Primer parcial Lenguajes y Paradigmas de Programación
Curso 2017-18

Nombre:_____

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.**
- **Debes implementar las funciones recursivas usando procesos recursivos, no recursión por la cola.**
- **La duración del examen es de 2 horas.**

Ejercicio 1 (1 punto) Contesta las siguientes **preguntas de tipo test** rodeando la letra de la respuesta correcta. Cada respuesta errónea penaliza con 0,05 puntos.

a.1) (0,2 puntos) Indica cuál de las siguientes afirmaciones **es cierta** sobre la genealogía de los lenguajes de programación:

a) El lenguaje Lisp es el padre de lenguajes dinámicos como C, C++, Java o Scala.
b) El lenguaje SIMULA es el padre de lenguajes orientados a objetos como Smalltalk, Objective-C o Java.
c) El lenguaje Fortran es el padre de lenguajes procedurales como APL, Prolog o Perl.
d) El lenguaje Algol es el padre de lenguajes funcionales como Scheme o Haskell.

a.2) (0,2 puntos) Indica cuál de las siguientes afirmaciones **es cierta** sobre la construcción de abstracciones en computación:

a) Las abstracciones permiten ahorrar tiempo y esfuerzo a la hora de atacar la complejidad del mundo real.
b) La posibilidad de construir abstracciones depende del lenguaje de programación escogido: hay lenguajes de alto nivel que no permiten construir abstracciones.
c) La construcción de abstracciones permite que el código sea más eficiente, al optimizar el rendimiento del compilador.
d) No importan los nombres escogidos en las abstracciones ya que siempre es posible refactorizar y cambiar un nombre por otro.

a.3) (0,2 puntos) Indica cuál de las siguientes afirmaciones **es falsa** sobre la programación funcional:

a) La programación funcional permite concatenar operaciones como filter o map sobre streams de datos.
b) La programación funcional permite realizar una programación evolutiva en la que se pueden construir fácilmente programas complejos a partir de la composición de programas más simples.
c) La programación funcional permite escribir código que se puede probar automáticamente mediante la utilización de funciones de orden superior.
d) La programación funcional permite programar fácilmente sistemas concurrentes con múltiples hilos de ejecución.

a.4) (0,2 puntos) Supongamos la función:

```
(define (maximo lista)
    (if (null? (cdr lista))
        (car lista)
        (mayor (car lista) (maximo (cdr lista)))))
```

¿Qué devuelve **la primera llamada recursiva** a la función `maximo` en la ejecución de la siguiente expresión?

```
(maximo '(30 15 21 0 3))
```

a) 15
b) 3
c) 21
d) 30

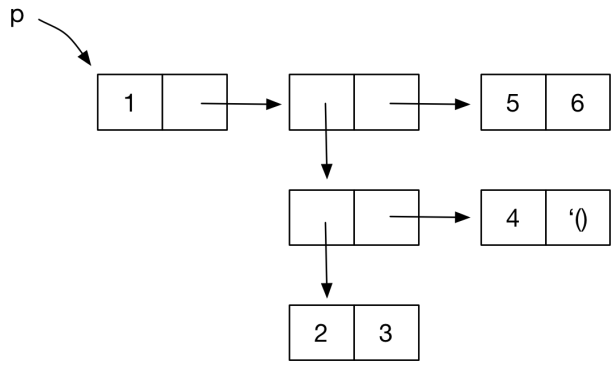
a.5) (0,2 puntos) Una clausura es:

a) Una expresión lambda.
b) Una función creada en tiempo de ejecución en el ámbito local de otra función.
c) Una evaluación de una expresión lambda que devuelve un procedimiento creado en tiempo de ejecución.
d) Una función constructora que devuelve un procedimiento creado por una expresión lambda.

Ejercicio 2 (2 puntos)

a) (0,5 puntos)

a.1) Escribe la sentencia en Scheme que, utilizando el mínimo número de llamadas a `cons` y `list`, genera el siguiente diagrama box-and-pointer:

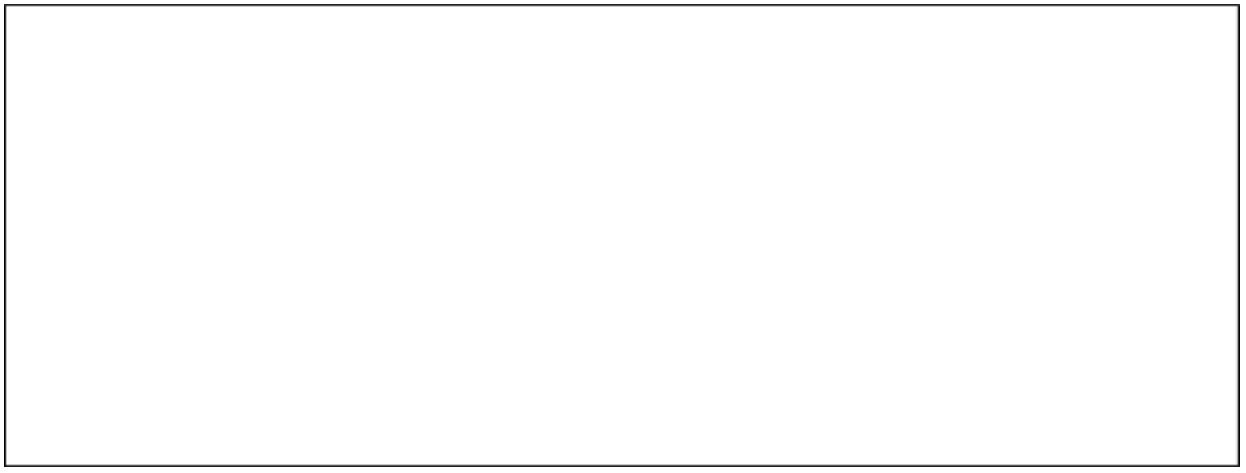


a.2) Escribe la sentencia en Scheme que devuelve el número 4 a partir de la variable `p` anterior.

