

C# Y .NET 8

Parte 8. Evaluador de expresiones algebraicas

2024-07

Rafael Alberto Moreno Parra
ramsoftware@gmail.com

Contenido

Tabla de ilustraciones.....	4
Acerca del autor.....	5
Licencia de este libro	5
Licencia del software	5
Marcas registradas	6
Dedicatoria	7
El problema.....	8
Evaluador de expresiones iterativo. Versión 4.0.....	9
El algoritmo.....	10
Paso 1: Retirar caracteres que no sean de una expresión algebraica	10
Paso 2: Verificando la sintaxis de la expresión algebraica	11
Paso 3: Conversión de funciones a una letra mayúscula	14
Paso 4: Dividiendo la cadena en partes	16
Paso 5: Generando las Piezas desde las Partes	19
Paso 6: Evaluando a partir de las Piezas.....	20
El código completo	22
Como usar el evaluador de expresiones.....	36
Con números reales	38
Con paréntesis	39
Con funciones	40
Con variables	41
Haciendo múltiples cálculos	42
Validando la sintaxis.....	44
Métricas: Desempeño con el evaluador interno de C#.....	48
Evaluador de expresiones usando un árbol binario. Programación recursiva	53
Fase 1. Sumas y restas	54
Fase 2. Multiplicación y división	64
Fase 3. Potencia	69
Fase 4. Paréntesis	74
Fase 5. Variables	80
Fase 6. Funciones matemáticas.....	87
Fase 7. Orientado a objetos.....	97
Fase 8. Evaluación de sintaxis y optimización	106

Comparativa de desempeño entre evaluadores 118

Tabla de ilustraciones

Ilustración 1: Uso del evaluador de expresiones	36
Ilustración 2: Resultado obtenido con el evaluador	37
Ilustración 3: Operando números reales.....	38
Ilustración 4: Operando con paréntesis.....	39
Ilustración 5: Operando con funciones.....	40
Ilustración 6: Operando con uso de variables.....	41
Ilustración 7: Generar múltiples valores de una ecuación al cambiar los valores.	43
Ilustración 8: Evaluando la sintaxis	45
Ilustración 9: Evaluando la sintaxis	46
Ilustración 10: Evaluando la sintaxis	47
Ilustración 11: Métrica compara ambos evaluadores	51
Ilustración 12: Métrica compara ambos evaluadores	52
Ilustración 13: Métrica compara ambos evaluadores	52
Ilustración 14: Árbol binario generado	55
Ilustración 15: Árbol binario generado	56
Ilustración 16: Ejemplo de ejecución del programa	59
Ilustración 17: Interpretación en viz-js.com	59
Ilustración 18: Resultado al evaluar	62
Ilustración 19: Validado con la calculadora	63
Ilustración 20: Árbol binario generado	63
Ilustración 21: Árbol binario generado	64
Ilustración 22: Ejecución del evaluador.....	68
Ilustración 23: Prueba con calculadora.....	68
Ilustración 24: Árbol binario generado	69
Ilustración 25: Ejemplo de ejecución	73
Ilustración 26: Prueba con la calculadora	73
Ilustración 27: Árbol binario generado	74
Ilustración 28: Árbol binario generado	85
Ilustración 29: Ejemplo de ejecución	86
Ilustración 30: Ejemplo de ejecución	95
Ilustración 31: Árbol binario generado	96
Ilustración 32: Ejemplo de ejecución	105
Ilustración 33: Ejemplo de ejecución	117
Ilustración 34: Comparativa de desempeño	123
Ilustración 35: Comparativa de desempeño	123
Ilustración 36: Comparativa de desempeño	123
Ilustración 37: Comparativa de desempeño	124

Acerca del autor

Rafael Alberto Moreno Parra

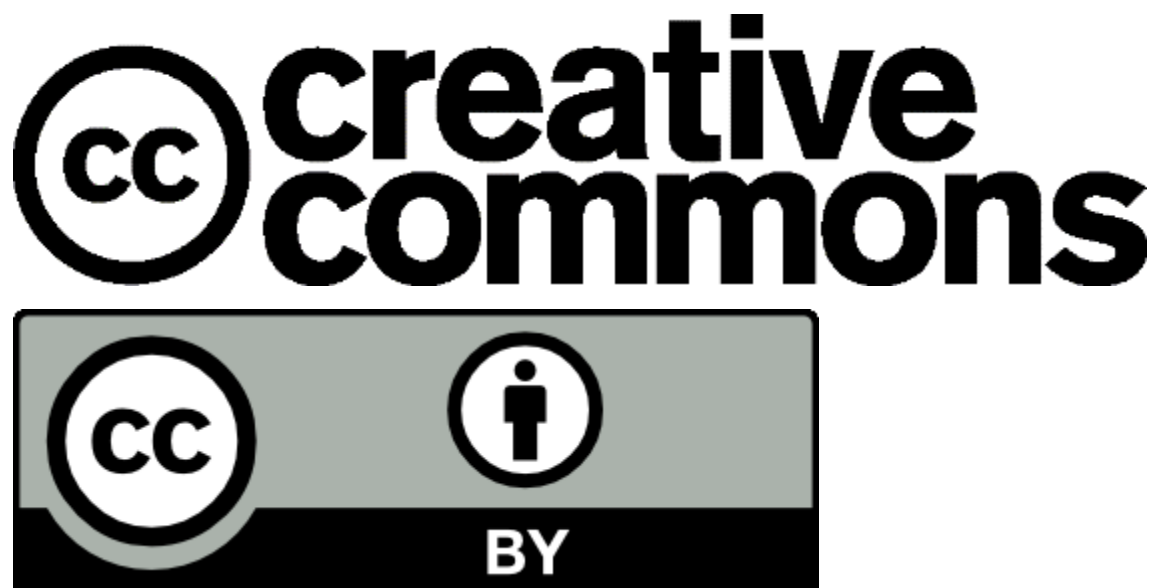
ramsoftware@gmail.com o enginelifelife@hotmail.com

Sitio Web: <http://darwin.50webs.com> (dedicado a la investigación de algoritmos evolutivos y vida artificial).

