TC2006 Lenguajes de Programación

Tarea 3: Programación Básica en Racket

(* Equipos de 3 integrantes *)

En esta tarea comenzarán a practicar con el lenguaje Racket. Deberán subir como asignación de Canvas el archivo en Racket con nombre **M_tarea3.rkt** (donde sustituyan M por sus matrículas separadas con guiones bajos) que contenga el código de las funciones implementadas para cada problema.
Internamente, mediante comentarios de Racket deberán incluir sus matrículas y nombres en el archivo, así como la descripción de lo que calcula cada función y el significado de cada parámetro formal de las mismas.
La respuesta al problema 4 lo deben agregar en el archivo **M_tarea3.pdf**.

1. Implementar la función recursiva **primo?** que determine si el valor del argumento dado como un número natural es un número primo. Un número primo es un número natural que solo tiene dos factores que son el número mismo y el uno. Los números 0 y 1 no se consideran primos.
   Probar con:
   ```
   >(primo? 1)          => #f
   >(primo? 4)          => #f
   >(primo? 17)         => #t
   ```

2. Implementar la función recursiva **sumdpar** que regrese la suma de los dígitos pares que conforman a un número entero no negativo que se le pasa como argumento.
   Probar con:
   ```
   >(sumdpar 0)         => 0
   >(sumdpar 8032)      => 10
   >(sumdpar 174)       => 4
   ```

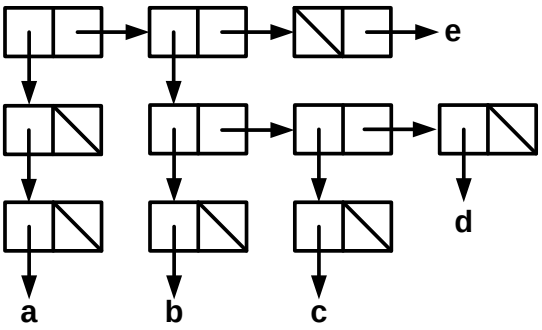
3. El número de combinaciones describe el número de formas en las que puedes elegir sin repetición, y sin importar el orden de elección, r cosas de un conjunto de n cosas. La notación para combinaciones es C(n,r).
   La cantidad de permutaciones posibles se calcula como: C(n,r) = n!/(r! * (n-r)!).
   Implementar la función recursiva **combinaciones** que calcule el número de combinaciones si se le pasan como argumentos "n" y "r".
   Probar con:
   ```
   >(combinaciones 9 3)     => 84
   >(combinaciones 4 4)     => 1
   >(combinaciones 8 1)     => 8
   ```

4. Para cada uno de los siguientes incisos obtén las dos representaciones faltantes de las tres vistas en clase: representación visual, representación interna, y representación con el constructor **cons** (no se permiten otros constructores como list o append).

   a. (1 2 ((3 4) (5)))

   b. (cons (cons (cons 'a 'b)(cons 'c (cons 'd '()))) '())

c.



5. Implementar la función recursiva bitor que calcule el or lógico entre bits dados como elementos de 2 listas dadas como sus argumentos. Asumir que las dos listas son del mismo tamaño.
   Probar con:

```
> (bitor '() '())                  => ()
> (bitor '(1 0 1) '(0 1 1))        => (1 1 1)
> (bitor '(1 0 1 0) '(0 0 1 0))    => (1 0 1 0)
```

6. Implementar la función recursiva hexadecimal que obtenga una lista de números y letras que represente la codificación hexadecimal del argumento entero que representa un número en binario (solo unos y ceros).
   Probar con:

```
> (hexadecimal 0)                  => (0)
> (hexadecimal 1011)               => (b)
> (hexadecimal 111111)             => (3 f)
```

7. Implementar la función recursiva decimal que convierta su argumento dado como un número binario representado por una lista de 1's y 0's en un valor en decimal.
   Probar con:

```
> (decimal '(1 0 1))               => 5
> (decimal '(1 0 1 0))             => 10
> (decimal '(1 1 1 1 1))           => 31
```

8. Implementar el predicado recursivo **expresion?** que determine si una expresión aritmética (sumas, restas, multiplicaciones y divisiones) en notación prefija dada como una lista imbricada (como Racket) se especificó correctamente. Asumir que las listas solo contendrán operadores y números.
   Probar con:

```
> (expresion? '(+ 1 2 3))                          => #t
> (expresion? '(2 + 3))                            => #f
> (expresion? '(/ (* (+ 4 5) 4) (/ (* 4 6) 2))     => #t
```

9. Implementar la función recursiva **palindromo** que genere el patrón palíndromo intercalando los símbolos a y b anidados N veces en sublistas.
   Probar con:

```
> (palindromo 1)        => (a)
> (palindromo 3)        => (a (b (a) b) a)
> (palindromo 6)        => (b (a (b (a (b (a) b) a) b) a) b)
```

10. Implementar la función recursiva **inversiontotal** que invierta una lista imbricada junto con todas sus listas anidadas.
    Probar con:

```
> (inversiontotal '(a b c d))           => (d c b a)
> (inversiontotal '(a (b c) d))         => (d (c b) a)
> (inversiontotal '(1 (2 (3 (4 5)) 6))) => ((6 ((5 4) 3) 2) 1)
```