Práctica 3: Funciones recursivas sobre listas

Entrega de la práctica

Para entregar la práctica debes subir a Moodle el fichero `practica03.rkt` con una cabecera inicial con tu nombre y apellidos, y las soluciones de cada ejercicio separadas por comentarios. Cada solución debe incluir:

- La **definición de las funciones** que resuelven el ejercicio.
- Un conjunto de **pruebas** que comprueben su funcionamiento utilizando el API `RackUnit`.

Ejercicio 1

a) Implementa la función recursiva `(contiene-prefijo prefijo lista-pal)` que recibe una cadena y una lista de palabras. Devuelve una lista con los booleanos resultantes de comprobar si la cadena es prefijo de cada una de las palabras de la lista.

Debes definir una función auxiliar `(es-prefijo? pal1 pal2)` que compruebe si la palabra 1 es prefijo de la palabra 2.

> **Pista**
>
> Puedes usar la función `(substring palabra inicio final)` que devuelve la subcadena de la `palabra` que va desde la posición `inicio` hasta la posición `final` (sin incluir).

Ejemplos:

```
(es-prefijo? "ante" "anterior") ; ⇒ #t
(contiene-prefijo "ante" '("anterior" "antígona" "antena"
```

```
3  "anatema"))
   ; ⇒ (#t #f #t #f)
```

b) Implementa la función recursiva `(inserta-pos dato pos lista)` que recibe un dato, una posición y una lista e inserta el dato en la posición indicada de la lista. Si la posición es 0, el dato se inserta en cabeza. Suponemos que la posición siempre será positiva y menor o igual que la longitud de la lista.

Ejemplos:

```
1  (inserta-pos 'b 2 '(a a a a)) ; ⇒ '(a a b a a)
2  (inserta-pos 'b 0 '(a a a a)) ; ⇒ '(b a a a a)
```

c) Implementa la función recursiva `(inserta-ordenada n lista-ordenada)` que recibe un número y una lista de números ordenados de menor a mayor y devuelve la lista resultante de insertar el número `n` en la posición correcta para que la lista siga estando ordenada.

Ejemplo:

```
1  (inserta-ordenada 10 '(-8 2 3 11 20)) ; ⇒ (-8 2 3 10 11 20)
```

Usando la función anterior `inserta-ordenada` implementa la función recursiva `(ordena lista)` que recibe una lista de números y devuelve una lista ordenada.

Ejemplo:

```
1  (ordena '(2 -1 100 4 -6)) ; ⇒ (-6 -1 2 4 100)
```

Ejercicio 2

a) Escribe la función `(expande-parejas pareja1 pareja2 ... pareja_n)` que recibe un número variable de argumentos y devuelve una lista donde se han "expandido" las parejas, creando una lista con tantos elementos como el número que indique cada pareja.

La función `expande-parejas` deberá llamar a una función recursiva `(expande-lista lista-parejas)` que trabaje sobre una lista de parejas.

Ejemplo:

```
(expande-parejas '(#t . 3) '("LPP" . 2) '(b . 4))
; ⇒ (#t #t #t "LPP" "LPP" b b b b)
```

Pista

Puedes definir una función auxiliar `(expande-pareja pareja)` que recibe una pareja y devuelve la lista expandida resultante de expandir sólo esa pareja.

b) Implementa la función recursiva `(expande2 lista)`. Recibe una lista en la que hay intercalados algunos números enteros positivos. Devuelve la lista original en la que se han expandido los elementos siguientes a los números, tantas veces como indica el número. La lista nunca va a contener dos números consecutivos y siempre va a haber un elemento después de un número.

Ejemplo:

```
(expande2 '(4 clase ua 3 lpp aulario))
; ⇒ (clase clase clase clase ua lpp lpp lpp aulario))
```

En el ejemplo, el 4 indica que el siguiente elemento (`clase`) se debe repetir 4 veces en la lista expandida y el 3 indica que el siguiente elemento (`lpp`) se va a repetir 3 veces.

Como en los anteriores ejercicios, te recomendamos implementar alguna función auxiliar.

Ejercicio 3

a) En matemáticas, el conjunto potencia de un conjunto es el conjunto formado por todos los subconjuntos del conjunto original. Vamos a representar este concepto con listas y diseñar una función recursiva que lo implemente.

Dada una lista de elementos, podemos representar su conjunto potencia como la lista de todas las sublistas posibles formadas con elementos de la lista original.

Ejemplo:

```
(conjunto-potencia '(1 2 3))
; ⇒ ((1 2 3) (1 2) (1 3) (1) (2 3) (2) (3) ())
```

Implementa la función recursiva `(conjunto-potencia lista)` que devuelva el conjunto potencia de la lista original.

Algunas pistas:

- El conjunto potencia de una lista vacía es una lista con la lista vacía como único elemento.
- El conjunto potencia de una lista se puede construir uniendo dos listas: la lista resultante de añadir el primer elemento a todas las listas del conjunto potencia sin el primer elemento y el conjunto potencia de la lista sin el primer elemento.

```
(conjunto-potencia '(1 2 3)) = ((1 2 3) (1 2) (1 3) (1)) +
                               ((2 3) (2) (3) ())
```

b) Implementa la función recursiva `(producto-cartesiano lista1 lista2)` que devuelva una lista con todas las parejas resultantes de combinar todos los elementos de la lista 1 con todos los elementos de la lista 2.

