Nombre:_____

Primer parcial Lenguajes y Paradigmas de Programación (Curso 2016-17)

Normas importantes

- **Los profesores no contestarán ninguna pregunta durante la realización del examen, exceptuando aquellas que estén relacionadas con algún posible error en el enunciado de alguna pregunta.**
- **Puedes definir y utilizar funciones auxiliares en aquellos ejercicios en los que sea posible.**
- **La duración del examen es de 2 horas.**

Ejercicio 1 (2,5 puntos)

a) Contesta las siguientes **preguntas de tipo test**. Cada respuesta errónea penaliza con 0,05 puntos.

a.1) (0,2 puntos) Cuál de las siguientes afirmaciones es cierta sobre el Lisp (sólo una):

a) Se desarrolló en la década de 1940.
b) Lo desarrolló el investigador de IBM John W. Backus.
c) Se desarrolló en el MIT para resolver problemas de Inteligencia Artificial.
d) El objetivo principal de su desarrollo es compilar las expresiones simbólicas en código máquina altamente eficiente.

a.2) (0,2 puntos) Cuál de las siguientes ordenaciones cronológicas de lenguajes de programación es correcta (sólo una):

a) Fortran, Java, C, Swift, Python
b) Fortran, Java, C, Python, Swift
c) C, Fortran, Python, Java, Swift
d) Fortran, C, Python, Java, Swift

a.3) (0,2 puntos) Cuál de las siguientes afirmaciones sobre los primeros computadores creados en la década de los 40 es cierta (sólo una):

a) La máquina Z3 tenía un conjunto de instrucciones Turing-completo.
b) Las válvulas de vacío se utilizaron desde el principio de la década, en computadores como el Z3, el ABC o el Mark-1.
c) Los computadores británicos "Máquina de Manchester" y EDSAC fueron los primeros en utilizar los programas almacenados en memoria.
d) Alan Turing trabajó junto con Von Neumann para definir la famosa arquitectura de este último.

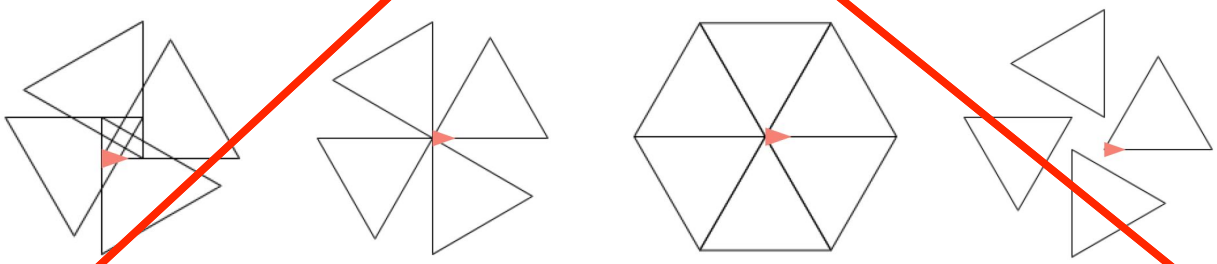
No se ha dado este curso

a.4) (0,2 puntos) Supongamos el siguiente código Scheme que usa la librería de gráficos de tortuga:

```
(define (figura a b)
  (if (> a 0)
      (begin
        (draw 100)
        (turn 120)
        (draw 100)
        (turn 120)
        (draw 100)
        (turn 120)
        (turn b)
        (figura (- a 1) b))))

(figura 4 90)
```

¿Cuál de las siguientes figuras se genera? Redondea la letra bajo la figura.



a) b) c) d)

No entra en este parcial

a.5) (0,2 puntos) Supongamos la siguiente función

```
(define (lista->str lista str)
    (if (null? lista)
        str
        (string-append (car lista) (lista->str (cdr lista) ""))))
```

Y supongamos la siguiente llamada a la función:

```
(lista->str '("hola" "como" "estas") "")
```

¿Cuál de las siguientes afirmaciones es correcta? (sólo una):

