v.8.8

Programación 1 - Práctica 5, parte 2.

1 Clasificando elementos de una lista

La Práctica 5 primera parte contenía una serie de problemas en los cuales había que *filtrar* los elementos de una lista, quedándose con aquellos que cumplían una determinada condición. Vimos en teoría que la función `filter` nos servía para resolver esta clase de problemas. La signatura de dicha función es la siguiente:

```
; filter : (X -> Boolean) List(X) -> List(X)
```

Dado un predicado `p` y una lista `l` con objetos en `X`, queremos devolver una lista con aquellos objetos de `l` para los cuales `p` evalúa a `#true`.

```
(define (filter p l)
             (cond [(empty? l) empty]
                   [else (if (p (first l))
                             (cons (first l) (filter p (rest l)))
                             (filter p (rest l)))]))
```

Algunos ejemplos:

```
(filter even? (list 1 2 3 4 5))
 ==
(list 2 4)
```

```
(filter string? (list 3 "Lista" #true "heterogénea"))
 ==
(list "Lista" "heterogénea")
```

Ejercicio 1. Resuelva los ejercicios de la sección 'Clasificando elementos de una lista' de la Práctica 5, primera parte"] utilizando `filter`.

Ejercicio 2. Diseñe una función `pares` que tome una lista de números `l` y devuelva una lista con los números pares de `l`.

Ejemplo:

```
pares (list 4 6 3 7 5 0)
= (list 4 6 0)
```

Ejercicio 3. Diseñe una función `cortas` que tome una lista de strings y devuelva una lista con aquellas palabras de longitud menor a 5.

Ejemplo:

```
cortas (list "Lista" "de" "palabras" "sin" "sentido")
= (list "de" "sin")
```

Ejercicio 4. Diseñe una función `cerca` que tome una lista de puntos del plano (representados mediante estructuras `posn`), y devuelva la lista de aquellos puntos que están a distancia menor a `MAX`, donde `MAX` es una constante de su programa.

Ejemplo (considerando `5` para la constante) :

```
cerca (list (make-posn 3 5) (make-posn 1 2) (make-posn 0 1) (make-posn 5 6))
= (list (make-posn 1 2) (make-posn 0 1))
```

Ejercicio 5. Diseñe una función `positivos` que tome una lista de números y se quede sólo con aquellos que son mayores a 0.

```
(positivos (list -5 37 -23 0 12))
= (list 37 12)
```

---

2 Aplicando transformaciones a cada elemento de una lista

En la Práctica 5 primera parte se presentan algunos problemas cuya solución se obtiene aplicando una determinada transformación a cada uno de los elementos de una lista.

Es decir, dada una función `f`, estamos interesados en transformar la lista:

$$[a0, a1, ..., an]$$

en

$$[f(a0), f(a1), ..., f(an)]$$

En clase de teoría vimos que la función `map` se podía utilizar para resolver problemas de este tipo. Podemos escribir su signatura como sigue:

```
; map : (X -> Y) List(X) -> List(Y)
```

Es decir, dada una función que transforma objetos de `X` en objetos de `Y`, y una lista con objetos en `X`, devuelve una lista con objetos en `Y`. Recordemos la definición vista en clase:

```
(define (map f l)
  (cond [(empty? l) empty]
        [else (cons (f (first l)) (map f (rest l)))]))
```

Con esta definición obtenemos, por ejemplo:

```
(map sqr (list 1 2 3 4 5))
 ==
(list 1 4 9 16 25)
```

```
(map string-length (list "Lista" "de" "palabras"))
 ==
(list 5 2 8)
```

Ejercicio 6. Diseñe la función `raices`, que dada una lista de números, devuelve una lista con las raíces cuadradas de sus elementos.

```
(raices (list 9 16 4))
= (list 3 4 2)
```

Ejercicio 7. Diseñe una función `distancias` que tome una lista de puntos del plano y devuelva una lista con la distancia al origen de cada uno.

Ejemplo:

```
(distancias (list (make-posn 3 4) (make-posn 0 4) (make-posn 12 5)))
= (list 5 4 13)
```

Ejercicio 8. Diseñe una función `anchos` que tome una lista de imágenes y devuelva una lista con el ancho de cada una.