b) (0,5 puntos) Dada la función recursiva `generica-rec`, rellena el hueco para implementar la función equivalente `generica-fos` utilizando una función de orden superior:

```
(define (generica-rec fun1 fun2 lista base)
  (if (null? lista)
      base
      (fun1 (fun2 (car lista))
            (generica-rec fun1 fun2 (cdr lista) base))))

(define (generica-rec-fos fun1 fun2 lista base)
```



```
)



(check-equal? (generica-rec + string-length '("hola" "adios") 0)
              (generica-fos + string-length '("hola" "adios") 0))
```

c) (0,5 puntos) Dadas las siguientes funciones:

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

(define (g x)
  (lambda (y)
    (- x y)))
```

Indica qué debe haber en los huecos para que las siguientes pruebas de `f` y `g` sean correctas:

prueba de `f`: `(check-equal? _____ 6)`

prueba de `g`: `(check-equal? _____ 3)`

d) (0,5 puntos) ¿Qué debería haber en el hueco para que el check-equal sea correcto?

```
(check-equal? (fold-left ___________ '() '((1 . 20) (5 . 8) (7 . 10))
              '((10 . 7) (8 . 5)  (20 . 1))
```

Ejercicio 3 (1,5 puntos)

a) (0,75 puntos) Escribe la **función recursiva** `(cruza-cero? lista)` que recibe una lista ordenada creciente de números y comprueba si los números pasan de negativo a positivo (cruzan por el cero).

Ejemplos:

```
(cruza-cero? '(-10 -5 -2 10 20)) ⇒ #t
(cruza-cero? '(-20 -12 -9 -3)) ⇒ #f
(cruza-cero? '(3 12 18 20 25)) ⇒ #f
```

b) (0,75 puntos) Define la **función recursiva** `restagrama` que verifica los siguientes check-equal?:

```
(check-equal? (restagrama '(c h o l a))
              '((c h o l a) (h o l a) (o l a) (l a) (a)))
(check-equal? (restagrama '(u n o))
              '((u n o) (n o) (o)))
```

Ejercicio 4 (2,75 puntos)

a) (0,75 puntos) Escribe la **función recursiva** `(crea-lista n elem)` que recibe un elemento y un número y devuelve una lista con n repeticiones del elemento.

Ejemplos:

```
(crea-lista 3 #\o) ⇒ '(#\o #\o #\o)
(crea-lista 5 2) ⇒ '(2 2 2 2 2)
```

b) (1 punto) Escribe la **función recursiva** `(expande-simbolos lista-parejas)` que **use la función anterior `crea-lista`** y devuelva una lista con los elementos definidos por las partes derechas de las parejas **con símbolos** tantas veces como indica el número de las partes izquierdas.

Pista: recuerda que la función para comprobar si un dato es un símbolo es `(symbol? dato)`.

Ejemplos:

```
(expande-simbolos '((2 . a) (4 . #\a) (3 . b) (5 . #f))) ⇒ '(a a b b b)
```

c) (1 punto) Usando una **composición de funciones de orden superior** implementa la función `(expande-simbolos-fos lista-parejas)` que haga lo mismo que la función anterior.

Pista: Puedes usar la función `(concatena lista)` que recibe una lista con listas y las concatena todas:

```
(define (concatena lista)
    (fold-right append '() lista))

(concatena '((a a) (b b b))) ⇒ (a a b b b)
```

Ejercicio 5 (2,75 puntos)

a) (0,75 puntos) Define las funciones `(añade-izq elem pareja-listas)` y `(añade-der elem pareja-listas)` que funcionen como indican los siguientes `check-equal?`:

```
(check-equal? (añade-izq 'a (cons '(b c) '(d))) (cons '(a b c) '(d)))
(check-equal? (añade-der 'a (cons '(b c) '(d))) (cons '(b c) '(a d)))
```

b) (1 punto) Define la función **recursiva** `(separar-pares-impares lista-num)` que, dada una lista de números, devuelva una **pareja** cuya parte izquierda sea una **lista** con los números pares y su parte derecha una **lista** con los números impares de la lista. Puedes usar las funciones del apartado anterior, suponemos que están bien definidas.

Ejemplo:

```
(separar-pares-impares '(3 6 8 1 5 4)) ⇒ {{6 8 4} . {3 1 5}}
```

c) (1 punto) Define `(separar-pares-impares-fos lista-num)` utilizando **funciones de orden superior,** que haga lo mismo que la función anterior. Igual que antes puedes usar las funciones del apartado a), suponemos que están bien definidas.