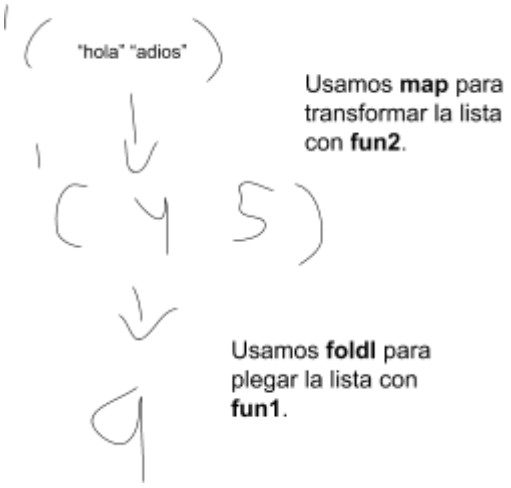
Ej 2

Apartado b: Dada la función recursiva generica-rec, rellena el hueco para implementar la función equivalente generica-fos utilizando una función de orden superior:

```
(define (generica-rec fun1 fun2 lista base)
        (if (null? lista)  base
                (fun1 (fun2 (car lista))
                        (generica-rec fun1 fun2 (cdr
                lista) base))))

(check-equal? (generica-rec + string-length '("hola"
"adios") 0)
                (generica-fos + string-length '("hola"
        "adios") 0))
```



(define (generica-fos fun1 fun2 lista base)
(foldl fun1 base (map fun2 lista)))

Apartado c: Indica qué debe haber en los huecos para que las siguientes pruebas de f y g sean correctas:

```
(define (f x)   (lambda (y)     (+ x y)))
(define (g x)   (lambda (y)     (- x y)))
```

prueba de f:
(check-equal? ((f 3) 3) 6)

prueba de g:
(check-equal? ((g 6) 3) 3)

Apartado d: ¿Qué debería haber en el hueco para que el check-equal sea correcto?

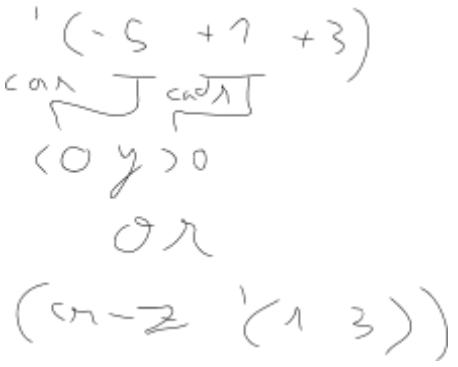
```
(check-equal? (fold-left (lambda (x base)
                           (cons (cons (cdr x) (car x)) base)) '()
                         '((1 . 20) (5 . 8) (7 . 10))) '((10 . 7) (8 . 5) (20 . 1)))
```

Ejercicio 3

Apartado a: Escribe la función recursiva (cruza-cero? lista) que recibe una lista ordenada creciente de números y comprueba si los números pasan de negativo a positivo (cruzan por el cero).

Ejemplos:

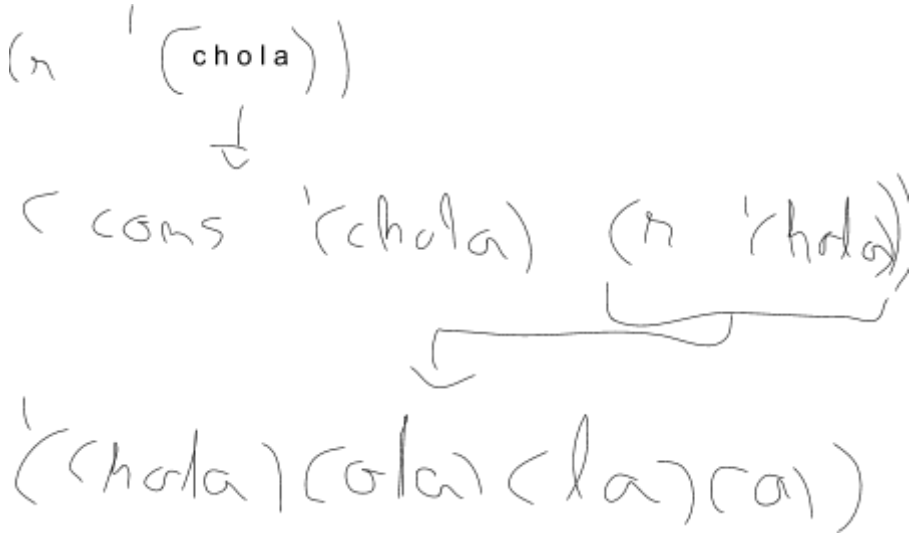
(cruza-cero? '(-10 -5 -2 10 20)) ⇒ #t
(cruza-cero? '(-20 -12 -9 -3)) ⇒ #f
(cruza-cero? '(3 12 18 20 25)) ⇒ #f



```
(define (cruza-cero? lista)
        (if (null? (cdr lista)) #f
                (or (and (<= (car lista) 0) (> (cadr lista) 0))
                    (cruza-cero? (cdr lista)))))
```