Github: <https://github.com/ramsoftware>

Youtube: <https://www.youtube.com/@RafaelMorenoP>

Licencia de este libro



Licencia del software

Todo el software desarrollado aquí tiene licencia LGPL “Lesser General Public License” [1]



Marcas registradas

En este libro se hace uso de las siguientes tecnologías registradas:

Microsoft ® Windows ® Enlace: <http://windows.microsoft.com/en-US/windows/home>

Microsoft ® Visual Studio 2022 ® Enlace: <https://visualstudio.microsoft.com/es/vs/>

Dedicatoria

A mis padres, a mi hermana....

Y a mi tropa gatuna: Sally, Suini, Grisú, Capuchina, Milú,
Arián, Frac y mis recordados Tinita, Tammy, Vikingo y
Michu.

El problema

Dada una expresión algebraica almacenada en una cadena "string", esta debe ser interpretada para devolver un valor real. Ejemplo:

Cadena: "3*4+1" → Resultado: 13

Hay que considerar que las expresiones algebraicas pueden tener:

1. Números reales
2. Variables (de la a .. z)
3. Operadores (suma, resta, multiplicación, división, potencia)
4. Uso de paréntesis
5. Uso de funciones (seno, coseno, tangente, valor absoluto, arcoseno, arcocoseno, arcotangente, logaritmo natural, valor techo, exponencial, raíz cuadrada).

Evaluador de expresiones iterativo. Versión 4.0

El algoritmo

Esta es la versión 4.0 del evaluador de expresiones.

La expresión algebraica está almacenada en una variable de tipo cadena (string). Por ejemplo:

```
string Cadena = "0.004 - (1.78 / 3.45 + h) * sen(k ^ x)";
```

La expresión algebraica puede tener números reales, operadores, paréntesis, variables y funciones. Luego hay que interpretarla y evaluarla siguiendo las estrictas reglas matemáticas.

Paso 1: Retirar caracteres que no sean de una expresión algebraica

Como la expresión algebraica ha sido digitada por un usuario final entonces se quita cualquier caracter(char) que no sea parte de una expresión algebraica. Los únicos caracteres permitidos son: "abcdefghijklmnopqrstuvwxyz0123456789.+*/^()". Esa parte la hace este código:

```
private int ChequeaSintaxis(string ExpOrig) {  
    /* Primero a minúsculas */  
    StringBuilder Minusculas = new StringBuilder(ExpOrig.ToLower());  
  
    /* Sólo los caracteres válidos */  
    string Valido = "abcdefghijklmnopqrstuvwxyz0123456789.+*/^()";  
    HashSet<char> Permite = new HashSet<char>(Valido);  
    StringBuilder ConLetrasValidas = new StringBuilder("(");  
    for (int Cont = 0; Cont < Minusculas.Length; Cont++)  
        if (Permite.Contains(Minusculas[Cont]))  
            ConLetrasValidas.Append(Minusculas[Cont]);  
    ConLetrasValidas.Append(')');  
}
```

Se agrega un paréntesis que abre al inicio y luego un paréntesis al final.

De ser: "0.004-(1.78/3.45+h)*sen(k^x)"

Pasa a: "(0.004-(1.78/3.45+h)*sen(k^x))"

Es necesario ese paso porque el evaluador busca los paréntesis para extraer la expresión interna. Se hace uso de StringBuilder para hacerlo más rápido. El uso de la estructura HashSet es como filtro para que sólo pasen los caracteres permitidos. Al final la variable "ConLetrasValidas" es la que tiene la expresión limpia.

Paso 2: Verificando la sintaxis de la expresión algebraica

Luego se verifica si la expresión cumple con las estrictas reglas sintácticas del álgebra. Se hacen 26 validaciones que son:

1. Un número seguido de una letra. Ejemplo: $2q-(\ast 3)$
2. Un número seguido de un paréntesis que abre. Ejemplo: $7-2(5-6)$
3. Doble punto seguido. Ejemplo: $3..1$
4. Punto seguido de operador. Ejemplo: $3.\ast 1$
5. Un punto y sigue una letra. Ejemplo: $3+5.w-8$
6. Punto seguido de paréntesis que abre. Ejemplo: $2-5.(4+1)\ast 3$
7. Punto seguido de paréntesis que cierra. Ejemplo: $2-(5.)\ast 3$
8. Un operador seguido de un punto. Ejemplo: $2-(4+.1)-7$
9. Dos operadores estén seguidos. Ejemplo: $2++4, 5-\ast 3$
10. Un operador seguido de un paréntesis que cierra. Ejemplo: $2-(4+)-7$
11. Una letra seguida de número. Ejemplo: $7-2a-6$
12. Una letra seguida de punto. Ejemplo: $7-a.-6$
13. Un paréntesis que abre seguido de punto. Ejemplo: $7-(.4-6)$
14. Un paréntesis que abre seguido de un operador. Ejemplo: $2-(\ast 3)$
15. Un paréntesis que abre seguido de un paréntesis que cierra. Ejemplo: $7-()-6$
16. Un paréntesis que cierra y sigue un número. Ejemplo: $(3-5)7$
17. Un paréntesis que cierra y sigue un punto. Ejemplo: $(3-5).$
18. Un paréntesis que cierra y sigue una letra. Ejemplo: $(3-5)t$
19. Un paréntesis que cierra y sigue un paréntesis que abre. Ejemplo: $(3-5)(4\ast 5)$
20. Hay dos o más letras seguidas (obviando las funciones)
21. Los paréntesis están desbalanceados. Ejemplo: $3-(2\ast 4))$
22. Doble punto en un número de tipo real. Ejemplo: $7-6.46.1+2$
23. Paréntesis que abre no corresponde con el que cierra. Ejemplo: $2+3)-2\ast (4 ,$
24. Inicia con operador. Ejemplo: $+3\ast 5$
25. Finaliza con operador. Ejemplo: $3\ast 5\ast$
26. Letra seguida de paréntesis que abre (obviando las funciones). Ejemplo: $4\ast a(6-2)$

