Problemario: Programación funcional, parte 2

Instrucciones

Utilizando el lenguaje de programación funcional indicado por tu profesor (Scheme, Racket, Clojure), resuelve los problemas que se presentan a continuación. Coloca tu código en un solo archivo. Cada función debe incluir un comentario con una breve descripción de lo que hace.

1. La función `insert` toma dos entradas: un número `n` y una lista `lst` que contiene números en orden ascendente. Devuelve una nueva lista con los mismos elementos de `lst` pero con `n` insertado en su lugar correspondiente.

   Ejemplos:

   ```
   (insert 14 '())
   ⇒ (14)

   (insert 4 '(5 6 7 8))
   ⇒ '(4 5 6 7 8)

   (insert 5 '(1 3 6 7 9 16))
   ⇒ (1 3 5 6 7 9 16)

   (insert 10 '(1 5 6))
   ⇒ (1 5 6 10)
   ```

2. La función `insertion-sort` toma una lista desordenada de números como entrada y devuelve una nueva lista con los mismos elementos pero en orden ascendente. Se debe usar la función de `insert` definida en el ejercicio anterior para escribir `insertion-sort`. No se debe utilizar la función `sort` o alguna similar predefinida.

   Ejemplos:

   ```
   (insertion-sort '())
   ⇒ ()

   (insertion-sort '(4 3 6 8 3 0 9 1 7))
   ⇒ (0 1 3 3 4 6 7 8 9)

   (insertion-sort '(1 2 3 4 5 6))
   ⇒ (1 2 3 4 5 6)

   (insertion-sort '(5 5 5 1 5 5 5))
   ⇒ (1 5 5 5 5 5 5)
   ```

3. La función `rotate-left` toma dos entradas: un número entero `n` y una lista `lst`. Devuelve la lista que resulta de rotar `lst` un total de `n` elementos a la izquierda. Si `n` es negativo, rota hacia la derecha.

   Ejemplos:

```
(rotate-left 5 '())
⇒ ()

(rotate-left 0 '(a b c d e f g))
⇒ (a b c d e f g)

(rotate-left 1 '(a b c d e f g))
⇒ (b c d e f g a)

(rotate-left -1 '(a b c d e f g))
⇒ (g a b c d e f)

(rotate-left 3 '(a b c d e f g))
⇒ (d e f g a b c)

(rotate-left -3 '(a b c d e f g))
⇒ (e f g a b c d)

(rotate-left 8 '(a b c d e f g))
⇒ (b c d e f g a)

(rotate-left -8 '(a b c d e f g))
⇒ (g a b c d e f)

(rotate-left 45 '(a b c d e f g))
⇒ (d e f g a b c)

(rotate-left -45 '(a b c d e f g))
⇒ (e f g a b c d)
```

4. La función `prime-factors` toma un número entero `n` como entrada (`n` ¿ 0) y devuelve una lista que contiene los factores primos de `n` en orden ascendente. Los factores primos son los números primos que dividen a un número de manera exacta. Si se multiplican todos los factores primos se obtiene el número original.

   Ejemplos:

```
(prime-factors 1)
⇒ ()

(prime-factors 6)
⇒ (2 3)

(prime-factors 96)
⇒ (2 2 2 2 2 3)

(prime-factors 97)
⇒ (97)

(prime-factors 666)
⇒ (2 3 3 37)
```

5. La función `gcd` toma dos números enteros positivos `a` y `b` como entrada, donde `a` > 0 y `b` > 0. Devuelve el *máximo común divisor* (GCD por sus siglas en inglés) de `a` y `b`. No se debe usar la función `gcd` o similar predefinida.

   NOTA: El GCD de dos enteros es el mayor entero positivo que divide ambos números de manera exacta. Por ejemplo, el GCD de 20 y 16 es 4.

   Ejemplos:

   ```
   (gcd 13 7919)
   ⇒ 1

   (gcd 20 16)
   ⇒ 4

   ⇒ (gcd 54 24)
   6

   (gcd 6307 1995)
   ⇒ 7

   (gcd 48 180)
   ⇒ 12

   (gcd 42 56)
   ⇒ 14
   ```

6. La función `deep-reverse` toma una lista como entrada. Devuelve una lista con los mismos elementos que su entrada pero en orden inverso. Si hay listas anidadas, estas también deben invertirse.

   Ejemplos:

   ```
   (deep-reverse '())
   ⇒ ()

   (deep-reverse '(a (b c d) 3))
   ⇒ (3 (d c b) a)

   (deep-reverse '((1 2) 3 (4 (5 6))))
   ⇒ (((6 5) 4) 3 (2 1))

   (deep-reverse '(a (b (c (d (e (f (g (h i j)))))))))
   ⇒ (((((((((j i h) g) f) e) d) c) b) a)
   ```

7. La función `insert-anywhere` toma dos entradas: un objeto `x` y una lista `lst`. Devuelve una nueva lista con todas las formas posibles en que se puede insertar `x` en cada posición de `lst`.

