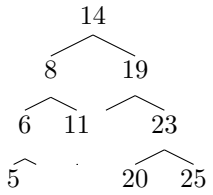
Árboles generales

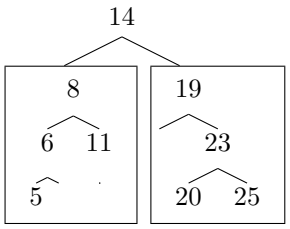
Á. Tasistro

Primavera de 2016

El siguiente es un ejemplo de árbol:



Su característica esencial es que se forma *recursivamente* combinando un *elemento* (en este caso, un entero) con otro*s* árboles (en este ejemplo, *dos*):



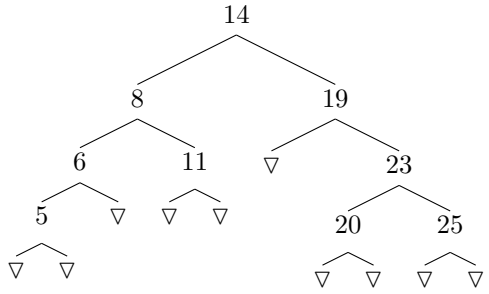
Esto debe compararse con la formación de *listas* que combina un elemento con *una* lista:

$[2, \underbrace{4, 6, 8, 10}]$.

En el caso de las listas llamamos al primer elemento la *cabeza* de la lista y al resto (la lista subsiguiente) su *cola*. En el caso de los árboles, el elemento en el tope es denominado la *raíz* del árbol[1], mientras que los árboles componentes son sus *subárboles*. Como ya es sabido, las listas pueden seguirse descomponiendo recursivamente una cantidad finita de veces hasta llegar a su caso base indivisible, que es en general la lista vacía. Como es también ya sabido, si escribimos esta (des)composición en notación explícita de Haskell se tiene, para la lista precedente:

`2 : (4 : (6 : (8 : (10 : []))))` [2].

Asimismo, si usamos la notación $\triangledown$ para denotar al árbol vacío, podemos reescribir más explícitamente la formación del árbol precedente:



Un árbol del tipo del de este ejemplo es llamado *binario*, porque cumple que:

- O bien se compone de un elemento y *dos* árboles *binarios* (caso recursivo),

[1] O sea que estos árboles crecen hacia abajo, al revés de lo que ocurre en la naturaleza. Esta forma de representaciónn "invertida" es la más frecuente, aunque no excluyente.

[2] Los paréntesis que aparecen allí pueden en general omitirse puesto que Haskell asume ese preciso orden de asociación del operador "cons", es decir, a la derecha.

- o bien es el árbol vacío (caso base).

Veamos entonces cómo introducir este tipo de árboles binarios en Haskell. Para ello repasemos primero cómo se procede en el caso de las listas. Uno tiene para empezar una forma de denotar la *lista vacía*, a saber `[]`. Luego se tiene una manera de *combinar* un elemento con una lista para formar una lista más larga. Éste es el constructor "cons", escrito en Haskell `:` cuando se usa en forma infija. Como ya hemos visto, la declaración de este tipo en Haskell sería:

```
data [a] = [] | (:)  a [a],
```

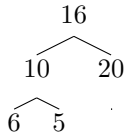
pero no es necesario efectuarla puesto que las listas están predefinidas en Haskell. Recordemos también que esta declaración está formulada en términos de un parámetro de tipo `a`, que indica que se están definiendo listas cuyos miembros pueden ser de *cualquier* tipo (aunque el mismo para todos ellos).

Para el caso de los árboles binarios del tipo del de arriba, tendremos, correspondientemente, una constante para el árbol vacío; llamémosla `Empty`. Y luego necesitaremos una *función constructora* que combine un elemento con dos árboles binarios para formar un árbol binario mayor. Si denominamos a esta función `Node`, entonces podemos introducir el tipo `BinTree` de los árboles binarios de la siguiente manera:

```
data BinTree a = Empty | Node a (BinTree a) (BinTree a).
```

Observemos que, al igual que en el caso de las listas, `BinTree` debe ser aplicado a un tipo, que será el de los elementos de los árboles del tipo resultante. Así podremos tener árboles de enteros `BinTree Integer`, de booleanos `BinTree Bool` y, en general, de cualquier tipo de elementos (aunque, nuevamente, el mismo para todos éstos). Asimismo, tal como ya hemos visto en varios casos concerniendo listas, podremos definir funciones que admitan como argumentos árboles cuyos elementos puedan ser de un tipo genérico.

