Práctica 2: Recursión, parejas y diagramas box-and-pointer

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.
- Un conjunto de **pruebas** que comprueben su funcionamiento utilizando el API `RackUnit`.

Ejercicios

Ejercicio 1

a.1) Implementa la función recursiva `(minimo lista)` que recibe una lista con números como argumento y devuelve el menor número de la lista. Suponemos listas de 1 o más elementos.

Para la implementación debes usar la función `menor` definida en la práctica anterior.

> **Pista**
>
> Podemos expresar el caso general de la recursión de la siguiente forma:
>
> > 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.
>
> Y el caso base:
>
> > El mínimo de una lista con un único número es ese número.

Ejemplos:

```
(minimo '(1 8 6 4 3)) ; ⇒ 1
(minimo '(1 -1 3 -6 4)) ; ⇒ -6
```

a.2) Vamos a investigar el funcionamiento de la recursión en la función `minimo`. Supongamos la siguiente llamada:

```
(minimo '(1 8 6 4 3)) ; ⇒ 1
```

- ¿Qué lista se pasa como parámetro a la primera llamada recursiva a la función?
- ¿Qué devuelve esa llamada recursiva?
- ¿Con qué argumentos se llama a la función `menor` que devuelve el resultado final?

b) Implementa la función recursiva `(concatena lista-chars)` que recibe una lista de caracteres y devuelve la cadena resultante de concatenarlos.

Ejemplos:

```
(concatena '(#\H #\o #\l #\a)) ; ⇒ "Hola"
(concatena '(#\S #\c #\h #\e #\m #\e #\space #\m #\o #\l #\a))
; ⇒ "Scheme mola"
```

c) Implementa la función `(contiene? cadena char)` que comprueba si una cadena contiene un carácter determinado. Debes usar la función `string->list` e implementar la función auxiliar recursiva `(contiene-lista? lista dato)`.

Ejemplos:

```
(contiene? "Hola" #\o) ; ⇒ #t
(contiene? "Esto es una frase" #\space) ; ⇒ #t
(contiene? "Hola" #\h) ; ⇒ #f
```

Ejercicio 2

a) Implementa la función recursiva `(binario-a-decimal lista-bits)` que reciba una lista de bits que representan un número en binario (el primer elemento será el bit más significativo) y devuelva el número decimal equivalente.

> **Pista**
>
> Puedes utilizar la función `length`.

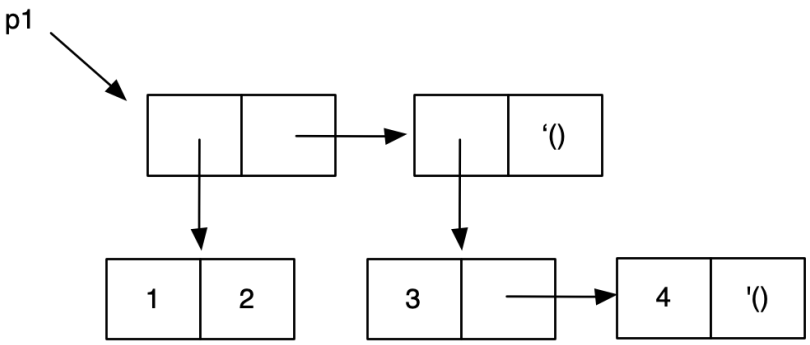
```
(binario-a-decimal '(1 1 1 1)) ; ⇒ 15
(binario-a-decimal '(1 1 0)) ; ⇒ 6
(binario-a-decimal '(1 0)) ; ⇒ 2
```

