Nombre:_____

Lenguajes y Paradigmas de Programación
Curso 2014-2015
Primer parcial

Normas importantes

- **La puntuación total del examen es de 10 puntos.**
- **Se debe contestar cada pregunta en las hojas que entregamos. Utiliza las últimas hojas para hacer pruebas. No olvides poner el nombre.**
- **La duración del examen es de 2 horas.**

Ejercicio 1 (1 punto)

a) (0,25 puntos) Explica brevemente con un pequeño párrafo (alrededor de 10 palabras) la contribución que consideres más importante de las siguientes personas a la historia de la Informática:

- Herman Hollerith: Inventó a finales del siglo XIX las tarjetas perforadas para codificación de información (censo USA). Permitieron introducir los programas y los datos en muchos de los primeros computadores.

- Alan Turing: Crea la máquina de Turing, que es un modelo computacional abstracto que permite estudiar matemáticamente cuáles son los límites del cálculo y la computación, si existen problemas que no pueden ser calculados por muy potente que sea el computador.

- John von Neumann: Propone la arquitectura básica de un computador digital de propósito general con programas almacenados en memoria.

- John McCarthy: Fue el creador del lenguaje Lisp, desarrollado a finales de los 50 en el MIT. Primer lenguaje del paradigma de programación funcional.

- John Backus: Fue el creador del lenguaje FORTRAN, el primer lenguaje comercial desarrollado por IBM.

b) (0,25 puntos) Ordena temporalmente los siguientes lenguajes de programación

- C
- Java
- Scheme
- Go
- Scala
- Python

C - Scheme - Python - Java - Scala - Go

c) (0,5 puntos) Explica qué es un símbolo en Scheme. Indica si son o no símbolos los siguientes datos:

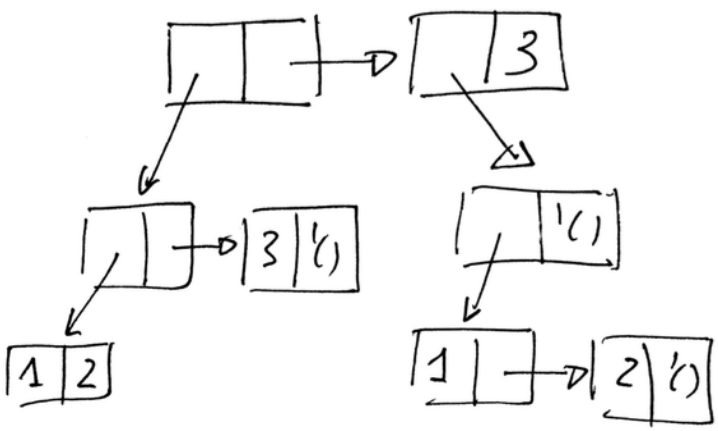
Un símbolo es un identificador que puede asociarse o ligarse (bind) a un valor (cualquier dato de primera clase). Los símbolos son tipos primitivos que pueden pasarse como parámetro o ligarse a variables. Para su definición se utiliza el quote, para evitar que el intérprete los evalúe.

1. `(symbol? cons)` → `___#f____`
2. `(symbol? "hola")` → `__#f_____`
3. `(symbol? 'a)` → `___#t____`
4. `(symbol? 'hola)` → `__#t_____`
5. `(define b 4)`
   `(symbol? b)` → `___#f_____`

Ejercicio 2 (1,5 puntos)

a) (0,5 puntos) Dibuja el diagrama caja y puntero (*Box & Pointer*) de la siguiente expresión y explica si genera una lista o no:

```
(cons (list (cons 1 2) 3)
      (cons (cons (list 1 2)
                  '())
            3))
```



No es una lista, porque la secuencia de partes derechas de parejas no termina en una lista vacía, sino en el número 3.

b) (0,5 puntos) Para cada una de las siguientes definiciones, escribe una expresión de Scheme correcta en la que se invoque a `f` y que devuelva 12.

```
(define f
     (lambda (x)
         (lambda ()
             (+ x x))))

____ ((f 6)) ________
```

```
(define (f x)
     (lambda (y)
             (* x y)))

_____ ((f 3) 4) _________
```

c) (0,5 puntos) Dado el siguiente código en Scheme, indica:

- qué definiciones de funciones se realizan
- qué invocaciones a funciones se realizan

