Práctica 2: Programación funcional en Scheme

Entrega de la práctica

Para entregar la práctica debes subir a Moodle el fichero `practica02.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.
- Una visualización por pantalla de uno de los ejemplos incluidos en el enunciado que **demuestre qué hace la función**, usando la función de `display`.
- Un conjunto de **pruebas** que comprueben su funcionamiento utilizando la librería `schemeunit`. Estas pruebas deben incluir los ejemplos proporcionados en los ejercicios y un mínimo de **2 casos de prueba sustancialmente distintos** a estos ejemplos.

Ejercicio 1

En clase de teoría hemos visto que el símbolo `cond` es una **forma especial**. Vamos a crear una función `new-cond` que tome como argumentos 2 booleanos (`c1` y `c2`) y 3 valores a devolver (`x`, `y` y `z`):

```
(define (new-cond c1 x c2 y z)
 (cond
 (c1 x)
 (c2 y)
 (else z)))
```

En principio, parece que la función `new-cond` es equivalente a la forma especial `cond`. Por ejemplo, el resultado de evaluar las siguientes expresiones es el mismo:

```
(cond ((= 2 1) (+ 1 1)) ((< 3 2) (+ 2 3)) (else (- 10 3)))
(new-cond (= 2 1) (+ 1 1) (< 3 2) (+ 2 3) (- 10 3))
```

Sin embargo, no siempre pasa esto. Por ejemplo, en las siguientes expresiones:

```
(cond ((< 2 2) 1) ((> 3 2) 2) (else (/ 3 0)))
(new-cond (< 2 2) 1 (> 3 2) 2 (/ 3 0))
```

a) Explica detalladamente por qué `cond` y `new-cond` funcionan de forma distinta. Puedes utilizar la función `display` para mostrar por pantalla la explicación. La función puede mostrar una cadena que tenga varias líneas:

```
(display "Esto es una prueba
de como display
puede mostrar
varias líneas
por pantalla.")
```

b) Comprueba ahora el funcionamiento de las primitivas `and` y `or`. ¿Son formas especiales o

funciones? ¿Por qué? Pon ejemplos y explica tu respuesta detalladamente.

Ejercicio 2

Implementa la función `(minimo lista)` que reciba una lista numérica como argumento y devuelva el menor número de la lista. Suponemos listas de 1 o más elementos. No puedes utilizar la función `min` de Scheme, aunque puedes definirte y utilizar una función auxiliar `menor`.

La **formulación recursiva** del caso general podemos expresarla de la siguiente forma:

```
;;
;; Formulación recursiva de (minimo lista):
;;
;; El mínimo de los elementos de una lista es el menor entre
;; el primer elemento de la lista y el mínimo del resto de la lista
;;
```

Ejemplos:

```
(minimo '(9 8 6 4 3))  ; ⇒ 3
(minimo '(9 8 3 6 4))  ; ⇒ 3
```

Ejercicio 3

Implementa la función recursiva `(ordenada-decreciente? lista-nums)` que recibe como argumento una lista de números y devuelve `#t` si los números de la lista están ordenados de forma decreciente o `#f` en caso contrario. Suponemos listas de 1 o más elementos.

Escribe primero la formulación recursiva del caso general, y después realiza la implementación en Scheme.

Ejemplos:

```
(ordenada-decreciente? '(99 59 45 23 -1))  ; ⇒ #t
(ordenada-decreciente? '(12 50 -1 293 1000))  ; ⇒ #f
(ordenada-decreciente? '(3))  ; ⇒ #t
```

Ejercicio 4

Vamos a volver a trabajar con los intervalos de números, continuando con los ejercicios planteados en la primera práctica.

Para **elevar el nivel de abstracción** de nuestras funciones, vamos a **representar un intervalo como una pareja de números**. Así vamos a poder pasar intervalos como parámetros de funciones, devolverlos como resultados de alguna función o guardarlos en listas.

Por ejemplo, el intervalo que empieza en 3 y termina en 12 lo representaremos con la pareja `(3 . 12)`.

También vamos a poder utilizar el símbolo `'vacio` para representar un intervalo vacío.

a) Implementa el predicado `(engloban-intervalos? a b)` que a diferencia de la función `engloba?` definida en la práctica 1, recibe como parámetros parejas en lugar de números. Puedes utilizar dicha función `engloba?` copiando su definición en esta práctica.

Ejemplos:

```
(define i1 (cons 4 9))
(define i2 (cons 3 10))
(define i3 (cons 12 15))
(define i4 (cons 8 19))

(engloban-intervalos? (cons 5 9) (cons 4 8))  ; ⇒ #f
(engloban-intervalos? i1 i2)  ; ⇒ #t
(engloban-intervalos? i3 'vacio)  ; ⇒ #t
```

b) Implementa la función `(union-intervalos a b)` que debe devolver el intervalo (pareja) que englobe a los intervalos `a` y `b`.

Pista: Puedes utilizar las funciones `min` y `max`.

Ejemplos:

```
(union-intervalos (cons 4 10) (cons 3 8))  ; ⇒ {3 . 10}
(union-intervalos i2 i3)  ; ⇒ {3 . 15}
(union-intervalos 'vacio i4)  ; ⇒ {8 . 19}
```

c) Implementa la función `(interseccion-intervalos a b)` que debe devolver la intersección de los intervalos `a` y `b`. En caso de que no exista intersección, se deberá devolver el símbolo `'vacio`. Puedes usar el predicado `intersectan?` definido en la práctica 1 copiando su definición.

Ejemplos:

```
(interseccion-intervalos (cons 4 10) (cons 8 15))  ; ⇒ {8 . 10}
(interseccion-intervalos i1 i3))  ; ⇒ vacio
(interseccion-intervalos 'vacio i4))  ; ⇒ vacio
```

Ejercicio 5

Implementa utilizando la recursión las funciones `(union-lista-intervalos lista-intervalos)` e `(interseccion-lista-intervalos lista-intervalos)` que devuelven el intervalo (pareja) resultante de la suma o intersección de una lista de intervalos. Debes utilizar las funciones definidas en el ejercicio anterior.

Escribe primero la formulación recursiva del caso general y realiza después la implementación en Scheme.

Ejemplo:

```
(union-lista-intervalos (list (cons 2 12) (cons -1 10) (cons 8 20))) ; ⇒ {-1 . 20}
(interseccion-lista-intervalos (list (cons 12 30) (cons -8 20) (cons 13 35))) ; ⇒ {13
. 20}
(interseccion-lista-intervalos (list (cons 25 30) (cons -8 20) (cons 13 35))) ; ⇒
```

```
vacio
```