Con la declaración que acabamos de dar, el árbol

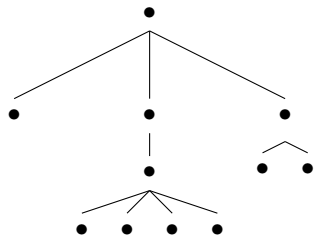


debe ser escrito como sigue:

```
Node 16 (Node 10 (Node 6 Empty Empty) (Node 5 Empty Empty)) (Node 20 Empty Empty).
```

Los elementos de los árboles son habitualmente llamados sus *nodos*. Cada nodo tiene asociado un número de *subárboles* (en el caso de los árboles binarios, siempre *dos*). Las raíces de estos subárboles, si existen, son llamados nodos *hijos* del nodo original. Así, en el árbol precedente, la raíz del árbol completo, con valor `16` tiene como nodos hijos a los que contienen los valores `10` y `20`. Se denominan *hojas* del árbol a aquellos nodos sin hijos. Se observa también que los nodos de todo árbol se organizan en *niveles*: el primer nivel está ocupado sólo por la raíz, el segundo por los hijos de ésta y, en general, cada nivel contiene todos los hijos de los nodos del nivel precedente. La *altura* de un árbol es la cantidad de niveles que posee.

Uno puede, naturalmente, imaginarse múltiples tipos de árboles, más allá de los binarios. En un caso bastante general, cada nodo puede tener asociada una *lista finita* de subárboles, como por ejemplo:



Volviendo a los árboles binarios, corresponde ahora la pregunta: ¿cómo se programan funciones sobre estas estructuras? Consideremos por ejemplo una función que calcule el *tamaño* (cantidad de nodos) de un árbol binario dado. Si la llamamos `size`, su declaración es la siguiente:

```
size ::  BinTree a -> Integer
```
porque claramente puede actuar sobre árboles binarios de cualquier tipo de elementos.
Para razonar sobre árboles binarios, uno recuerda su *definición*. Un árbol binario es:

- O bien el arbol vacío `Empty`.
- O bien un árbol no vacío formado por un elemento y dos subárboles. En el caso de los árboles binarios, los subárboles suelen ser denominados *izquierdo* y *derecho*. La forma de los árboles no vacíos es entonces `Node` *x izq der*, donde *x* es el elemento (nodo raíz) y los otros dos, sus subárboles. Por lo tanto, *x* es de tipo `a`, mientras *izq* y *der* son de tipo `BinTree a`.

Se trata, como ya es evidente, de una definición *recursiva* (el término técnico es *inductiva*) y el razonamiento correspondiente también debe ser recursivo:

1. Uno piensa la solución correcta para el *caso base* (en este caso, el árbol vacío `Empty`).
2. Uno piensa cómo combinar las soluciones para los subárboles componentes (obtenidas mediante las *llamadas recursivas*) de forma de obtener una solución correcta para el árbol no vacío compuesto. En concreto, uno debe pensar cuál es la solución para `Node` *x izq der* utilizando las llamadas recursivas sobre *izq* y *der*.

En nuestro caso, esto da:
```
size Empty = 0
size (Node x izq der) = size izq + size der + 1,
```
porque:

1. El tamaño del árbol vacío es claramente 0.
2. El tamaño de un árbol binario no vacío (`Node` *x izq der*) se obtiene sumando los tamaños de los subárboles, más una unidad correspondiente a la raíz.

Las soluciones recursivas pueden entenderse (como ya hemos visto) de la siguiente manera:

- Todo árbol (binario) se forma comenzando con un árbol vacío (*caso base* o *arranque*) y luego
- Combinando dos árboles binarios ya formados con un elemento para formar un árbol mayor (*caso recursivo* o *propagación*).

Correspondientemente, una función se puede definir para todo árbol binario de la siguiente manera:

1. Dando la solución correcta para el árbol vacío (*caso base* o *arranque correcto*).
2. Combinando las soluciones para los subárboles componentes de forma de obtener la solución correcta para el árbol mayor (*paso recursivo* o *propagación correcta*).

Simétricamente, puede verse que:

- El caso recursivo descompone el árbol no vacío en sus componentes (subárboles más pequeños) y utiliza las soluciones en éstos para combinarlas en la solución correcta para el árbol mayor dado.
- El caso base se ocupa de que la recursión termine correctamente.

?1. Programar una función que *sume* todos los nodos de un árbol binario de enteros.

