Funciones recursivas

Lenguajes de Programación — Tarea 1

Eduardo Acuña Yeomans*

11 de agosto de 2022

Resuelve los siguientes problemas en un archivo llamado `problemas.rkt`. Asegúrate que tu código sea correcto escribiendo un archivo de pruebas `problemas-pruebas.rkt`. Se proporcionan dos archivos como plantillas, sin embargo, las pruebas proporcionadas son insuficientes para una solución satisfactoria de la tarea.

1. Implementa un procedimiento `countdown` que toma un natural y regresa una lista de los naturales que son menores o iguales en orden descendente:

```
(test-case "countdown"
  (check-equal? (countdown 5)
                '(5 4 3 2 1 0)))
```

2. Implementa un procedimiento `insertL` que toma dos símbolos y una lista y regrese una nueva lista con el segundo símbolo insertado antes de cada aparición del primer símbolo, utiliza únicamente `eqv?` para comparar.

```
(test-case "insertL"
  (check-equal? (insertL 'x 'y '(x z z x y x))
                '(y x z z y x y y x)))
```

3. Implementa un procedimiento `remv-1st` que toma un símbolo y una lista y regresa una nueva lista con la primera aparición del símbolo eliminada.

```
(test-case "remv-1st"
  (check-equal? (remv-1st 'x '(x y z x))
                '(y z x))
  (check-equal? (remv-1st 'y '(x y z y x))
                '(x z y x))
  (check-equal? (remv-1st 'z '(a b c))
                '(a b c)))
```

4. Implementa un procedimiento `map` que toma un procedimiento `p` de un argumento y una lista `ls` y regresa una nueva lista que contiene los resultados de aplicar `p` a los elementos de `ls`.

```
(test-case "map"
  (check-equal? (map sub1 '(1 2 3 4))
                '(0 1 2 3)))
```

*eduardo.acuna@unison.mx

5. Implementa un procedimiento `filter` que toma un predicado y una lista y regresa una nueva lista que contiene los elementos que satisfacen el predicado.

```
(test-case "filter"
  (check-equal? (filter even? '(1 2 3 4 5 6))
                '(2 4 6)))
```

6. Implementa un procedimiento `zip` que toma dos listas y forma una nueva lista, cada elemento en esta es un par formado de la combinación de los elementos correspondientes a las dos listas de entrada. Si las dos listas no tienen la misma longitud, ignora la cola de la mas larga.

```
(test-case "zip"
  (check-equal? (zip '(1 2 3) '(a b c))
                '((1 . a) (2 . b) (3 . c)))
  (check-equal? (zip '(1 2 3 4 5 6) '(a b c))
                '((1 . a) (2 . b) (3 . c)))
  (check-equal? (zip '(1 2 3) '(a b c d e f))
                '((1 . a) (2 . b) (3 . c))))
```

7. Implementa un procedimiento `list-index-ofv` que toma un elemento y una lista y regresa el índice de ese elemento en la lista (base 0).

```
(test-case "list-index-ofv"
  (check-eqv? (list-index-ofv 'x '(x y z x x)) 0)
  (check-eqv? (list-index-ofv 'x '(y z x x)) 2))
```

8. Implementa un procedimiento `append` que toma dos listas, `ls1` y `ls2` y regresa la concatenación de `ls1` y `ls2`.

```
(test-case "append"
  (check-equal? (append '(42 120) '(1 2 3))
                '(42 120 1 2 3))
  (check-equal? (append '(a b c) '(cat dog))
                '(a b c cat dog)))
```

9. Implementa un procedimiento `reverse` que toma una lista y regresa una lista con los mismos elementos en orden inverso.

```
(test-case "reverse"
  (check-equal? (reverse '(a 3 x))
                '(x 3 a)))
```

10. Implementa un procedimiento `repeat` que toma una lista y un natural y regresa una nueva lista con secuencias repetidas de la lista de entrada, donde la cantidad de repeticiones es igual al natural dado.

```
(test-case "repeat"
  (check-equal? (repeat '(4 8 11) 4)
                '(4 8 11 4 8 11 4 8 11 4 8 11)))
```

11. Implementa un procedimiento `same-lists*` que toma dos listas (cuyos elementos posiblemente son a su vez listas) y regresa `#t` si son iguales y `#f` de lo contrario.

```
(test-case "same-lists*"
  (check-true (same-lists* '() '()))
  (check-true (same-lists* '(1 2 3 4 5) '(1 2 3 4 5)))
  (check-false (same-lists* '(1 2 3 4) '(1 2 3 4 5)))
  (check-false (same-lists* '(a (b c) d) '(a (b) c d)))
  (check-true (same-lists* '((a) b (c d) d) '((a) b (c d) d))))
```

12. Las expresiones `(a b)` y `(a . (b . ()))` son equivalentes. Sabiendo esto, reescribe la expresión `((w x) y (z))` usando tantos puntos como sea posible. Asegúrate de probar tu solución usando el predicado `equal?`
13. Implementa un procedimiento `binary->natural` que toma una lista de ceros y unos representando un numero binario sin signo en orden inverso y que regrese ese numero.

```
(test-case "binary->natural"
  (check-eqv? (binary->natural '()) 0)
  (check-eqv? (binary->natural '(0 0 1)) 4)
  (check-eqv? (binary->natural '(0 0 1 1)) 12)
  (check-eqv? (binary->natural '(1 1 1 1)) 15)
  (check-eqv? (binary->natural '(1 0 1 0 1)) 21)
  (check-eqv? (binary->natural '(1 1 1 1 1 1 1 1 1 1 1 1 1)) 8191))
```

