Nombre y Apellido: ____________________________ **Comisión:** ____________

PARADIGMAS DE PROGRAMACIÓN
3er Parcial: Programación Funcional

Fecha: 09/12/2023
INSTRUCCIONES:
- ✓ **Las soluciones de los Ejercicios 1, 2a y 2b deben entregarse en esta hoja.**
- ✓ **Resolver los Ejercicios 2c, 3 y 4 cada uno en una hoja separada.**
- ✓ **Poner nombre en TODAS las hojas que entregue.**
- ✓ **Numerar TODAS las hojas indicando en cada una el total (nro. hoja/total).**

Ejercicio 1 (25 puntos)

a) En Cálculo Lambda, en el término: (λz.λf.(P f) x)

	z y x son argumentos nominales.
	No puede aplicarse la Regla Beta hasta conocer la forma normal de P.
	Es equivalente a [z/x] λf.(P f)
	P no puede incluir ninguna variable con nombre f.
V	Ninguna de las anteriores es correcta.

b) En el Cálculo Lambda:

	Un término está en forma normal si tiene todas las variables ligadas.
	Siempre que un término tiene exactamente 1 redex se puede asegurar que luego de reducirlo se obtiene su forma normal.
	La estrategia de reducción de orden aplicativo siempre encuentra la forma normal, pero la cantidad de reducciones necesarias es mayor que en el orden normal.
V	Siendo P y Q términos equivalentes, si Q no tiene forma normal entonces P tampoco tiene forma normal.
	Ninguna de las afirmaciones anteriores es correcta.

c) Dada la siguiente abstracción funcional: λz.λy.λx.((z y) x)

	No puede ser evaluada con x como argumento ya que x ocurre en la abstracción funcional.
V	Luego de ser evaluada, el resultado es una abstracción funcional.
	No puede ser evaluada con el argumento z por ambigüedad en la aplicación de la regla Beta.
	Luego de ser evaluada con t como argumento, el resultado es un término sin variables libres.
	Ninguna de las afirmaciones anteriores es correcta.

d) En Scheme:

F	A diferencia de Cálculo Lambda, toda expresión tiene forma normal.
V	En una evaluación el argumento efectivo puede ser una abstracción funcional.
F	Al igual que en Cálculo Lambda, toda aplicación funcional tiene un único argumento.
F	Dada una aplicación funcional, si la misma incluye funciones de orden superior se empleará la estrategia de evaluación de orden normal.
F	Ninguna de las afirmaciones anteriores es correcta.

e) En Scheme, la siguiente expresión:

```
(((lambda (x) (lambda (y) ((car y) x (x y)))) cdr) (list cons))
```

	Retorna una lista de longitud 2.
V	Retorna una lista de longitud 1.
	Retorna la lista vacía.
	Retorna una función.
	Produce un error.
	Ninguna de las anteriores es correcta.

Ejercicio 2 (15 puntos)

Considere el siguiente término del Cálculo Lambda:

((λt.λw.(t (t w)) λx.w) (λx.x p))

a) Marque en la expresión anterior, cuántos y cuáles (si hubiera) son los redex que contiene.

b) Para cada variable, indicar el número de ocurrencias libres y ligadas de la misma, completando la siguiente tabla:

Variable	Ocurrencias Libres	Ocurrencias Ligadas

c) Hallar la forma normal del término indicando claramente la estrategia (orden normal u orden aplicativo) y las reglas utilizadas en cada paso.

Ejercicio 3 (25 puntos)

Se dispone de puntos en el plano 2D que son representados en Scheme mediante una lista de 2 elementos: **(x y)**, correspondientes a las coordenadas del punto.

Se le solicita definir las funciones:

a) **(distancia P_1 P_2)** que dados 2 puntos P_1 y P_2 devuelve la distancia euclidiana entre los mismos. Recuerde que la fórmula para calcular esta distancia es:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

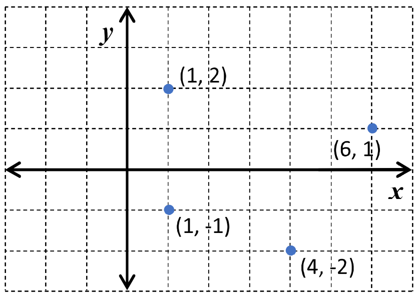
b) **(mas-cercano P)** que al ser evaluada con un punto **P** como argumento, devuelve una función de 2 argumentos $\mathbf{P_1}$ y $\mathbf{P_2}$, que también son puntos. Esta función, al ser evaluada con 2 puntos como argumentos, retorna #t si $\mathbf{P_1}$ está a menor distancia de **P** que $\mathbf{P_2}$, y #f en caso contrario.

c) **(ordenados? P lista-puntos)** que dado un punto y una lista de puntos, retorna #t si los puntos de la lista ya están ordenados desde el más cercano al más lejano al punto **P**, y #f en caso contrario. Puede asumir que la lista de puntos tendrá al menos 1 elemento. Para definir esta función **<u>debe utilizar</u>** la función solicitada en el punto **b)**.