Para hacer esto, se genera una copia de la expresión , y en esa copia se convierten las funciones (como seno, coseno, tangente) en la misma cadena "a+(", esto es para validar.

```
/* Validación de sintaxis, se genera una copia
 * y allí se reemplaza las funciones por a+( */
StringBuilder sbSintax = new StringBuilder();
sbSintax.Append(ConLetrasValidas);
sbSintax.Replace("sen(", "a+(");
sbSintax.Replace("cos(", "a+(");
sbSintax.Replace("tan(", "a+(");
sbSintax.Replace("abs(", "a+(");
sbSintax.Replace("asn(", "a+(");
sbSintax.Replace("acs(", "a+(");
sbSintax.Replace("atn(", "a+(");
sbSintax.Replace("log(", "a+(");
sbSintax.Replace("exp(", "a+(");
sbSintax.Replace("sqr(", "a+(");
```

Con esta técnica, si el usuario digitó una función que no existe, el programa encontrará este error. El siguiente código hace las 26 validaciones:

```
for (int Cont = 1; Cont < sbSintax.Length - 2; Cont++) {
    char cA = sbSintax[Cont];
    char cB = sbSintax[Cont + 1];

    if (Char.IsDigit(cA) && Char.IsLower(cB)) return 1;
    if (Char.IsDigit(cA) && cB == '(') return 2;
    if (cA == '.' && cB == '.') return 3;
    if (cA == '.' && EsUnOperador(cB)) return 4;
    if (cA == '.' && Char.IsLower(cB)) return 5;
    if (cA == '.' && cB == '(') return 6;
    if (cA == '.' && cB == ')') return 7;
    if (EsUnOperador(cA) && cB == '.') return 8;
    if (EsUnOperador(cA) && EsUnOperador(cB)) return 9;
    if (EsUnOperador(cA) && cB == ')') return 10;
    if (Char.IsLower(cA) && Char.IsDigit(cB)) return 11;
    if (Char.IsLower(cA) && cB == '.') return 12;
    if (Char.IsLower(cA) && Char.IsLower(cB)) return 13;
    if (Char.IsLower(cA) && cB == '(') return 14;
    if (cA == '(' && cB == '.') return 15;
    if (cA == '(' && EsUnOperador(cB)) return 16;
    if (cA == '(' && cB == ')') return 17;
    if (cA == ')' && Char.IsDigit(cB)) return 18;
    if (cA == ')' && cB == '.') return 19;
    if (cA == ')' && Char.IsLower(cB)) return 20;
    if (cA == ')' && cB == '(') return 21;
```

```

}

/* Valida el inicio y fin de la expresión */
if (EsUnOperador(sbSintax[1])) return 22;
if (EsUnOperador(sbSintax[sbSintax.Length - 2])) return 23;

/* Valida balance de paréntesis */
int ParentesisAbre = 0; /* Contador de paréntesis que abre */
int ParentesisCierra = 0; /* Contador de paréntesis que cierra */
for (int Cont = 1; Cont < sbSintax.Length - 1; Cont++) {
    switch (sbSintax[Cont]) {
        case '(': ParentesisAbre++; break;
        case ')': ParentesisCierra++; break;
    }
    if (ParentesisCierra > ParentesisAbre) return 24;
}
if (ParentesisAbre != ParentesisCierra) return 25;

/* Validar los puntos decimales de un número real */
int TotalPuntos = 0;
for (int Cont = 0; Cont < sbSintax.Length; Cont++) {
    if (EsUnOperador(sbSintax[Cont])) TotalPuntos = 0;
    if (sbSintax[Cont] == '.') TotalPuntos++;
    if (TotalPuntos > 1) return 26;
}
}

```

Se hace uso de esta función:

```

/* Retorna true si el Caracter es un operador matemático */
private static bool EsUnOperador(char Caracter) {
    switch (Caracter) {
        case '+':
        case '-':
        case '*':
        case '/':
        case '^':
            return true;
        default:
            return false;
    }
}

```

Si no hay ningún error sintáctico, entonces el programa retorna el valor cero.

Paso 3: Conversión de funciones a una letra mayúscula

Una vez ha pasado la prueba de sintaxis, se reemplazan las funciones (que son de tres letras) por una letra mayúscula. Esta es la tabla de conversión:

Función	Descripción	Letra con que se reemplaza
Sen	Seno	A
Cos	Coseno	B
Tan	Tangente	C
Abs	Valor absoluto	D
Asn	Arcoseno	E
Acs	Arcocoseno	F
Atn	Arcotangente	G
Log	Logaritmo Natural	H
Exp	Exponencial	I
Sqr	Raíz cuadrada	J

Estas son las instrucciones que lo hacen:

```
/* Deja la expresión para ser analizada.
 * Reemplaza las funciones de tres letras
 * por una letra mayúscula. Cambia los ))
 * por )+0) porque es requerido al crear las piezas */
this.Analizada.Length = 0;
this.Analizada.Append(ConLetrasValidas);
this.Analizada.Replace("sen", "A");
this.Analizada.Replace("cos", "B");
this.Analizada.Replace("tan", "C");
this.Analizada.Replace("abs", "D");
this.Analizada.Replace("asn", "E");
this.Analizada.Replace("acs", "F");
this.Analizada.Replace("atn", "G");
```

```
this.Analizada.Replace("log", "H");  
this.Analizada.Replace("exp", "I");  
this.Analizada.Replace("sqr", "J");
```

