

3 | LISTAS EN LISP

Las listas fueron originalmente la principal estructura de datos en Lisp. De hecho, el nombre del lenguaje es un acrónimo de “LISt Processing”. Las implementaciones modernas de Lisp incluyen otras estructuras de datos. El desarrollo de programas en Lisp refleja esta historia. Las versiones iniciales del programa suelen hacer un uso intensivo de listas, que posteriormente se convierten a otros tipos de datos, más rápidos o especializados. Este capítulo muestra qué es lo que uno puede hacer con las listas y las usa para ejemplificar algunos conceptos generales de Lisp.

3.1 CONSES

En el capítulo anterior se introdujeron las funciones primitivas `cons/2`, `car/1` y `cdr/1` para el manejo de listas. Lo que `cons` hace es combinar dos objetos, en uno formado de dos partes llamado **cons**. Conceptualmente, un `cons` es un par de apuntadores: el primero es el `car` y el segundo el `cdr`.

Los `conses` proveen una representación conveniente para cualquier tipo de pares. Los dos apuntadores del `cons` pueden dirigirse a cualquier objeto, incluyendo otros `conses`. Es esta última propiedad, la que explotamos para definir **listas** en términos de `cons`. Una lista puede definirse como el par formado por su primer elemento y el resto de la lista. La mitad del `cons` apunta a ese primer elemento; la otra mitad al resto de la lista.

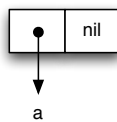


Figura 1: Una lista de un elemento, como `cons a nil`.

Cuando aplicamos `cons` a un elemento y una lista vacía, el resultado es un solo `cons` mostrado en la figura 1. Como cada `cons` representa un par de apuntadores, `car` regresa el objeto apuntado como primer componente del `cons` y `cdr` el segundo:

```
1 CL-USER> (cons 'a nil)
2 (A)
3 CL-USER> (car '(a))
4 A
5 CL-USER> (cdr '(a))
6 NIL
```

Cuando construimos una lista con múltiples componentes, el resultado es una cadena de `conses`. La figura 2 ilustra la siguiente construcción:

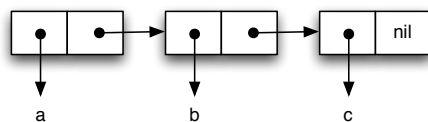


Figura 2: Una lista de varios elementos.

```

1 CL-USER> (list 'a 'b 'c)
2 (A B C)
3 CL-USER> (cdr (list 'a 'b 'c))
4 (B C)

```

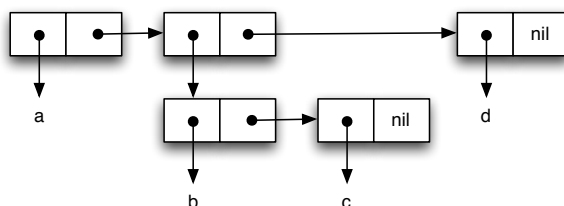


Figura 3: Una lista de varios elementos, incluida otra lista.

Las listas pueden tener elementos de cualquier tipo, incluidas otras listas, como lo ilustran las siguientes definiciones y la figura 3:

```

1 CL-USER> (list 'a (list 'b 'c) 'd)
2 (A (B C) D)

```

Las listas que no incluyen otras listas, se conocen como **listas planas**. En caso contrario, decimos que se trata de una **lista anidada**.

La función `consp/1` regresa `true` si su argumento es un `cons`, así que podemos definir `listp/1` que regresa `true` si su argumento es una lista como sigue:

```

1 CL-USER> (defun mi-listp (x)
2   (or (null x) (consp x)))
3 MI-LISTP
4 CL-USER> (defun mi-atomp (x)
5   (not (consp x)))
6 MI-ATOMP
7 CL-USER> (mi-listp '(1 2 3))
8 T
9 CL-USER> (mi-atomp '(1 2 3))
10 NIL
11 CL-USER> (mi-listp nil)
12 T
13 CL-USER> (mi-atomp nil)
14 T

```

la definición de `mi-atomp` se basa en que todo lo que no es un `cons` es un **átomo**. Observen que `nil` es lista y átomo a la vez.

