Tema 4: Procedimientos y estructuras recursivas

Contenidos

Bibliografía

- Abelson y Sussman: Cap 1.2 (Procedures and the processes they generace), Introducción capítulo 2 (Building Abstraction with Data) pp. 79–89 y pp. 107–113

Objetivos

Semana 2

- Entender estructuras jerárquicas formadas por listas estructuradas
- Entender la implementación en Scheme de las estructuras jerárquicas como listas que contienen listas u hojas
- Ser capaz de entender y diseñar procedimientos recursivos que trabajen con estructuras jerárquicas (listas estructuradas)
- Entender la implementación en Scheme de la estructura de un árbol genérico
- Entender y ser capaz de diseñar funciones recursivas mutuas que operen sobre árboles genéricos y sobre listas de árboles genéricos (*bosques*)
- Entender y ser capaz de utilizar `map` para diseñar funciones recursivas que operen sobre árboles genéricos

2. Listas estructuradas

Hemos visto temas anteriores que las listas en Scheme se implementan como un estructura de datos recursiva, formadas a partir de parejas y de una lista vacía. Una vez conocida su implementación, vamos a volver a estudiar las listas desde un nivel de abstracción alto, usando las funciones `car` y `cdr` para obtener el primer elemento y el resto de la lista y la

función `cons` para añadir un nuevo elemento a su cabeza.

En la mayoría de funciones y ejemplos que hemos visto hasta ahora las listas están formadas por datos y el recorrido por la lista es un recorrido lineal, una iteración por sus elementos.

En este apartado vamos a ampliar este concepto y estudiar cómo trabajar con *listas que contienen otras listas*.

2.1. Definición y ejemplos

Las listas en Scheme pueden tener cualquier tipo de elementos, incluido otras listas.

Llamaremos *lista estructurada* a una lista que contiene otras sublistas. Lo contrario de lista estructurada es una *lista plana*, una lista formada por elementos que no son listas. Llamaremos *hojas* a los elementos de una lista que no son sublistas.

A las listas estructuradas cuyas hojas son símbolos se les denomina *expresiones-S* (S-expression).

Por ejemplo, la lista estructurada:

```
'(a b (c d e) (f (g h)))
```

es una lista estructurada con 4 elementos:

- El elemento `'a`, una hoja
- El elemento `'b`, otra hoja
- La lista plana `'(c d e)`
- La lista estructurada `'(f (g h))`

Una lista formada por parejas la consideraremos una lista plana, ya que no contiene ninguna sublista. Por ejemplo, la lista

```
'((a . 3) (b . 5) (c . 12))
```

es una lista plana de tres elementos (hojas) que son parejas.

2.1.1. Definiciones en Scheme

Vamos a escribir las definiciones anteriores en código de Scheme.

Un dato es una hoja si no es una lista:

```
(define (hoja? dato)
   (not (list? dato)))
```

Una definición recursiva de lista plana:

> Una lista es plana si y solo si el primer elemento es una hoja y el resto es plana.

Y el caso base:

> Una lista vacía es plana.

En Scheme:

```
(define (plana? lista)
   (or (null? lista)
      (and (hoja? (car lista))
           (plana? (cdr lista)))))
```

Una lista es estructurada cuando alguno de sus elementos es otra lista:

```
(define (estructurada? lista)
   (if (null? lista)
      #f
      (or (list? (car lista))
          (estructurada? (cdr lista)))))
```

Ejemplos:

```
(plana? '(a b c d e f)) ⇒ #t
(plana? '((a . 1) (b . 2) (c . 3))) ⇒ #t
(plana? '(a (b c) d)) ⇒ #f
(plana? '(a () b)) ⇒ #f
(estructurada? '(1 2 3 4)) ⇒ #f
(estructurada? '((a . 1) (b . 2) (c . 3))) ⇒ #f
(estructurada? '(a () b)) ⇒ #t
(estructurada? '(a (b c) d)) ⇒ #t
```

Realmente bastaría con haber hecho una de las dos definiciones y escribir la otra como la negación de la primera:

```
(define (estructurada? lista)
   (not (plana? lista)))
```

O bien:

```
(define (plana? lista)
   (not (estructurada? lista)))
```

2.1.2. Ejemplos de listas estructuradas

Las listas estructuradas son muy útiles para representar información jerárquica en donde queremos representar elementos que contienen otros elementos.