Apartado b: Define la función recursiva restagrama que verifica los siguientes check-equal?:

(check-equal? (restagrama '(c h o l a)) '((c h o l a) (h o l a) (o l a) (l a) (a)))
(check-equal? (restagrama '(u n o)) '((u n o) (n o) (o)))

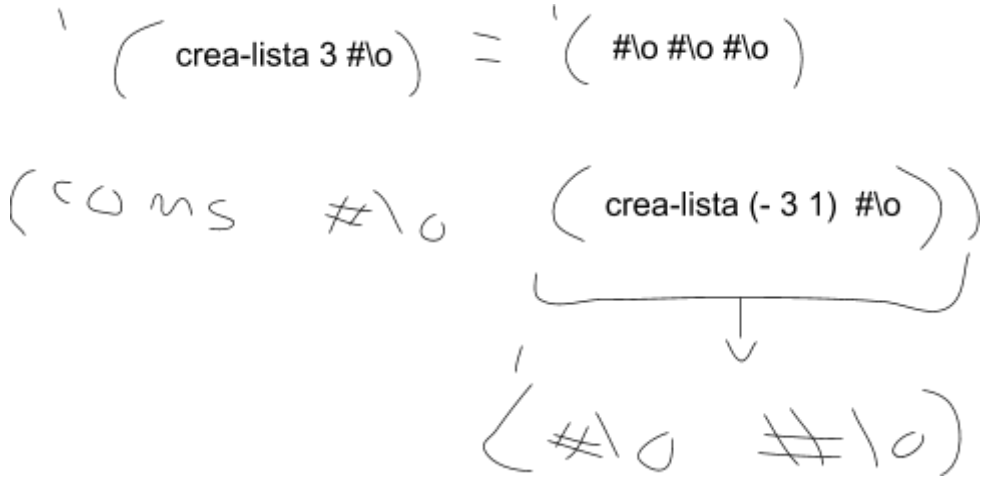



```
(define (restagrama lista)
        (if (null? lista) '()
                (cons lista (restagrama (cdr lista)))))
```

Ejercicio 4

Apartado a: Escribe la función recursiva (crea-lista n elem) que recibe un elemento y un número y devuelve una lista con n repeticiones del elemento.

Ejemplos:
(crea-lista 3 #\o) ⇒ '(#\o #\o #\o)
(crea-lista 5 2) ⇒ '(2 2 2 2 2)

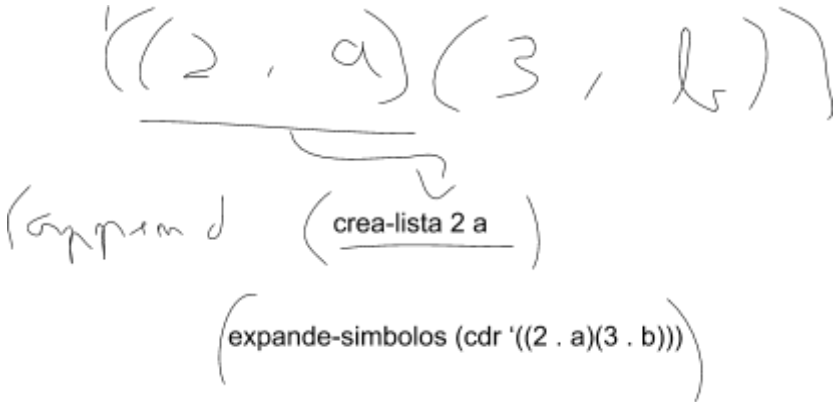



```
(define (crea-lista n elem)
        (if (= n 0) '()
                (cons elem (crea-lista (- n 1) elem))))
```

