012. Une dos listas tipo cierre de cremallera

Escriba una función **`zip2`** que toma dos listas y genera una lista de pares, donde el primer elemento de un par es un elemento de la primer lista, y el segundo elemento de un par es un elemento de la segunda lista. Si una lista es más corta que la otra, utilice **`null`** en lugar del elemento faltante.

Al final entonces la lista resultante tendrá la longitud que corresponde a la lista más larga. Ejemplo:

```
(zip2 ‘(1 2 3 4 5)  ‘(a b c d e)) -> ((1 a) (2 b) (3 c) (4 d) (5 e))
(zip2 ‘(1 2 3 4 5) ‘(a b c)) -> ((1 a) (2 b) (3 c) (4 ()) (5 ()))
(zip2 ‘(1 2)  ‘(a b c d e)) -> ((1 a) (2 b) (() c) (() d) (() e))
(zip2 ‘() ‘()) -> ‘()
```

Utilizamos el enfoque de casos para el pensamiento recursivo.

Veamos el primer caso, con 5 elementos en cada lista. Si cortamos la cabeza de cada lista, nos quedamos con
`1 / ‘(2 3 4 5)`
`a / ‘(b c d e)`

Ahora vemos que las colas de la lista tienen la misma forma que el problema original: son dos listas que deben *zipearse* juntas, así que podemos aplicar recursivamente la función que estamos escribiendo, de tal manera que si todo sale bien, la salida será:

```
(zip2 ‘(2 3 4 5)  ‘(b c d e)) -> ((2 b) (3 c) (4 d) (5 e))
```

Bien, ahora debemos preguntarnos, ¿cómo podemos combinar los elementos **`1`**, **`a`**, y la lista **`((2 b) (3 c) (4 d) (5 e))`** para obtener **`((1 a) (2 b) (3 c) (4 d) (5 e))`**?

Podemos crear una lista **`(1 a)`** y ***`cons`**earla* en el resultado recursivo:

```
(cons (list (car xs) (car ys)) (zip2 (cdr xs) (cdr ys)))
```
donde **`xs`**, **`ys`** son los argumentos de nuestra función.

En cada vuelta de la recursión, tanto **`xs`** como **`ys`** se van acortando. En algún momento una de ellas o ambas a la vez quedarán vacías. Resolvamos primero el caso que ambas quedan vacías a la vez.

```
(define zip2
  (lambda (xs ys)
    (if (and (null? xs) (null? ys))
      null
      (cons (list (car xs) (car ys)) (zip2 (cdr xs) (cdr ys))))))
```

Podemos comprobar cómo funciona este código por ejemplo con:

```
(zip2 ‘(1 2 3 4 5)  ‘(a b c d e)) -> ((1 a) (2 b) (3 c) (4 d) (5 e))
(zip2 ‘() ‘()) -> ‘()
```

Si queremos agregar la funcionalidad de listas de distinta longitud, vamos a tener que elaborar mejor los argumentos de list, pues pueden ser el car de uno de los argumentos o null si es que se han acabado los elementos de una lista. Primero rescribimos el código usando let, que nos dará más capacidad expresiva:

```
(define zip2
  (lambda (xs ys)
    (if (and (null? xs) (null? ys))
      null
      (let [(x (car xs))
            (y (car ys))]
        (cons (list x y) (zip2 (cdr xs) (cdr ys)))))))
```

Este cambio no debe alterar el comportamiento de la función.

Ahora podemos pensar con claridad: si ambas están vacías, entonces se retorna **`null`**. Eso está correcto; si no lo controlamos de esta manera podemos caer en el caso que ambas se vacían y el programa emite al final una infinita cantidad de pares **`(() ())`**, en lugar de terminar.

Analicemos cómo obtenemos el valor de x y de y. Son el primer elemento de cada lista, y eso es correcto mientras haya elementos, sino debería ser null; o sea, nos falta un condicional para eso. Veamos:

```
(define zip2
  (lambda (xs ys)
    (if (and (null? xs) (null? ys))
      null
      (let [(x (if (null? xs) null (car xs)))
            (y (if (null? ys) null (car ys)))]
        (cons (list x y) (zip2 (cdr xs) (cdr ys)))))))
```

Sin embargo, nos queda un problema por resolver, pues no podemos acortar la lista vacía (eso da un error), así que hay que alterar la llamada recursiva **`(zip2 (cdr xs) (cdr ys))`** para que sólo haga el **`cdr`** de la lista que aún tiene elementos.

Aprovechamos la forma especial **`let`** para crear otros dos identificadores— **`xz`**, **`yz`** —asociados con las colas adecuadas a la recursión: si una lista es vacía, se debe seguir con **`null`**, si una lista no es vacía, se debe usar su **`cdr`**. Nos queda:

```
(define zip2
  (lambda (xs ys)
    (if (and (null? xs) (null? ys))
      null
      (let [(x  (if (null? xs) null (car xs)))
            (xz (if (null? xs) null (cdr xs)))
            (y  (if (null? ys) null (car ys)))
            (yz (if (null? ys) null (cdr ys)))]

        (cons (list x y) (zip2 xz yz))))))
```

Note que el cuerpo de la función es el mismo: poner el par de los primeros elementos como primer elemento de la salida.

Podemos verificar ahora el caso de listas de longitud desigual:

```
(zip2 ‘(1 2 3 4 5) ‘(a b c)) -> ((1 a) (2 b) (3 c) (4 ()) (5 ()))
(zip2 ‘(1 2)  ‘(a b c d e)) -> ((1 a) (2 b) (() c) (() d) (() e))
```

Con funciones de orden superior

```
(define zip2
  (lambda (xs ys)
    (foldr
      (lambda (x y acum)
        (cons (list x y) acum))
      ‘()
      xs
      ys)))
```