14. Implementa la división usando recursividad. Tu procedimiento de división `div` debe solo funcionar cuando el segundo número divide por completo al primero.

```
(test-case "div"
  (check-eqv? (div 25 5) 5)
  (check-eqv? (div 36 6) 6))
```

15. Implementa un procedimiento recursivo `append-map` similar a `map` pero que concatena los resultados de aplicar el primer argumento a cada elemento del segundo.

```
(test-case "append-map"
  (check-equal? (append-map countdown (countdown 5))
                '(5 4 3 2 1 0 4 3 2 1 0 3 2 1 0 2 1 0 1 0 0)))
```

16. Implementa un procedimiento `set-difference` que toma dos listas sin elementos repetidos `s1` y `s2` y regresa una lista con todos los elementos de `s1` que no son elementos de `s2`.

```
(test-case "set-difference"
  (check-equal? (set-difference '(1 2 3 4 5) '(2 6 4 8))
                '(1 3 5)))
```

17. Implementa un procedimiento `foldr` que toma tres argumentos: un operador binario, un acumulador inicial y una lista. Este procedimiento recorre la lista de derecha a izquierda y en cada elemento invoca al operador con el elemento de la lista y el valor del acumulador, luego actualiza el acumulador con el resultado de esta invocación. El resultado de este procedimiento es el acumulador final después de recorrer toda la lista.

```
(test-case "foldr"
  (check-equal? (foldr cons '() '(1 2 3 4))
                '(1 2 3 4))
  (check-eqv? (foldr + 0 '(1 2 3 4))
              10)
  (check-eqv? (foldr * 1 '(1 2 3 4))
              24))
```

18. En matemáticas, el conjunto potencia de un conjunto S, denotado $\mathcal{P}(S)$, es el conjunto de todos los subconjuntos de S, incluyendo el conjunto vacío y el propio S.

$$S = \{x, y, z\}$$

$$\mathcal{P}(S) = \{\{\}, \{x\}, \{y\}, \{z\}, \{x, y\}, \{x, z\}, \{y, z\}, \{x, y, z\}\}$$

El procedimiento `powerset` toma una lista y regresa el conjunto potencia de los elementos en la lista.

```
(test-case "powerset"
  (check-equal? (powerset '(3 2 1))
                '((3 2 1) (3 2) (3 1) (3) (2 1) (2) (1) ()))
  (check-equal? (powerset '())
                '(())))
```

19. El producto cartesiano es definido sobre una lista de conjuntos (representados como listas sin duplicados). El resultado es una lista de tuplas (representadas como listas). Cada tupla tiene en la primera posición un elemento del primer conjunto, en la segunda posición un elemento del segundo conjunto, etc. La lista resultante debe contenter todas las combinaciones. El orden en la lista resultante no es relevante.

```
(test-case "cartesian-product"
  (check-equal? (cartesian-product '((5 4) (3 2 1)))
                '((5 3) (5 2) (5 1) (4 3) (4 2) (4 1))))
```

20. Implementa los siguientes procedimientos pero utilizando `foldr`:
    - `insertL-fr`
    - `filter-fr`
    - `map-fr`
    - `append-fr`
    - `reverse-fr`
    - `binary->natural-fr`
    - `append-map-fr`
    - `set-difference-fr`
    - `powerset-fr`

21. Considera una función f definida de la siguiente manera

$$f(n) = \begin{cases} n/2 & \text{si } n \equiv 0 \pmod 2 \\ 3n+1 & \text{si } n \equiv 1 \pmod 2 \end{cases}$$

Tu objetivo es completar la definición de `snowball` (ver listado 1), la cuál cuando se le da de entrada un entero positivo siempre regresa 1. Este procedimiento se debe comportar

exactamente como la f de arriba. Lo que acompletes de la respuesta debe ser muy corto (una linea) y no debe de usar `lambda`.

```
(test-case "snowball"
  (check-eqv? (snowball 12) 1)
  (check-eqv? (snowball 120) 1)
  (check-eqv? (snowball 9999) 1))
```

22. Un *quine* es un programa cuya salida es el código fuente del programa original, en Racket, los valores simples son quines:

```
> 5
5
> #t
#t
```

Decimos que un quine que no es un valor simple es un quine interesante:

```
> ((lambda (x) (list x (list 'quote x)))
   '(lambda (x) (list x (list 'quote x))))
((lambda (x) (list x (list 'quote x)))
 '(lambda (x) (list x (list 'quote x))))
```

Escribe tu propio quine interesante y definelo como `quine`.

```
(test-case "snowball"
  (let ((ns (make-base-namespace)))
    (check-equal? (eval quine ns) quine)
    (check-equal? (eval (eval quine ns) ns) quine)))
```

No todas las listas son quines, asegurate de probar tu quine usando `eval` como se muestra arriba.

```
(define snowball
  (letrec
      ((odd-case
        (lambda (fix-odd)
          (lambda (x)
            (cond
              ((and (exact-integer? x) (positive? x) (odd? x))
               (snowball (add1 (* x 3))))
              (else (fix-odd x))))))
       (even-case
        (lambda (fix-even)
          (lambda (x)
            (cond
              ((and (exact-integer? x) (positive? x) (even? x))
               (snowball (/ x 2)))
              (else (fix-even x))))))
       (one-case
        (lambda (fix-one)
          (lambda (x)
            (cond
              ((zero? (sub1 x)) 1)
              (else (fix-one x))))))
       (base
        (lambda (x)
          (error 'error "Invalid value ~s~n" x))))
    ...))
```

Listing 1: Impementación incompleta de `snowball`