Por ejemplo, las expresiones de Scheme son listas estructuradas:

```
'(= 4 (+ 2 2))
'(if (= x y) (* x y) (+ (/ x y) 45))
'(define (factorial x) (if (= x 0) 1 (* x (factorial (- x 1)))))
```

El análisis sintáctico de una oración puede generar una lista estructurada de símbolos, en donde se agrupan los distintos elementos de la oración:

```
'((Juan) (compró) (la entrada (de los Miserables)) (el viernes por la tarde))
```

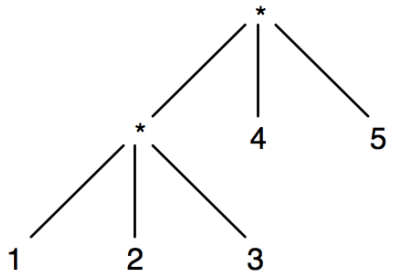
Una página HTML, con sus distintos elementos, unos dentro de otros, también se puede representar con una lista estructurada:

```
'((<h1> Mi lista de la compra </h1>)
  (<ul> (<li> naranjas </li>)
        (<li> tomates </li>)
        (<li> huevos </li>) </ul>))
```

2.1.3. *Pseudo árboles* con niveles

Las listas estructuradas definen una estructura de niveles, donde la lista inicial representa el primer nivel, y cada sublista representa un nivel inferior. Los datos de las listas representan las hojas.

Por ejemplo, la representación en forma de niveles de la lista `'((1 2 3) 4 5)` es la siguiente:



La estructura no es un árbol propiamente dicho, porque todos los datos están en las hojas.

Otro ejemplo. ¿Cuál sería la representación en niveles de la siguiente lista estructurada?:

```
'(let ((x 12)
       (y 5))
   (+ x y)))
```

2.2. Funciones recursivas sobre listas estructuradas

2.2.1. Número de hojas

Veamos como primer ejemplo la función que cuenta el número de hojas de una lista estructurada.

Podemos definir la función obteniendo el primer elemento y el resto de la lista, y contando recursivamente el número de hojas del primer elemento y del resto. Al ser una lista estructurada, el primer elemento puede ser a su vez otra lista, por lo que llamamos a la recursión para contar sus hojas.

La definición de este caso general usando *pseudocódigo* es:

> El número de hojas de una lista estructurada es la suma del número de hojas de su primer elemento (que puede ser otra lista) y del número de hojas del resto.

Como casos base, podemos considerar cuando la lista es vacía (el número de hojas es 0) y cuando la lista no es tal, sino que es un dato (una hoja), en cuyo caso es 1. La implementación en Scheme es:

```
(define (num-hojas lista)
   (cond
      ((null? lista) 0)
      ((hoja? lista) 1)
      (else (+ (num-hojas (car lista))
               (num-hojas (cdr lista))))))
```

Hay que hacer notar que el parámetro `lista` puede ser tanto una lista como un dato atómico. Estamos aprovechándonos de la característica de Scheme de ser débilmente tipeado para hacer un código bastante conciso.

Por ejemplo:

```
(num-hojas '(1 2 (3 4 (5) 6) (7))) ⇒ 7
```

2.2.2. Versión con funciones de orden superior

Podemos usar también las funciones de orden superior `map` y `apply` para obtener una versión más concisa. Como una lista estructurada tiene como elementos otras listas, podemos mapear *la propia función que estamos definiendo* sobre sus elementos. El resultado será una

lista de números (el número de hojas de cada componente), que podemos sumar aplicando la función `+`:

```
(define (num-hojas lista)
  (cond
    ((null? lista) 0)
    ((hoja? lista) 1)
    (else (apply + (map num-hojas lista)))))
```

2.2.3. Altura de una lista estructurada

La *altura* de una lista estructurada viene dada por su número de niveles: una lista plana tiene una altura de 1, la lista `'((1 2 3) 4 5)` tiene una altura de 2.

