007. Invertir el orden de los elementos de una lista

Hay varias opciones de enunciado que podemos estudiar. Veamos cada una por turno. Éste es un enunciado que en realidad encierra varios problemas diferentes.

007.1 Escribe una función Scheme que toma una lista de elementos como entrada y su salida es una lista con los mismos elementos pero en orden inverso. Aquellos elementos que son listas internamente no son invertidos, sólo los del primer nivel. Ejemplos de invocación son:

```
(reversa ‘(a b c d))                    ->  ‘(d c b a)
(reversa ‘(a (1 2 3 4) c (x y (z)) d)) ->  ‘(d (x y (z)) c (1 2 3 4) a)
```

Revisemos nuestro enfoque recursivo para el problema.

a) Identificamos un caso de entrada y su salida esperada, para trabajar con algo concreto en mente.
b) Identificamos una estructura recursiva en los datos: en este caso es una lista que tiene cabeza y cola
c) Calculamos a mano cuál sería el resultado de nuestra función cuando la aplicamos a la cola
d) Pensamos cómo combinar la cabeza con el resultado anterior, para obtener el resultado deseado.

Vamos a aplicar este método al primer ejemplo más arriba.

a) **`(a b c d) -> (d c b a)`**
b) **`a`** --- **`(b c d)`** [este resultado lo debe conseguir la llamada recursiva]
c) **`(b c d) -> (d c b)`**
d) ¿Cómo combinar a con **`(d c b)`** para conseguir **`(d c b a)`**? Parece que hay que insertar a al final de la lista **`(d c b)`**.

Con eso en mente ya podemos escribir la parte recursiva de la función:

```
(define reversa
  (lambda (lst)
    (let [(cabeza (car lst))
          (cola (cdr lst))]
      (append (reversa cola)
              (list cabeza)))))
```

Sólo pusimos nombres descriptivos con **`let`**, y escribimos la fórmula de la combinación que analizamos en el punto d). Es claro que este programa no funciona porque estamos haciendo la llamada recursiva pero no hay un caso de solución inmediata, así que la recursión va a proseguir hasta que la lista se vacíe y ocasione un error en el **`car`**.

Ahora nos concentramos en que pasa cuando la lista lst se reduce en cada llamada. Y eso nos hace pensar en cuál es el caso límite de la inversión de lista: ¿es cuando la lista está vacía, o cuando tiene un solo elemento? ¿Tiene sentido decir que una lista vacía tiene invertidos sus elementos, cuando por definición no hay elementos en una lista vacía? Éste es el momento de pensar esas cosas. El enunciado no las aclara.

Supongamos que una lista vacía es aquella que tiene sus elementos en orden inverso (Mmmmhhh, no suena bien). Entonces el código debe comprobar ese argumento de entrada, y retornar el resultado que hemos acordado:

```
(define reversa
  (lambda (lst)
    (if (null? lst) null
      (let [(cabeza (car lst))
            (cola (cdr lst))]
        (append (reversa cola)
                (list cabeza))))))
```

Con esto completamos esta versión del programa.

Supongamos ahora que una lista vacía no puede ser un argumento a la función. Entonces el caso límite de la lista lst mientras se reduce será cuando contenga un único elemento. Una lista de un elemento es idéntica a su lista invertida. Veamos cómo queda, con control de error y todo:

```
(define reversa
  (lambda (lst)
    (if (null? lst) (error "reversa: la entrada no puede estar vacia")
      (let [(cabeza (car lst))
            (cola (cdr lst))]
        (if (null? cola)
            lst
            (append (reversa cola)
                    (list cabeza)))))))
```

007.2 Escribe una función Scheme que toma una lista de elementos como entrada y su salida es una lista con los mismos elementos pero en orden inverso. Aquellos elementos que son listas son invertidos también.

Si echamos una mirada al código vemos que el núcleo del programa es:

```
            (append (reversa cola)
                    (list cabeza)))
```

La cola de la lista se invierte, y se le agrega al final la cabeza de la lista. Esta cabeza tiene dos formas posibles, lo que nos da dos casos extras: puede ser un átomo, o puede ser una lista. El caso del átomo es el que ya vimos: lo envolvemos en una lista (para defenderlo del **append** que trabaja con listas) y aplicamos **append**.

Qué hay que hacer si cabeza es una lista, pues invertirla recursivamente, envolverla con **list** y agregarla al final con **append**. Escribamos eso:

```
(define reversa
  (lambda (lst)
    (if (null? lst) null
      (let [(cabeza (car lst))
            (cola (cdr lst))]
        (append (reversa cola)
                (list (if (list? cabeza) (reversa cabeza) cabeza)))))))
```

Hemos vuelto a la versión que considera las listas vacías invertibles.

Note la sustitución de:

```
            (append (reversa cola)
                    (list cabeza)))
```

por:

```
        (append (reversa cola)
                (list (if (list? cabeza) (reversa cabeza) cabeza)))
```

Lo único que cambiamos es el argumento de **list**, que dependiendo de si **cabeza** es una lista se lo invierte, o no en otro caso.

