y los casos de prueba.

```
; Sigma: Natural (Natural -> Number) -> Number
; Dados un número natural n y una función f, devuelve la sumatoria
; de todos los valores entre f(0) y f(n), incluyendo ambos extremos.
(check-expect (Sigma 0 sqr) 0)
(check-expect (Sigma 4 sqr) 30)
(check-expect (Sigma 10 identity) 55)
```

Finalmente, procedemos con la definición, teniendo en cuenta el caso base y el caso recursivo.

```
(define (Sigma n f)
  (cond [(zero? n) (f n)]
        [else (+ (f n) (Sigma (sub1 n) f))]))
```

1.2 Ejercicio 11

Se nos pide diseñar una función `componer`, que dados:

- una función `f: Number -> Number`,
- un natural `n`, y
- un número `x`,

devuelva el resultado de aplicar `n` veces la función `f` a `x`.

Como siempre, comenzamos con la declaración de propósito y los casos de prueba, dentro de los cuales los primeros 2 fueron provistos por el ejercicio:

```
; componer: (Number -> Number) -> Natural -> Number
; Dados una función f, un número natural n y un valor x,
; devuelve el resultado de aplicar n veces la función f a x.
(check-expect (componer sqr 2 5) 625)
(check-expect (componer add1 5 13) 18)
(check-expect (componer identity 9999 100) 100)
(check-expect (componer sqrt 3 256) 2)
```

Por último, implementamos la función. En el caso base de la recursión devolveremos el valor `x` ya que es lo que consideramos aplicarle `f` 0 veces a `x`. Para el caso recursivo simplemente devolvemos el valor de aplicarle `f` a la llamada recursiva de `componer` con la misma función `f`, una unidad menos de `n` y el mismo valor de `x`:

```
(define (componer f n x)
  (cond
    [(zero? n) x]
    [(positive? n) (f (componer f (sub1 n) x))]))
```

1.3 Ejercicio 15

En este ejercicio el objetivo es diseñar una función que tome un número natural `m` y un número que representa el ángulo `ang`, para luego graficar cuadrados de lado m^2, $(m-1)^2$... 1^2, aumentando en 20 el ángulo en cada paso.

Comenzaremos, como siempre, con el diseño del ejercicio, con las constantes correspondientes.

Es preciso notar que para este tipo de ejercicios la función que se encargue de graficar cada uno de los cuadrados va a ser la misma. Ya lo que lo único que va a cambiar es el grado de rotación y el tamaño del lado.

Con esto en mente, pensando en modularización, podemos comenzar con una función `graficar-cuadrados` cuyo objetivo sea simplemente el de generar un cuadrado rotado, dado un tamaño lado y un ángulo.

> Recordemos que como esta función devuelve elementos de tipo `imagen`, no resulta necesario hacer tests. Esto NO significa que no realicemos tests en nuestra computadora y le "creamos" al resultado que programamos

```
; graficar-cuadrados : Natural Number -> Image
; La función graficar-cuadrados toma un valor de lado lad y un ángulo ang,
; y a partir de ahí grafica un cuadrado de lado lad y lo rota en un ángulo ang.
(define (graficar-cuadrados lad ang)
(rotate ang (square lad "outline" "slateblue")))
```

Vale aclarar que el `outline` representa que el cuadrado no esté relleno (es lo opuesto a `solid`), y el color `slateblue` es un color arbitrario.

Habiendo realizado la función que se encarga de graficar cada cuadrado, solo nos resta crear la función `cuadrado` para resolver el problema.

```
(define FONDO (empty-scene 200 200))

; cuadrado : Natural Number -> Image
; La función cuadrados toma un valor inicial m y un ángulo ang,
; y dibuja una serie de m cuadrados con lados de tamaños de los cuadrados perfectos de 1 a m,
; rotando en 20 cada cuadrado.
(define (cuadrado m ang)
  (cond
```

```
      [(zero? m) FONDO]
      [(positive? m) (place-image (graficar-cuadrados (sqr m) ang) 100 100 (cuadrado (sub1 m) (+ ang 20)))]))
```

Como la mayoría de los problemas recursivos que trabajamos, la función tiene dos casos. Si m vale cero, existen dos posibilidades:

1. El valor inicial era cero, y solo hace falta dibujar el fondo

2. El valor inicial era mayor a cero, y ya terminé de dibujar todos los elementos hasta 1

Si m no vale cero, necesito dibujar el cuadrado de lado m^2 y con un ángulo ang, es decir `(graficar-cuadrados (sqr m) ang)`, en el centro y luego dibujar todos los cuadrados restantes.

Para dibujar los cuadrados restantes, utilizo la función `cuadrado`, con el valor `(m-1)` y un ángulo equivalente a `(+ ang 20)`. Esto irá dibujando cuadrado a cuadrado, reduciendo m en 1 y aumentando el ángulo en 20 en cada paso, hasta finalmente llegar a cero (**caso base**), donde simplemente dibujará el `FONDO`.