Para calcular la altura de una lista estructurada tenemos que obtener (de forma recursiva) la altura de su primer elemento, y la altura del resto de la lista, sumarle 1 a la altura del primer elemento y devolver el máximo de los dos números (te recomiendo hacer un dibujo con pseudo-árboles para entender mejor la idea).

Como casos base, la altura de una lista vacía o de una hoja (dato) es 0.

En Scheme:

```
(define (altura lista)
   (cond
      ((null? lista) 0)
      ((hoja? lista) 0)
      (else (max (+ 1 (altura (car lista)))
                 (altura (cdr lista))))))
```

Por ejemplo:

```
(altura '(1 (2 3) 4)) ⇒ 2
(altura '(1 (2 (3)) 3)) ⇒ 3
```

Y la segunda versión, usando la función de orden superior `map`:

```
(define (altura lista)
  (cond
    ((null? lista) 0)
    ((hoja? lista) 0)
    (else (+ 1 (apply max (map altura lista))))))
```

2.2.4. Otras funciones recursivas

Vamos a diseñar otras funciones recursivas que trabajan con la estructura jerárquica de las listas estructuradas.

- `(pertenece? lista)` : busca una hoja en una lista estructurada
- `(cuadrado-lista lista)` : eleva todas las hojas al cuadrado (suponemos que la lista estructurada contiene números)
- `(map-lista f lista)` : similar a map, aplica una función a todas las hojas de la lista estructurada y devuelve el resultado (otra lista estructurada)

`pertenece?`

Comprueba si el dato x aparece en la lista estructurada.

```
(define (pertenece? x lista)
  (cond
    ((null? lista) #f)
    ((hoja? lista) (equal? x lista))
    (else (or (pertenece? x (car lista))
              (pertenece? x (cdr lista))))))
```

Ejemplos:

```
(pertenece? 'a '(b c (d (a)))) ⇒ #t
(pertenece? 'a '(b c (d e (f)) g)) ⇒ #f
```

`cuadrado-lista`

Devuelve una lista estructurada con la misma estructura y sus números elevados al cuadrado.

```
(define (cuadrado-lista lista)
  (cond ((null? lista) '())
        ((hoja? lista) (* lista lista))
        (else (cons (cuadrado-lista (car lista))
                    (cuadrado-lista (cdr lista))))))
```

Por ejemplo:

```
(cuadrado-lista '(2 3 (4 (5)))) ⇒ (4 9 (16 (25))
```

`map-lista`

Devuelve una lista estructurada igual que la original con el resultado de aplicar a cada uno de sus hojas la función f

```
(define (map-lista f lista)
  (cond ((null? lista) '())
        ((hoja? lista) (f lista))
        (else (cons (map-lista f (car lista))
                    (map-lista f (cdr lista))))))
```

Por ejemplo:

```
(map-lista (lambda (x) (* x x)) '(2 3 (4 (5)))) ⇒ (4 9 (16 (25))
```

2.3. Ejercicios

Planteamos a continuación algunos ejercicios para que practiquéis las funciones recursivas sobre las listas estructuradas.

`tres-elementos?`

Escribe la función `(tres-elementos? lista)` que tome una lista estructurada como argumento. Devolverá #t si cada subsista que aparece y cada uno de sus elementos tienen 3 elementos. Suponemos que nunca se llamará a la función con una lista vacía.

