

El árbol de sintaxis de una expresión aritmética

Una especificación algebraica en Nereus — material para alumnos

1. ¿Qué es un árbol de sintaxis?

Cuando escribimos una expresión aritmética como `((a+b)*c)/(c+d)`, los paréntesis nos dicen *en qué orden* se combinan las operaciones. Un **árbol de sintaxis** (en inglés, *Abstract Syntax Tree* o AST) es una forma de representar esa estructura sin necesidad de paréntesis: la *forma del árbol* ya contiene el orden.

La regla es simple y tiene solo dos casos:

- una **hoja** guarda un **operando** (un número o una variable: `a`, `b`, `3`...);
- un **nodo interno** guarda un **operador** (`+`, `-`, `*`, `/`, `**`) y tiene exactamente **dos hijos**: la sub-expresión izquierda y la derecha.

El operador `**` es la **potencia**: el árbol `node("**", b, e)` representa la expresión `b**e`, es decir *b elevado a e*. Aunque se escribe con dos caracteres, es un operador **binario** igual que los otros cuatro — combina exactamente dos sub-expresiones — así que encaja en la misma estructura sin cambiar nada del árbol.

Como cada operador combina exactamente dos cosas, el árbol es **binario**: cada nodo tiene a lo sumo dos hijos. Por eso podemos construirlo *heredando* de una estructura que ya conocemos: el árbol binario genérico `BTree`.

2. Heredar de BTree

`BTree` es el árbol binario "puro": sabe guardar una etiqueta en cada nodo y tener un subárbol izquierdo y uno derecho. Ofrece dos constructores — `BTree_Empty()` (árbol vacío) y `BTree_Create(izq, der, etiqueta)` — y observadores para mirar la raíz y los hijos.

Nuestro árbol de sintaxis, `ExprTree`, es un **subtipo** de `BTree`: reutiliza toda esa maquinaria y solo agrega lo que es propio de las expresiones aritméticas.

¿Por qué BTree y no BSTree? Un `BSTree` (árbol binario de *búsqueda*) exige un invariante de orden: izquierda < raíz < derecha. Un árbol de sintaxis *no* cumple eso — la posición de cada nodo la decide la expresión, no un orden de claves. La estructura correcta para heredar es el `BTree` a secas.

3. La especificación en Nereus

En Nereus describimos el tipo con **constructores** (cómo se arma un valor) y **operaciones** (qué se puede preguntar sobre él), y damos su significado con **axiomas** (ecuaciones).

Los dos constructores propios

BASIC CONSTRUCTORS leaf, node

```
-- una hoja: guarda un operando
leaf: String -> ExprTree;

-- un nodo operador con dos hijos; La precondition
-- obliga a que el símbolo sea uno de los cinco operadores
node: String(s) * ExprTree * ExprTree -> ExprTree
  pre: (s = "+") or (s = "-") or (s = "*") or (s = "/")
      or (s = "**");
```

Y cada constructor se *define* en términos de los que se heredan de `BTree`:

```
leaf(x)      = BTree_Create(BTree_Empty(), BTree_Empty(), x);
node(s, l, r) = BTree_Create(l, r, s);
```

Es decir: una hoja es un `BTree` con dos subárboles vacíos cuya etiqueta es el operando; un nodo es un `BTree` cuya etiqueta es el símbolo del operador y cuyos hijos son las dos sub-expresiones.

Cuatro operaciones para observar el árbol

Operación	Qué responde
<code>isLeaf(t)</code>	¿es <code>t</code> una hoja (un operando) o un nodo operador?
<code>numOperands(t)</code>	cuántos operandos (hojas) tiene el árbol
<code>numOperators(t)</code>	cuántos operadores (nodos internos) tiene
<code>render(t)</code>	reconstruye la expresión aritmética parentizada

Cada operación se define con dos axiomas: uno para la hoja y otro para el nodo.

```
isLeaf(leaf(x))      = True;
isLeaf(node(s, l, r)) = False;

numOperands(leaf(x))      = 1;
numOperands(node(s, l, r)) = numOperands(l) + numOperands(r);

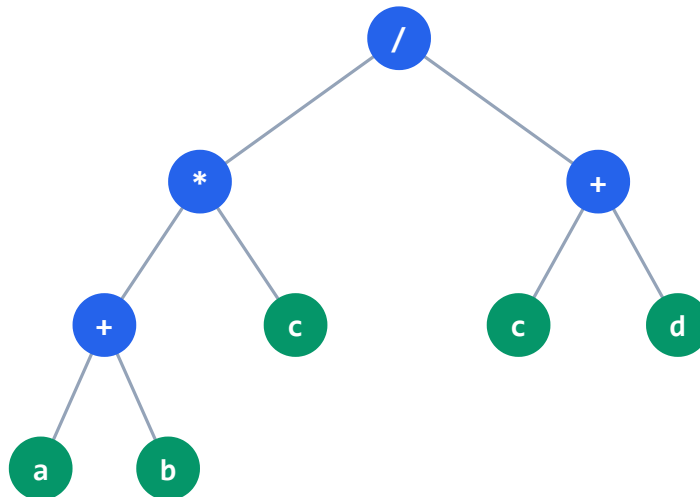
numOperators(leaf(x))      = 0;
numOperators(node(s, l, r)) = 1 + numOperators(l) + numOperators(r);

render(leaf(x))      = x;
render(node(s, l, r)) = "(" + render(l) + s + render(r) + ");
```

La idea clave: los axiomas son *recursivos*. Cada uno resuelve el caso de la hoja directamente, y el caso del nodo lo reduce a la misma pregunta sobre los dos hijos. Como el árbol es finito, la recursión siempre termina en hojas.