3.2 CONS E IGUALDAD

Cada vez que invocamos a `cons`, Lisp reserva memoria para dos apuntadores, así que si llamamos a `cons` con el mismo argumento dos veces, Lisp regresa dos valores que aparentemente son el mismo, pero en realidad se trata de diferentes objetos:

```
1 CL-USER> (eql (cons 1 nil) (cons 1 nil))
2 NIL
```

Sería conveniente contar con una función `mi-eql` que regrese `true` cuando dos listas tienen los mismos elementos, aunque se trate de objetos distintos:

```
1 CL-USER> (defun mi-eql (lst1 lst2)
2           (or (eql lst1 lst2)
3               (and (consp lst1)
4                     (consp lst2)
5                     (mi-eql (car lst1) (car lst2))
6                     (mi-eql (cdr lst1) (cdr lst2))))))
7 MI-EQL
8 CL-USER> (mi-eql (cons 1 nil) (cons 1 nil))
9 T
```

en realidad, lisp cuenta con una función predefinida `equal/2` que cumple con nuestros objetivos. Como la definición de nuestro `eql` lo sugiere, si dos objetos son `eql`, también son `equal`.

Uno de los secretos para comprender Lisp es darse cuenta de que las variables tienen valores, en el mismo sentido en que las listas tienen elementos. Así como los `conses` tienen apuntadores a sus elementos, las variables tienen apuntadores a sus valores.

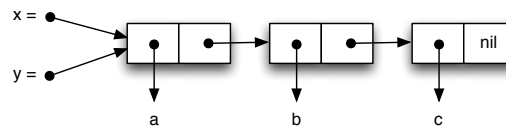


Figura 4: Una lista de varios elementos, incluida otra lista.

La diferencia entre Lisp y otros lenguajes de programación donde manipulamos los apuntadores explícitamente, es que en el primero caso, el lenguaje administra los apuntadores por uno. Ya vimos ejemplos de esto relacionados con la creación de listas. Algo similar pasa con las variables. Por ejemplo, si asignamos a dos variables la misma lista:

```
1 CL-USER> (setf x '(a b c))
2 (A B C)
3 CL-USER> (setf y x)
```

```

4 | (A B C)
5 | CL-USER> (eql x y)
6 | T

```

¿Qué sucede al hacer la segunda asignación (línea 3)? En realidad, la localidad de memoria asociada a x no contiene la lista (a, b, c) sino un apuntador a esta lista. El `setf` en cuestión copia ese mismo apuntador a la localidad de memoria asociada a y . Es decir, Lisp copia el apuntador relevante, no la lista completa. La figura 4 ilustra este caso. Por eso `eql` en la llamada de la línea 5, regresa `true`.

3.3 CONSTRUYENDO LISTAS

Ya hemos visto ejemplos de construcción de listas con `list` y `cons`. Ahora imaginen que deseamos una función `copia-lista` que tome una lista como su primer argumento y regrese una copia de ella. La lista resultante tiene los mismos elementos que la lista original, pero está contenida en cónsules diferentes:

```

1 | CL-USER> (defun copia-lista (lst)
2 |           (if (atom lst)
3 |               lst
4 |               (cons (car lst) (copia-lista (cdr lst)))))
5 |
6 | COPIA-LISTA
7 | CL-USER> (setf x '(a b c)
8 |           y (copia-lista x))
9 | (A B C)
10 | CL-USER> (eql x y)
11 | NIL

```

evidentemente, x y su copia nunca serán `eql` pero si `equal`, al menos que x sea `nil`.