En este caso hemos empleado la forma **if** como una expresión condicional, y no como se usaría en un lenguaje imperativo como estructura de control a nivel de sentencias: lo que nos interesa es el valor que devuelve el **if**.

007.3 Escribe una función Scheme que toma una lista de elementos como entrada y su salida es una lista con los mismos elementos pero en orden inverso. La lista de entrada es una lista binaria donde el primer elemento es un átomo y el segundo es una lista de elementos anidados, salvo en el último nivel, que consiste de una lista de un solo elemento. Ejemplo: **‘(a (b (c))))**

La lista invertida tiene la misma estructura de anidamiento, pero los elementos están en orden inverso. En nuestro ejemplo: **(reversa ‘(a (b (c)))) -> (c (b (a)))**

En este problema el primer ciclo de la recursión no parece ayudarnos. Puede parecer que tenemos una lista como argumento, y eso nos llevaría a pensar que la podemos cortar con **car** y **cdr** para obtener un subproblema de la misma forma y que aplicamos recursivamente

nuestra solución, pero no es así, el argumento es un árbol, no una lista. Probemos este punto antes de seguir adelante. Podemos hacer un diagrama con las celdas cons (queda de ejercicio para el lector) y podemos analizar el caso recurrente, que haremos a continuación:

Tomemos un ejemplo más grande:

```
'(a (b (c (d (e (f)))))) -> (f (e (d (c (b (a))))))
```

Supongamos ahora que cortamos la cabeza a y pasamos la cola a la función para que calcule la llamada recursiva:

cabeza: **a** / cola: **(b (c (d (e (f))))))**

Entonces esperamos que al llamar a **(reversa (b (c (d (e (f))))))) nos devuelva (f (e (d (c (b (a))))))**. Sin embargo, **reversa** es una función de un solo argumento, y claramente necesita conocer la cabeza **a** para poder insertarla en la última posición. No alcanza conocer la cabeza al final de la llamada recursiva, como hacíamos en el caso del factorial.

Lo que necesitamos es que reversa disponga de un acumulador local, una pila, donde vaya apilando las cabezas que va desapilando (cortando) de la lista argumento. Podemos escribir reversa con un argumento adicional para este acumulador, donde se va acumulando los resultados parciales, y por eso lo denominamos res.

El resultado parcial se envuelve en una lista, y se agrega como primer elemento de esa nueva capa de listas al elemento desapilado del argumento. Entonces tenemos:

```
'(a (b (c (d (e (f))))))
(reversa ‘(b (c (d (e (f))))) ‘(a) )
(reversa ‘(c (d (e (f)))) (cons b (list (a)) )
(reversa ‘(d (e (f))) (cons c (list (b (a))) )
(reversa ‘(e (f)) (cons d (list (c (b (a)))) )
(reversa ‘(f) (cons e (list (d (c (b (a))))) )
```

Y aquí está la tentación de invocar recursivamente una vez más a reversa. PEro note un tema muy importante: el argumento de entrada ha dejado de tener la forma especificada en el enunciado. Ya no es una estructura anidada, sino simple, y eso es el caso base. El código que implementa estas transformaciones de entrada en salida está a continuación, y se obtiene de mirar con cuidado la secuencia de valores más arriba. Como ejercicio, intente descubrirlo antes de mirar la solución.

```
(define reversa
  (lambda (lst res)
    (if (null? (cdr lst)) (cons (car lst) (list res))
      (reversa (cadr lst) (cons (car lst) (list res))))))
```

Para cumplir con el enunciado, que nos pide una función reversa de un solo argumento, podemos rescribirla como función auxiliar, visible dentro de la solicitada. La función principal construye el valor del inicial del acumulador, en un proceso que podemos denominar “cebado” en analogía con los motores a combustión y la autoexcitación inicial mediante baterías de generadores eléctricos.

```
(define reversa
  (lambda (lst)
    (define rev
      (lambda (lst res)
        (if (null?  (cdr lst)) (cons (car lst) (list res))
          (rev (cadr lst) (cons (car lst) (list res))))))
    (rev (cadr lst) (list (car lst)))))
```

Usemos algunas pruebas cuyos argumentos son listas anidadas:

```
(display (reversa '(b (c)))) ;-> (b (a))
(display (reversa '(a (b (c))))) ;-> (c (b (a)))
(display (reversa '(a (b (c (d (e (f)))))))))
```

El caso base no tiene forma de lista anidada, y por lo tanto no es un candidato aceptable para la función reversa:

```
(display (reversa '(c))) ; -> error
```

Hemos usado un **`define`** dentro de otro **`define`** para crear una función auxiliar **`rev`**, visible solamente dentro de **`reversa`**. Con esto cumplimos con la consigna respecto a cuántos argumentos debía aceptar la función solución del enunciado.