Indica en qué número de línea se realiza cada invocación o cada definición.

```
1. (define (foo x)
2.    (lambda (y)
3.        (* x y)))
4. (define (bar x)
5.     (+ x x))
6. (define a (foo 10))
7. (bar (a 3))
```

3 definiciones y 3 invocaciones:

Línea 1 - Definición de la función foo
Línea 4 - Definición de la función bar
Línea 6 - Invocación a la función foo, que devuelve una función que se guarda en la variable a
Línea 7 - Invocación a la función a y después a la función bar

Ejercicio 3 (3 puntos)

a) (1,5 puntos) Implementa una función recursiva `(triangulo n)` que devuelva una lista con n listas, cada una de las cuales debe tener n asteriscos. Puedes usar funciones auxiliares, pero todas las funciones que uses deben ser recursivas.

Ejemplo:

```
(triangulo 10) ⇒
((* * * * * * * * * *)
 (* * * * * * * * *)
 (* * * * * * * *)
 (* * * * * * *)
 (* * * * * *)
 (* * * * *)
 (* * * *)
 (* * *)
 (* *)
 (*))
```

```
(define (crea-fila n)
  (if (= n 0)
      '()
      (cons '* (crea-fila (- n 1)))))

(define (triangulo n)
  (if (= n 0)
      '()
      (cons (crea-fila n)
            (triangulo (- n 1)))))
```

b) (1,5 puntos) Dada una lista con al menos dos números escribe una función recursiva `(mayor-pareja lista)` que devuelva la pareja de números consecutivos cuya suma es la mayor de todas las sumas de números consecutivos de la lista.

Ejemplo:

```
(mayor-pareja '(1 5 8 2 20))
⇒ (2 . 20)
(mayor-pareja '(1 15 8 2 20))
⇒ (15 . 8)
```

```
(define (mayor-suma p1 p2)
  (if (> (+ (car p1) (cdr p1))
         (+ (car p2) (cdr p2)))
      p1
      p2))

(define (primera-pareja lista)
  (cons (car lista)
        (cadr lista)))

(define (mayor-pareja lista)
  (if (null? (cddr lista))
      (primera-pareja lista)
      (mayor-suma (primera-pareja lista)
                  (mayor-pareja (cdr lista)))))
```

Ejercicio 4 (3 puntos)

a) (1,5 puntos) Escribe la función `(expande lista-parejas)` **utilizando funciones de orden superior**. La función recibe una lista de parejas que contienen un dato y un número y devuelve una lista donde se han "expandido" las parejas, repitiendo tantos elementos como el número que indica cada pareja. Debes escribir también la implementación de cualquier función auxiliar que consideres necesaria.

Ejemplo:

```
(expande (list (cons 'a 4) (cons "hola" 2) (cons #f 3)))
⇒ (a a a a "hola" "hola" #f #f #f)
```

```
(define (expande-pareja pareja)
  (if (= (cdr pareja) 0)
      '()
      (cons (car pareja)
            (expande-pareja (cons (car pareja)
                                  (- (cdr pareja) 1))))))

(define (expande lista-parejas)
  (apply append (map expande-pareja lista-parejas)))
```

b) (1,5 punto) Define las funciones recursivas `(make-saludos lista-de-cadenas)` y `(aplica-lista-funcs lista-de-funciones dato)`. La función `make-saludos` recibe una lista de cadenas (saludos) y devuelve una lista de funciones de un argumento que concatenan un saludo con el argumento. La función `aplica-lista-funcs` recibe una lista de funciones de un argumento y un dato y devuelve una lista con los resultados de aplicar cada función al dato.

Nota: para concatenar cadenas puedes usar la función `string-append`

Ejemplo:

```
(make-saludos '("Hola " "Adios " "Cómo estás "))
⇒ (#<procedure> #<procedure> #<procedure>)

(aplica-lista-funcs (make-saludos '("Hola " "Adios " "Cómo estás "))
"Pepe")
⇒ ("Hola Pepe" "Adios Pepe" "Cómo estás Pepe")
```

