ÁRBOLES DE EXPRESIONES

Separata para estudiantes de programación (C#)

Elaborado por: Docente — Especialista en C# y Estructuras de Datos

Objetivos de aprendizaje

- Comprender el concepto de árbol de expresiones y su relación con las notaciones infija, prefija y posfija.
- Dominar la precedencia y asociatividad de operadores (+, −, ×, ÷) y el rol de los paréntesis en el análisis sintáctico.
- Construir árboles de expresiones a partir de expresiones infijas utilizando el algoritmo de Shunting-Yard y/o pilas.
- Evaluar árboles de expresiones mediante recorridos (postorden) y aplicar comprobaciones de validez.
- Implementar en C# una representación de árbol de expresiones, un analizador (parser) y un evaluador seguro.
- Conocer el uso de System.Linq.Expressions para generar y compilar árboles de expresiones en C#.

1. Definición y contexto

Un árbol de expresiones es una representación en forma de árbol binario (para operadores binarios) de una expresión aritmética. Cada nodo interno corresponde a un operador y cada hoja corresponde a un operando (número o variable). La estructura captura explícitamente la jerarquía impuesta por la precedencia de operadores y por los paréntesis, evitando ambigüedades propias de la notación infija.

Notaciones: la misma expresión puede escribirse en tres formas equivalentes:

- Infija: los operadores se ubican entre operandos. Ej.: 3 + 4 × 2.
- Prefija (Polaca): los operadores preceden a sus operandos. Ej.: + 3 × 4 2.
- Posfija (Polaca inversa): los operadores siguen a sus operandos. Ej.: 3 4 2 × +.

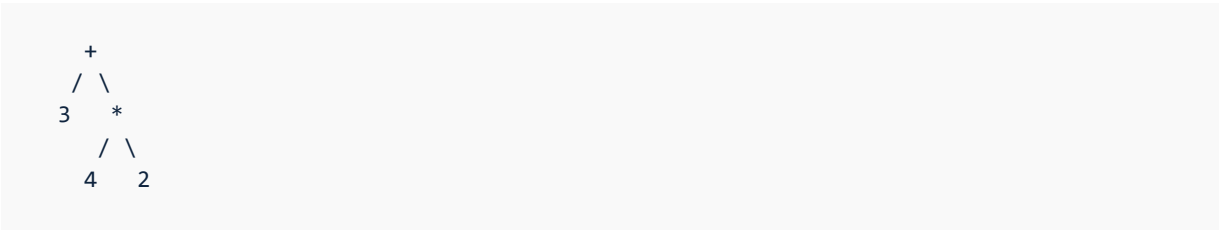
2. Precedencia y asociatividad

Para operadores básicos: la multiplicación (*) y la división (/) tienen mayor precedencia que la suma (+) y la resta (−). La asociatividad para estos operadores binarios suele ser de izquierda a derecha. Los paréntesis alteran explícitamente la precedencia obligando a evaluar primero las subexpresiones entre paréntesis.

Operador	Precedencia	Asociatividad
()	Más alta	N/A
*****	Alta	Izquierda
/	Alta	Izquierda
+	Baja	Izquierda
-	Baja	Izquierda

3. Árboles de expresiones: construcción y lectura

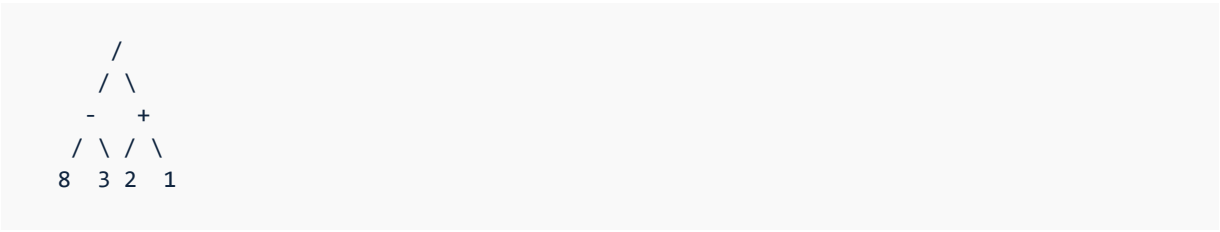
Ejemplo 1: Expresión infija 3 + 4 * 2



Recorridos:

- Prefijo (preorden): + 3 * 4 2
- Infijo (inorden con paréntesis): (3 + (4 * 2))
- Posfijo (postorden): 3 4 2 * +

Ejemplo 2: Expresión infija (8 - 3) / (2 + 1)



Recorridos:

- Prefijo: / - 8 3 + 2 1
- Infijo: ((8 - 3) / (2 + 1))
- Posfijo: 8 3 - 2 1 + /

Ejemplo 3: Expresión infija 7 + (6 / 3) - 2

4. Construcción desde notación posfija (algoritmo con pila)

Dada una expresión en notación posfija, el árbol se construye con una pila de nodos:
1) Si el token es un operando, crear un nodo hoja y apilarlo.
2) Si el token es un operador binario, desapilar los dos nodos superiores (derecha, izquierda), crear un nodo operador con esos dos hijos y apilar el resultado.
3) Al finalizar, el tope de la pila es la raíz del árbol.

Ejemplo: posfija '3 4 2 * +'

```
Pila    Token    Acción
[]      3        push(3)
[3]     4        push(4)
[3,4]   2        push(2)
[3,4,2] *        pop->2, pop->4; make *(4,2); push(*)
[3,*]   +        pop->*, pop->3; make +(3, *); push(+)
[+]     (fin)    raíz = +
```

5. De infija a árbol: Shunting-Yard y construcción del árbol

El algoritmo de Dijkstra (Shunting-Yard) transforma una expresión infija en posfija respetando precedencia, asociatividad y paréntesis. Luego, a partir de la posfija, se aplica el algoritmo con pila del apartado anterior para obtener el árbol.

1. Para cada token de la expresión infija:
2. Si es número/variable: enviar a la salida.
3. Si es operador: mientras haya operadores en la pila con mayor o igual precedencia (y asociatividad izquierda), pasarlos a la salida; luego apilar el operador actual.
4. Si es '(': apilar.
5. Si es ')': desapilar hasta encontrar '(' (que se descarta).
6. Al final, vaciar la pila de operadores a la salida.

Ejemplo: infija '(8 - 3) / (2 + 1)' → posfija '8 3 - 2 1 + /' → árbol del apartado 3.

6. Evaluación del árbol (postorden)

- Si el nodo es hoja: retornar su valor.
- Si el nodo es '+': eval(left) + eval(right).
- Si el nodo es '−': eval(left) − eval(right).
- Si el nodo es '*': eval(left) * eval(right).
- Si el nodo es '/': validar división entre cero y retornar eval(left) / eval(right).