   Ejemplos:

```
(insert-everywhere 1 '())
⇒ ((1))

(insert-everywhere 1 '(a))
⇒ ((1 a) (a 1))

(insert-everywhere 1 '(a b c))
⇒ ((1 a b c) (a 1 b c) (a b 1 c) (a b c 1))

(insert-everywhere 1 '(a b c d e))
⇒ ((1 a b c d e)
   (a 1 b c d e)
   (a b 1 c d e)
   (a b c 1 d e)
   (a b c d 1 e)
   (a b c d e 1))

(insert-everywhere 'x '(1 2 3 4 5 6 7 8 9 10))
⇒ ((x 1 2 3 4 5 6 7 8 9 10)
   (1 x 2 3 4 5 6 7 8 9 10)
   (1 2 x 3 4 5 6 7 8 9 10)
   (1 2 3 x 4 5 6 7 8 9 10)
   (1 2 3 4 x 5 6 7 8 9 10)
   (1 2 3 4 5 x 6 7 8 9 10)
   (1 2 3 4 5 6 x 7 8 9 10)
   (1 2 3 4 5 6 7 x 8 9 10)
   (1 2 3 4 5 6 7 8 x 9 10)
   (1 2 3 4 5 6 7 8 9 x 10)
   (1 2 3 4 5 6 7 8 9 10 x))
```

8. La función `pack` toma una lista `lst` como entrada. Devuelve una lista de listas que agrupan los elementos iguales consecutivos.

   Ejemplos:

```
(pack '())
⇒ ()

(pack '(a a a a b c c a a d e e e e))
⇒ ((a a a a) (b) (c c) (a a) (d) (e e e e))

(pack '(1 2 3 4 5))
⇒ ((1) (2) (3) (4) (5))

(pack '(9 9 9 9 9 9 9 9 9))
⇒ ((9 9 9 9 9 9 9 9 9))
```

9. La función `compress` toma una lista `lst` como entrada. Devuelve una lista en la que los elementos repetidos consecutivos de `lst` se reemplazan por una sola instancia. El orden de los elementos no debe modificarse.

   Ejemplos:

```
(compress '())
⇒ ()

(compress '(a b c d))
⇒ '(a b c d)

(compress '(a a a a b c c a a d e e e e))
⇒ (a b c a d e)

(compress '(a a a a a a a a a a))
⇒ (a)
```

10. La función `encode` toma una lista `lst` como entrada. Los elementos consecutivos en `lst` se codifican en listas de la forma: (`n e`), donde `n` es el número de ocurrencias del elemento `e`.

    Ejemplos:

```
(encode '())
⇒ ()

(encode '(a a a a b c c a a d e e e e))
⇒ ((4 a) (1 b) (2 c) (2 a) (1 d) (4 e))

(encode '(1 2 3 4 5))
⇒ ((1 1) (1 2) (1 3) (1 4) (1 5))

(encode '(9 9 9 9 9 9 9 9 9 9))
⇒ ((9 9))
```

11. La función `encode-modified` toma una lista `lst` como entrada. Funciona igual que el problema anterior, pero si un elemento no tiene duplicados simplemente se copia en la lista resultante. Solo los elementos que tienen repeticiones consecutivas se convierten en listas de la forma: (`n e`).

    Ejemplos:

```
(encode-modified '())
⇒ ()

(encode-modified '(a a a a b c c a a d e e e e))
⇒ ((4 a) b (2 c) (2 a) d (4 e))

(encode-modified '(1 2 3 4 5))
⇒ (1 2 3 4 5)

(encode-modified '(9 9 9 9 9 9 9 9 9 9))
⇒ ((9 9))
```

12. La función de `decode` toma como entrada una lista codificada `lst` que tiene la misma estructura que la lista resultante del problema anterior. Devuelve la versión decodificada de `lst`.

    Ejemplos:

    ```
    (decode '())
    ⇒ ()

    (decode '((4 a) b (2 c) (2 a) d (4 e)))
    ⇒ (a a a a b c c a a d e e e e)

    (decode '(1 2 3 4 5))
    ⇒ (1 2 3 4 5)

    (decode '((9 9)))
    ⇒ (9 9 9 9 9 9 9 9 9)
    ```

13. La función `args-swap` toma como entrada una función de dos argumentos `f` y devuelve una nueva función que se comporta como `f` pero con el orden de sus dos argumentos intercambiados. En otras palabras:

    $$((\texttt{args-swap}\ f)\ x\ y) \equiv (f\ y\ x)$$

    Ejemplos:

    ```
    ((args-swap list) 1 2)
    ⇒ (2 1)

    ((args-swap /) 8 2)
    ⇒ 1/4

    ((args-swap cons) '(1 2 3) '(4 5 6))
    ⇒ ((4 5 6) 1 2 3)

    ((args-swap map) '(-1 1 2 5 10) /)
    ⇒ (-1 1 1/2 1/5 1/10)
    ```

14. La función `there-exists-one?` toma dos entradas: una función booleana de un argumento `pred` y una lista `lst`. Devuelve *verdadero* si hay exactamente un elemento en `lst` que satisface `pred`, en otro caso devuelve *falso*.