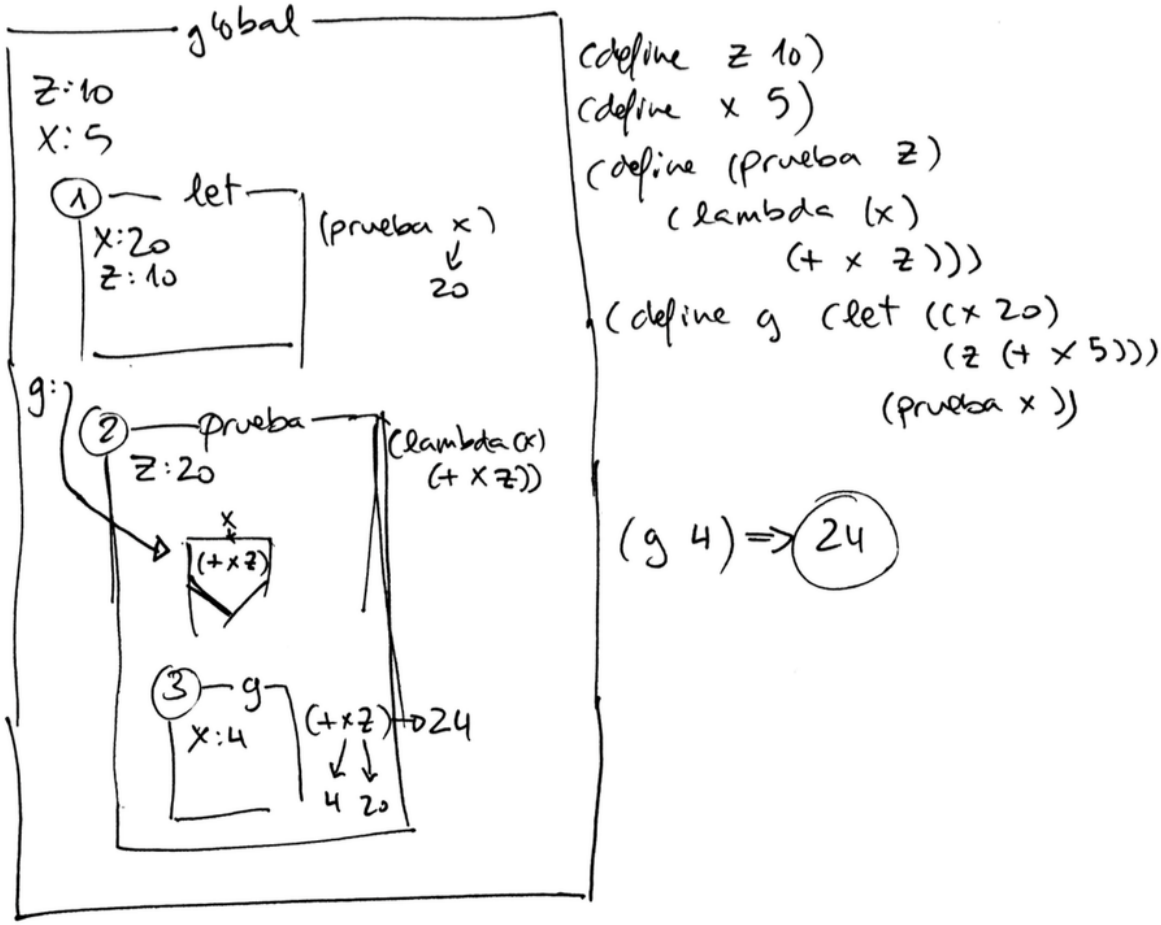
```
(define (make-saludos lista-cadenas)
  (if (null? lista-cadenas)
      '()
      (cons (lambda(x) (string-append (car lista-cadenas) x))
            (make-saludos (cdr lista-cadenas)))))

(define (aplica-lista-funcs lista-funciones dato)
  (if (null? lista-funciones)
      '()
      (cons ((car lista-funciones) dato)
            (aplica-lista-funcs (cdr lista-funciones) dato))))
```

Ejercicio 5 (1,5 puntos)

Dados el siguiente código en Scheme, dibuja en un diagrama los ámbitos que se generan. Junto a cada ámbito escribe un número indicando en qué orden se ha creado. ¿Cuál es el resultado? ¿Cuántos ámbitos se crean? ¿Se crea alguna clausura?

```
(define z 10)
(define x 5)
(define (prueba z)
    (lambda (x) (+ x z)))
(define g (let ((x 20)
                (z (+ x 5)))
    (prueba x))
(g 4)
```



El resultado es 24. Se crean 3 ámbitos locales, en el orden indicado en el dibujo. Sí que se crea una clausura, la función a la que apunta la variable "g".