4. El ejemplo: $((a+b)*c)/(c+d)$

Veamos cómo esa expresión se transforma en un árbol. Leyéndola "de afuera hacia adentro", la última operación en aplicarse es la división $/$: ella es la **raíz**. Su hijo izquierdo es $(a+b)*c$ y el derecho es $c+d$. Repitiendo el razonamiento en cada sub-expresión llegamos a este árbol:



Árbol de sintaxis de $((a+b)*c)/(c+d)$. En azul los operadores; en verde los operandos.

Escrito como término Nereus, ese árbol es:

```
node("/",
  node("*",
    node("+", leaf("a"), leaf("b")),
    leaf("c")),
  node("+", leaf("c"), leaf("d")))
```

Observá que **el término y el árbol son lo mismo**: cada `node` es un nodo azul y cada `leaf` es una hoja verde.

5. `render`: del árbol al string

La operación `render` recorre el árbol y vuelve a armar la expresión con paréntesis. Apliquemos sus axiomas paso a paso, de adentro hacia afuera:

```
render(leaf("a")) = "a"    render(leaf("b")) = "b"
render(node("+", leaf "a", leaf "b")) = "(" + "a" + "+" + "b" + ")" = "(a+b)"
render(node("*", ..., leaf "c")) = "(" + "(a+b)" + "*" + "c" + ")" = "((a+b)*c)"
render(node("+", leaf "c", leaf "d")) = "(" + "c" + "+" + "d" + ")" = "(c+d)"
render(node("/", ..., ...)) = "(" + "((a+b)*c)" + "/" + "(c+d)" + ")"
                             = "(((a+b)*c)/(c+d))"
```

El resultado, `((a+b)*c)/(c+d)`, es la forma **canónica totalmente parentizada**: *toda* operación queda entre paréntesis, incluso la división de la raíz. El par de paréntesis más externo es redundante — por eso solemos escribir `((a+b)*c)/(c+d)` — pero incluirlo hace que cada subexpresión tenga exactamente la misma forma, y eso simplifica los axiomas.

Cuidado con el doble sentido del `+`. En el axioma `render(node(s,l,r)) = "(" + render(l) + s + render(r) + ")"` conviven dos `+` que *no* son lo mismo:

- el `+` que une los trozos del resultado es la **concatenación de cadenas**: todos sus argumentos son `String`, así que Nereus lo interpreta como el `+` del tipo `String`, no como una suma;
- el `+` *operador aritmético* que el árbol representa no aparece aquí como operación: viaja como **dato**, es el valor `"+"` guardado en un nodo (la variable `s`).

Por eso, cuando `s = "+"`, `render` simplemente *pega* ese carácter como texto entre los dos lados; nunca suma nada. `render` **imprime** la expresión, no la **evalúa**: la aritmética no se calcula.

La potencia `**` no necesita ningún axioma propio: como `render` intercala el símbolo `s` sea cual sea, un nodo `**` se imprime igual que cualquier otro. Por ejemplo, la expresión `a**2 + b` — árbol `node("+", node("**", leaf("a"), leaf("2")), leaf("b"))` — se reconstruye así:

```
render(node("**", leaf "a", leaf "2")) = "(" + "a" + "**" + "2" + ")" = "(a**2)"
render(node("+", ..., leaf "b"))      = "(" + "(a**2)" + "+" + "b" + ")" = "((a**2)+b)"
```

Que `**` se escriba con *dos* caracteres es irrelevante para el árbol: el símbolo es un `String` cualquiera y `render` lo concatena entero. Lo único que distingue a `**` de `+` `-` `*` `/` es estar listado en la precondition de `node`; la estructura del árbol binario no cambia en absoluto.

6. Contar operandos y operadores

Las operaciones `numOperands` y `numOperators` también recorren el árbol recursivamente. Para nuestro ejemplo:

Subárbol	numOperands	numOperators
<code>node("+", leaf a, leaf b)</code>	<code>1 + 1 = 2</code>	<code>1 + 0 + 0 = 1</code>
<code>node("**", ..., leaf c)</code>	<code>2 + 1 = 3</code>	<code>1 + 1 + 0 = 2</code>
<code>node("+", leaf c, leaf d)</code>	<code>1 + 1 = 2</code>	<code>1 + 0 + 0 = 1</code>
raíz <code>node("/", ...)</code>	<code>3 + 2 = 5</code>	<code>1 + 2 + 1 = 4</code>

La expresión tiene **5 operandos** (las hojas `a`, `b`, `c`, `c`, `d`) y **4 operadores** (`+`, `*`, `/`, `+`). Contar las hojas y los nodos del dibujo confirma ambos números.

7. Para practicar

1. Dibujá el árbol de sintaxis de $(a*b)+(c*d)$ y escribí su término con `leaf` y `node`.
2. Para el árbol del ejercicio anterior, calculá a mano `render`, `numOperands` y `numOperators`.
3. ¿Por qué la precondition de `node` es importante? ¿Qué pasaría si permitiéramos `node("?", 1, r)`?
4. El árbol de $a+b+c$ no es único: puede ser $(a+b)+c$ o $a+(b+c)$. Dibujá los dos árboles y comprobá que `render` los distingue.
5. Dibujá el árbol de sintaxis de $(a**2)+(b**2)$ (suma de cuadrados), escribí su término con `leaf` y `node` y calculá a mano `render`, `numOperands` y `numOperators`.