Ejemplo:

```
(anchos (list (circle 30 "solid" "red") (rectangle 10 30 "outline" "blue")))
= (list 60 10)
```

Ejercicio 9. Diseñe la función `signos`, que dada una lista de números, devuelve una lista con el resultado de aplicarle a cada elemento la función `sgn2` definida en la práctica 1.

```
(signos (list 45 32 -23 0 12))
= (list 1 1 -1 0 1)
```

Ejercicio 10. Diseñe una función `cuadrados` que tome una lista de números y devuelva otra lista donde los elementos que aparezcan sean el cuadrado de los elementos de la lista original.

Ejemplo:

```
(cuadrados (list 1 2 3))  = (list 1 4 9)
```

Ejercicio 11. Diseñe una función `longitudes` que tome una lista de cadenas y devuelva una lista de números que corresponda con la longitud de cada cadena de la lista original.

Ejemplo:

```
(longitudes(list "hola" "cómo" "estás?")) = (list 4 4 6)
```

Ejercicio 12. Diseñe la función `convertirFC`, que convierte una lista de temperaturas medidas en Fahrenheit a una lista de temperaturas medidas en Celsius.

---

3 Operando con los elementos de una lista

Finalmente, algunos ejercicios de la Práctica 5 primera parte buscaban obtener un valor como resultado de realizar una opecion que involucra a todos los elementos de la lista. Es decir, dada una función `f` y una lista

$$[a0, a1, ..., an]$$

estos ejercicios se resolvian haciendo

```
(f a0 (f a1 (... (f an-1 an))))
```

En los ejercicios 25 y 26, dicha función es el producto (`*`) y la concatenación de Strings (`string-append`) respectivamente.

Tal como vimos en teoría, podemos encontrar un patrón en común en la solución de estos ejercicios, y a este patron de solución le llamamos `fold`. La función `fold` recibe tres argumentos:

- La función `f` con la que se quiere operar los elementos de la lista;
- Un valor `c`, que es el resultado esperado para la lista vacía;
- La lista `l` a transformar.

Al evaluarse la expresión

```
(fold f c (cons a0 (cons a1 (... (cons an '()))))))
```

se obtiene

```
(f a0 (f a1 (... (f an c))))
```

Detengase y piense en la signatura de la función `fold`.

Algunos ejemplos concretos:

```
(fold * 1 (list 1 2 3 4 5))
 ==
120
```

```
(fold string-append "" (list "Pro" "gra" "ma" "ción."))
 ==
"Programación"
```

Usando *racket*, la definición de `fold` quedaría:

```
(define (fold f c l)
  (cond [(empty? l) c]
        [else (f (first l) (fold f c (rest l)))]))
```

Ejercicio 13. Diseñe una función `prod` que multiplica los elementos de una lista de números. Para la lista vacía, devuelve `1`.

```
(prod (list 1 2 3 4 5))
= 120
```

Ejercicio 14. Diseñe una función `pegar` que dada una lista de strings, devuelve el string que se obtiene de concatenar todos los elementos de la lista.

```
(pegar (list "Las " "lis" "tas " "son " "complicadas" "."))
= "Las listas son complicadas."
```

Ejercicio 15. Diseñe una función `max` que devuelve en máximo de una lista de naturales. Para la lista vacía, devuelve `0`.

```
(max (list 23 543 325 0 75))
```

```
= 543
```

Ejercicio 16. Diseñe una función `todos-verdaderos`, que dada una lista de booleanos, devuelve `#true` si todos los elementos de la lista son `#true`, y `#false` en caso contrario. Por convención, devuelve `#true` para la lista vacía.

Ejemplos:

```
(todos-verdaderos (list #t #f #t)) = #f
(todos-verdaderos (list #t #t #t #t)) = #t
```

¿Racket le permite usar el operador booleano `and` como argumento para la función `fold`? En caso de que no sea posible, defina su propia función `and2` que se comporte exactamente como `and`.

Esto sucede porque la sintaxis de Racket no clasifica a `and` como una función debido al cortocircuito.

Ejercicio 17. Diseñe una función `largo`, que dada una lista, devuelve el largo de la lista.

Ejemplos:

```
(largo (list 1 "hola" #t -5)) = 4
```

4 Definiciones locales

El uso de patrones es una buena práctica de programación para el diseño de programas. Como ya se ha mencionado, los patrones evitan la repetición de código y simplifican la escritura y la legibilidad de los programas. En general, la mayoría de los lenguajes de programación disponen de estos patrones, aunque cada uno presenta sus particularidades. En este sentido, Racket presenta algunas limitaciones, que generan que algunos programas deban ser escritos de una forma distinta a la vista en clase. Veamos algunos ejemplos.

