v.8.8

Programación 1 - Práctica 5, Primera Parte

1 Listas

1.1 Introducción

Es habitual que las personas usen listas en un montón de tareas diferentes. Por ejemplo, solemos hacer una lista de lo que necesitamos comprar en el supermercado, escribimos una lista con las tareas del día o armamos una lista con las personas que invitaremos a una fiesta. Como vimos en la teoría, *racket* nos brinda una serie de expresiones para definir y manipular listas de cualquier tipo de valores. Por ejemplo, una lista vacia viene dada por la expresión:

```
'()
```

Como `#true` o `5`, `'()` es un valor. En este caso un valor del tipo lista.

Cuando agregamos un elemento a una lista, estamos construyendo otra lista. Para ello, usamos el constructor `cons`. Por ejemplo,

```
(cons "Juan" '())
```

construye una lista de un elemento a partir de la lista vacía `'()` y la cadena `"Juan"`.

Una vez que contamos con una lista de un elemento, podemos construir listas de dos elementos usando nuevamente el constructor `cons`. Un ejemplo sería:

```
(cons "Pedro" (cons "Juan" '()))
```

Otro ejemplo posible:

```
(cons "María" (cons "Juan" '()))
```

A su vez, podemos construir una lista de tres elementos:

```
(cons "Jorge" (cons "María" (cons "Juan" '())))
```

También podemos construir listas de números. Un ejemplo de una lista que contiene los 10 dígitos decimales podría ser la siguiente:

```
(cons 0
  (cons 1
    (cons 2
      (cons 3
        (cons 4
          (cons 5
            (cons 6
              (cons 7
                (cons 8
```

```
                    (cons 9 '()))))))))))
```

La construcción de esta lista se obtuvo de aplicar 10 veces el constructor `cons` y la lista vacía `'()`.

En ciencias de la computación, las definiciones autorreferenciadas juegan un rol fundamental. Veamos, por ejemplo, cómo podemos definir una lista de nombres de personas presentes en una agenda:

```
Contactos es:
- una lista vacia '() o
- una expresión del tipo (cons un-nombre-persona Contactos)
```

> Segun esta definición, `Contactos` puede ser una lista vacía o una lista de contactos encabezada por el String *un-nombre-persona*.

Ejercicio 1. Cree una lista de Contactos con cinco nombres.

Ejercicio 2. Explique por qué la siguiente lista

```
(cons "1" (cons "2" '()))
```

es un ejemplo de lista de Contactos y por qué `(cons 2 '())` no lo es.

Miremos nuevamente la definición dada y prestemos atención a los ejemplos dados hasta ahora. Podrá notar que todos caen dentro de dos categorías: o bien son listas vacías `'()` o bien usan el constructor `cons` para agregar algo a una lista ya existente. El truco está en decir qué tipo de "listas existentes" queremos permitir. Una de las ventajas de las definiciones de datos autorreferenciadas es que permiten definir datos sin imponer restricciones de tamaño.

Ejercicio 3. Diseñe las listas de valores booleanos. Este tipo de dato debe permitir listas de cualquier longitud.

1.2 Operaciones sobre listas

Una vez que entendimos qué son las listas, debemos poder operar sobre ellas.

DrRacket provee un conjunto de operadores que podemos usar para manipular listas:

Operador	Tipo de Operador	Función
`'()`	Constructor	Representa la lista vacía
`empty?`	Predicado	Reconoce únicamente la lista vacía
`cons`	Constructor	Agrega un elemento a una lista
`first`	Selector	Devuelve el primer elemento de la lista
`rest`	Selector	Devuelve la lista sin su primer elemento
`cons?`	Predicado	Reconoce listas no vacías

Estos operadores son suficientes para definir cualquier función que opere sobre listas. Sin embargo, y para simplificar la escritura de nuestros programas, *racket* nos provee una forma más sencilla de escribir listas.

Para denotar la lista vacía, escribiremos `empty`, mientras que para definir una lista con varios elementos, utlizaremos el operador `list`. Cuando querramos definir la lista

```
(cons a0 (cons a1 (.... (cons an empty))))
```

podremos escribir

```
(list a0 a1 ... an).
```

Por ejemplo, la lista que contiene los 10 dígitos decimales puede reescribirse como sigue:

```
(list 0 1 2 3 4 5 6 7 8 9).
```

Observación 1: Es importante notar que `list` no es un constructor, sino un operador que nos provee el lenguaje para simplificar la escritura de las listas en nuestros programas.