Apartado b: Escribe la función recursiva (expande-simbolos lista-parejas) que use la función anterior crea-lista y devuelva una lista con los elementos definidos por las partes derechas de las parejas con símbolos tantas veces como indica el número de las partes izquierdas.

Pista: recuerda que la función para comprobar si un dato es un símbolo es (symbol? dato).
Ejemplos:

(expande-simbolos '((2 . a) (4 . #\a) (3 . b) (5 . #f))) ⇒ '(a a b b b)

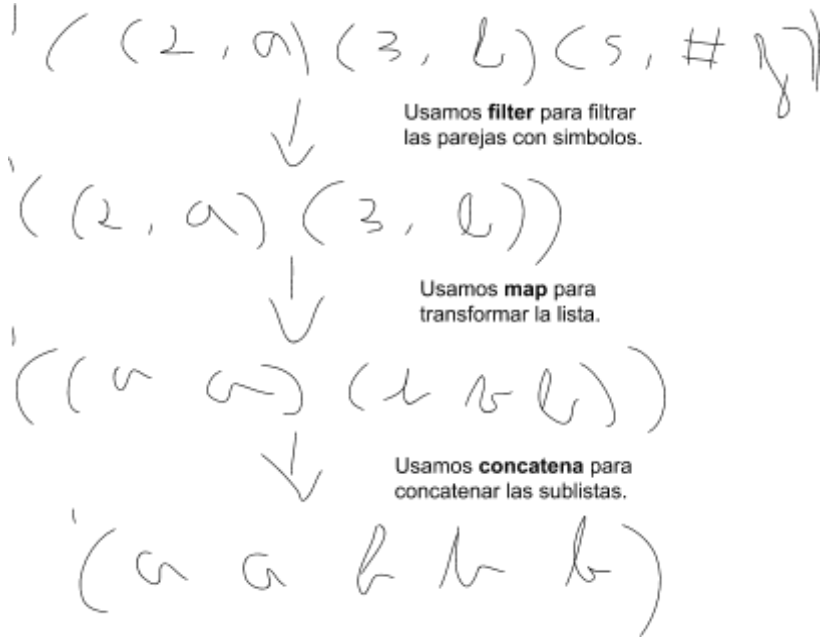



```
(define (expande-simbolos lista-parejas)
        (cond ((null? lista-parejas) '())
              ((not (symbol? (cdar lista-parejas))) (expande-simbolos (cdr
        lista-parejas)))
              (else   (append (crea-lista (caar lista-parejas) (cdar lista-parejas))
                              (expande-simbolos (cdr lista-parejas))))))
```

Apartado c: Usando una composición de funciones de orden superior implementa la función (expande-simbolos-fos lista-parejas) que haga lo mismo que la función anterior.

Pista: Puedes usar la función (concatena lista) que recibe una lista con listas y las concatena todas:

```
(define (concatena lista)
        (fold-right append '() lista))
```
(concatena '((a a) (b b b))) ⇒ (a a b b b)




```
(define (expande-simbolos-fos lista-parejas)
        (concatena (map (lambda (x)
                          (crea-lista (car x) (cdr x)))
                        (filter (lambda(x)
                                  (symbol? (cdr x))) lista-parejas))))
```