Ejemplos:

```
(tres-elementos? '(1 2 3)) ⇒ #t
(tres-elementos? '((1 2 3) 2 3)) ⇒ #t
(tres-elementos? '((1 2 3) (4 5 6))) ⇒ #f
(tres-elementos? '(1 2 (3 3))) ⇒ #f
```

`diff-listas`

Escribe la función `(diff-listas l1 l2)` que tome como argumentos dos listas estructuradas con la misma estructura, pero con diferentes elementos, y devuelva una lista de parejas que contenga los elementos que son diferentes.

Ejemplos:

```
(dif-listas '(a (b ((c)) d e) f) '(1 (b ((2)) 3 4) f))
⇒ ((a . 1) (c . 2) (d . 3) (e . 4))
(diff-listas '() '())
⇒ ()
(diff-listas '((a b) c) '((a b) c))
⇒ ()
```

`transformar`

Define un procedimiento llamado `(transformar plantilla lista)` que reciba dos listas como argumento: la lista plantilla será una lista estructurada y estará compuesta por números enteros positivos con una estructura jerárquica, como `(2 (3 1) 0 (4))`. La segunda lista será una lista plana con tantos elementos como indica el mayor número de plantilla más uno (en el caso anterior serían 5 elementos). El procedimiento deberá devolver una lista estructurada donde los elementos de la segunda lista se sitúen (en situación y en estructura) según indique la plantilla.

Ejemplos:

```
(transformar '((0 1) 4 (2 3)) '(hola que tal estas hoy))
⇒ ((hola que) hoy (tal estas))
(transformar '(1 4 3 2 5 (0)) '(vamos todos a aprobar este examen))
⇒ (todos este aprobar a examen (vamos))
```

`nivel-hoja`

Escribe la función `(nivel-hoja dato lista)` que recorra la lista buscando el dato y devuelva el nivel en que se encuentra. Suponemos que el dato está en la lista y no está repetido.

Ejemplos:

```
(nivel-hoja 2 '(1 2 (3))) ⇒ 1
(nivel-hoja 2 '(1 (2) 3)) ⇒ 2
(nivel-hoja 2 '(1 3 4 ((2))) ⇒ 4
```

`hojas-nivel`

Define la función `(hojas-nivel lista)` que reciba una lista estructurada y devuelva una lista de parejas con las hojas y la profundidad en que se encuentra cada hoja.

Ejemplos:

```
(hojas-nivel '(2 3 4 (5) ((6))))
⇒ ((2.1) (3.1) (4.1) (5.2) (6.3))
(hojas-nivel '(2 (5 (8 9) 10 1)))
⇒ ((2.1) (5.2) (8.3) (9.3) (10.2) (1.2))
```

`acumula`

Define la función (acumula lista nivel) que reciba una lista estructurada compuesta por números y un número de nivel. Devolverá la suma de todos los nodos hoja que tengan una profundidad mayor o igual que el nivel indicado.

Ejemplos:

```
(acumula '(2 3 4 (5) ((6))) 2) ⇒ 11
(acumula '(2 3 4 (5) ((6))) 3) ⇒ 6
(acumula '(2 (5 (8 9) 10 1)) 3) ⇒ 17
```

3. Árboles

3.1. Definición de árboles en Scheme

3.1.1. Representación de árboles con listas

Vamos a explicar la forma de representar árboles en Scheme.

Un árbol contiene un dato en su raíz y un número cualquiera de hijos, que también son árboles.

En Scheme utilizaremos una lista de *n+1* elementos para representar un árbol con n hijos: el primer elemento será el dato de la raíz y el resto serán los árboles hijos. Los nodos hoja serán por tanto listas de un elemento, el dato.

```
Árbol => (dato hijo-1 hijo-2 ... hijo-n)
```

Por ejemplo, el árbol:

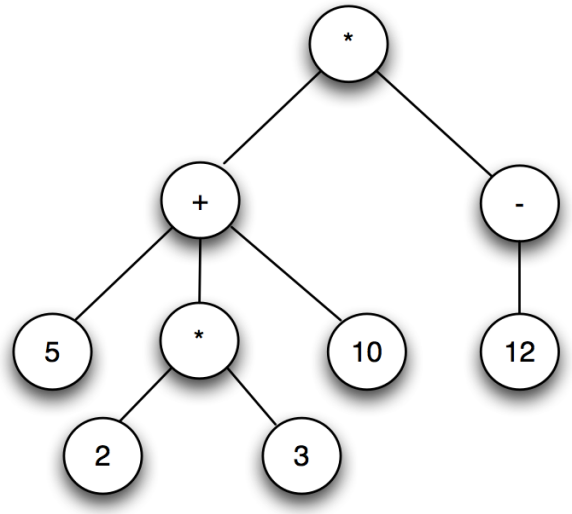
```
   10
  /|\
 / | \
2  3  5
```

Lo representaremos en Scheme con la siguiente lista estructurada:

```
(10 (2) (3) (5))
```

Los nodos 2, 3 y 5 son nodos hoja, y también árboles. Por eso los representamos como listas cuyo único elemento es el dato y sin árboles hijos.

Veamos un ejemplo más complicado. En los nodos podemos guardar tanto números como símbolos.



Un ejemplo de árbol genérico

Lo representamos con la siguiente lista estructurada:

```
(* (+ (5) (* (2) (3)) (10)) (- (12)))
```

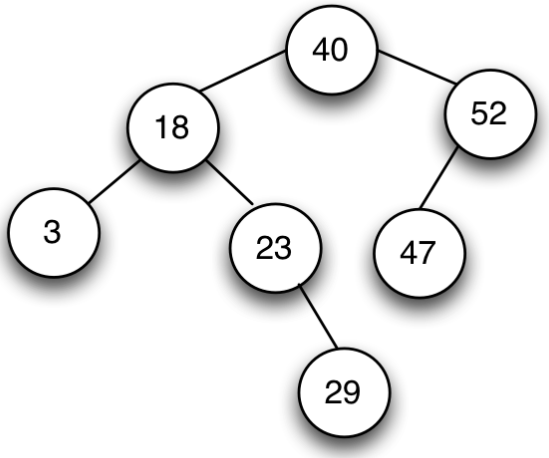
Si tenemos un estructura como la anterior, las funciones `car` y `cdr` devuelven lo siguiente:

- `(car tree)` : devuelve el dato de la raíz del árbol
- `(cdr tree)` : devuelve la lista de árboles hijos (lista vacía, cuando es un nodo hoja)

En algunas funciones denominaremos *bosque* a la lista de árboles hijos.

Podemos construir un árbol a partir de un dato y una lista de hijos (puede ser vacía) añadiendo el dato a la cabeza de la lista de hijos con la función `cons` .

Otro ejemplo más. ¿Cómo se implementa en Scheme el árbol de la siguiente figura?



Se haría con la siguiente lista estructurada:

```
(40 (18 (3) (23 (29)))
    (52 (47)))
```

3.1.2. Los árboles son listas estructuradas

Es importante darse cuenta de que ya que los árboles se implementan con listas estructuradas, cualquier función que reciba listas estructuradas también podrá recibir un árbol. Pero lo tratará como una lista estructurada, no como un árbol. Por ejemplo, si le pasamos a la función `num-hojas` el árbol anterior devolverá 7, que es número de hojas de la lista estructurada, pero no el número de hojas del árbol.

```
(num-hojas '((40 (18 (3) (23 (29))) (52 (47))))) ⇒ 7
```

3.1.3. Barrera de abstracción

En un lenguaje débilmente tipeado como Scheme es conveniente darles un nombre distinto a las funciones de acceso a los datos del árbol, para dejar claro que son funciones que trabajan sobre árboles, y no sobre listas.

Por eso definimos las siguientes funciones que proporcionan una *barrera de abstracción* sobre el nuevo tipo de dato *tree*. Una *barrera de abstracción* es un conjunto de funciones que permiten trabajar con un tipo de datos escondiendo su implementación.