Observación 2: Para utilizar estas abreviaturas de listas en *DrRacket*, es necesario cargar el lenguaje: *Estudiante Principiante con Abreviaturas de Listas*.

1.3 Programando con listas

Supongamos que por algún motivo Ud. mantiene una lista con los nombres de todos sus amigos. Pensemos que Ud. es alguien muy popular y que esa lista crece tanto que necesita de un programa que le permita buscar un nombre en la misma. Para transformar esta idea en un ejemplo concreto, pensemos en el siguiente problema:

Ud. se encuentra trabajando sobre la lista de contactos de un nuevo teléfono. El dueño del teléfono puede agregar y borrar nombres y consultar esta lista para cualquier nombre. Pensemos que a Ud. se la ha asignado la tarea de diseñar la función que toma la lista de contactos y determina si contiene el nombre "Marcos"

Veremos que siguiendo la receta de diseño vista en la **Práctica 2**, podemos obtener una solución al problema planteado.

1. **Diseño de datos**

   El tipo de datos *Contactos* definido anteriormente es apropiado para representar una lista de nombres que usará la función que queremos definir `contiene-Marcos?`.

   O sea que con esto completamos el primer paso de la receta.

2. **Signatura y declaración de propósito**

   Debemos proponer una signatura adecuada para la función `contiene-Marcos?` y brindar una declaración de propósito.

   Lo hacemos de la siguiente manera:

   ```
   ; contiene-Marcos? : Contactos -> Booleano
   ; dada una lista de Contactos, determina si "Marcos" es un elemento de la misma
   ```

3. **Ejemplos**

Para completar el 3er paso debemos proponer ejemplos ilustrativos que sirvan para validar si la función `contiene-Marcos?` cumple con su propósito.

Algunos ejemplos fáciles de calcular son:

```
(check-expect (contiene-Marcos? '()) #false)
(check-expect (contiene-Marcos? (cons "Sara" (cons "Pedro"  (cons "Esteban" '())))) #false)
(check-expect (contiene-Marcos? (cons "A" (cons "Marcos" (cons "C" '())))) #true)
(check-expect (contiene-Marcos? (cons "Juan" '())) #false)
(check-expect (contiene-Marcos? (cons "Marcos" '())) #true)
```

4. **Definición de la función**

En este paso, la idea es diseñar una función que se corresponda con el tipo de datos que va a usar. Dado que la definición de *Contactos* tiene dos cláusulas, el cuerpo de la función debe contener una expresión `cond` con dos condiciones que determinan cuál de los dos tipos posibles de listas recibió la función. Estas condiciones son `(empty? l)` y `(cons? l)`.

Un primer esbozo de la función sería el siguiente:

```
(define (contiene-Marcos? l) (cond [(empty? l) ...]
                                   [(cons? l) ...]))
```

> **Observación:** Podemos reemplazar `(cons? l)` con `else`, pues nuestra función recibe sólo listas como dato de entrada.

Si la primera condición es verdadera, entonces `l` es la lista vacía `'()`. En este caso no quedan dudas que la función `contiene-Marcos?` deberá devolver `#false`. Podemos ir completando la definición de la siguiente manera:

```
(define (contiene-Marcos? l) (cond [(empty? l) #false]
                                   [(cons? l) ...]))
```

En caso de que la segunda condición sea la verdadera, entonces `l` es una lista con al menos un elemento. Gracias a los operadores `first` y `rest` podemos separar el primer elemento de la lista y el resto de la misma.

Podemos verificar entonces si el primer elemento de la lista coincide con el nombre "*Marcos*" usando el siguiente código:

```
(string=? (first l) "Marcos")
```

Si esta comparación resulta verdadera, entonces sabemos que la función `contiene-Marcos?` debe devolver `#true`. Si resulta falsa, será necesario buscar el nombre "*Marcos*" en el resto de la lista `(rest l)`.

Pero, ¿cómo hacemos para buscar a "*Marcos*" en la lista `(rest l)`? Por suerte tenemos la función `contiene-Marcos?` que, según su declaración de propósito, determina si una lista contiene o no a "*Marcos*". Esto nos lleva al siguiente código:

```
(contiene-Marcos? (rest l))
```

Combinando ambos casos obtenemos el código final de la función `contiene-Marcos?`:

```
(define (contiene-Marcos? l) (cond [(empty? l) #false]
                                   [(cons? l) (if (string=? (first l) "Marcos")
                                                  #true
                                                  (contiene-Marcos? (rest l)))]))
```

5. **Evaluar el código en los ejemplos**

   Podemos correr el código en *DrRacket* y veremos que los casos de prueba funcionan correctamente. Por esto no es necesario pasar al 6to paso de la receta.