Existe otro constructor de listas que toma como argumento varias listas para construir una sola:

```

1 | CL-USER> (append '(1 2) '(3) '(4))
2 | (1 2 3 4)

```

3.4 COMPRESIÓN DE DATOS *run-length*

Consideremos un ejemplo para utilizar los conceptos introducidos hasta ahora. RLE (*run-length encoding*) es un algoritmo de compresión de datos muy sencilla. Funciona como los meseros: si los comensales pidieron una tortilla española, otra, otra más y una ensalada verde; el mesero pedirá tres tortillas españolas y una ensalada verde. El código de esta estrategia es como sigue:

```

1 | (defun rle (lst)
2 |   (if (consp lst)
3 |       (compress (car lst) 1 (cdr lst))
4 |       lst))

```

```

5
6 (defun compress (elt n lst)
7   (if (null lst)
8       (list (n-elts elt n))
9       (let ((sig (car lst)))
10        (if (eql sig elt)
11            (compress elt (+ n 1) (cdr lst))
12            (cons (n-elts elt n)
13                  (compress sig 1 (cdr lst)))))))
14
15 (defun n-elts (elt n)
16   (if (> n 1)
17       (list n elt)
18       elt))

```

Una llamada a rle sería como sigue:

```

1 CL-USER> (rle '(1 1 1 0 1 0 0 0 1))
2 ((3 1) 0 1 (4 0) 1)

```

Ejercicio sugerido. Programen una función inversa a rle, esto es dado una lista que es un código rle, esta función regresa la cadena original. Observen que este método de comprensión no tiene pérdida de información. Prueben su solución con la ayuda de un generador de listas de n elementos aleatorios.

3.5 FUNCIONES BÁSICAS

A continuación definiremos una biblioteca mínima de operaciones sobre listas, que por cuestiones de eficiencia serán declaradas:

```

1 (proclaim '(inline last1 single append1 conc1 mklist))
2 (proclaim '(optimize speed))
3
4 (defun last1 (lst)
5   (car (last lst)))
6
7 (defun single (lst)
8   (and (consp lst) (not (cdr lst))))
9
10 (defun append1 (lst obj)
11   (append lst (list obj)))
12
13 (defun conc1 (lst obj)
14   (nconc lst (list obj)))
15
16 (defun mklist (obj)
17   (if (listp obj) obj (list obj)))

```

La función last1 regresa el último elemento de una lista. La función predefinida last regresa el último cons de una lista, no su último elemento. Generalmente obtenemos tal elemento usando (car(last ...)). ¿Vale la pena definir una nueva función

para una función predefinida? La respuesta es afirmativa cuando la nueva función reemplaza efectivamente a la función predefinida.

Observen que `last1` no lleva a cabo ningún chequeo de error. En general, ninguna de las funciones del curso harán chequeo de errores. En parte esto se debe a que de esta manera los ejemplos serán más claros; y en parte se debe a que no es razonable hacer chequeo de errores en utilidades tan pequeñas. Si intentamos:

```
1 | CL-USER> (last1 "prueba")
2 | value "prueba" is not of the expected type LIST.
3 | [Condition of type TYPE-ERROR]
```

el error es capturado y reportado por la función predefinida `last`. Cuando las utilidades son tan pequeñas, forman una capa de abstracción tan delgada, que comienzan por ser transparentes. Uno puede ver en `last1` para interpretar los errores que ocurren en sus funciones subyacentes.

La función `single` prueba si algo es una lista de un elemento. Los programas Lisp necesitan hacer esta prueba bastantes veces. Al principio, uno está tentado a utilizar la traducción natural del español a Lisp:

```
1 | (= (length lst) 1)
```

pero escrita de esta forma, la función sería muy ineficiente.

Las funciones `append1` y `conc1` agregan un elemento al final de una lista, `conc1` de manera destructiva. Estas funciones son pequeñas, pero se usan tantas veces que vale la pena incluirlas en la librería. De hecho, `append1` ha sido predefinida en muchos dialectos Lisp.