Por ejemplo, una expresión como:

$(1.81 - \exp(1.30 + 1.95 + \log(1.70 - 1.98)) - (1.1) * \operatorname{asn}(1.100 - 1.84 * 1.19) - 1.58 / \operatorname{atn}(1.50 - 1.40))$

Al convertirla queda así:

$(1.81 - I(1.30 + 1.95 + H(1.70 - 1.98) + 0) - (1.1) * E(1.100 - 1.84 * 1.19) - 1.58 / G(1.50 - 1.40) + 0)$

Paso 4: Dividiendo la cadena en partes

Se toma la cadena y se divide en partes diferenciadas dentro de un LIST: número real, operador, variable, paréntesis que abre, paréntesis que cierra y función. Por ejemplo:

"(0.4-(1.7/3.4+h)*A(k^x)+0)"

Queda en:

(0.4	-	(1.7	/	3.4	+	h)	*	A	(k	^	x)	+	0)
---	-----	---	---	-----	---	-----	---	---	---	---	---	---	---	---	---	---	---	---	---

Las variables y números constantes quedan en una lista dinámica llamada Valores de tipo double (las variables guardan sus respectivos valores allí). De esa forma cuando el evaluador necesite un valor para consultarlo o modificarlo, lo busca directamente en esa lista. Este cambio algorítmico da como resultado, que esta versión del evaluador (la 4.0), sea más rápida que la versión anterior del evaluador de expresiones.

Lista **Valores**

Posición 0 a 25	26	27	28	29
Variables de la expresión	0.4	1.7	3.4	0

Por lo que la lista de partes queda ahora así:

([26]	-	([27]	/	[28]	+	[7])	*	A	([10]	^	[23])	+	[29])
---	------	---	---	------	---	------	---	-----	---	---	---	---	------	---	------	---	---	------	---

Lo que está entre [] es la posición del valor en la lista de Valores.

En C# esta sería la lista de Valores

```
private List<double> Valores = new List<double>();
```


Las partes se guardan aquí:

```
/* Listado de partes en que se divide la expresión
Toma una expresión, por ejemplo:
1.68 + sen( 3 / x ) * ( 2.92 - 9 )
y la divide en partes así:
[1.68] [+] [sen()] [3] [/] [x] []] [*] [(] [2.92] [-] [9] []]
Cada parte puede tener un número, un operador, una función,
un paréntesis que abre o un paréntesis que cierra.
En esta versión 4.0, las constantes, piezas y variables guardan
sus valores en la lista Valores.
En partes, se almacena la posición en Valores */
private List<ParteEvl4> Partes = new List<ParteEvl4>();
```

Y la clase de las partes es así:

```
internal class ParteEvl4 {
    /* Constantes de los diferentes tipos
    * de datos que tendrán las piezas */
    private const int ESFUNCION = 1;
    private const int ESPARABRE = 2;
    private const int ESOPERADOR = 4;
    private const int ESNUMERO = 5;
    private const int ESVARIABLE = 6;

    /* Acumulador, función, paréntesis que abre,
    * paréntesis que cierra, operador,
    * número, variable */
    public int Tipo;

    /* Código de la función 0:seno, 1:coseno, 2:tangente,
    * 3: valor absoluto, 4: arcoseno, 5: arcocoseno,
    * 6: arcotangente, 7: logaritmo natural, 8: valor tope,
    * 9: exponencial, 10: raíz cuadrada */
    public int Funcion;

    /* + suma - resta * multiplicación / división ^ potencia */
    public int Operador;

    /* Posición en lista de valores del número literal
    * por ejemplo: 3.141592 */
    public int posNumero;

    /* Posición en lista de valores del
    * valor de la variable algebraica */
```

```

public int posVariable;

/* Posición en lista de valores del valor de la pieza.
 * Por ejemplo:
 * 3 + 2 / 5 se convierte así:
 * |3| |+| |2| | / | |5|
 * |3| |+| |A| A es un identificador de acumulador */
public int posAcumula;

public ParteEv14(int TipoParte, int Valor) {
    this.Tipo = TipoParte;
    switch (TipoParte) {
        case ESFUNCION: this.Funcion = Valor; break;
        case ESNUMERO: this.posNumero = Valor; break;
        case ESVARIABLE: this.posVariable = Valor; break;
        case ESPARABRE: this.Funcion = -1; break;
    }
}

public ParteEv14(char Operador) {
    this.Tipo = ESOPERADOR;
    switch (Operador) {
        case '+': this.Operador = 0; break;
        case '-': this.Operador = 1; break;
        case '*': this.Operador = 2; break;
        case '/': this.Operador = 3; break;
        case '^': this.Operador = 4; break;
    }
}
}

```

Paso 5: Generando las Piezas desde las Partes

Las Piezas tienen esta estructura: [Pieza] Función Parte1 Operador Parte2

Esta sería la clase que representa las piezas:

```
internal class PiezaEvl4 {  
    /* Posición donde se almacena el valor que genera  
    * la pieza al evaluarse */  
    public int PosResultado;  
  
    /* Código de la función 0:seno, 1:coseno, 2:tangente,  
    * 3: valor absoluto, 4: arcoseno, 5: arcocoseno,  
    * 6: arcotangente, 7: logaritmo natural, 8: valor tope,  
    * 9: exponencial, 10: raíz cuadrada, 11: raíz cúbica */  
    public int Funcion;  
  
    /* Posición donde se almacena la primera parte de la pieza */  
    public int pA;  
  
    /* 0 suma 1 resta 2 multiplicación 3 división 4 potencia */  
    public int Operador;  
  
    /* Posición donde se almacena la segunda parte de la pieza */  
    public int pB;  
}
```