Empecemos por definir una función `eliminar-0`, que dada una lista de números, devuelve la lista luego de eliminar todas las ocurrencias del valor númerico 0.

Ejemplos:

```
(eliminar-0 (list 0 0 1 -5 0 2.5 0)) = (list 1 -5 2.5)
```

Notar que estamos ante una clásica aplicación del patrón `filter`. Primero, vamos a definir un predicado `distinto-0?`, que dado un número `n`, devuelve `#false` si `n` es 0 y `#true` en caso contrario.

```
; distinto-0?: Number -> Boolean
; Determina si un número es distinto de 0
(define (distinto-0? n) (not (= n 0)))
```

Luego, usamos el patrón `filter` para filtrar la lista y quedarnos con aquellos elementos que cumplan `distinto-0?`.

```
; eliminar-0?: List(Number) -> List(Number)
; Elimina de una lista numérica todas las ocurrencias de 0
```

```
(define (eliminar-0 l) (filter distinto-0? l))
```

Listo, fue sencillo.

¿Qué ocurre si ahora necesitamos una función que elimine de una lista todas las ocurrencias del valor numérico 1? De forma análoga a lo que hicimos recién, podemos definir un predicado `distinto-1?` y luego una función `eliminar-1` que la llame mediante el patrón `filter`. De modo que si necesitamos filtrar por ambos valores a la vez, tendremos en nuestro programa dos funciones que repiten la mayor parte de su código. Y lo mismo aplica para cualquier valor particular que se quiera eliminar.

Este tipo de enfoque es extremadamente ineficiente y una mala práctica de programación en general. Siempre buscamos evitar la repetición de código. Además, sería imposible contemplar todas las posibles funciones porque hay infinitos números.

Una alternativa consiste en definir una constante que contenga el número que se desea eliminar de la lista.

```
; M: Number
(define M 1) ; Número a eliminar de la lista

; distinto-M?: Number -> Boolean
; Determina si un número dado es distinto a la constante M
(define (distinto-M? n) (not (= n M)))

; eliminar-M: List(Number) -> List(Number)
; Elimina de una lista numérica todas las ocurrencias de la constante M
(define (eliminar-M l) (filter distinto-M? l))
```

Ahora, basta con cambiar el valor de la constante para alterar el comportamiento del programa.

Aunque este enfoque es mejor que el anterior, todavía tiene algunos problemas. Por un lado, todavía es necesario tocar el código del programa antes de ejecutarlo. Por el otro, en cada ejecución la constante asume un único valor, luego no podemos al mismo tiempo eliminar las ocurrencias del 0 y del 1.

Todos estos inconvenientes se solucionarían pudiendo definir una función `eliminar`, que dada una lista numérica y un número `m`, devuelva la lista luego de eliminar todas las ocurrencias de `m`.

Ejemplos:

```
(eliminar (list 0 0 1 -5 0 2.5 0) 0) = (list 1 -5 2.5)
(eliminar (list 0 0 1 -5 0 2.5 0) 1) = (list 0 0 -5 0 2.5 0)
```

¿Cómo puede definirse `eliminar` mediante el patrón `filter`? Dado que `eliminar-0?` se define a partir de `distinto-0?`, es intuitivo pensar que también tenemos que adaptar esta última función. Es decir, empecemos por definir la función `distinto?` que determine si dos números dados son distintos.

```
; distinto?: Number -> Number -> Boolean
; Determina si dos números son distintos
```

```
(define (distinto? m n) (not (= m n)))
```

Luego, intentemos usar el patrón `filter`.

```
; eliminar?: List(Number) -> Number -> List(Number)
; Elimina de una lista numérica todas las ocurrencias de un número dado
(define (eliminar l m) (filter (distinto? m) l))
```

Pruebe estas definiciones y vea qué ocurre.

Al evaluar la expresión `(eliminar (list 0 0 1 -5 0 2.5 0) 0)`, Racket devuelve el siguiente error:

`"distinto?: expects 2 arguments, but found only 1"`.

Es decir, Racket no permite aplicar "parcialmente" una función. Sino que busca dos argumentos para `distinto?`, y al encontrar únicamente uno, lanza un error de sintaxis.

Nos gustaría que Racket entienda la expresión `(distinto? m)` y la interprete como una función, que dado un número `n`, determine si `m` y `n` son distintos. Esto se conoce como **currificación**.