Si juntamos todo, vemos que el diseño final obtenido es el siguiente:

```
; contiene-Marcos? : Contactos -> Booleano
; dada una lista de Contactos, determina si "Marcos" es un elemento de la misma

(check-expect (contiene-Marcos? '()) #false)
(check-expect (contiene-Marcos? (cons "Sara" (cons "Pedro"  (cons "Esteban" '())))) #false)
(check-expect (contiene-Marcos? (cons "A" (cons "Marcos" (cons "C" '())))) #true)
(check-expect (contiene-Marcos? (cons "Juan" '())) #false)
(check-expect (contiene-Marcos? (cons "Marcos" '())) #true)

(define (contiene-Marcos? l) (cond [(empty? l) #false]
                                   [(cons? l) (if (string=? (first l) "Marcos")
                                                  #true
                                                  (contiene-Marcos? (rest l)))]))
```

Ejercicio 4. Use *DrRacket* para evaluar la función `contiene-Marcos?` usando la siguiente lista como entrada:

```
(cons "Eugenia"
  (cons "Lucía"
    (cons "Dante"
      (cons "Federico"
        (cons "Marcos"
          (cons "Gabina"
            (cons "Laura"
              (cons "Pamela" '()))))))))
```

Ejercicio 5. Diseñe la función `contiene?` que determine si un string aparece en una lista de string.

DrRacket provee una función `member?` que toma un valor *x*y una lista *l* y chequea si el valor *x* está en la lista *l*. Por ejemplo:

```
> (member? "Laura" (cons "a" (cons "b" (cons "Laura" '()))))
#true
```

> No use `member?` para definir la función `contiene?`.

Ejercicio 6. Realice la evaluación *paso a paso* usando *DrRacket* para la siguiente expresión:

```
(contiene-Marcos? (cons "Marcos" (cons "C" '())))
```

Use también el evaluador *paso a paso* para esta otra expresión:

```
(contiene-Marcos? (cons "A" (cons "Marcos" (cons "C" '()))))
```

¿Qué pasa si reemplazamos a "*Marcos*" con "*B*"?

Ejercicio 7. Aquí vemos un tipo de dato que nos permite representar listas de montos de dinero:

```
Una Lista-de-montos es:
- '()
- (cons NumeroPositivo Lista-de-montos)
Lista-de-montos representa una lista con montos de dinero
```

Cree algunos ejemplos que pertenezcan a este tipo de datos para asegurarse de entender bien la definición. Una vez hecho esto, diseñe la función `suma` que tome como entrada una lista con montos de dinero y devuelva como resultado la suma de los montos presentes en dicha lista. Use el evaluador *paso a paso* de *DrRacket* para ver cómo se evalúa `(suma l)` para una lista no muy larga de montos `l`.

Ejercicio 8. Ahora consideremos la siguiente definición:

```
Una Lista-de-numeros es:
- '()
- (cons Numero Lista-de-numeros)
```

Algunos elementos de este tipo de datos son apropiados para la función `suma` del ejercicio anterior y otros no.

Diseñe la función `pos?` que tome una Lista-de-numeros y determine si todos los elementos de la lista son positivos. En otras palabras, si `(pos? l)` devuelve `#true`, entonces `l` es un elemento del tipo de dato Lista-de-montos.

Use el evaluador *paso a paso* de *DrRacket* para entender cómo se evalúan las siguientes expresiones: `(pos? (cons 5 '()))` y `(pos? (cons -1 '()))`.

Por último, diseñe la función `checked-suma`. Esta función recibe como entrada una Lista-de-numeros y devuelve la suma de sus elementos si la lista pertenece a Lista-de-montos; sino deberá devolver un string indicando un error.

Ejercicio 9. Diseñe:

- `todos-verdaderos`, una función que recibe como entrada una lista de valores booleanos y devuelve `#true` unicamente si todos los elementos de la lista son `#true`.
- `uno-verdadero`, una función que recibe como entrada una lista de valores booleanos y devuelve `#true` si al menos uno de los elementos de la lista es `#true`.

Ejercicio 10. Diseñe la función `cant-elementos` que dada una lista, devuelve la cantidad de elementos que contiene.

Ejercicio 11. Diseñe la función `promedio`, que devuelve el promedio de una lista de números. **Sugerencia:** No reinvente la rueda, ¡use los ejercicios anteriores!

2 Clasificando elementos de una lista

Ejercicio 12. Diseñe la función `pares`, que dada una lista de números *l*, devuelve una lista con los números pares de *l*.