No entra en este parcial

No entra en este parcial

A. La función usa recursión por la cola y la llamada devuelve "holacomoestas"
B. La función no usa recursión por la cola y la llamada devuelve la cadena vacía ("")
C. La función no usa recursión por la cola y la llamada devuelve "holacomoestas"
D. La función usa recursión por la cola y la llamada devuelve la cadena vacía ("")

================== FIN DE LAS PREGUNTAS DE TIPO TEST ==================

b) (0,5 puntos) Escribe a la derecha de cada expresión el resultado que mostraría en la pantalla DrRacket. En el caso en que la expresión genere un error indícalo (no hace falta explicarlo).

a) `(cons (cons 1 2)`
`(cons 1 '()))`

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

c) `(append 2 '(1 2 3))`

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

e) `(append '(1 2 3) 2)`

c) (0,4 puntos) Rellena el hueco en el siguiente algoritmo de la versión memoizada de Pascal:

No entra en este parcial

```
(define (pascal-memo fil col lista)
  (cond ((= col 0) 1)
        ((= col fil) 1)

        ____________________________

        (else (put (cons fil col)
                   (+ (pascal-memo (- fil 1) (- col 1) lista)
                      (pascal-memo (- fil 1) col lista))
                   lista))))
```

d) (0,6 puntos) Rellena los huecos en las siguientes expresiones para que las siguientes pruebas no fallen:

d.1)
```
(check-equal?
       (filter
              [                                    ]
              '(3 8 90 99 100 102 105))
       '(102 105))
```

d.2)
```
(check-equal?
       (__________
                  (lambda (x)
                       (+ x 2))
                  '(1 12 9 4))
       '(3 14 11 6))
```

d.3)
```
(check-equal?
      (fold-right
            (lambda(x y)
                (+ (car x) y))
            0
            (map (lambda (x)
                    (cons (* 10 (car x)) (cdr x)))
                 (list (cons 2 3) (cons 3 4) (cons 5 5))))


      _________________)
```

Ejercicio 2 (1,25 puntos)

Escribe la **función recursiva** `(tiradas-consecutivas? lista-parejas n)` que recibe una lista de parejas de números que representan una tirada de dos dados y un número `n`, y debe comprobar si hay al menos dos tiradas consecutivas en las que se haya obtenido dicha cantidad `n`. Es decir, la suma de los 2 dados en cada una de ambas tiradas consecutivas debe ser igual a `n`.

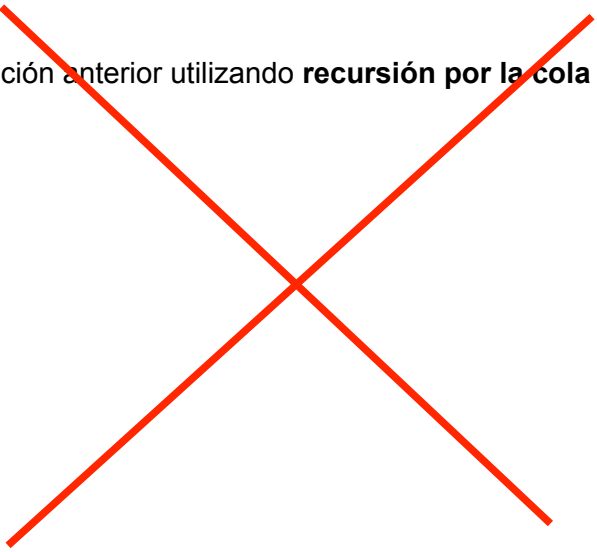
```
(tiradas-consecutivas? (list (cons 2 4) (cons 3 4) (cons 5 2)
                             (cons 6 4) (cons 3 3)) 7) ; ⇒ #t
(tiradas-consecutivas? (list (cons 2 4) (cons 3 4) (cons 5 2)
                             (cons 6 4) (cons 3 3)) 6) ; ⇒ #f
```

Ejercicio 3 (2 puntos)

a) (1 punto) Escribe la **función recursiva** (`numeros-en-intervalo a b`) que recibe dos números enteros que representan los extremos de un intervalo y devuelve la lista de números enteros entre esos dos valores

```
(numeros-en-intervalo 4 8) ; ⇒ {5  6  7}
(numeros-en-intervalo 1 2) ; ⇒ {}
```

b) (1 punto) Escribe la función anterior utilizando **recursión por la cola** (tail recursion)