La función `mklist` nos asegura que su argumento sea una lista. Muchas funciones Lisp están escritas para regresar una lista o un elemento. Supongamos que `lookup` es una de estas funciones. Si queremos coleccionar el resultado de aplicar esta función a todos los miembros de una lista, podemos escribir:

```
1 | (mapcar #'(lambda (d) (mklist (lookup d)))
2 | data)
```

Veamos ahora otros ejemplos de utilidades más grandes que operan sobre listas:

```
1 | (defun longer (x y)
2 |   (labels ((compare (x y)
3 |             (and (consp x)
4 |                  (or (null y)
5 |                      (compare (cdr x) (cdr y))))))
6 |     (if (and (listp x) (listp y))
7 |         (compare x y)
8 |         (> (length x) (length y))))
9 |
10 | (defun filter (fn lst)
11 |   (let ((acc nil))
12 |     (dolist (x lst)
13 |       (let ((val (funcall fn x)))
14 |         (if val (push val acc))))
15 |     (nreverse acc)))
```

```

16
17 (defun group (source n)
18   (if (zerop n) (error "zero length"))
19   (labels ((rec (source acc)
20             (let ((rest (nthcdr n source)))
21                 (if (consp rest)
22                     (rec rest (cons (subseq source 0 n) acc))
23                     (nreverse (cons source acc))))))
24     (if source (rec source nil) nil)))

```

Al comparar la longitud de dos lista, lo más inmediato es usar (`>(length x) (length y)`), pero esto es ineficiente. En particular si una de las listas es mucho más corta que la otra. Lo mejor, es usar `longer` y recorrerlas en paralelo con la función local `compare`, en caso de que `x` e `y` sean listas. Si este no es el caso, por ejemplo, si los argumentos de `longer` son cadenas de texto, solo entonces usaremos `length`.

La función `filter` aplica su primer argumento `fn` a cada elemento de la lista `lst` guardando aquellos resultados diferentes a `nil`.

```

1 CL-USER> (filter #'null '(nil t nil t 5 6))
2 (T T)
3 CL-USER> (filter #'(lambda (x)
4                 (when (> x 0) x))
5               '(-1 2 -3 4 -5 6))
6 (2 4 6)
7 > (filter #'(lambda (x)
8           (if (numberp x) (1+ x)))
9         '(a 1 2 b 3 c d 4))
10 (2 3 4 5)

```

Observen el uso del acumulador `acc` en la definición de `filter`. La combinación de `push` y `nreverse` es la forma estándar de producir una lista acumulador en Lisp.

La función `group` agrupa una lista `lst` en sublistas de tamaño `n`:

```

1 CL-USER> (group '(a b c d e f g) 2)
2 ((A B) (C D) (E F) (G))

```

Esta función si lleva a cabo un chequeo de error, porque si `n = 0` `group` entra en un ciclo infinito.

Otras funciones (doblemente recursivas) sobre listas son:

```

1 (defun flatten (x)
2   (labels ((rec (x acc)
3             (cond ((null x) acc)
4                   ((atom x) (cons x acc))
5                   (t (rec (car x) (rec (cdr x) acc))))))
6     (rec x nil)))
7
8 (defun prune (test tree)
9   (labels ((rec (tree acc)
10            (cond ((null tree) (nreverse acc))
11                  ((consp (car tree))
12                   (rec (cdr tree)
13                       (cons (rec (car tree) nil) acc))))
13     (rec tree nil)))

```

```

14         (t (rec (cdr tree)
15              (if (funcall test (car tree))
16                  acc
17                  (cons (car tree) acc))))))
18     (rec tree nil)))

```

estas funciones recurren sobre listas anidadas para hacer su trabajo. La primera de ellas, `flatten`, es una aplanadora de listas. Si su argumento es una lista anidada, regresa los elementos de la lista original, pero eliminando el anidamiento:

```

1 CL-USER 1 > (flatten '(a (b c) ((d e) f)))
2 (A B C D E F)

