008. Elimina duplicados de una lista

Escriba una función que toma una lista de elementos con o sin duplicados y devuelve una lista con los elementos en el mismo orden, pero sólo la primer copia de los duplicados.

```
(dedup ‘(1 2 3 4 2 5))   -> ‘(1 2 3 4 5)
(dedup ‘(1 2 3 4 5))     -> ‘(1 2 3 4 5)
(dedup ‘(1 1 1 2 3 4 5)) -> ‘(1 2 3 4 5)
```

Una variante

Hagamos una variante del enunciado, en la cual el elemento que se deja en el resultado es el último que se encuentra en la lista

```
(dedup-cola ‘(1 2 3 4 2 5))   -> ‘(1 3 4 2 5)
(dedup-cola ‘(1 2 3 4 5))     -> ‘(1 2 3 4 5)
(dedup-cola ‘(1 1 2 3 4 5 1)) -> ‘(2 3 4 5 1)
```

Todo elemento de la lista de entrada debe estar en la lista de salida, por lo tanto se toma la cabeza y se la agrega como primer elemento del resultado, excepto que la cabeza pertenece a la cola: con lo cual quiere decir que la cabeza es el repetido que hay que eliminar.

```
(define dedup-cola
  (lambda (lst)
    (if (null? lst) '()
      (let [ (cabeza (car lst))
             (cola   (cdr lst)) ]
        (if (member cabeza cola)
          (dedup-cola cola)
          (cons cabeza (dedup-cola cola)))))))
```

Retomamos el problema original

El problema original requiere que sea el primer ejemplar de cada duplicado el que quede, pero tenemos un programa que deja el duplicado del final. O sea que es un tema de orden, pues bien, si invertimos la lista, lo tenemos resuelto:

```
(define dedup
  (lambda (lst)
    (reverse (dedup-cola (reverse lst)))))
```

Una mejora en el ambiente

Nos han solicitado una función **dedup**, pero para facilitarnos el trabajo, hemos creado una función auxiliar **dedup-cola**, la cual ahora ensucia el ambiente de nombres de la aplicación. ¿Podemos crear una función auxiliar que solamente sea visible para nuestra función **dedup**?

```
(define dedup
  (lambda (lst)

    (letrec [ (dedup-cola
                (lambda (xs)
                  (if (null? xs) '()
                    (let [(cabeza (car xs))
                          (cola   (cdr xs)) ]
                      (if (member cabeza cola)
                        (dedup-cola cola)
                        (cons cabeza (dedup-cola cola))))))) ]

    (reverse (dedup-cola (reverse lst))))))
```

La forma **letrec** (**let rec**ursivo) nos permite asociar localmente el identificador **dedup-cola**, y usarlo en el cuerpo más adelante. Fuera de **dedup**, nadie puede verlo.

Esto es parecido a cuando en matemática escribimos cosas como:

$$x = \frac{-b \pm \sqrt{\det}}{2a}$$

donde:

$$\det = b^2 - 4\,a\,c$$

El **let** y el **letrec** son el equivalente del **donde**, que nos permite definir nombres abreviados para expresiones más o menos complejas o repetitivas.

Una implementación sin truco con reverse

Está bien, veamos cómo escribir una implementación sin en truco del doble reverse de la lista. Aunque haz de reconocer que estuvo *pretty cool*!

Nuestro problema con la definición que pide el primer ejemplar de todos los duplicados es que cuando analizamos la cabeza de la lista y nos damos cuenta que es el que debe quedar, si lo montamos en la recursión como hace **dedup-cola**, lo perdemos de vista en ls recursiones anidadas.

Tenemos que conseguir una estructura de datos que conserve los elementos ya aprobados, de manera que cuando analicemos un elemento nuevo, podamos comprobar si ya lo pusimos en el resultado o no. Esta estructura se parece sospechosamente al resultado final solicitado.

Entonces lo que podemos hacer es utilizar una suerte de acumulador donde vayamos ingresando los elementos correctos, y pasar este acumulador en las llamadas recursivas para usarlo de lista de comprobación. Eventualmente cuando lleguemos al caso base, el resultado será exactamente el contenido del acumulador.

Empezamos construyendo dedup con dos argumentos: uno es la lista a simplificar, y el otro es el acumulador con el resultado inicial, como iremos agregando elementos, el elemento neutro será la lista vacía.

Recuperamos los casos de prueba para tenerlos presentes mientras programamos:

```
(dedup ‘(1 2 3 4 2 5) ‘())   -> ‘(1 2 3 4 5)
(dedup ‘(1 2 3 4 5) ‘())     -> ‘(1 2 3 4 5)
(dedup ‘(1 1 1 2 3 4 5) ‘()) -> ‘(1 2 3 4 5)
```

```
(define dedup
  (lambda (xs res)
    (if (null? xs) (reverse res)
      (let [(cabeza (car xs))
            (cola (cdr xs)) ]
        (if (member cabeza res)
          (dedup cola res)
          (dedup cola (cons cabeza res)))))))
```

Como se van agregando mediante **cons** los elementos al resultado, quedan en orden inverso, y usamos **reverse** para enderezarlos.
En vez de ingresarlos con **cons** a la lista de resultados, los podemos ir agregando al final con **append**, para mantener el orden correcto:

```
(define dedup
  (lambda (xs res)
    (if (null? xs) res
      (let [(cabeza (car xs))
            (cola (cdr xs)) ]
        (if (member cabeza res)
          (dedup cola res)
          (dedup cola (append res (list cabeza))))))))
```