Esas Piezas se construyen desde las Partes. Se sigue el orden de evaluación de los paréntesis y operadores. De:

([26]	-	([27]	/	[28]	+	[7])	*	A	([10]	^	[23]))
---	------	---	---	------	---	------	---	-----	---	---	---	---	------	---	------	---	---

Queda así:

Pieza	Función	Parte1	Operador	Parte2
[29]	A	[10]	^	[23]
[30]		[27]	/	[28]
[31]		[30]	+	[7]
[32]		[31]	*	[29]
[33]		[26]	-	[32]

Paso 6: Evaluando a partir de las Piezas

Yendo de pieza en pieza se evalúa toda la expresión. Este método es el más crítico en cuanto a velocidad por lo que se ha optimizado. ¿Por qué? Lo usual es que se obtenga varios valores de la misma ecuación. Por ejemplo, para hacer un gráfico matemático de una ecuación $Y=F(X)$, hay que darle varios valores de X, la ecuación ya está analizada (convertida en piezas), sólo es cambiar el valor de X. Este evaluador se ha diseñado considerando que se requieren varios valores de la misma ecuación, por este motivo fue preferible sacrificar tiempo en el análisis (que la convierte en piezas) para darle gran velocidad en la ejecución.

A continuación, el código que evalúa la expresión ya analizada.

```
/* Evalúa la expresión convertida en piezas */
public double Evaluar() {
    double Resulta = 0;
    PiezaEvl4 tmp;

    /* Va de pieza en pieza */
    for (int Posicion = 0; Posicion < Piezas.Count; Posicion++) {
        tmp = Piezas[Posicion];

        switch (tmp.Operador) {
            case 0:
                Resulta = Valores[tmp.pA] + Valores[tmp.pB];
                break;
            case 1:
                Resulta = Valores[tmp.pA] - Valores[tmp.pB];
                break;
            case 2:
                Resulta = Valores[tmp.pA] * Valores[tmp.pB];
                break;
            case 3:
                Resulta = Valores[tmp.pA] / Valores[tmp.pB];
                break;
            default:
                Resulta = Math.Pow(Valores[tmp.pA], Valores[tmp.pB]);
                break;
        }

        switch (tmp.Funcion) {
            case 0: Resulta = Math.Sin(Resulta); break;
            case 1: Resulta = Math.Cos(Resulta); break;
            case 2: Resulta = Math.Tan(Resulta); break;
            case 3: Resulta = Math.Abs(Resulta); break;
            case 4: Resulta = Math.Asin(Resulta); break;
            case 5: Resulta = Math.Acos(Resulta); break;
            case 6: Resulta = Math.Atan(Resulta); break;
            case 7: Resulta = Math.Log(Resulta); break;
        }
    }
}
```

```
        case 8: Resulta = Math.Exp(Resulta); break;
        case 9: Resulta = Math.Sqrt(Resulta); break;
    }

    Valores[tmp.PosResultado] = Resulta;
}
return Resulta;
}
```

El código completo

Este es el código completo del evaluador de expresiones.

H/Eval4/Evaluador4.cs

```
using System.Globalization;
using System.Text;

/* Evaluador de expresiones versión 4 (enero de 2024)
 * Autor: Rafael Alberto Moreno Parra
 * Correo: ramsoftware@gmail.com ; enginelife@hotmail.com
 * URL: http://darwin.50webs.com
 * GitHub: https://github.com/ramsoftware
 * */

namespace ArbolBinarioEvaluador {

    internal class ParteEvl4 {

        /* Constantes de los diferentes tipos
         * de datos que tendrán las piezas */
        private const int ESFUNCION = 1;
        private const int ESPARABRE = 2;
        private const int ESOPERADOR = 4;
        private const int ESNUMERO = 5;
        private const int ESVARIABLE = 6;

        /* Acumulador, función, paréntesis que abre,
         * paréntesis que cierra, operador,
         * número, variable */
        public int Tipo;

        /* Código de la función 0:seno, 1:coseno, 2:tangente,
         * 3: valor absoluto, 4: arcoseno, 5: arcocoseno,
         * 6: arcotangente, 7: logaritmo natural, 8: exponencial
         * 9: raíz cuadrada */
        public int Funcion;

        /* + suma - resta * multiplicación / división ^ potencia */
        public int Operador;

        /* Posición en lista de valores del número literal
         * por ejemplo: 3.141592 */
        public int posNumero;

        /* Posición en lista de valores del
         * valor de la variable algebraica */
        public int posVariable;
    }
}
```

```

/* Posición en lista de valores del valor de la pieza.
 * Por ejemplo:
 * 3 + 2 / 5 se convierte así:
 * |3| |+| |2| | / | |5|
 * |3| |+| |A| A es un identificador de acumulador */
public int posAcumula;

public ParteEvl4(int TipoParte, int Valor) {
    this.Tipo = TipoParte;
    switch (TipoParte) {
        case ESFUNCION: this.Funcion = Valor; break;
        case ESNUMERO: this.posNumero = Valor; break;
        case ESVARIABLE: this.posVariable = Valor; break;
        case ESPARABRE: this.Funcion = -1; break;
    }
}

public ParteEvl4(char Operador) {
    this.Tipo = ESOPERADOR;
    switch (Operador) {
        case '+': this.Operador = 0; break;
        case '-': this.Operador = 1; break;
        case '*': this.Operador = 2; break;
        case '/': this.Operador = 3; break;
        case '^': this.Operador = 4; break;
    }
}

}

internal class PiezaEvl4 {
    /* Posición donde se almacena el valor que genera
     * la pieza al evaluarse */
    public int PosResultado;

    /* Código de la función 0:seno, 1:coseno, 2:tangente,
     * 3: valor absoluto, 4: arcoseno, 5: arcocoseno,
     * 6: arcotangente, 7: logaritmo natural, 8: valor tope,
     * 9: exponencial, 10: raíz cuadrada */
    public int Funcion;

    /* Posición donde se almacena la primera parte de la pieza */
    public int pA;

    /* 0 suma 1 resta 2 multiplicación 3 división 4 potencia */
    public int Operador;
}