```

La segunda función, `prune`, elimina de una lista anidada a aquellos elementos atómicos que satisfacen el predicado `test`, de forma que:

```

1 CL-USER 2 > (prune #'evenp '(1 2 (3 (4 5) 6) 7 8 (9)))
2 (1 (3 (5)) 7 (9))

```

3.6 MAPEOS

Otra clase de funciones ampliamente usadas en Lisp son los mapeos, que aplican una función a la secuencia de sus argumentos. La más conocida de estas funciones es `mapcar`. Definiremos otras funciones de mapeo a continuación:

```

1 (defun map0-n (fn n)
2   (mapa-b fn 0 n))
3
4 (defun map1-n (fn n)
5   (mapa-b fn 1 n))
6
7 (defun mapa-b (fn a b &optional (step 1))
8   (do ((i a (+ i step))
9       (result nil))
10      ((> i b) (nreverse result))
11      (push (funcall fn i) result)))
12
13 (defun map-> (fn start test-fn succ-fn)
14   (do ((i start (funcall succ-fn i))
15       (result nil))
16       ((funcall test-fn i) (nreverse result))
17       (push (funcall fn i) result)))
18
19 (defun mappend (fn &rest lsts)
20   (apply #'append (apply #'mapcar fn lsts)))
21
22 (defun mapcars (fn &rest lsts)
23   (let ((result nil))
24     (dolist (lst lsts)
25       (dolist (obj lst)
26         (push (funcall fn obj) result)))

```

```

27     (nreverse result)))
28
29     (defun rmapcar (fn &rest args)
30       (if (some #'atom args)
31           (apply fn args)
32           (apply #'mapcar
33                  #'(lambda (&rest args)
34                      (apply #'rmapcar fn args))
35                      args)))

```

Las primeras tres funciones mapean funciones sobre rangos de números sin tener que hacer cons para guardar la lista resultante. Las primeras dos `map0-n` y `map1-n` funcionan con rangos positivos de enteros:

```

1 CL-USER 3 > (map0-n #'1+ 5)
2 (1 2 3 4 5 6)
3
4 CL-USER 4 > (map1-n #'1+ 5)
5 (2 3 4 5 6)

```

Ambas fueron escritas usando `mapa-b` que funciona para cualquier rango de números:

```

1 CL-USER> (mapa-b #'1+ 1 4 0.5)
2 (2 2.5 3.0 3.5 4.0 4.5 5.0)

```

A continuación se implementa un mapeo más general con `map->`, el cual trabaja para cualquier tipo de secuencias de objetos de cualquier tipo. La secuencia comienza con el objeto dado como segundo argumento; el final de la secuencia está dado por el tercer argumento como una función; y los sucesores del primer elemento se generan de acuerdo a la función que se da como cuarto argumento. Con esta función es posible navegar en estructuras de datos arbitrarias, así como operar sobre secuencias de números. Por ejemplo, `mapa-b` puede definirse en términos de `map->` como:

```

1 (defun my-mapa-b (fn a b &optional (step 1))
2   (map-> fn
3         a
4         #'(lambda(x) (> x b))
5         #'(lambda(X) (+ x step))))

```

La función `mapcars` es útil cuando queremos aplicar `mapcar` a varias listas. Las siguientes dos expresiones (`mapcar #'sqrt (append list1 list2)`) y (`mapcars #'sqrt list1 list2`), son equivalentes. Sólo que la segunda versión no hace conses innecesarios.

Finalmente `rmapcar` es un acrónimo para `mapcar` recursivo:

```

1 CL-USER 11 > (rmapcar #'princ '(1 2 (3 4 (5) 6) 7 (8 9)))
2 123456789
3 (1 2 (3 4 (5) 6) 7 (8 9))
4
5 CL-USER 12 > (rmapcar #'+ '(1 (2 (3) 4)) '(10 (20 (30) 40)))
6 (11 (22 (33) 44))

```

