Fecha: 05/07/2024 Nombre y Apellido:

PARADIGMAS DE PROGRAMACIÓN
3er Parcial: Programación Funcional

INSTRUCCIONES:
- ✓ **Las soluciones de los Ejercicios 1, 2a y 2b deben entregarse en estas hojas.**
- ✓ **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) La gramática del Cálculo Lambda define los siguientes tipos de expresiones:

	Constantes, variables, y operadores aritméticos.
	Variables, funciones, y números enteros.
	Variables, abstracciones funcionales y aplicaciones funcionales.
	Condiciones, bucles, y estructuras de datos.
	Ninguna de las opciones anteriores es correcta.

b) En el Paradigma Funcional:

	En el cuerpo de una función una variable puede cambiar su valor mediante una instrucción de asignación.
	Las funciones siempre producen la misma salida para las mismas entradas y no tienen efectos secundarios.
	Una función puede tener diferentes salidas para las mismas entradas dependiendo del estado del programa.
	Una función no puede ser evaluada más de una vez dentro del cuerpo de otra función.
	Ninguna de las anteriores es correcta.

c) En el Cálculo Lambda:

	Un término solamente puede ser reducido cuando tiene forma normal.
	La aplicación reiterada de la Regla Beta a un término siempre garantiza encontrar su forma normal.
	La forma normal de un término puede variar en función de la estrategia de reducción utilizada.
	La aplicación de la Regla Beta a un término da como resultado un término equivalente.
	Ninguna de las anteriores.

d) El término del Cálculo Lambda: λx.λf.(x (f f))

	No se puede expresar en Scheme ya que no posee variables libres.
	Presenta a **x** como variable libre y ligada.
	En Scheme definiría una función sin nombre de orden superior.
	Incluye un redex.
	Ninguna de las afirmaciones anteriores es correcta.

e) Dada la siguiente expresión en Scheme:

```
(define f (lambda (n) (cons n (lambda () (* 2 n)))))
```

Al evaluar:

```
(f (+ 3 2))
```

	Produce un error al querer formar un par cuyo segundo elemento es una función.
	Produce como resultado el par (#<procedure> . #<procedure>)
	Produce como resultado el par (5 . #<procedure>)
	Produce como resultado el par (5 . 10)
	Ninguna de las afirmaciones anteriores es correcta.

Ejercicio 2 (15 puntos)

Dado el siguiente término del Cálculo Lambda:

$$(\lambda c.\lambda w.(\lambda z.\lambda z.(f\ z)\ (c\ w))\ w)$$

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 las reglas utilizadas en cada paso.

Ejercicio 3 (25 puntos)

Defina en *Scheme* las siguientes funciones de orden superior:

a) **(aplica-cond *lista-pares*)**

La función **aplica-cond** será evaluada con una lista compuesta por listas de 2 funciones, con la siguiente estructura:

((*condición*$_1$ *fun*$_1$) (*condición*$_2$ *fun*$_2$) … (*condición*$_m$ *fun*$_m$))

Donde cada una de las funciones *condición*$_i$ y *fun*$_i$ (con `i = 1 … m`), son funciones de 1 argumento.

Al evaluar **aplica-cond** el resultado será una función de 1 argumento, la cual al evaluarse con un *valor* realizará el siguiente proceso: Siguiendo el orden de la lista, se evaluará cada función *condición*$_i$ con el *valor* hasta encontrar alguna que no retorne #f, en cuyo caso el resultado será aquel de evaluar la correspondiente *fun*$_i$ con el *valor* dado. Si todas las funciones *condición*$_i$ retornan #f, se retornará el *valor* directamente.