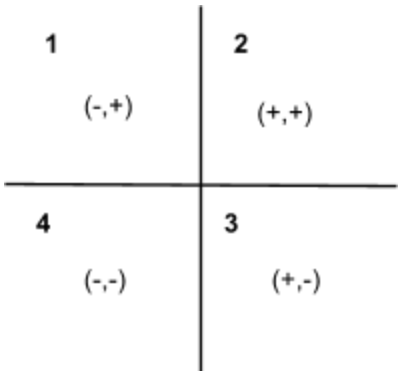
No entra en este parcial

Ejercicio 4 (1,25 punto)

Implementa utilizando **funciones de orden superior** la función `(puntos-coordenada lista-puntos coord n)` que recibe una lista de parejas que representan puntos 2D (coordenadas `x` `y`), un símbolo que representa una coordenada y un número que representa el valor de la coordenada, y devuelve la lista de puntos que tienen ese mismo valor de coordenada.

```
(puntos-coordenada (list (cons 2 3) (cons 3 4) (cons 4 1)
                         (cons 8 4) (cons 4 5)) 'x  4)
; ⇒ {{4 . 1} {4 . 5}}
(puntos-coordenada (list (cons 2 3) (cons 3 4) (cons 4 1)
                         (cons 8 4) (cons 4 5)) 'y  3)
; ⇒  {{2 . 3}}
```

Ejercicio 5 (1,5 puntos)



a) (0,5 puntos) Define la función `(en-cuadrante punto)` que devuelva el cuadrante en el que se encuentra un punto 2D (pareja con las coordenadas x y) según la figura de la derecha. Suponemos que el punto (0,0) está en el cuadrante 2.

Ejemplos:

```
(en-cuadrante (cons -3 -2)) ; ⇒ 4
(en-cuadrante (cons 3 -2)) ; ⇒ 3
```

b) (1 punto) Implementa utilizando **funciones de orden superior** la función `(suma-cuadrante lista-puntos n)` que devuelva la pareja resultante de sumar las coordenadas x e y de todos los puntos de la lista situados en el cuadrante indicado. Debes utilizar la función auxiliar definida en el apartado anterior. Por ejemplo:

```
(define lista (list (cons 1 1) (cons -1 2) (cons 3 4) (cons -4 10) (cons -2 -2)))
(suma-cuadrante lista 2) ; ⇒ {4 . 5}
(suma-cuadrante lista 1) ; ⇒ {-5 . 12}
(suma-cuadrante lista 3) ; ⇒ {0 . 0}
```

Ejercicio 6 (1,5 puntos)

Dadas estas dos funciones, `suma-p1` y `suma-p2`:

```
(define (suma-p1 pred elem lista)
  (if (pred elem)
      (cons (+ 1 (car lista)) (cdr lista))
      lista))

(define (suma-p2 pred elem lista)
  (if (pred elem)
      (cons (car lista)
            (cons (+ 1 (cadr lista))
                  '()))
      lista))
```

a) (0,5 puntos) Escribe un `check-equal?` para cada una de ellas, utilizando predicados distintos.

- En el primer check-equal debes escribir una expresión en la que se prueba la función `suma-p1` y el resultado esperado sea la lista {2 3}.
- En el segundo check-equal debes escribir una expresión en la que se prueba la función `suma-p2` y el resultado esperado sea la lista {3 1}.

b) (1 punto) Utilizando las funciones anteriores, define la **función recursiva** `(cuenta-2 p1 p2 lista)` que recibe 2 predicados y una lista, y devuelve una lista de 2 elementos donde cada uno sea el total de elementos de la lista que cumplen cada uno de los predicados. Ten en cuenta que hay elementos que pueden cumplir más de un predicado.

Ejemplos:

```
(cuenta-2 string? number? '("hola" 1 2 3 "que" bye 4 5 "tal" hello 6)) ; ⇒ {3 6}
(cuenta-2 (lambda (x) (>= x 10))
          even?
          '(8 30 3 25 4 7 22)) ; ⇒ {3 4}
```