    Ejemplos:

    ```
    (there-exists-one? positive? '())
    ⇒ #f

    (there-exists-one? positive? '(-1 -10 4 -5 -2 -1))
    ⇒ #t

    (there-exists-one? negative? '(-1))
    ⇒ #t

    (there-exists-one? symbol? '(4 8 15 16 23 42))
    ⇒ #f

    (there-exists-one? symbol? '(4 8 15 sixteen 23 42))
    ⇒ #t
    ```

15. La función `linear-search` toma tres entradas: una lista `lst`, un valor `x`, y una función de igualdad `eq-fun`. Busca secuencialmente `x` en `lst` usando `eq-fun` para comparar `x` con los elementos contenidos en `lst`. La función `eq-fun` debe aceptar dos argumentos, `a` y `b`, y devolver *verdadero* si se debe considerar que `a` es igual a `b`, o *falso* en caso contrario.

    La función `linear-search` devuelve el índice donde se encuentra la primera ocurrencia de `x` en `lst` (el primer elemento de la lista se encuentra en el índice 0), o *falso* si no se encontró.

    Ejemplos:

    ```
    (linear-search '() 5 =)
    ⇒ #f

    (linear-search '(48 77 30 31 5 20 91 92 69 97 28 32 17 18 96) 5 =)
    ⇒ 4

    (linear-search '("red" "blue" "green" "black" "white") "black" string=?)
    ⇒ 3

    (linear-search '(a b c d e f g h) 'h equal?)
    ⇒ 7
    ```

16. La derivada de una función $f(x)$ con respecto a la variable x se define como:

$$f'(x) \equiv \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

    Donde f debe ser una función continua. Escribe la función `deriv` que toma `f` y `h` como entradas, y devuelve una nueva función que toma `x` como argumento, y que representa la derivada de `f` dado un cierto valor de `h`.

    Los ejemplos verifican las siguientes derivadas:

$$\begin{aligned} f(x) &= x^3 \\ f'(x) &= 3x^2 \\ f''(x) &= 6x \\ f'''(x) &= 6 \\ f'(5) &= 3 \cdot 5^2 = 75 \\ f''(5) &= 6 \cdot 5 = 30 \\ f'''(5) &= 6 \end{aligned}$$

    Ejemplos:

    ```
    (define f (lambda (x) (* x x x)))
    (define df (deriv f 0.001))
    (define ddf (deriv df 0.001))
    (define dddf (deriv ddf 0.001))

    (df 5)
    ⇒ 75.01500100002545

    (ddf 5)
    ⇒ 30.006000002913424

    (dddf 5)
    ⇒ 5.999993391014868
    ```

17. El *método de Newton* es un algoritmo para encontrar la raíz de una función a partir del cálculo de aproximaciones sucesivamente mejores. Se puede resumir de la siguiente manera:

$$x_n = \begin{cases} 0 & \text{Si } n = 0 \\ x_{n-1} - \frac{f(x_{n-1})}{f'(x_{n-1})} & \text{Si } n > 0 \end{cases}$$

Algunas cosas que es necesario señalar:

- f debe ser una función real diferenciable.
- Entre mayor sea el valor de n, mejor es la aproximación.
- x_0 es una aproximación inicial, y es recomendable que sea un valor cercano a la solución. Esto permite calcular el resultado de manera más rápida. Sin embargo, por simplicidad, siempre suponemos aquí que $x_0 = 0$.

Escribe la función `newton` que recibe `f` y `n` como entradas, y devuelve el valor correspondiente de x_n. Usa la función `deriv` del problema anterior para calcular f', con $h = 0.0001$.

Ejemplos:

```
(newton (lambda (x) (- x 10)) 1)
⇒ 10.000000000023306

(newton (lambda (x) (+ (* 4 x) 2)) 1)
⇒ -0.5000000000000551

(newton (lambda (x) (+ (* x x x) 1)) 50)
⇒ -0.9999999980685114

(newton (lambda (x) (+ (cos x) (* 0.5 x))) 5)
⇒ -1.029866529322135
```

18. La *regla de Simpson* es un método de integración numérica. Su fórmula es:

$$\int_a^b f = \frac{h}{3}(y_0 + 4y_1 + 2y_2 + 4y_3 + 2y_4 + \cdots + 2y_{n-2} + 4y_{n-1} + y_n)$$

Donde n es un entero positivo par (si se incrementa el valor de n se obtiene una mejor aproximación), mientras que h y y_k se definen de la siguiente manera:

$$h = \frac{b-a}{n}$$

$$y_k = f(a + kh)$$

Escribe la función `integral`, que recibe como entradas lo siguiente: `a`, `b`, `n` y `f`. Devuelve el valor de la integral definida, usando la regla de Simpson.

Los ejemplos resuelven las siguientes integrales definidas con `n` = 10:

$$\int_0^1 x^3\,dx = \frac{1}{4}$$

$$\int_1^2 \int_3^4 xy \cdot dx \cdot dy = \frac{21}{4}$$

Ejemplos:

```
(integral 0 1 10 (lambda (x) (* x x x)))
⇒ 1/4

(integral 1 2 10
  (lambda (x)
    (integral 3 4 10
      (lambda (y)
        (* x y)))))
⇒ 21/4
```