Constructores

Funciones que permiten construir un nuevo árbol:

```
(define (make-tree dato lista-hijos)  (cons dato lista-hijos))
(define (make-hoja-tree dato) (make-tree dato '()))
```

Selectores

Funciones que obtienen los elementos de un árbol:

```
(define (dato-tree tree)  (car tree))
(define (hijos-tree tree) (cdr tree))
(define (hoja-tree? tree) (null? (hijos-tree tree)))
```

El árbol anterior se puede construir con las siguientes llamadas a los constructores:

```
(define tree
  (make-tree '*
             (list (make-tree
                    '+ (list (make-hoja-tree 5)
```

```
                    (make-tree '*
                               (list (make-hoja-tree 2)
                                     (make-hoja-tree 3)))
                    (make-hoja-tree 10)))
          (make-tree '- (list (make-hoja-tree 12))))))
```

Cuando trabajamos con árboles genéricos y hacemos funciones recursivas que los recorren, es muy importante considerar en cada caso con qué tipo de dato estamos trabajando y usar la barrera de abstracción adecuada:

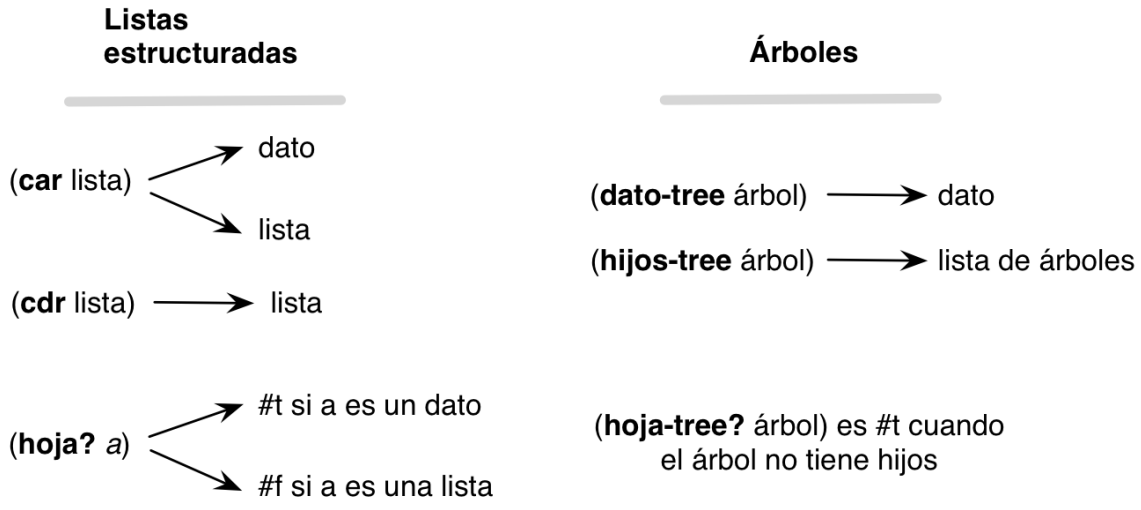
- La función `hijos-tree` devuelve una lista de árboles, que podemos recorrer usando `car` y `cdr`
- El `car` de una lista de árboles (devuelta por `hijos-tree`) es un árbol y debemos de usar las funciones de su barrera de abstracción: `dato-tree` e `hijos-tree`
- La función `dato-tree` devuelve un dato de árbol, del tipo que guardemos en el árbol

Por ejemplo, para obtener el número 2 en el árbol anterior habría que acceder al primer hijo del árbol, después al segundo hijo de este subárbol, y por último al dato del primero. Recordemos que `hijos-tree` devuelve la lista de árboles hijos, por lo que utilizaremos las funciones `car` y `cdr` para recorrerlas y obtener los elementos que nos interesen:

```
(dato-tree (car (hijos-tree (cadr (hijos-tree (car (hijos-tree tree)))))))
⇒ 2
```

Es importante diferenciar la barrera de abstracción de los árboles de la de las listas estructuradas. Aunque un árbol se implementa en Scheme con una lista estructurada, a la hora de definir funciones sobre árboles hay que trabajar con las funciones definidas arriba.