Ejercicio 5

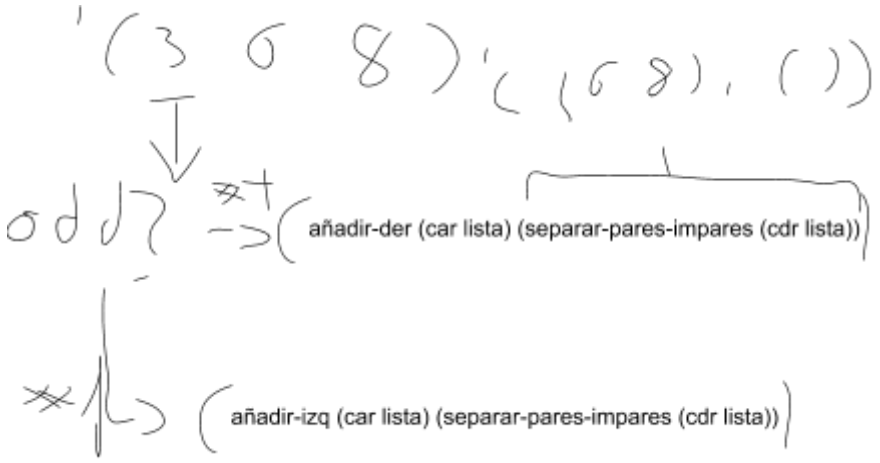
Apartado a: Define las funciones (añade-izq elem pareja-listas) y (añade-der elem pareja-listas) que funcionen como indican los siguientes check-equal?:

(check-equal? (añade-izq 'a (cons '(b c) '(d))) (cons '(a b c) '(d)))
(check-equal? (añade-der 'a (cons '(b c) '(d))) (cons '(b c) '(a d)))

```
(define (añade-izq elem pareja-listas)
        (cons (cons elem (car pareja-listas)) (cdr pareja-listas)))
(define (añade-der elem pareja-listas)
        (cons (car pareja-listas) (cons elem (cdr pareja-listas))))
```

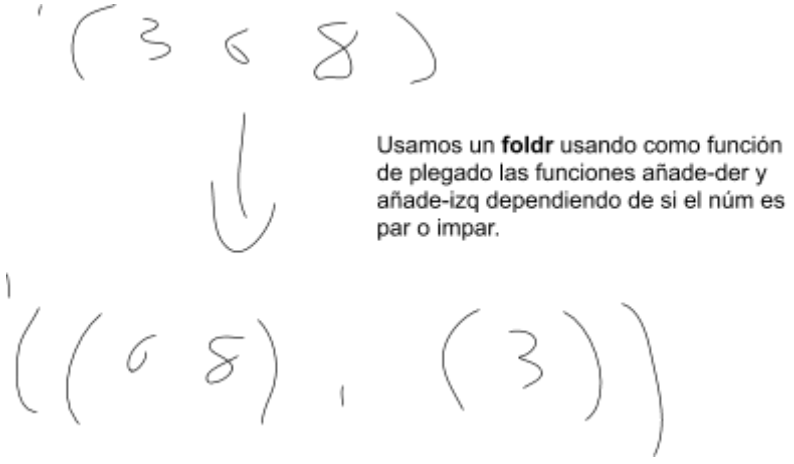
Apartado b: Define la función recursiva (separar-pares-impares lista-num) que, dada una lista de números, devuelva una pareja cuya parte izquierda sea una lista con los números pares y su parte derecha una lista con los números impares de la lista. Puedes usar las funciones del apartado anterior, suponemos que están bien definidas.

Ejemplo:
(separar-pares-impares '(3 6 8 1 5 4)) ⇒ {{6 8 4} . {3 1 5}}




```
(define (separar-pares-impares lista)
        (cond ((null? lista-num) '(() . ()))
                ((odd? (car lista-num)) (añade-der (car lista) (separar-pares-impares (cdr lista))))
                (else (añade-izq (car lista) (separar-pares-impares (cdr lista))))))
```

Apartado c: Define (separar-pares-impares-fos lista-num) utilizando funciones de orden superior, que haga lo mismo que la función anterior. Igual que antes puedes usar las funciones del apartado a), suponemos que están bien definidas.




```
(define (separar-pares-impares-fos lista-num)
        (foldr (lambda (num base)
                       (if (even? num)
                                (añade-izq num base)
                                (añade-der num base))) '(() . ()) lista-num))
```

Bibliografía

Los enunciados de los ejercicios resueltos, y los resúmenes, se han elaborado a partir del material publicado en https://domingogallardo.github.io/ , material del que es propietario el Departamento de Ciencia de la Computación e Inteligencia Artificial de la Universidad de Alicante, Domingo Gallardo, Cristina Pomares, Antonio Botía y Francisco Martínez.