?2. Programar funciones que reciban un árbol binario cualquiera y un predicado sobre elementos del árbol y:

1. Determine si *todos* los nodos satisfacen el predicado.

2. Determine si *algún* nodo satisface el predicado.

?3. Programar una función que calcule la *altura* de un árbol binario dado. *Sugerencia*: programar una función que calcule el *máximo* de dos enteros dados.

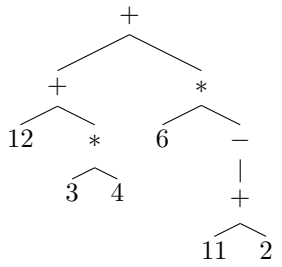
?4. Un *árbol binario ordenado* (de enteros) es o bien vacío o está formado por una raíz entera y dos subárboles binarios ordenados tales que todos los nodos del subárbol izquierdo son menores que la raíz y todos los del subárbol derecho son mayores o iguales que la raíz.

1. Observar cuáles de los ejemplos de árboles binarios colocados a lo largo del repartido son ordenados.
2. Escribir una función que reciba un árbol binario ordenado y un entero y verifique si éste se encuentra o no en el árbol, efectuando el menor número de comparaciones posible.

?5. Una *linealización* de un árbol es cualquier proceso por el cual éste se convierte en una *lista* de sus elementos. Escribir funciones de linealización para árboles binarios de modo de que:

1. Cada nodo preceda a todo su subárbol izquierdo y éste al derecho (*preorden* o *primer orden*).
2. El subárbol izquierdo preceda a cada nodo y éste al subárbol derecho (*en orden* o *segundo orden*).
3. El subárbol izquierdo preceda al derecho y éste al nodo (*postorden* o *tercer orden*).

Una aplicación extremadamente conveniente del concepto de árbol es la representación de *expresiones* de lenguajes. Por ejemplo, considérese la siguiente expresión aritmética:
$12 + 3 * 4 + 6 * (-(11 + 2))$.
Si el lector procede a evaluarla de acuerdo a las convenciones comunes verá que la tarea requiere cierto esfuerzo para determinar el orden correcto de realización de las operaciones. Sería más sencillo si la expresión se nos presentara en esta forma:



Lo que ocurre es que la representación en forma de árbol muestra explícitamente la manera en que los operadores deben ser aplicados. Uno puede proceder muy fácilmente con la evaluación asociando a cada operador el resultado de aplicarlo a los valores de sus hijos, comenzando por los niveles inferiores hasta culminar con el resultado final en la raíz. En otras palabras, la representación en forma de árbol resuelve de por sí ("gratuitamente") el problema del orden en que las operaciones deben ser realizadas. Sería desde este punto de vista muy ventajoso si tuviéramos la posibilidad y costumbre de escribir las expresiones aritméticas directamente como árboles en lugar de linealmente. Desafortunadamente, la representación de árbol resulta, por su bidimensionalidad, costosa como representación concreta. Es por ello que los árboles como el precedente son denominados de *sintaxis abstracta*, pues exhiben la *estructura* de cada expresión de manera directa o "ideal". Y, en contrapartida, las formas lineales de las expresiones son llamadas de *sintaxis concreta*, pues proveen maneras específicas de escribirlas en renglones, ya sea a mano o por medio de interfaces clásicas como el teclado. Nótese que un mismo árbol de sintaxis abstracta puede escribirse en forma lineal de múltiples maneras. Por ejemplo, además de la expresión lineal de la cual partimos, la siguiente también representa el árbol de arriba:

$12 + (3 * 4) + 6 * (-(11 + 2))$.

Allí hemos agregado un par de paréntesis que podrían criticarse por ser redundantes, *dada la convención usual de precedencia del producto sobre la suma*. Pero, más allá de tal observación, el uso de paréntesis "redundantes" no es incorrecto y puede ser conveniente. Puede inclusive llevarse a la máxima expresión como en:

$((12 + (3 * 4)) + (6 * (-(11 + 2))))$.

El proceso por el cual se pasa de una sintaxis concreta al correspondiente árbol de sintaxis abstracta se denomina *análisis sintáctico* o (en inglés) *parsing*. El proceso opuesto es llamado de *linealización*. Los procesadores de lenguajes de programación, sean ellos compiladores o intérpretes, reciben strings de símbolos y controlan que respeten las reglas de la sintaxis concreta del lenguaje; si esto ocurre, traducen la sintaxis concreta al correspondiente árbol de sintaxis abstracta. Una vez hecho esto es posible evaluar (interpretar) la expresión, ya en forma de árbol, o traducirla (compilarla) linealizándola en otro lenguaje.