Intentemos definir esta versión currificada de `distinto?`. Proponemos una función `distinto-m?` que, dado un número `n`, determine si `m` y `n` son distintos.

```
; distinto-m?: Number -> Boolean
; Determina si un número n es distinto a m
(define (distinto-m? n) (not (= m n)))
```

Pruebe esta definición y vea qué sucede.

Al correr el programa, Racket devuelve el siguiente error:

`"m: this variable is not defined"`.

Es decir, la definición de la función usa una variable `m` que no es un argumento de la función y tampoco es el nombre de una función o constante definida anteriormente.

Estamos en efecto ante una limitación en la sitaxis de Racket, que no nos permite escribir lo que buscamos. Afortunadamente, Racket provee algunos mecanismos adicionales para arreglarlo. A continuación, estudiaremos el concepto de **definiciones locales**.

Primeramente, revise que DrRacket esté usando el sublenguaje de estudiante intermedio. Esto habilita nuevas expresiones con la siguiente sintaxis:

`(local (definiciones ...) expr)`

y permiten definir localmente constantes y funciones, las cuales pueden ser llamadas en la expresión `expr`. Por ejemplo:

```
(local (; M: Number
        (define M 0)
        ; distinto-M?: Number -> Boolean
        ; Determina si un número es distinto a la constante M
        (define (distinto-M? n) (not (= M n))))
```

```
   (distinto-M? 3))
```

Para evaluar esta expresión local, Racket hace los siguientes pasos.

```
 (local ((define M 0)
         (define (distinto-M? n) (not (= M n))))
   (distinto-M? 3))
```

```
== Evaluación de local

(distinto-M? 3)

== Definición local de distinto-M?

(not (= M 3))

== Definición local de M

(not (= 0 3))

== Evaluación de =

(not #false)

== Evaluación de not

#true
```

Estas definiciones son locales, en el sentido de que su alcance se limita únicamente a la expresión local. Es decir, llamar a estas definiciones (`M` o `distinto-M?`) fuera de la expresión local genera un error de sintaxis (salvo que también estén definidas afuera).

Así usadas, las definiciones locales no son demasiado interesantes. Salvo por el hecho de que permiten limitar el alcance de las definiciones al lugar donde se necesitan, se obtiene un programa "equivalente" definiéndolas globalmente (como lo veníamos haciendo).

Sin embargo, el verdadero potencial de las definiciones locales aparece cuando se usan en la definición de una función. En particular, se pueden aprovechar para simular funciones currificadas, debido a que pueden acceder a los argumentos de las mismas.

Volvamos a la definición de la función `eliminar`. Esta vez, vamos a definir localmente a la función `distinto-m?`.

```
 (define (eliminar l m)
   (local (; distinto-m?: Number -> Boolean
           ; Determina si un número n es distinto a m
           (define (distinto-m? n) (not (= m n))))
     (filter distinto-m? l)))
```

Pruebe esta definición y vea qué ocurre.

Al evaluar la expresión `(eliminar (list 0 0 1 -5 0 2.5 0) 0)`, Racket devuelve exactamente el resultado esperado. ¿Qué está sucediendo realmente? Hagamos algunos pasos de evaluación.

```
(eliminar (list 0 0 1 -5 0 2.5 0) 0)
```

```
== Definición de la función eliminar

  (local ((define (distinto-m? n) (not (= 0 n))))
    (filter distinto-m? (list 0 0 1 -5 0 2.5 0)))

== Evaluación de local

(filter distinto-m? (list 0 0 1 -5 0 2.5 0))
```

Cuando Racket evalúa `local`, primero define localmente la función `distinto-m?` y luego evalúa la expresión local. Pero es importante notar que `distinto-m?` ya quedó definida para el valor de `m` tomado por argumento por la función `eliminar`. Por supuesto, la evaluación paso a paso prosigue de la forma habitual.

En conclusión, mediante definiciones locales es posible extender la variedad de funciones que se pueden definir por medio de patrones. En particular, es posible aplicar patrones para definir funciones que clasifiquen, transformen u operen los elementos de una lista en base a un valor dado de entrada.

A continuación, se proponen algunos ejercicios para poner en práctica estos conceptos. Tenga en cuenta los siguientes criterios al resolverlos.

- **Las definiciones locales no reemplazan las buenas prácticas que aprendimos hasta ahora.** En particular, aquellas funciones que no necesiten definiciones locales, las escribiremos sin ellas. Y aquellas funciones que no necesiten ser escritas localmente, las escribiremos globalmente (como veníamos haciendo).
- **Las definiciones locales también deben ser diseñadas**. Lo único que no incluiremos son los ejemplos (o check-expect) ya que no es posible evaluarlos localmente.