Analice detenidamente estos ejemplos:

```
(define suma1 (lambda (x) (+ x 1)))
(define cuadrado (lambda (x) (* x x)))

(define lista-pares-1 (list (list even? suma1)
                            (list odd? cuadrado)))

(define lista-pares-2 (list (list number? cuadrado)
                            (list pair? cdr)
                            (list symbol? list)))

> (aplica-cond lista-pares-1)
#<procedure>

> ((aplica-cond lista-pares-1) 6)
7

> ((aplica-cond lista-pares-2) '(d e f))
(e f)

> ((aplica-cond lista-pares-2) 'a)
(a)

> ((aplica-cond lista-pares-2) "cadena")
"cadena"
```

b) (map-condicional *lista-pares lista*)

Esta función aplica a cada elemento de ***lista*** las condiciones y funciones de ***lista-pares***, retornando una lista con los resultados.

Importante: Para resolver este ítem **debe utilizar** la función definida en el pto. **a)**

Considere los siguientes ejemplos:

```
> (map-condicional lista-pares-1 '(3 4 5 6))
(9 5 25 7)

> (map-condicional lista-pares-2 '(a (b c) 10 (d e f) g))
((a) (c) 100 (e f) (g))
```

Ejercicio 4 (35 puntos)

Con motivo de la realización de la Copa América 2024, se desea representar con listas en Scheme la información de los juegos disputados. Para un juego, se utilizará una lista con:

- Nombre de la Selección 1.
- Lista de jugadores de la Selección 1 que convirtieron goles.
- Nombre de la Selección 2.
- Lista de jugadores de la Selección 2 que convirtieron goles.
- Fecha de disputa.

Si una selección no convierte ningún gol en el juego, la lista de jugadores será una lista vacía.

Por ejemplo, para representar **Ecuador 3** vs **Jamaica 0**, disputado el **26/06/2024**, siendo **Palmer, Minda y Palmer** los jugadores de Ecuador que convirtieron los goles, tenemos la lista:

```
'(ecuador (palmer minda palmer) jamaica () "26/06/2024")
```


Por otro lado, los resultados de un grupo de la primera fase son representados como una lista de juegos:

```
(define grupo-b
  '((ecuador (sarmiento) venezuela (cadiz bello) "22/06/2024")
    (mexico (arteaga) jamaica () "22/06/2024")
    (ecuador (palmer minda palmer) jamaica () "26/06/2024")
    (venezuela (rondon) mexico () "26/06/2024")
    (mexico () ecuador () "30/06/2024")
    (jamaica () venezuela (bello rondon ramirez) "30/06/2024")
 ))
```

Defina las siguientes funciones en Scheme:

a) (**cantidad-empates** *lista-juegos*): retorna la cantidad de juegos empatados. Es decir, la cantidad de juegos en los que ambas selecciones convirtieron la misma cantidad de goles. Ejemplos:

```
> (cantidad-empates grupo-b)
1

> (cantidad-empates '())
0
```

b) (**goles-convertidos** *selección lista-juegos*): retorna la cantidad de goles convertidos por la *selección* indicada en ***todos*** los juegos que disputó. Ejemplos:

```
> (goles-convertidos 'ecuador grupo-b)
4

> (goles-convertidos 'mexico grupo-b)
1

> (goles-convertidos 'jamaica grupo-b)
0
```

c) (**goleadores** *selección lista-juegos*): retorna una lista ***sin repetidos*** de los jugadores que convirtieron al menos un gol en la *selección* dada, considerando todos los juegos que disputó. Ejemplos:

```
> (goleadores 'venezuela grupo-b)
(cadiz bello rondon ramirez)

> (goleadores 'jamaica grupo-b)
()

> (goleadores 'ecuador grupo-b)
(sarmiento palmer paez minda)
```

d) (**lista-rivales** *lista-juegos*): retorna una ***función*** de un argumento. Esta función, al ser evaluada con una selección, devuelve una lista con los rivales de la selección en la *lista-juegos*. Ejemplos:

```
> (define rivales-grupo-b (lista-rivales grupo-b))

> (rivales-grupo-b 'jamaica)
(mexico ecuador venezuela)

> (rivales-grupo-b 'canada)
()
```