Ejemplo:

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

Ejercicio 13. 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 14. Diseñe la función `mayores`, que dada una lista de números `l` y un número `n`, devuelve una lista con aquellos elementos de `l` que son mayores a `n`.

Ejercicio 15. 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` de origen, 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 16. Diseñe una función llamada `positivos` que tome una lista de números y se quede sólo con aquellos que son mayores a 0.

Ejemplo:

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

Ejercicio 17. Diseñe la función `eliminar`, que dada una lista de números *l* y un numero *n*, devuelve la lista que resulta de eliminar en *l* todas las ocurrencias de *n*.

Ejemplos:

```
(eliminar (list 1 2 3 2 7 6) 2)
= (list 1 3 7 6)

  (eliminar (list 1 2 3 2 7 6) 0)
= (list 1 2 3 2 7 6)
```

3 Aplicando transformaciones a cada elemento de una lista

Ejercicio 18. Diseñe la función `raices`, que dada una lista de números, devuelve una lista con las raíces cuadradas de sus elementos.

Ejemplo:

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

Ejercicio 19. Diseñe la función `distancias`, que dada una lista de puntos del plano, 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 20. Diseñe la función `anchos`, que dada una lista de imágenes, 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 21. 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, segunda parte.

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

Ejercicio 22. Diseñe la función `cuadrados`, que dada una lista de números, devuelva la lista que resulta de elevar al cuadrado cada uno de los elementos de la lista original.

Ejemplo:

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

Ejercicio 23. Diseñe la función `longitudes`, que dada una lista de cadenas, devuelve la lista de sus longitudes (es decir, la cantidad de caracteres que contienen).

Ejemplo:

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

Ejercicio 24. Diseñe la función `convertirFC`, que dada una lista de temperaturas en grados Fahrenheit, devuelve esta lista de temperaturas convertidas en grados Celsius.

4 Operando con los elementos de una lista

Ejercicio 25. Diseñe la función `prod`, que multiplica los elementos de una lista de números entre sí. Para la lista vacía, devuelve `1`.

Ejemplo:

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

Ejercicio 26. Diseñe la función `pegar`, que dada una lista de strings, devuelve el string que se obtiene de concatenar todos los elementos de la lista.

Ejemplo:

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

Ejercicio 27. Diseñe la función `maximo` que devuelve el máximo de una lista de números naturales. Para la lista vacía, devuelve `0`.

Ejemplo:

```
(maximo (list 23 543 325 0 75))
= 543
```

Ejercicio 28. Diseñe la función `sumdist`, que dada una lista de puntos del plano, devuelva la suma de sus distancias al origen.

Ejemplo:

```
(sumdist (list (make-posn 3 4) (make-posn 0 4) (make-posn 12 5)))
= 22
```

Ejercicio 29. Diseñe la función `sumcuad`, que dada una lista de números, devuelve la suma de sus cuadrados. Para la lista vacía, devuelve `0`.

Ejemplo:

```
(sumcuad (list 1 2 3))
= 14
```

5 Más estados

En esta sección estudiaremos programas interactivos cuyos estados incluyen listas. Como hemos mencionado anteriormente, las listas permiten definir datos sin imponer restricciones de tamaño. En consecuencia, pueden capturar estados mucho más complejos que los vistos hasta ahora. Esto nos permite incorporar una gran diversidad de nuevas funcionalidades a nuestros programas. Por ejemplo, pueden usarse para llevar un registro de todos los eventos capturados desde el inicio de programa y el orden en que ocurrieron. A continuación veremos algunas aplicaciones.

Ejercicio 30. En el ejercicio 8 de la Práctica 3 diseñamos un programa que dibuja estrellas sobre un cielo vacío. En particular, habíamos elegido como estado a una imagen. Imaginemos ahora que queremos agregar la siguiente funcionalidad a nuestro programa:

- Cada vez que se presiona la tecla `"backspace"` se debe borrar la estrella más reciente. Si no había estrellas en el cielo, entonces no hace nada.

Observemos que la imagen no permite identificar por sí misma cuál es la estrellas más reciente, por lo tanto debemos pensar en otro estado. Necesitamos que el estado lleve un registro de todas las coordenadas donde hay estrellas, convenientemente ordenadas de la más

reciente a la más antigua (ya veremos por qué). Las listas son ideales para esta tarea, por lo tanto podemos considerar el siguiente diseño de datos:

```
; Un estado es una List(posn)
; que representa las coordenadas donde hay estrellas,
; ordenadas de la más reciente a la más antigua.
```

Modifique su programa para manejar la nueva representación para un estado. Tenga en cuenta los siguientes puntos.