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

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

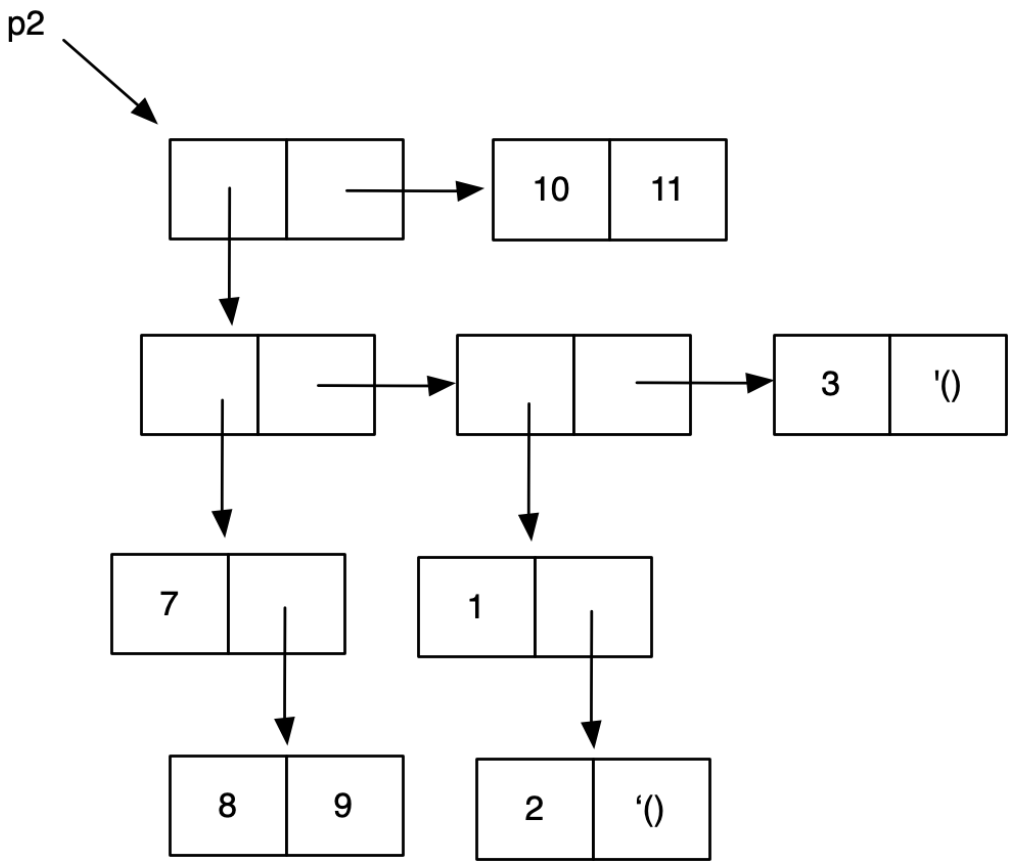
Ejercicio 3

a.1) Dado el siguiente *box & pointer*, escribe la expresión en Scheme que define `p1` usando el mínimo número de llamadas a `list` y `cons`. No debes utilizar expresiones con `quote`.



a.2) Escribe las expresiones que devuelven 2 y 4 a partir de `p1`.

b.1) Dado el siguiente diagrama caja y puntero, escribe la expresión en Scheme que define `p2` usando el mínimo número de llamadas a `list` y `cons`.



b.2) Escribe las expresiones que devuelven 9 y 2 a partir de `p2`.

Ejercicio 4

Vamos a programar una versión simplificada del *blackjack* o 21. Dos jugadores juegan un número de cartas y suman todos sus valores. Gana el que se acerque más al 21 sin pasarse.

Representaremos las cartas como en la práctica 1, y podemos usar las funciones allí definidas para obtener su valor. Suponemos que el valor es el propio de la carta (no seguiremos la regla del juego original en el que las figuras valen 10).

Cada jugador tendrá una lista de cartas y deberá indicar un número `n` que representa el número de cartas de esa lista con las que se queda.

Tendremos que implementar la función `(blackjack cartas1 n1 cartas2 n2)` que recibe la lista de cartas del jugador 1, el número de cartas que se queda el jugador 1, la lista de cartas del jugador 2 y el número de cartas del jugador 2.

Por ejemplo, supongamos las siguientes cartas del jugador 1 y del jugador 2:

```
(define cartas1 '(3O 5E AC 2B 5O 5C 4B))
(define cartas2 '(CE AO 3B AC 2E SC 4C))
```

Supongamos que el jugador 1 se queda con 5 cartas de su lista y el jugador 2 con 3. Las primeras 5 cartas del jugador 1 suman 16 y las 3 primeras cartas del jugador 2 suman 15. Ganaría el jugador 1.

```
(blackjack cartas1 5 cartas2 3) ; ⇒ 1
```

La función `blackjack` devolverá 1 si gana el jugador 1, 2 si gana el jugador 2, 0 si empatan y -1 si los dos jugadores se pasan de 21.

Debes implementar la función `blackjack` y una función auxiliar recursiva que sea la que sume los valores de las `n` primeras cartas de una lista. Si `n` es mayor que el número de cartas, devolverá la suma de todas las cartas de la lista.

Puedes definir cualquier otra función auxiliar que necesites.

Otros ejemplos:

```
(blackjack cartas1 5 cartas2 4) ; ⇒ 0
(blackjack cartas1 5 cartas2 3) ; ⇒ 1
(blackjack cartas1 3 cartas2 4) ; ⇒ 2
(blackjack cartas1 7 cartas2 6) ; ⇒ -1
```

Ejercicio 5

a) Implementa las funciones `(suma-izq pareja n)` y `(suma-der pareja n)` definidas de la siguiente forma:

- `(suma-izq pareja n)` : devuelve una nueva pareja con la parte izquierda incrementada en `n` .

- `(suma-der pareja n)` : devuelve una nueva pareja con la parte derecha incrementada en `n` .

Ejemplos:

```
(suma-izq (cons 10 20) 3)  ; ⇒ (13 . 20)
(suma-der (cons 10 20) 5)  ; ⇒ (10 . 25)
```

b) Implementa la función recursiva `(suma-impares-pares lista-num)` que devuelva una pareja cuya parte izquierda sea la suma de los números impares de la lista y la parte derecha la suma de los números pares. Debes utilizar las funciones auxiliares definidas en el apartado anterior. También puedes utilizar las funciones predefinidas `even?` y `odd?` .

Ejemplos:

```
(suma-impares-pares '(3 2 1 4 8 7 6 5)) ; ⇒ (16 . 20)
(suma-impares-pares '(3 1 5))           ; ⇒ (9 . 0)
```

Ejercicio 6

Implementa la función recursiva `(cadena-mayor lista)` que recibe un lista de cadenas y devuelve una pareja con la cadena de mayor longitud y dicha longitud. En el caso de que haya más de una cadena con la máxima longitud, se devolverá la última de ellas que aparezca en la lista.

En el caso en que la lista sea vacía se devolverá la pareja con la cadena vacía y un 0 (la longitud de la cadena vacía).

Pista: puedes utilizar la función `string-length`

```
(cadena-mayor '("vamos" "a" "obtener" "la" "cadena" "mayor")) ;
⇒  ("obtener" . 7)
(cadena-mayor '("prueba" "con" "maximo" "igual")) ; ⇒
("maximo" . 6)
(cadena-mayor '()) ; ⇒ ("" . 0)
```

Lenguajes y Paradigmas de Programación, curso 2019-20

Domingo Gallardo, Cristina Pomares, Antonio Botía, Francisco Martínez