7. Implementación en C# (árbol propio + parser + evaluación)

Implementación didáctica en C#: tokenización, Shunting-Yard, construcción y evaluación.

```
// ------------------------------------------------------------
// Árbol de expresiones en C# (didáctico, .NET 6+ compatible)
// ------------------------------------------------------------
using System;
using System.Collections.Generic;
using System.Globalization;

namespace ExpressionTrees
{
    public abstract class Node
    {
        public abstract double Evaluate();
        public abstract string ToInfix();
    }

    public sealed class NumberNode : Node
    {
        public double Value { get; }
        public NumberNode(double value) => Value = value;
        public override double Evaluate() => Value;
        public override string ToInfix() => Value.ToString(CultureInfo.InvariantCulture);
    }

    public sealed class BinaryNode : Node
    {
        public char Op { get; }
        public Node Left { get; }
        public Node Right { get; }
        public BinaryNode(char op, Node left, Node right)
        {
            Op = op; Left = left; Right = right;
        }
        public override double Evaluate()
        {
            var a = Left.Evaluate();
            var b = Right.Evaluate();
            switch (Op)
            {
                case '+': return a + b;
                case '-': return a - b;
                case '*': return a * b;
                case '/': if (b == 0)
                                throw new DivideByZeroException("División entre cero.");
```

```
                        return a / b;
            default: throw new
                     InvalidOperationException($"Operador no soportado: {Op}");
        }
    }
    public override string ToInfix()
    {
        int prec = Precedence(Op);
        string L = WrapIfNeeded(Left, prec);
        string R = WrapIfNeeded(Right, prec, true, Op);
        return $"{L} {Op} {R}";
    }

    private static int Precedence(char op)
    {
        switch (op)
        {
            case '+':
            case '-': return 1;
            case '*':
            case '/': return 2;
            default: return 0;
        }
    }

    private static string WrapIfNeeded(Node child, int parentPrec,
            bool rightSide=false, char op='+')
    {
        if (child is BinaryNode bn)
        {
            int childPrec = Precedence(bn.Op);
            if (childPrec < parentPrec) return $"({child.ToInfix()})";
            if (childPrec == parentPrec && rightSide && (op == '-' || op == '/'))
                return $"({child.ToInfix()})";
        }
        return child.ToInfix();
    }
}

public static class Parser
{
    public static Node Parse(string expr)
    {
        var postfix = InfixToPostfix(Tokenize(expr));
        return BuildTree(postfix);
    }

    // 1) Tokenización simple
    private static IEnumerable<string> Tokenize(string s)
    {
        int i = 0;
        while (i < s.Length)
        {
            if (char.IsWhiteSpace(s[i])) { i++; continue; }
            if ("+-*/()".IndexOf(s[i]) >= 0)
            {
                yield return s[i++].ToString();
```

```
                }
                else if (char.IsDigit(s[i]) || s[i]=='.')
                {
                    int j = i;
                    while (j < s.Length && (char.IsDigit(s[j]) || s[j]=='.')) j++;
                    yield return s.Substring(i, j-i);
                    i = j;
                }
                else
                {
                    throw new ArgumentException($"Símbolo inválido: '{s[i]}'");
                }
            }
        }

        // 2) Shunting-Yard (infija -> posfija)
        private static Queue<string> InfixToPostfix(IEnumerable<string> tokens)
        {
            var output = new Queue<string>();
            var ops = new Stack<string>();

            Func<string,int> Prec = op =>
            {
                switch (op)
                {
                    case "+":
                    case "-": return 1;
                    case "*":
                    case "/": return 2;
                    default: return 0;
                }
            };

            Func<string,bool> IsLeftAssoc = op => true; // todos izquierda

            foreach (var tk in tokens)
            {
                double dummy;
                if (double.TryParse(tk, NumberStyles.Float,
                    CultureInfo.InvariantCulture, out dummy))
                {
                    output.Enqueue(tk);
                }
                else if ("+-*/".Contains(tk))
                {
                    while (ops.Count > 0 && "+-*/".Contains(ops.Peek()) and \
                           ((Prec(ops.Peek()) > Prec(tk)) or (Prec(ops.Peek()) == Prec(tk)
and IsLeftAssoc(tk))))
                    {
                        output.Enqueue(ops.Pop());
                    }
                    ops.Push(tk);
                }
                else if (tk == "(")
                {
                    ops.Push(tk);
                }
```

```
                else if (tk == ")")
                {
                    while (ops.Count > 0 and ops.Peek() != "(")
                        output.Enqueue(ops.Pop());
                    if (ops.Count == 0) throw new ArgumentException("Paréntesis
desbalanceados.");
                    ops.Pop(); // descartar '('
                }
            }
            while (ops.Count > 0)
            {
                var op = ops.Pop();
                if (op == "(" or op == ")") throw new ArgumentException("Paréntesis
desbalanceados.");
                output.Enqueue(op);
            }
            return output;
        }

        // 3) Construcción del árbol desde posfija
        private static Node BuildTree(Queue<string> postfix)
        {
            var stack = new Stack<Node>();
            while (postfix.Count > 0)
            {
                var tk = postfix.Dequeue();
                double num;
                if (double.TryParse(tk, NumberStyles.Float,
                    CultureInfo.InvariantCulture, out num))
                {
                    stack.Push(new NumberNode(num));
                }
                else if (tk.Length == 1 && "+-*/".Contains(tk))
                {
                    if (stack.Count < 2) throw new
                       ArgumentException("Expresión inválida.");
                    var right = stack.Pop();
                    var left = stack.Pop();
                    stack.Push(new BinaryNode(tk[0], left, right));
                }
                else
                {
                    throw new ArgumentException($"Token inesperado: {tk}");
                }
            }
            if (stack.Count != 1) throw new ArgumentException("Expresión inválida.");
            return stack.Pop();
        }
    }

    // Ejemplo de uso
    class Program
    {
        static void Main()
        {
            var root = Parser.Parse("(8 - 3) / (2 + 1)");
            Console.WriteLine(root.ToInfix());   // ((8 - 3) / (2 + 1))
```

```
            Console.WriteLine(root.Evaluate());  // 1.666...
        }
    }
}
```

