v.7.6

Programación 1 - Práctica 5, segunda parte.

1 Clasificando los elementos de una lista

La sección 2 de 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 12, 13, 15 y 16 de la Práctica 5, primera parte utilizando `filter`.

2 Aplicando una transformación a los elementos de una lista

En la sección 3 de 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 2. Resuelva los ejercicios de la sección 3 de la Práctica 5, primera parte utilizando `map`.

---

3 Operando los elementos de una lista

Finalmente, algunos ejercicios de la sección 4 de la Práctica 5, primera parte buscaban obtener un valor como resultado de realizar una opecion que involucra a todos los elemntos 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))))
```

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 3. Utilizando `fold`, rehaga los siguientes ejercicios de la Práctica 5, primera parte:

- ejercicio 9
- ejercicio 10
- ejercicio 25
- ejercicio 26
- ejercicio 27

Ejercicio 4. Considere las siguientes definiciones:

```
(define CIRCULOS (list (circle 10 "solid" "blue")
                       (circle 20 "solid" "yellow")
                       (circle 30 "solid" "blue")
                       (circle 40 "solid" "yellow")
                       (circle 50 "solid" "blue")
                       (circle 60 "solid" "yellow")))

(define FONDO (empty-scene 200 200))
```

Evalúe la expresión `(fold overlay FONDO CIRCULOS)`.

4 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 5. 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 6. 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 7. 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 8. 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 9. 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 10. Diseñe la función `sag`, que dada una lista de imágenes, devuelva la suma de las áreas de aquellas imágenes `"Gordas"`.

> Vea la Práctica 1 para recordar cuándo decíamos que una imagen era "Gorda".

Ejemplo:

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

Ejercicio 11. 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 12. 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 13. 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 14. 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")
```