```

```

    /* Posición donde se almacena la segunda parte de la pieza */
    public int pB;
}

internal class Evaluador4 {
    /* Constantes de los diferentes tipos
     * de datos que tendrán las piezas */
    private const int ESFUNCION = 1;
    private const int ESPARABRE = 2;
    private const int ESPARCIERRA = 3;
    private const int ESOPERADOR = 4;
    private const int ESNUMERO = 5;
    private const int ESVARIABLE = 6;
    private const int ESACUMULA = 7;

    /* Expresión algebraica convertida de la
     * original dada por el usuario */
    private StringBuilder Analizada = new();

    /* Donde guarda los valores de variables, constantes y piezas */
    private List<double> Valores = new List<double>();

    /* Listado de partes en que se divide la expresión
     Toma una expresión, por ejemplo:
     1.68 + sen( 3 / x ) * ( 2.92 - 9 )
     y la divide en partes así:
     [1.68] [+] [sen() [3] [/] [x] []] [*] [(] [2.92] [-] [9] []]
     Cada parte puede tener un número, un operador, una función,
     un paréntesis que abre o un paréntesis que cierra.
     En esta versión 4.0, las constantes, piezas y variables guardan
     sus valores en la lista Valores.
     En partes, se almacena la posición en Valores */
    private List<ParteEvl4> Partes = new List<ParteEvl4>();

    /* Listado de piezas que ejecutan
     Toma las partes y las divide en piezas con
     la siguiente estructura:

     acumula = funcion valor operador valor

     donde valor puede ser un número, una variable o
     un acumulador

     Siguiendo el ejemplo anterior sería:
     A = 2.92 - 9
     B = sen( 3 / x )
     C = B * A
     D = 1.68 + C

```



```

    Esas piezas se evalúan de arriba a abajo y así
    se interpreta la ecuación */
private List<PiezaEvl4> Piezas = new List<PiezaEvl4>();

/* Analiza la expresión */
public int Analizar(string ExpresionOriginal) {
    Partes.Clear();
    Piezas.Clear();
    Valores.Clear();

    /* Hace espacio para las 26 variables que
     * puede manejar el evaluador */
    for (int Variables = 1; Variables <= 26; Variables++)
        Valores.Add(0);

    int Sintaxis = ChequeaSintaxis(ExpresionOriginal);
    if (Sintaxis == 0) {
        CrearPartes();
        CrearPiezas();
    }
    return Sintaxis;
}

/* Retorna mensaje de error sintáctico */
public string MensajeError(int CodigoError) {
    string Msj = "";
    switch (CodigoError) {
        case 1:
            Msj = "1. Número seguido de letra";
            break;

        case 2:
            Msj = "2. Número seguido de paréntesis que abre";
            break;

        case 3:
            Msj = "3. Doble punto seguido";
            break;

        case 4:
            Msj = "4. Punto seguido de operador";
            break;

        case 5:
            Msj = "5. Punto y sigue una letra";
            break;
    }
}

```

```
case 6:
    Msj = "6. Punto seguido de paréntesis que abre";
    break;

case 7:
    Msj = "7. Punto seguido de paréntesis que cierra";
    break;

case 8:
    Msj = "8. Operador seguido de un punto";
    break;

case 9:
    Msj = "9. Dos operadores estén seguidos";
    break;

case 10:
    Msj = "10. Operador seguido de paréntesis que cierra";
    break;

case 11:
    Msj = "11. Letra seguida de número";
    break;

case 12:
    Msj = "12. Letra seguida de punto";
    break;

case 13:
    Msj = "13. Letra seguida de otra letra";
    break;

case 14:
    Msj = "14. Letra seguida de paréntesis que abre";
    break;

case 15:
    Msj = "15. Paréntesis que abre seguido de punto";
    break;

case 16:
    Msj = "16. Paréntesis que abre y sigue operador";
    break;

case 17:
    Msj = "17. Paréntesis que abre y luego cierra";
    break;
```

```

    case 18:
        Msj = "18. Paréntesis que cierra y sigue número";
        break;

    case 19:
        Msj = "19. Paréntesis que cierra y sigue punto";
        break;

    case 20:
        Msj = "20. Paréntesis que cierra y sigue letra";
        break;

    case 21:
        Msj = "21. Paréntesis que cierra y luego abre";
        break;

    case 22:
        Msj = "22. Inicia con operador";
        break;

    case 23:
        Msj = "23. Finaliza con operador";
        break;

    case 24:
        Msj = "24. No hay correspondencia entre paréntesis";
        break;

    case 25:
        Msj = "25. Paréntesis desbalanceados";
        break;

    case 26:
        Msj = "26. Dos o más puntos en número real";
        break;
}
return Msj;
}

/* Da valor a las variables que tendrá
 * la expresión algebraica */
public void DarValorVariable(char varAlgebra, double Valor) {
    Valores[varAlgebra - 'a'] = Valor;
}

/* Evalúa la expresión convertida en piezas */
public double Evaluar() {
    double Resulta = 0;