<u>Ejemplos</u>:

```
> (distancia '(1 2) '(1 -1))
3
> (distancia '(1 2) '(4 -2))
5
> (distancia '(1 2) '(6 1))
5.0990195135927845
> (mas-cercano '(1 2))
#<procedure>
> ((mas-cercano '(1 2)) '(1 -1) '(6 1))
#t
> ((mas-cercano '(1 2)) '(4 -2) '(1 -1))
#f
> (ordenados? '(1 2) '((1 -1) (4 -2) (6 1)))
#t
> (ordenados? '(1 2) '((6 1) (4 -2)))
#f
```

<u>Representación gráfica</u>:



Ejercicio 4 (35 puntos)

El árbol de Calkin–Wilf es el árbol definido por las siguientes reglas:

- El nodo raíz es el par (1 . 1)
- Los hijos del nodo (x . y) son (x . x + y) y (x + y . y)

Por ejemplo, los 4 primeros niveles del árbol de Calkin-Wilf son:

```
                              (1.1)
                                |
                   +------------+------------+
                   |                         |
                 (1.2)                     (2.1)
                   |                         |
            +------+------+           +------+------+
            |             |           |             |
          (1.3)         (3.2)       (2.3)         (3.1)
            |             |           |             |
        +---+---+     +---+---+   +---+---+     +---+---+
        |       |     |       |   |       |     |       |
      (1.4)   (4.3) (3.5)   (5.2) (2.5)  (5.3) (3.4)   (4.1)
```

a) Definir la función **(sucesores par)** que retorna la lista de los hijos del **par = (x . y)** en el árbol de Calkin–Wilf. Por ejemplo:

```
> (sucesores (cons 3 2))
((3 . 5) (5 . 2))
> (sucesores (cons 3 1))
((3 . 4) (4 . 1))
> (sucesores (cons 2 1))
((2 . 3) (3 . 1))
```

b) Definir la función **(siguientes xs)** que retorna la lista formada por los hijos de los elementos de la lista **xs** en el árbol de Calkin–Wilf. Por ejemplo:

```
> (siguientes (list (cons 3 2) (cons 1 1)))
((3 . 5) (5 . 2) (1 . 2) (2 . 1))
> (siguientes (list (cons 3 2) (cons 1 1) (cons 3 1)))
((3 . 5) (5 . 2) (1 . 2) (2 . 1) (3 . 4) (4 . 1))
```

c) Definir la función **(niveles-calkin-wilf N)** que retorna una lista de listas que representan los diferentes niveles del árbol de Calkin-Wilf. Por ejemplo:

```
> (niveles-calkin-wilf 1)
(((1 . 1)))
> (niveles-calkin-wilf 2)
(((1 . 1)) ((1 . 2) (2 . 1)))
> (niveles-calkin-wilf 3)
(((1 . 1)) ((1 . 2) (2 . 1)) ((1 . 3) (3 . 2) (2 . 3) (3 . 1)))
> (niveles-calkin-wilf 4)
(((1 . 1))
 ((1 . 2) (2 . 1))
 ((1 . 3) (3 . 2) (2 . 3) (3 . 1))
 ((1 . 4) (4 . 3) (3 . 5) (5 . 2) (2 . 5) (5 . 3) (3 . 4) (4 . 1)))
```

d) Definir la función **(detecta-calkin-wilf pred)** que recibe como argumento una función **pred** de 1 argumento que retorna un valor booleano, y recorre el árbol de Calkin-Wilf buscando el 1er nodo para el cual la evaluación de **pred** con ese nodo retorna #t. El recorrido del árbol será de arriba hacia abajo y de izquierda a derecha. Al definir la función **detecta-calkin-wilf**, puede asumir que siempre existirá dicho nodo.

Considere las siguientes definiciones y ejemplos:

```
(define suma
     (lambda (n) (lambda (nodo) (= n (+ (car nodo) (cdr nodo))))))
(define producto
     (lambda (n) (lambda (nodo) (= n (* (car nodo) (cdr nodo))))))

> (detecta-calkin-wilf (suma 2))
(1 . 1)
> (detecta-calkin-wilf (suma 5))
(3 . 2)
> (detecta-calkin-wilf (producto 15))
(3 . 5)
> (detecta-calkin-wilf (lambda (nodo) (= (car nodo) 5)))
(5 . 2)
```