003. Determinar si un elemento pertenece a una lista dada.

Buscamos definir una función **`pertenece?`** que toma dos argumentos y devuelve un booleano. Los argumentos son, respectivamente, un elemento y una lista de elementos. Cuando el elemento se encuentra en una o más posiciones de la lista, la función retorna verdadero, y si el elemento no se encuentra en la lista, retorna falso.

Veamos un par de ejemplos con elementos numéricos por simplicidad. Supongamos que el elemento es el primer argumento, y la lista es el segundo:

```
(pertenece? 2 '(2 3 4 5 6 7)) -> #t
(pertenece? 5 '(1 2 3 4 5 6)) -> #t
(pertenece? 5 '(1 2 3 4 6 7)) -> #f
(pertenece? 5 '()) -> #f
```

Una lista es una secuencia de elementos. Por tanto tiene un elemento distinguido que es el primero, y cada elemento tiene un sucesor, con excepción del último.

Scheme nos proporciona una forma de partir una lista en cabeza y cola, con **`car`** y **`cdr`**, respectivamente. Lo interesante es que la cola de una lista es también una lista. Esto nos permite pensar en la lista como una estructura de datos recurrente.

Podemos definir nuestra solución entonces de esta manera:

> *Un elemento dado pertenece a una lista si está en la cabeza de la lista, o pertenece a la cola de la lista.*

No hay otro lugar donde el elemento se pueda esconder.

Veamos cómo sería una invocación a la función que vamos a escribir:

```
(pertenece? 2 '(2 3 4 5 6 7)) -> #t
(pertenece? 2 '(1 5 3 4 2 6)) -> #t
(pertenece? 2 '()) -> #f
```

Tenemos claramente tres casos posibles:

a) el elemento coincide con la cabeza
b) el elemento no coincide con la cabeza pero está en la cola de la lista
c) la lista no contiene ningún elemento

Estos casos cubren todas las alternativas; nuestra función debe tener en cuenta los tres.

Si el primer elemento de la lista es el elemento buscado, la función devuelve **`#t`** y termina su ejecución. Entonces el código quedaría:

```
(define pertenece?
  (lambda (elem lst)
    (if (equal? elem (car lst))
        #t
        #f)))

(pertenece? 3 '(3 2 1 4)) -> #t
```

Con este código podemos acertar siempre que el elemento sea la cabeza de la lista. Nos falta ahora pensar qué sucede cuando no es la cabeza. Claramente descartamos la cabeza, porque no nos resuelve nada. El resto o cola de la lista se obtiene con **`(cdr lst)`**, la cual por definición es una lista también.

La función queda entonces:

```
(define pertenece?
  (lambda (elem lst)
    (if (equal? elem (car lst))
        #t
        (pertenece? elem (cdr lst)))))

(pertenece? 3 '(3 2 1 4)) -> #t
(pertenece? 3 '(1 2 3 4)) -> #t
```

Nos queda el caso en el cual la lista **`lst`** es la lista vacía. Esta situación puede darse por invocar la función con el segundo argumento con **`'()`** o puede suceder porque en cada invocación recursiva de **`pertenece?`** nos estamos quedando con la cola de **`lst`**, y así la vamos acortando hasta que eventualmente si **`elem`** no se encuentra, la lista **`lst`** será vacía, y esto va a producir un error al intentar **`(car lst)`**, que no está definido si **`lst`** es nula.

Insertamos el condicional que controla si está vacía, antes de la verificación de igualdad. Por cierto, es falso que un elemento pertenece a una lista vacía.

```
(define pertenece?
  (lambda (elem lst)
    (if (null? lst) #f
      (if (equal? elem (car lst))
          #t
          (pertenece? elem (cdr lst))))))

(pertenece? 3 '()) -> #f
(pertenece? 3 '(1 2 4 5)) -> #f
(pertenece? 3 '(3 2 1 4)) -> #t
(pertenece? 3 '(1 2 3 4)) -> #t
```

Hasta aquí llegamos con una solución completa a nuestro problema.

Una expansión de la idea: expresiones booleanas

Ahora bien, podemos intentar pensar desde la matemática el problema, en lugar de enfrentarlo desde la estructura de datos de la secuencia.

El elemento pertenece a la lista o no pertenece. ¿Cuándo estamos seguros que no pertenece? Cuando la lista está vacía.

¿Cuándo estamos seguros que pertenece a la lista? Cuando coincide con la cabeza o pertenece a la cola. Vamos a escribir eso como una fórmula booleana:

$$pertenece\ elem\ lst = \neg(null?\ lst) \wedge ((equal?\ elem\ (car\ lst)) \vee (pertenece?\ elem\ (cdr\ lst)))$$

Cuya codificación en Scheme queda:

```
(define pertenece?
  (lambda (elem lst)
    (and
      (not (null? lst))
      (or
         ((equal? elem (car lst))
         (pertenece? elem (cdr lst))))))
```

Y verificamos que los casos anteriores funcionan con esta expresión de la función:

```
(pertenece? 3 '()) -> #f
(pertenece? 3 '(1 2 4 5)) -> #f
(pertenece? 3 '(3 2 1 4)) -> #t
(pertenece? 3 '(1 2 3 4)) -> #t
```

Es una primitiva en Scheme

En Scheme[1] disponemos de las primitivas **`member`**, **`memv`** y **`memq`** que son similares a **`pertenece?`**.

`memq` usa **`eq?`**, **`memv`** usa **`eqv?`**, y **`member`** usa **`equal?`** para la comparación. Todas devuelven **`#f`** o bien la sublista que contiene en su cabeza al elemento buscado.

[1] https://schemers.org/Documents/Standards/R5RS/ R5RS is the Revised[5] Report on the Algorithmic Language Scheme