Ejercicio 18. Diseñe una función `mayores`, que dada una lista de números y un número `m`, devuelve la lista de todos los números mayores a `m`.

Ejemplos:

```
(mayores (list -5 0 8 5 6 2) 5) = (list 8 6)
```

Ejercicio 19. Diseñe una función `largas`, que dada una lista de strings y un número `m`, devuelve la lista de todas las strings de largo mayor a `m`.

Ejemplos:

```
(largas (list "Hola" "estudiantes" "de" "LCC" "Rosario") 4) =
    (list "estudiantes" "Rosario")
```

Ejercicio 20. Diseñe una función `lejos`, que dada una lista de puntos en el plano (representados mediante estructuras `posn`) y un número `m`, devuelve la lista de puntos que están a distancia mayor a `m` del origen.

Ejemplos:

```
(lejos (list (make-posn 3 5) (make-posn 1 2) (make-posn 0 1) (make-posn 5 6)) 4)
= (list (make-posn 3 5) (make-posn 5 6))
```

Ejercicio 21. Diseñe una función `sumar`, que dada una lista de números y un número `m`, devuelve la lista que resulta de sumar `m` a cada uno de los elementos de la lista.

Ejemplos:

```
(sumar (list 5 3 -4) 10) = (list 15 13 6)
```

Ejercicio 22. Diseñe una función `elevar`, que dada una lista de números y un número `m`, devuelve la lista que resulta de elevar a la `m` a cada uno de los elementos de la lista.

Ejemplos:

```
(elevar (list 3 0 2 1) 3) = (list 27 0 8 1)
```

---

5 Más ejercicios

Los problemas de esta sección se pueden resolver utilizando las funciones presentadas en las secciones precedentes (en varios de ellos necesitará usar más de una). Intente utlizar `map`, `fold` y `filter` para construir sus soluciones.

Veamos un ejemplo:

Ejercicio 23. Diseñe una función `sumcuad` que dada una lista de números, devuelve la suma de sus cuadrados. Para la lista vacía, devuelve `0`.

Dada una lista `l`, podemos dividir este problema en dos tareas:

- Calcular los cuadrados de todos los elementos de `l`, y
- sumar estos valores.

Diseñamos una solución para cada tarea:

```
; cuadrados : ListN -> ListN
; calcula los cuadrados de todos los elementos de una lista de números
(check-expect (cuadrados (list 1 2 3 4 5)) (list 1 4 9 16 25))
(check-expect (cuadrados empty) empty)
(check-expect (cuadrados (list 11 13 9)) (list 121 169 81))
(define (cuadrados l) (map sqr l))
```

```
; suma : ListN -> Number
; suma todos los elementos de una lista de números
(check-expect (suma (list 1 2 3 4 5)) 15)
(check-expect (suma empty) 0)
(check-expect (suma (list 11 13 9)) 33)
(define (suma l) (fold + 0 l))
```

Ahora podemos simplemente combinar ambas partes para resolver el problema:

```
; sumcuad : ListN -> Number
; suma los cuadrados de una lista de números
(check-expect (sumcuad (list 1 2 3 4 5)) 55)
(check-expect (sumcuad empty) 0)
```

```
(check-expect (sumcuad (list 11 13 9)) 371)
(define (sumcuad l) (suma (cuadrados l)))
```

La idea es entonces que la función a definir se pueda construir combinando otras más sencillas sobre listas, y que cada una de estas últimas se puedan definir usando `map`, `fold` y `filter`.

> Para `map`, `fold` y `filter` puedes usar las definiciones que vimos en esta práctica o utilizar las funciones que vienen con *DrRacket* . Si elige esta última opción, debes tener en cuenta dos cosas: 1) Debes cargar el lenguaje *Estudiante Intermedio* 2) En *DrRacket* la función incluida para `fold` se llama **`foldr`**.

Ejercicio 24. Diseñe la función `sumdist`, que dada una lista `l` de estructuras `posn`, devuelve la suma de las distancias al origen de cada elemento de `l`.

Ejemplo:

```
(sumdist (list (make-posn 3 4) (make-posn 0 3)))
= 8
```

Ejercicio 25. Diseñe una función `multPos`, que dada una lista de números `l`, multiplique entre sí los números positivos de `l`.

Ejemplo:

```
(multPos (list 3 -2 4 0 1 -5))
= 12
```

Ejercicio 26. Diseñe una función `sumAbs`, que dada una lista de números, devuelve la suma de sus valores absolutos.

