010. Concatenar dos listas

Escriba un programa concatena que toma dos listas como argumento y retorna una lista cuyos elementos son todos los de la primer lista seguidos de todos los elementos de la segunda lista.

Algunos ejemplos de invocación a concatena:

```
(concatena ‘() ‘(a b c)) -> ‘(a b c)
(concatena ‘(a b c) ‘()) -> ‘(a b c)
(concatena ‘(a b c) ‘(d e)) -> ‘(a b c d e)
(concatena ‘() ‘()) -> ‘()
```

Se puede apreciar que la lista vacía es el elemento neutro de la operación concatena.

¿Cómo pensamos en la solución recurrente para un problema?

Siempre prestamos atención a los valores que se ingresan como entrada a nuestra función y los valores esperados como resultado.

La forma de trabajar recursivamente es poder expresar el problema completo de manera de reducirlo de tamaño (un número menos, un elemento menos en la lista, un nodo menos si es un árbol, etc.). El problema reducido debe tener la misma ***forma*** que el problema original. Si el problema original toma una lista de números como entrada, el problema reducido también toma una lista de números.

El problema reducido se puede resolver mediante una llamada recursiva a la función que estamos escribiendo, pues satisface la descripción del enunciado de la misma.

La llamada recursiva nos proporciona un resultado parcial.

Al reducir el problema, hemos sacado algo de la estructura original, un elemento. Puede ser la cabeza de la lista, o un carácter de un string, o un nodo de un árbol.

Ahora viene el momento de la composición: con el elemento que saqué y con el resultado parcial debo combinarlos para conseguir el resultado final

Revisemos algunos de los programas que ya estudiamos:

a) la longitud de la lista: como saque la cabeza, a la longitud de la cola debo incrementarla en uno para conseguir la longitud total
b) sumar una lista de números: agregar la cabeza a la suma de la cola para tener la suma total
c) pertenece?: la cabeza es el que busco, o el que busco está en la cola de la lista

d) enésimo elemento: cada vez que saco una cabeza, decremento el indicador de posición buscada
e) factorial: decremento el número solicitado y calculo su factorial, luego multiplico el factorial parcial por el número solicitado y obtengo el factorial buscado

En cada caso, hemos encontrado una manera de transformar el problema completo, en uno con la misma forma pero un poco más pequeño. Esta reducción se continúa hasta que se consigue un subproblema cuya resolución es inmediata, o al menos no requiere de llamadas recursivas a nuestra función. En ese momento, la solución inmediata se obtiene y se compone con aquello que se separó para reducir el problema, se combina creando una solución de nivel superior que se devuelve como resultado recursivo. Esto sucede así, desenvolviendo la recursión hasta que se llega a la llamada de primer nivel, donde se compone la solución total.

La concatenación

A los fines de la explicación siguiente, vamos a utilizar el símbolo ▫ para denotar la operación **cons** en formato infijo.

Una lista **‘(a b c)** se puede describir mediante la expresión **a ▫ b ▫ c ▫ ‘()**

Supongamos que nuestra función tiene argumentos **xs** y **ys**.

Consideremos un algoritmo que se aprovecha de la lista vacía como elemento neutro de la operación concatenar. Si logramos transformar una lista de entrada en la lista vacía, entonces para ese caso la respuesta es el valor del otro argumento, como se aprecia en los dos primeros ejemplos más arriba. Analicemos el caso en el cual trabajamos sobre el primer argumento:

```
(concatena ‘(a b c) ‘(d e))
  a ▫ (concatena ‘(b c) ‘(d e))
  a ▫ ‘(b c d e)
‘(a b c d e)
```

El algoritmo funciona quitando un elemento de **xs** al cual luego va a agregar en el resultado recursivo. Vemos el esquema de funcionamiento antes apuntado:

1. hacemos más *pequeño* el problema
2. lo pasamos por la misma función que estamos escribiendo
3. obtenemos un resultado parcial
4. componemos aquello que extrajimos del problema original con el resultado parcial
5. retornamos el resultado total.

Veamos el código Scheme:

```
(define concatena
  (lambda (xs ys)
    (if (null? xs) ys
      (cons (car xs) (concatena (cdr xs) ys)))))
```

La tentación de arrancar por los casos base

Se puede ver que hay tres casos particulares:

a) cuando **xs** es vacía, el resultado es **ys**
b) cuando **ys** es vacía, el resultado es **xs**
c) cuando ambas son vacías, el resultado es la lista vacía.

La tentación entonces pasa por poner condicionales para que en cada uno de los tres casos, se devuelva el resultado inmediato.

Esto sucede cuando pensamos la recursión como un caso base y un caso recursivo, buscamos identificar cada uno de ellos, y escribir el código que los comtemple.

Pero si bien muchos problemas muy simples obedecen a ese esquema, lo cierto es que no se trata del caso general, el que es más poderoso.

En el caso de concatena, b) y c) son casos especiales ya contemplados por el código que escribimos. Alguno puede argumentar que considerar los casos especiales de maner explícita hace más eficiente el programa, y eso es cierto en el caso b).

Debemos recordar que nuestro trabajo como programadores es "*hacer bien las cosas buenas*", es decir, que lo importante son las "cosas buenas"; imagino que "hacer bien las cosas erradas" no es un buen lema para la profesión. Primero hacemos que sea correcto, que responda con las respuestas correctas, luego lo hacemos eficiente (si de verdad *esa* eficiencia importa cuando lo medimos).