```

```

PiezaEvl4 tmp;

/* Va de pieza en pieza */
for (int Posicion = 0; Posicion < Piezas.Count; Posicion++) {
    tmp = Piezas[Posicion];

    switch (tmp.Operador) {
        case 0:
            Resulta = Valores[tmp.pA] + Valores[tmp.pB];
            break;
        case 1:
            Resulta = Valores[tmp.pA] - Valores[tmp.pB];
            break;
        case 2:
            Resulta = Valores[tmp.pA] * Valores[tmp.pB];
            break;
        case 3:
            Resulta = Valores[tmp.pA] / Valores[tmp.pB];
            break;
        default:
            Resulta = Math.Pow(Valores[tmp.pA], Valores[tmp.pB]);
            break;
    }

    switch (tmp.Funcion) {
        case 0: Resulta = Math.Sin(Resulta); break;
        case 1: Resulta = Math.Cos(Resulta); break;
        case 2: Resulta = Math.Tan(Resulta); break;
        case 3: Resulta = Math.Abs(Resulta); break;
        case 4: Resulta = Math.Asin(Resulta); break;
        case 5: Resulta = Math.Acos(Resulta); break;
        case 6: Resulta = Math.Atan(Resulta); break;
        case 7: Resulta = Math.Log(Resulta); break;
        case 8: Resulta = Math.Exp(Resulta); break;
        case 9: Resulta = Math.Sqrt(Resulta); break;
    }

    Valores[tmp.PosResultado] = Resulta;
}
return Resulta;
}

/* Divide la expresión en partes, yendo de caracter en caracter */
private void CrearPartes() {
    StringBuilder Numero = new StringBuilder();
    for (int Pos = 0; Pos < this.Analizada.Length; Pos++) {
        char Letra = this.Analizada[Pos];
        switch (Letra) {

```

```

case '.':
case '0':
case '1':
case '2':
case '3':
case '4':
case '5':
case '6':
case '7':
case '8':
case '9':    /* Si es un dígito o un punto
              * va acumulando el número */
    Numero.Append(Letra); break;
case '+':
case '-':
case '*':
case '/':
case '^':
    /* Si es un operador matemático entonces verifica
     * si hay un número que se ha acumulado */
    if (Numero.Length > 0) {
        Valores.Add(Convierte(Numero));
        Partes.Add(new ParteEvl4(ESNUMERO, Valores.Count - 1));
        Numero.Clear();
    }
    /* Agregar el operador matemático */
    Partes.Add(new ParteEvl4(Letra));
    break;

case '(':    /* Es paréntesis que abre */
    Partes.Add(new ParteEvl4(ESPARABRE, 0));
    break;

case ')':
    /* Si es un paréntesis que cierra entonces verifica
     * si hay un número que se ha acumulado */
    if (Numero.Length > 0) {
        Valores.Add(Convierte(Numero));
        Partes.Add(new ParteEvl4(ESNUMERO, Valores.Count - 1));
        Numero.Clear();
    }

    /* Si sólo había un número o variable
     * dentro del paréntesis le agrega + 0
     * (por ejemplo: sen(x) o 3*(2) ) */
    if (Partes[Partes.Count - 2].Tipo == ESPARABRE ||
        Partes[Partes.Count - 2].Tipo == ESFUNCION) {
        Partes.Add(new ParteEvl4(ESOPERADOR, 0));
    }

```

```

        Valores.Add(0);
        Partes.Add(new ParteEvl4(ESNUMERO, Valores.Count - 1));
    }

    /* Adiciona el paréntesis que cierra */
    Partes.Add(new ParteEvl4(ESPARCIERRA, 0));
    break;
case 'A':    /* Seno */
case 'B':    /* Coseno */
case 'C':    /* Tangente */
case 'D':    /* Valor absoluto */
case 'E':    /* Arcoseno */
case 'F':    /* Arcocoseno */
case 'G':    /* Arcotangente */
case 'H':    /* Logaritmo natural */
case 'I':    /* Exponencial */
case 'J':    /* Raíz cuadrada */
    Partes.Add(new ParteEvl4(ESFUNCION, Letra - 'A'));
    Pos++;
    break;
default:
    Partes.Add(new ParteEvl4(ESVARIABLE, Letra - 'a'));
    break;
}
}
}

private double Convierte(StringBuilder Numero) {
    string Cad = Numero.ToString();
    return double.Parse(Cad, CultureInfo.InvariantCulture);
}

/* Convierte las partes en las piezas finales de ejecución */
private void CrearPiezas() {
    int Contador = Partes.Count - 1;
    do {
        ParteEvl4 tmpParte = Partes[Contador];
        if (tmpParte.Tipo == ESPARABRE || tmpParte.Tipo == ESFUNCION) {

            /* Evalúa las potencias */
            GeneraPiezaOpera(4, 4, Contador);

            /* Luego evalúa multiplicar y dividir */
            GeneraPiezaOpera(2, 3, Contador);

            /* Finalmente evalúa sumar y restar */
            GeneraPiezaOpera(0, 1, Contador);

```

```

    /* Agrega la función a la última pieza */
    if (tmpParte.Tipo == ESFUNCION) {
        Piezas[Piezas.Count - 1].Funcion = tmpParte.Funcion;
    }

    /* Quita el paréntesis/función que abre y
     * el que cierra, dejando el centro */
    Partes.RemoveAt(Contador);
    Partes.RemoveAt(Contador + 1);
}
Contador--;
} while (Contador >= 0);
}

/* Genera las piezas buscando determinado operador */
private void GeneraPiezaOpera(int OperA, int OperB, int Inicia) {
    int Contador = Inicia + 1;
    do {
        ParteEvl4 tmpParte = Partes[Contador];
        if (tmpParte.Tipo == ESOPERADOR &&
            (tmpParte.Operador == OperA || tmpParte.Operador == OperB)) {
            ParteEvl4 tmpParteIzq = Partes[Contador - 1];
            ParteEvl4 tmpParteDer = Partes[Contador + 1];

            /* Crea Pieza */
            PiezaEvl4 NuevaPieza = new PiezaEvl4();
            NuevaPieza.Funcion = -1;

            switch (tmpParteIzq.Tipo) {
                case ESNUMERO:
                    NuevaPieza.pA = tmpParteIzq.posNumero;
                    break;

                case ESVARIABLE:
                    NuevaPieza.pA = tmpParteIzq.posVariable;
                    break;

                default:
                    NuevaPieza.pA = tmpParteIzq.posAcumula;
                    break;
            }

            NuevaPieza.Operador = tmpParte.Operador;

            switch (tmpParteDer.Tipo) {
                case ESNUMERO:
                    NuevaPieza.pB = tmpParteDer.posNumero;
                    break;
            }
        }
    } while (Contador >= 0);
}