Ejemplo:

```
(sumAbs (list 3 -2 4 0 1 -5))
= 15
```

Ejercicio 27. Diseñe la función `raices`, que dada una lista de números `l`, devuelve una lista con las raíces cuadradas de los números no negativos de `l`.

Ejemplo:

```
(raices (list 16 -4 9 0))
= (list 4 3 0)
```

Ejercicio 28. Diseñe la función `saa`, que dada una lista de imágenes, devuelva la suma de las áreas de aquellas imágenes `"Anchas"`.

Ejemplo:

```
(saa (list (circle 20 "solid" "red")
           (rectangle 40 20 "solid" "blue")
           (rectangle 10 20 "solid" "yellow")
           (rectangle 30 20 "solid" "green")))
= 1400
```

Ejercicio 29. Diseñe la función `algun-pos`, que toma una lista de listas de números y devuelve `#true` si y sólo si para alguna lista la suma de sus elementos es positiva.

Ejemplos:

```
(algun-pos (list (list 1 3 -4 -2) (list 1 2 3 -5) (list 4 -9 -7 8 -3)))
= #true
(algun-pos (list empty (list 1 2 3)))
= #true
(algun-pos (list (list -1 2 -3 4 -5) empty (list -3 -4)))
= #false
```

Ejercicio 30. Diseñe la función `long-lists`, que toma una lista de listas y devuelve `#true` si y sólo si las longitudes de todas las sublistas son mayores a 4.

Ejemplos:

```
(long-lists (list (list 1 2 3 4 5) (list 1 2 3 4 5 6) (list 87 73 78 83 33)))
= #true
(long-lists (list '() '() (list 1 2 3)))
= #false
(long-lists (list (list 1 2 3 4 5) empty))
= #false
```

Ejercicio 31. Diseñe una función `todos-true` que toma una lista de valores de cualquier tipo, y devuelve `#true` si y sólo si todos los valores booleanos de la lista son verdaderos. Caso contrario, devuelve `#false`.

Ejemplos:

```
(todos-true (list 5 #true "abc" #true "def"))
= #true
(todos-true (list 1 #true (circle 10 "solid" "red") -12 #false))
= #false
```

Presente al menos dos ejemplos más en su diseño.

Ejercicio 32. Dada la definición de la estructura `alumno`:

```
(define-struct alumno [nombre nota faltas])
; alumno  (String, Number, Natural). Interpretación
; - nombre representa el nombre del alumno.
; - nota representa la calificación obtenida por el alumno (entre 0 y 10).
; - faltas: número de clases a las el alumno no asistió.
```

Diseñe las siguientes funciones:

- `destacados`, que dada una lista de alumnos, devuelve una lista con el nombre de aquellos alumnos que sacaron una nota mayor o igual a 9.

  Ejemplo:

  ```
  (destacados (list (make-alumno "Ada Lovelace" 10 20)
                    (make-alumno "Carlos Software" 3.5 12)))
  ```

```
  = (list "Ada Lovelace")
```

- `condicion`, que dado un alumno, determine su condición de acuerdo a las siguientes reglas:
  - si la nota es mayor o igual a 8, su condición es `"promovido"`.
  - Si la nota es menor a 6, su condición es `"libre"`.
  - En cualquier otro caso, la condición es `"regular"`.
- `exito`, que dada una lista de alumnos, devuelve `#true` si ninguno está libre. Caso contrario, devuelve `#false`.

  Ejemplo:

```
(exito (list (make-alumno "Juan Computación" 5 13)
             (make-alumno "Carlos Software" 3.5 12)
             (make-alumno "Ada Lovelace" 10 20)))
= #false
```

- `faltas-regulares`, que dada una lista de alumnos, devuelve la suma de las ausencias de los alumnos regulares.

  Ejemplo:

```
(faltas-regulares (list (make-alumno "Juan Computación" 7 2)
                        (make-cliente "Carlos Software" 3.5 4)
                        (make-alumno "Ada Lovelace" 10 1)))

= 2
```

- `promovidos-ausentes`, que dada una lista de alumnos, devuelve una lista con el nombre de aquellos alumnos promovidos que no asistieron a tres o más clases.

  Ejemplo:

```
(promovidos-ausentes (list (make-alumno "Juan Computación" 9 3)
                        (make-cliente "Carlos Software" 3.5 2)
                        (make-alumno "Ada Lovelace" 10 1)))
= (list "Juan Computación")
```