Veamos ahora cómo introducir en Haskell estos tipos de árboles de sintaxis abstracta que resultan de tanta utilidad. Consideremos a estos efectos expresiones aritméticas donde los operadores son la suma, el producto y el signo $-$ (también llamado "negación aritmética"). Pensemos en cuál es el *caso base* de estos árboles de expresiones. Observando el ejemplo de la página precedente, uno nota que se trata de los numerales o constantes enteras. Además de ellos, están los operadores, que son constructores de expresiones (árboles) compuestas. Esto nos motiva a introducir:

1. Un constructor que transforme cada número entero en un árbol de expresión, que será, obviamente, atómico.

2. Un constructor recursivo por cada operador. Será binario en los casos de la suma y el producto y unario en el caso de la negación aritmética.

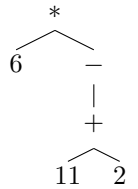
Lo anterior nos da:

```
data  ExprArit  =  Num Integer
                |  Neg ExprArit
                |  Sum ExprArit ExprArit
                |  Prod ExprArit ExprArit
```

Entonces por ejemplo la expresión siguiente:

$6 * -(11 + 2)$,

que corresponde al árbol



se escribirá linealmente, de acuerdo a la declaración Haskell precedente:

`Prod (Num 6) (Neg (Sum (Num 11) (Num 2)))`.

La precedente es notación *prefija*. Utilizando los apóstrofes invertidos es posible utilizar los operadores binarios en forma *infija*, de modo que la siguiente expresión es también válida en Haskell:

`Num 6 ‘Prod‘ Neg(Num 11 ‘Sum‘ Num 2)`.

Aquí cabe recordar que podemos ahorrarnos algunos paréntesis gracias a que la *aplicación* de los operadores prefijos (específicamente, de `Num`) tiene prioridad sobre cualquier operador infijo. La expresión lineal así obtenida puede hacerse aún más económica utilizando nombres más cortos para los operadores. Es posible también utilizar secuencias de símbolos, siempre que comiencen por `:`. De modo que la siguiente sería también una declaración válida de nuestro tipo de expresiones aritméticas:

```
data  ExprArit  =  K Integer
                |  N ExprArit
                |  (:+) ExprArit ExprArit
                |  (:*) ExprArit ExprArit,
```

en cuyo caso la expresión de arriba se podría escribir:

`K 6 :* N(K 11 :+ K 2)`,

que ya es bastante similar a la "natural" $6 * -(11 + 2)$.

Programemos ahora una función que *cuente* las *constantes* (números enteros) intervinientes en una expresión dada. Por ejemplo, en la expresión precedente intervienen $6, 11$ y 2, es decir, tres constantes. No nos importa si una constante interviene más de una vez —es decir, en tal caso la contaremos tantas veces como aparezca. Si llamamos `cant_cnst` a esta función, resulta que su declaración es:

`cant_cnst :: ExprArit -> Integer`.

El método de razonamiento por la forma de los árboles nos da las siguientes ecuaciones a completar:

```
cant_cnst (K x) = ?1
cant_cnst (N e) = ?2
cant_cnst (e1 :+ e2) = ?3
cant_cnst (e1 :* e2) = ?4.
```

En el caso base tenemos *una* constante (`K x`) de modo que el resultado es, claramente, 1. Los otros tres son los casos recursivos —es decir, debemos tener presente que `e` (en la segunda ecuación) así como `e1` y `e2` en la tercera y cuarta ecuación, son *árboles de expresión* cuya complejidad es arbitraria y que contendrán cada uno una cierta cantidad de constantes a contar. Esas cantidades se van a obtener por medio de las *llamadas recursivas*. Entonces, ¿cuál es la cantidad de constantes en `N e`? Bueno, es claramente la misma que la que hay en `e`. De modo que obtenemos la ecuación:

`cant_cnst (N e) = cant_cnst e`,

donde, como es debido, la llamada recursiva achica el argumento. Razonando similarmente, llegamos a la solución completa:

```
cant_cnst (K x) = 1
cant_cnst (N e) = cant_cnst e
cant_cnst (e1 :+ e2) = cant_cnst e1 + cant_cnst e2
cant_cnst (e1 :* e2) = cant_cnst e1 + cant_cnst e2.
```

?6. Programar una función que cuente la cantidad de operadores *binarios* intervinientes en una expresión dada.

?7. Programar una función que *evalúe* una expresión dada (es decir, que calcule su resultado efectivo).

?8. Programar una función que simplifique una expresión dada eliminando toda "doble negación", es decir, los signos aplicados consecutivamente.