El siguiente esquema resumen las características de la barrera de abstracción de listas y árboles:



3.2. Funciones recursivas

Vamos a diseñar las siguientes funciones recursivas:

- `(suma-datos-tree tree)` : devuelve la suma de todos los nodos
- `(to-list-tree tree)` : devuelve una lista con los datos del árbol
- `(cuadrado-tree tree)` : eleva al cuadrado todos los datos de un árbol manteniendo la estructura del árbol original
- `(map-tree f tree)` : devuelve un árbol con la estructura del árbol original aplicando la función f a subdatos.
- `(altura-tree tree)` : devuelve la altura de un árbol

Todas comparten un patrón similar de recursión mutua.

3.2.1. `(suma-datos-tree tree)`

Vamos a implementar una función recursiva que sume todos los datos de un árbol.

Un árbol siempre va a tener un dato y una lista de hijos (que puede ser vacía) que obtenemos con las funciones `dato-tree` e `hijos-tree` . Podemos plantear entonces el problema de sumar los datos de un árbol como la suma del dato de su raíz y lo que devuelva la llamada a una función auxiliar que sume los datos de su lista de hijos (un bosque):

```
(define (suma-datos-tree tree)
    (+ (dato-tree tree)
       (suma-datos-bosque (hijos-tree tree))))
```

Esta función suma los datos de **UN** árbol. La podemos utilizar entonces para construir la siguiente función que suma una lista de árboles:

```
(define (suma-datos-bosque bosque)
   (if (null? tree)
       0
       (+ (suma-datos-tree (car bosque)) (suma-datos-bosque (cdr bosque)))))
```

Tenemos una recursión mutua: para sumar los datos de una lista de árboles llamamos a la suma de un árbol individual que a su vez llama a la suma de sus hijos, etc. La recursión termina cuando calculamos la suma de un árbol hoja. Entonces se pasa a `suma-datos-bosque` una lista vacía y ésta devolverá 0.

Por ejemplo:

```
(suma-datos-bosque '()) ⇒ 0
(suma-datos-tree '(12)) ⇒ 12
(suma-datos-tree '(12 (4) (5) (6))) ⇒ 27
```

Versión alternativa con funciones de orden superior

Al igual que hacíamos con las listas estructuradas, es posible conseguir una versión más concisa y elegante utilizando funciones de orden superior:

```
(define (suma-datos-tree tree)
    (+ (dato-tree tree)
       (apply + (map suma-datos-tree (hijos-tree tree)))))
```

La función `map` aplica la propia función que estamos definiendo a cada uno de los árboles de `(hijos-tree tree)`, devolviendo una lista de números. Esta lista de número la sumamos haciendo un `apply +`. Una traza de su funcionamiento sería la siguiente:

```
(suma-datos-tree '(1 (2 (3) (4)) (5) (6 (7)))) ⇒
(+ 1 (apply + (map suma-datos-tree '((2 (3) (4))
                                     (5)
                                     (6 (7))))) ⇒
(+ 1 (apply + '(9 5 13))) ⇒
(+ 1 27) ⇒
28
```

3.2.2. `(to-list-tree tree)`

Queremos diseñar una función `(to-list-tree tree)` que devuelva una lista con los datos del árbol en un recorrido *preorden*.

```
(define (to-list-tree tree)
   (cons (dato-tree tree)
         (to-list-bosque (hijos-tree tree))))

(define (to-list-bosque bosque)
   (if (null? bosque)
       '()
       (append (to-list-tree (car bosque))
               (to-list-bosque (cdr bosque)))))
```

La función utiliza una *recursión mutua*: para listar todos los nodos, añadimos el dato a la lista de nodos que nos devuelve la función `to-list-bosque`. Esta función coge una lista de árboles (un *bosque*) y devuelve la lista *inorden* de sus nodos. Para ello, concatena la lista de los nodos de su primer elemento (el primer árbol) a la lista de nodos del resto de árboles (que devuelve la llamada recursiva).

