

Nombre: _____

Lenguajes y Paradigmas de Programación

Curso 2015-16

Primer parcial

Normas importantes

- Puedes crear funciones auxiliares.
- La puntuación total del examen es de 10 puntos. Se debe contestar cada pregunta en las hojas que entregamos. Utiliza la última hoja para hacer pruebas o si necesitas espacio adicional. No olvides poner el nombre.
- La duración del examen es de 2 horas.

Ejercicio 1

a) Contesta las siguientes preguntas de tipo test. Cada respuesta errónea penaliza con 0,1 puntos.

a.1) (0,25 puntos) Suponemos que `lista` es una lista de parejas y queremos que `(fold-right f ... lista)` devuelva un entero. Los argumentos de la función `(f x y)` deben ser (rodea la solución correcta):

- A. `x`: entero, `y`: pareja
- B. `x`: pareja, `y`: entero**
- C. `x`: entero, `y`: entero
- D. `x`: pareja, `y`: pareja
- E. ninguno de los anteriores, `f` tiene sólo 1 argumento

a.2) (0,25 puntos) Indica cuál de las siguientes afirmaciones sobre la historia de los computadores es correcta (sólo una):

- A. La máquina de Turing representó un avance tecnológico fundamental para el desarrollo del primer computador electrónico de propósito general.
- B. Los relés electromecánicos fueron inventados a comienzos del siglo XX y permitieron un avance importante en la velocidad de los primeros computadores.
- C. Sería posible construir un computador de propósito general con programa almacenado en memoria usando relés electromecánicos. Funcionaría correctamente, pero tendría el problema de ser muy lento y muy grande.**
- D. Estados Unidos lideró el desarrollo de los computadores en los años 40 y construyó la gran mayoría de computadores electrónicos de esa época, con figuras como John von Neumann, Max Newmann o Maurice Wilkes.

a.3) (0,25 puntos) Lee las siguientes afirmaciones sobre la historia y conceptos de los lenguajes de programación:

1. Los dos primeros lenguajes de programación de alto nivel originaron dos paradigmas muy distintos: FORTRAN el paradigma imperativo y LISP el paradigma funcional.
2. El desarrollo de nuevos lenguajes de programación se realiza exclusivamente en empresas informáticas con grandes recursos como Apple o IBM.
3. El lenguaje ensamblador es muy eficiente porque tiene flexibilidad suficiente para construir abstracciones que nos permiten razonar y comunicar ideas sobre los problemas que estamos programando.
4. El origen de la programación orientada a objetos está a finales de los años 60, en un lenguaje denominado SIMULA.

¿Qué combinación de las respuestas anteriores es cierta (sólo una)?

- A. 1 y 2
- B. 2 y 4
- C. 1 y 4**
- D. Sólo la 1

a.4) (0,25 puntos)Cuál de las siguientes afirmaciones es cierta sobre el paradigma declarativo frente al imperativo (sólo una):

- A. La programación declarativa es un tipo particular de programación funcional.
- B. La programación imperativa utiliza la mutación de variables, frente a la declarativa en la que se utiliza la asignación y los pasos de ejecución.
- C. En la programación declarativa no existen constantes, ni variables, ni identificadores.
- D. En la programación declarativa no está permitido modificar el valor de una variable una vez definido.**

*** FIN DE LAS PREGUNTAS DE TIPO TEST ***


b) (0,3 puntos) ¿Qué va a aparecer por pantalla cuando se ejecute el siguiente código?

```
(define x 1)
(display "-AAA-")
(define (prueba x)
  (lambda (y)
    (/ x y)))
(display "-BBB-")
(define f (prueba (/ 3 0)))
(display "-CCC-")
(display (f 0))
(display "-DDD-")
```

-AAA--BBB-
/:division by zero

c) (0,3 puntos) El siguiente programa lanza un mensaje de error

```
(define (prueba z)
  ((lambda (x) (+ x z)) 4))
(define f (prueba 2))
(f 4)
```

 *application: not a procedure;
expected a procedure that can be applied to arguments
given: 6
arguments...:*

Explica qué expresión produce el error y por qué.

La expresión que produce el error es `(f 4)`, debido a que `f` no es una función, ya que en `f` se guarda el resultado de `(prueba 2)` que es el número resultante de la expresión `((lambda (x) (+ x z)) 4)`

d) (0,4 puntos) Supongamos el siguiente código:

```
(define x 1)
(define y (+ x 2))
(define (prueba x)
  (let ((x (+ y 3))
        (z (+ x 4)))
    (lambda (z)
      (+ x y z))))
(define f (prueba (+ x 5)))
(f 6)
```

1. Escribe la expresión o expresiones que se evalúan en la invocación a (f 6)

(+ x y z)

2. Indica para cada variable que interviene en la expresión o expresiones anteriores: su valor y su ámbito (global, local o capturada) en la invocación a (f 6)