- El estado inicial es `empty`.
- La función `interpretar` toma una lista de coordenadas y debe devolver la imagen que resuelta de dibujar sobre el cielo vacío una estrella en cada coordenada de la lista, con su respectivo color y tamaño.
- Cada vez que se hace click sobre la imagen, el manejador del mouse debe agregar la coordenada del click al inicio de la lista.
- El manejador del teclado debe volver al estado inicial si se presiona la barra espaciadora.
- Por último, modifique el manejador del teclado para que, cada vez que se presione la tecla `"backspace"`, se borre el elemento inicial de la lista siempre que sea posible. De lo contrario, el estado no se debe modificar.
- Piense qué cambios serían necesarios en el manejador del mouse y del teclado si, por el contrario, las coordenadas se guardasen en la lista de la más antigua a la más reciente, ¿cuál implementación le parece más sencilla?

Si prestamos atención, lo único que nuestro programa necesita de la lista es saber si está vacía, agregar un nuevo elemento y sacar el elemento más reciente. Esta abstracción es de hecho tan habitual en ciencias de la computación que tiene su propio nombre: "pila". Una de las maneras más habitual de implementar pilas es mediante listas, como lo hicimos nosotros. Más adelante en la carrera estudiarán pilas con mayor profundidad.

Ejercicio `31`. En este juego dos jugadores "1" y "2" se enfrentan para saber quién de ellos tiene mejor memoria. Un turno de juego se desarrolla de la siguiente forma:

- El jugador "1" comienza tecleado alguna de las flechas, por ejemplo `"up"`. Suponemos que siempre se anuncia en voz alta la flecha tecleada para dar aviso al otro jugador.
- El jugador "2" debe repetir la misma flecha que el jugador "1" y teclear una nueva, por ejemplo `"up"`,`"right"`.
- El jugador "1" debe repetir la secuencia del jugador "2" y teclear una nueva flecha, por ejemplo `"up"`,`"right"`,`"left"`.
- De esta forma se van desarrollando los turnos hasta que alguno de los jugadores se equivoca y pierde el juego. Por ejemplo, si a continuación el jugador "2" teclea `"up"`,`"right"`,`"right"`,`"right"`, entonces pierde pues la tercer tecla debió ser `"left"` en lugar de `"right"`.

Diseñe un programa interactivo para este juego. Tener en cuenta los siguientes puntos.

- Puede considerar que un estado es una estructura con dos listas, tal que la primera almacena la secuencia de teclas ingresada por el jugador anterior y la segunda la secuencia de teclas ingresada por el jugador actual.

```
(define-struct Turno [ant act])
; Turno es (List(String), List(String)), donde
; ant es la lista de teclas ingresada por el jugador anterior,
; act es la lista de teclas ingresada por el jugador actual.

; El estado del programa es un Turno,
; inicialmente ambas listas son vacías.
(define ESTADO-INICIAL (make-Turno empty empty))
```

- La función `interpretar` debe escribir sobre una escena vacía el texto "Turno: Jugador 1" en color rojo o "Turno: Jugador 2" en color azul, dependiendo de a quién le toque jugar. Observe que es posible deducir de quién es el turno en base a la paridad del largo de la lista `ant`: le toca al jugador "1" si es par y al "2" en caso contrario.

- El manejador del teclado debe capturar las flechas tecleadas y agregarlas a la lista `act`, hasta que el turno termina. Las demás teclas deben ignorarse.

  Volviendo al ejemplo mencionado más arriba, cuando el jugador "1" teclea su primer flecha, se debe pasar del estado inicial a un estado igual a: `(make-Turno empty (list "up"))`.

- Cuando termina el turno, ambas listas deben ser comparadas para determinar si el juego continua o termina.

- Si el juego continua, entonces el nuevo estado debe tener como primer lista a `act` y como segunda lista a `empty`.

  Volviendo al ejemplo mencionado más arriba, cuando el jugador "1" termina su primer turno, se debe pasar a un estado igual a: `(make-Turno (list "up") empty)`.

- Si el juego termina, entonces el programa también termina y debe escribir sobre una escena vacía el texto "Ganador: Jugador 1" en color rojo o "Ganador: Jugador 2" en color azul, dependiendo de quién haya ganado. Puede usar un estado especial, por ejemplo el valor 0, para representar un juego terminado y aprovechar la cláusula `stop-when` de `big-bang` para llamar a una función que muestre la pantalla final.