Ejemplo:

```
(producto-cartesiano '(1 2) '(1 2 3))
; ⇒ ((1 . 1) (1 . 2) (1 . 3) (2 . 1) (2 . 2) (2 . 3))
```

Ejercicio 4

a) Indica qué devuelven las siguientes expresiones en Scheme. Puede ser que alguna expresión contenga algún error. Indícalo también en ese caso y explica qué tipo de error. Hazlo sin el intérprete, y después comprueba con el intérprete si tu respuesta era correcta.

```
((lambda (x) (* x x)) 3) ; ⇒ ?
((lambda () (+ 6 4))) ; ⇒ ?
((lambda (x y) (* x (+ 2 y))) (+ 2 3) 4) ; ⇒ ?
((lambda (x y) (* x (+ 2 x))) 5) ; ⇒ ?

```

```
6
7  (define f (lambda (a b) (string-append "***" a b "***")))
8  (define g f)
9  (procedure? g) ; ⇒ ?
10 (g "Hola" "Adios") ; ⇒ ?
```

b) Hemos visto en teoría que la forma especial `define` para construir funciones es *azucar sintáctico* y que el intérprete de Scheme la convierte en una expresión equivalente usando la forma especial `lambda`.

Escribe cuál sería las expresiones equivalentes, usando la forma especial `lambda` a las siguientes definiciones de funciones:

```
1  (define (suma-3 x)
2     (+ x 3))
3
4  (define (factorial x)
5     (if (= x 0)
6        1
7        (* x (factorial (- x 1)))))
```

c) Suponiendo las siguientes definiciones de funciones indica qué devolverían las invocaciones. Puede ser que alguna expresión contenga algún error. Indícalo también en ese caso y explica qué tipo de error.

Hazlo sin el intérprete, y después comprueba con el intérprete si tu respuesta era correcta.

```
1  (define (doble x)
2     (* 2 x))
3
4  (define (foo f g x y)
5     (f (g x) y))
6
7  (define (bar f p x y)
8     (if (and (p x) (p y))
9         (f x y)
10        'error))
11
12 (foo + 10 doble 15) ; ⇒ ?
13 (foo doble + 10 15) ; ⇒ ?
14 (foo + doble 10 15) ; ⇒ ?
15 (foo string-append (lambda (x) (string-append "***" x)) "Hola"
16 "Adios") ; ⇒ ?
```

```
17
18  (bar doble number? 10 15) ; ⇒ ?
19  (bar string-append string? "Hola" "Adios") ; ⇒ ?
    (bar + number? "Hola" 5) ; ⇒ ?
```

Ejercicio 5

a) Implementa la función recursiva `(crea-baraja lista-parejas)` que recibe una lista de parejas con un palo y un valor (ambos caracteres) y devuelve una lista de cartas tal y como se representaban en la práctica anterior.

Por ejemplo:

```
1  (crea-baraja '((#\u2660 . #\A) (#\u2663 . #\2)
2                 (#\u2665 . #\3) (#\u2666 . #\R)))
3  ; ⇒ (A♠ 2♣ 3♥ R♦)
```

b) Vamos a trabajar con cartas de póker. Los valores posibles son A, 2, 3, ..., 8, 9, 0, J, Q, K (representamos el número 10 con el carácter y el símbolo 0). Y los palos son los símbolos correspondientes a los caracteres UTF #\u2660, #\u2663, #\u2665 y #\u2666: ♠, ♣, ♥ y ♦.

```
1  (define palos '(#\u2660 #\u2663 #\u2665 #\u2666))
2  (define valores '(#\A #\2 #\3 #\4 #\5 #\6 #\7 #\8 #\9 #\0 #\J
   #\Q #\K))
```

Implementa la función `(baraja-poker)` que devuelve una lista con todas las cartas de una baraja de póker. No es una función recursiva. Debes usar la función anterior `crea-baraja` y la función `producto-cartesiano` definida en el ejercicio 3.

```
1  (baraja-poker)
2  ; ⇒ (A♠ 2♠ 3♠ ...K♠ A♣ 2♣ ... K♣ A♥ 2♥ ... K♥ A♦ 2♦ ... Q♦ K♦)
```

No hace falta que hagas ninguna prueba de la función. Basta con que dejes en el código la llamada a la función y al ejecutar el programa verás por pantalla que la baraja es correcta.

c) Implementa la función recursiva `(mezcla lista)` que recibe una lista y la devuelve mezclada (sus elementos se han intercambiando en posiciones

aleatorias).

```
(mezcla '(1 2 3 4 5 6)) ; ⇒ (2 1 6 4 5 3)
```

Llama a la función con la baraja de póker para devuelva la baraja mezclada y la muestre por pantalla.

> **Pista**
>
> Puedes usar la función `inserta-pos` definida anteriormente y la función `(random inicial final)` que devuelve un número aleatorio entre el valor inicial (incluido) y el valor final (sin incluir).

Ejercicio 6

Implementa una función recursiva `(filtra-simbolos lista-simbolos lista-num)` que recibe una lista de símbolos y una lista de números enteros (ambas de la misma longitud) y devuelve una lista de parejas. Cada pareja está formada por el símbolo de la i-ésima posición de `lista-simbolos` y el número entero situado esa posición de `lista-num`, siempre y cuando dicho número se corresponda con la longitud de la cadena correspondiente al símbolo. Puedes utilizar las funciones predefinidas `string-length` y `symbol->string`.

Ejemplo:

```
(filtra-simbolos '(este es un ejercicio de examen) '(2 1 2 9 1
6))
; ⇒ ((un . 2) (ejercicio . 9) (examen . 6))
```