'x' es capturada y vale 6

'y' es global y vale 3

'z' es local y vale 6

3. Indica el resultado final de la invocación a (f 6)

15

Ejercicio 2 (2 puntos)

a) (1,5 puntos) Escribe una función recursiva (`ordenada-lista-parejas?` `lista-parejas`) que recibe una lista con parejas de números y comprueba si están en orden creciente. Una pareja de números es mayor que otra cuando sus dos números suman más.

```
(ordenada-lista-parejas '((1 . 2) (4 . 5) (6 . 4) (4 . 8))) ; => #t  
(ordenada-lista-parejas '((3 . 4) (4 . 5) (1 . 2))) ; => #f
```

```
(define (suma-pareja pareja)  
  (+ (car pareja) (cdr pareja)))  
  
(define (parejas-ordenadas? lista)  
  (if (null? (cdr lista))  
      #t  
      (and (< (suma-pareja (car lista))  
              (suma-pareja (cadr lista)))  
           (parejas-ordenadas? (cdr lista)))))
```

b) (0,5 puntos) Escribe una función (`suma-lista-parejas-fos` `lista-parejas`) que use la función de orden superior (FOS) `fold-right` para sumar todos los números de una lista de parejas:

```
(suma-lista-parejas-fos '((3 . 4) (4 . 5) (1 . 2))) ; => 19  
(suma-lista-parejas-fos '()) ; => 0
```

```
(define (suma-parejas lista)  
  (fold-right (lambda (pareja suma)  
               (+ (suma-pareja pareja) suma))  
              0  
              lista))
```

Ejercicio 3 (2 puntos)

Escribe la función (resultados-quiniela lista-parejas) que devuelve una lista de 1, X, 2 a partir de una lista de parejas que representa resultados de partidos de fútbol. El resultado es 1 cuando el número izquierdo de la pareja es mayor que el derecho, un 2 cuando es al revés, y una X cuando los dos números son iguales.

(resultados-quiniela '((1 . 0) (2 . 2) (4 . 1) (1 . 2))) ; => {1 X 1 2}

Escribe dos implementaciones:

a) (1,5 puntos) Recursiva

```
(define (resultado pareja)
  (cond
    ((> (car pareja) (cdr pareja)) 1)
    ((= (car pareja) (cdr pareja)) 'X)
    (else 2)))

(define (resultados-futbol lista-resultados)
  (if (null? lista-resultados)
      '()
      (cons (resultado (car lista-resultados))
            (resultados-futbol (cdr lista-resultados)))))
```

b) (0,5 puntos) Con FOS

```
(define (resultados-futbol-FOS lista-resultados)
  (map resultado lista-resultados))
```

Ejercicio 4 (2 puntos)

a) (0,25 puntos) Implementa la función recursiva (`pertenece? dato lista`) que devuelva `#t` o `#f` dependiendo de si un dato está o no en una lista.

```
(pertenece? 'a '(h o l a)) ; => #t
(pertenece? 'b '(a d i o s)) ; => #f
```

```
(define (pertenece? elem lista)
  (if (null? lista)
      #f
      (or (equal? elem (car lista))
          (pertenece? elem (cdr lista)))))
```

b) (1 punto) Implementa la función recursiva (`listas-contiene-elem elem lista`) que recibe una lista de listas, y devuelve otra lista de las listas que contienen el elemento `elem`.

```
(listas-contiene-elem 'e
  (list '(s i) '(h e) '(h e c h o) '(p r a c t i c a s) '(a p r o b a r e)))
; => {{h e} {h e c h o} {a p r o b a r e}}
```

```
(listas-contiene-elem 1 '((3 5) (5 8 1) (2 1 0) (3 4 5 6 7)))
;=> {{5 8 1} {2 1 0}}
```

```
(define (listas-contiene-elem elem lista)
  (if (null? lista)
      '()
      (if (pertenece? elem (car lista))
          (cons (car lista) (listas-contiene-elem elem (cdr lista)))
          (listas-contiene-elem elem (cdr lista)))))
```

c) (0,75 puntos) Implementa la función anterior con FOS

```
(define (listas-contiene-elem-FOS elem lista)
  (filter (lambda(sublista) (pertenece? elem sublista)) lista))
```

Ejercicio 5 (2 puntos)

a) (1,25 puntos) Implementa una función recursiva (`dias-mes n mes`) que devuelva una lista de parejas que representan los días de un determinado mes, desde el día 1 hasta el día `n` indicado como parámetro.

```
(dias-mes 5 'Enero)
; => {{1 . Enero} {2 . Enero} {3 . Enero} {4 . Enero} {5 . Enero}}
```

```
(define (añadir-final-lista x lista)
  (append lista (list x)))
```

```
(define (dias-mes n mes)
  (if (= n 0)
      '()
      (añadir-final-lista (cons n mes)
                          (dias-mes (- n 1) mes))))
```

b) (0,75 puntos) Implementa la función (`calendario-fijo n lista-meses`) utilizando funciones de orden superior, que devuelva una lista de parejas que representan los días de todos los meses indicados en `lista-meses` y con los primeros `n` días de cada mes. Todos los meses tendrán el mismo número de días.

```
(calendario-fijo 3 '(Enero Febrero))
; => {{1 . Enero} {2 . Enero} {3 . Enero} {1 . Febrero} {2 . Febrero} {3 .
Febrero}}
```

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

```
(define (calendario-fijo n lista-meses)
  (aplana-lista (map (lambda (mes) (dias-mes n mes)) lista-meses)))
```