Ejemplo:

```
(to-list-tree '(* (+ (5) (* (2) (3)) (10)) (- (12))))
⇒ (* + 5 * 2 3 10 - 12)
```

Una definición alternativa usando funciones de orden superior:

```
(define (to-list-tree tree)
  (if (null? (hijos-tree tree))
      (list (dato-tree tree))
      (cons (dato-tree tree)
            (apply append (map to-list-tree (hijos-tree tree))))))
```

Esta versión es muy elegante y concisa. Usa la función `map` que aplica una función a los elementos de una lista y devuelve la lista resultante. Como lo que devuelve `(hijos-tree tree)` es precisamente una lista de árboles podemos aplicar a sus elementos cualquier función definida sobre árboles. Incluso la propia función que estamos definiendo (¡confía en la recursión!).

3.2.3. `(cuadrado-tree tree)`

Veamos ahora la función `(cuadrado-tree tree)` que toma un árbol de números y devuelve un árbol con la misma estructura y sus datos elevados al cuadrado:

```
(define (cuadrado-tree tree)
   (make-tree (cuadrado (dato-tree tree))
                  (cuadrado-bosque (hijos-tree tree))))

(define (cuadrado-bosque bosque)
   (if (null? bosque)
       '()
       (cons (cuadrado-tree (car bosque))
             (cuadrado-bosque (cdr bosque)))))
```

Ejemplo:

```
(cuadrado-tree '(2 (3 (4) (5)) (6)))
⇒ (4 (9 (16) (25)) (36))
```

Versión 2, con `map` :

```
(define (cuadrado-tree tree)
   (make-tree (cuadrado (dato-tree tree))
                 (map cuadrado-tree (hijos-tree tree))))
```

3.2.4. `map-tree`

La función `map-tree` es una función de orden superior que generaliza la función anterior. Definimos un parámetro adicional en el que se pasa la función a aplicar a los elementos del

árbol.

```
(define (map-tree f tree)
   (make-tree (f (dato-tree tree))
              (map-bosque f (hijos-tree tree))))

(define (map-bosque f bosque)
   (if (null? bosque)
       '()
       (cons (map-tree f (car bosque))
             (map-bosque f (cdr bosque)))))
```

Ejemplos:

```
(map-tree cuadrado '(2 (3 (4) (5)) (6)))
⇒ (4 (9 (16) (25)) (36))
(map-tree (lambda (x) (+ x 1)) '(2 (3 (4) (5)) (6)))
⇒ (3 (4 (5) (6)) (7))
```

Con `map`:

```
(define (map-tree f tree)
  (make-tree (f (dato-tree tree))
             (map (lambda (x)
                    (map-tree f x)) (hijos-tree tree))))
```

3.2.5. `altura-tree`

Vamos por último a definir una función que devuelve la altura de un árbol (el nivel del nodo de mayor nivel). Un nodo hoja tiene de altura 0.

Solución 1:

```
(define (altura-tree tree)
   (if (hoja-tree? tree)
       0
       (+ 1 (max-altura-bosque (hijos-tree tree)))))

(define (max-altura-bosque bosque)
    (if (null? bosque)
        0
        (max (altura-tree (car bosque))
             (max-altura-bosque (cdr bosque)))))
```

Ejemplos:

```
(altura-tree '(2)) ⇒ 0
(altura-tree '(4 (9 (16) (25)) (36))) ⇒ 2
```

Solución 2:

La función `max-altura-bosque` puede implementarse de una forma más concisa todavía usando las funciones `apply` y `map`:

```
(define (max-altura-bosque bosque)
   (apply max (map altura-tree bosque)))
```

La función `map` mapea la función `altura-tree` a todos los elementos del *bosque* (lista de árboles) devolviendo una lista de números, de la que obtenemos el máximo aplicando (`apply`) la función `max` .