8. Árboles de expresiones con System.Linq.Expressions (C#)

C# ofrece una API de árboles de expresiones en el espacio de nombres System.Linq.Expressions, que permite construir expresiones como árboles de objetos, inspeccionarlas y compilarlas en delegados ejecutables. A continuación, un ejemplo programático equivalente a (8 - 3) / (2 + 1).

```
using System;
using System.Linq.Expressions;

class LinqExpressionDemo
{
    static void Main()
    {
        var eight = Expression.Constant(8.0);
        var three = Expression.Constant(3.0);
        var two   = Expression.Constant(2.0);
        var one   = Expression.Constant(1.0);

        var minus = Expression.Subtract(eight, three);
        var plus  = Expression.Add(two, one);
        var div   = Expression.Divide(minus, plus);

        var lambda = Expression.Lambda<Func<double>>(div);
        var fn = lambda.Compile();
        Console.WriteLine(fn()); // 1.666...
    }
}
```

9. Ejemplos resueltos paso a paso

Ejemplo A: 3 + 4 * 2

7. Infija: 3 + 4 * 2
8. Posfija (Shunting-Yard): 3 4 2 * +
9. Árbol: como en el apartado 3, Ejemplo 1.
10. Evaluación: 3 + (4 * 2) = 11.

Ejemplo B: (5 + 2) * (9 - 4) / 3

11. Infija: (5 + 2) * (9 - 4) / 3
12. Posfija: 5 2 + 9 4 - * 3 /
13. Árbol: raíz '/', hijo izquierdo '*', hijo derecho '3'.
14. Evaluación: ((5 + 2) * (9 - 4)) / 3 = (7 * 5) / 3 = 35 / 3 ≈ 11.6667.

Ejemplo C: 7 + (6 / 3) - 2

15. Infija: 7 + (6 / 3) - 2
16. Posfija: 7 6 3 / + 2 -
17. Árbol: como en el apartado 3, Ejemplo 3.
18. Evaluación: 7 + (2) - 2 = 7.

10. Validación y errores comunes

- Paréntesis desbalanceados: debe detectarse durante Shunting-Yard.
- Tokens inválidos: cualquier símbolo no numérico y no operador debe provocar error.
- Bajo flujo de pila para operadores: insuficiencia de operandos.
- División entre cero: validar en evaluación.
- Formato numérico inválido (p. ej., múltiples puntos decimales): rechazar en tokenización.

11. Complejidad computacional

Tanto la conversión infija→posfija (Shunting-Yard) como la construcción del árbol y su evaluación se ejecutan en tiempo lineal O(n) respecto al número de tokens, y espacio O(n) por el uso de pilas y la estructura del árbol.

12. Ejercicios propuestos

19. Construya el árbol de expresiones y evalúe: 10 - 2 * 3.
20. Convierta a posfija y evalúe: (4 + 6) / (1 + 2).
21. Dibuje el árbol y dé los tres recorridos (prefijo, infijo, posfijo) de: 2 + 3 * (5 - 1).
22. Implemente en C# un método que valide paréntesis desbalanceados y localice el índice del error.

13. Referencias y fuentes verificables

- Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2007). Compilers: Principles, Techniques, and Tools (2nd ed.). Pearson. (Capítulos sobre análisis sintáctico y árboles de sintaxis).
- Dijkstra, E. W. (1961). 'Algol 60 translation' y notas sobre Shunting-Yard. Manuscritos EWD. Repositorios de la Universidad de Texas / Edsger W. Dijkstra Archive.
- Microsoft Learn. 'Expression trees (C#)'. Documentación oficial de System.Linq.Expressions.
- Knuth, D. E. (1997). The Art of Computer Programming, Vol. 1: Fundamental Algorithms (3rd ed.). Addison-Wesley.