```

```

        case ESVARIABLE:
            NuevaPieza.pB = tmpParteDer.posVariable;
            break;

        default:
            NuevaPieza.pB = tmpParteDer.posAcumula;
            break;
    }

    /* Añade a lista de piezas y crea una
     * nueva posición en Valores */
    Valores.Add(0);
    NuevaPieza.PosResultado = Valores.Count - 1;
    Piezas.Add(NuevaPieza);

    /* Elimina la parte del operador y la siguiente */
    Partes.RemoveAt(Contador);
    Partes.RemoveAt(Contador);

    /* Retorna el contador en uno para tomar
     * la siguiente operación */
    Contador--;

    /* Cambia la parte anterior por parte que acumula */
    tmpParteIzq.Tipo = ESACUMULA;
    tmpParteIzq.posAcumula = NuevaPieza.PosResultado;
}
Contador++;
} while (Partes[Contador].Tipo != ESPARCIERRA);
}

private int ChequeaSintaxis(string ExpOrig) {
    /* Primero a minúsculas */
    StringBuilder Minusculas = new StringBuilder(ExpOrig.ToLower());

    /* Sólo los caracteres válidos */
    string Valido = "abcdefghijklmnopqrstuvwxyz0123456789.+-*/^() ";
    HashSet<char> Permite = new HashSet<char>(Valido);
    StringBuilder ConLetrasValidas = new StringBuilder("");
    for (int Cont = 0; Cont < Minusculas.Length; Cont++)
        if (Permite.Contains(Minusculas[Cont]))
            ConLetrasValidas.Append(Minusculas[Cont]);
    ConLetrasValidas.Append(')');

    /* Agrega +0) donde exista )) porque es
     * necesario para crear las piezas */
    string nuevo = ConLetrasValidas.ToString();
}

```



```

while (nuevo.IndexOf("))") != -1)
    nuevo = nuevo.Replace("))", ") + 0)");
ConLetrasValidas = new StringBuilder(nuevo);

/* Validación de sintaxis, se genera una copia
 * y allí se reemplaza las funciones por a+( */
StringBuilder sbSintax = new StringBuilder();
sbSintax.Append(ConLetrasValidas);
sbSintax.Replace("sen(", "a+(");
sbSintax.Replace("cos(", "a+(");
sbSintax.Replace("tan(", "a+(");
sbSintax.Replace("abs(", "a+(");
sbSintax.Replace("asn(", "a+(");
sbSintax.Replace("acs(", "a+(");
sbSintax.Replace("atn(", "a+(");
sbSintax.Replace("log(", "a+(");
sbSintax.Replace("exp(", "a+(");
sbSintax.Replace("sqr(", "a+(");

for (int Cont = 1; Cont < sbSintax.Length - 2; Cont++) {
    char cA = sbSintax[Cont];
    char cB = sbSintax[Cont + 1];

    if (Char.IsDigit(cA) && Char.IsLower(cB)) return 1;
    if (Char.IsDigit(cA) && cB == '(') return 2;
    if (cA == '.' && cB == '.') return 3;
    if (cA == '.' && EsUnOperador(cB)) return 4;
    if (cA == '.' && Char.IsLower(cB)) return 5;
    if (cA == '.' && cB == '(') return 6;
    if (cA == '.' && cB == ')') return 7;
    if (EsUnOperador(cA) && cB == '.') return 8;
    if (EsUnOperador(cA) && EsUnOperador(cB)) return 9;
    if (EsUnOperador(cA) && cB == ')') return 10;
    if (Char.IsLower(cA) && Char.IsDigit(cB)) return 11;
    if (Char.IsLower(cA) && cB == '.') return 12;
    if (Char.IsLower(cA) && Char.IsLower(cB)) return 13;
    if (Char.IsLower(cA) && cB == '(') return 14;
    if (cA == '(' && cB == '.') return 15;
    if (cA == '(' && EsUnOperador(cB)) return 16;
    if (cA == '(' && cB == ')') return 17;
    if (cA == ')' && Char.IsDigit(cB)) return 18;
    if (cA == ')' && cB == '.') return 19;
    if (cA == ')' && Char.IsLower(cB)) return 20;
    if (cA == ')' && cB == '(') return 21;
}

/* Valida el inicio y fin de la expresión */
if (EsUnOperador(sbSintax[1])) return 22;

```

```

    if (EsUnOperador(sbSintax[sbSintax.Length - 2])) return 23;

    /* Valida balance de paréntesis */
    int ParentesisAbre = 0; /* Contador de paréntesis que abre */
    int ParentesisCierra = 0; /* Contador de paréntesis que cierra */
    for (int Cont = 1; Cont < sbSintax.Length - 1; Cont++) {
        switch (sbSintax[Cont]) {
            case '(': ParentesisAbre++; break;
            case ')': ParentesisCierra++; break;
        }
        if (ParentesisCierra > ParentesisAbre) return 24;
    }
    if (ParentesisAbre != ParentesisCierra) return 25;

    /* Validar los puntos decimales de un número real */
    int TotalPuntos = 0;
    for (int Cont = 0; Cont < sbSintax.Length; Cont++) {
        if (EsUnOperador(sbSintax[Cont])) TotalPuntos = 0;
        if (sbSintax[Cont] == '.') TotalPuntos++;
        if (TotalPuntos > 1) return 26;
    }

    /* Deja la expresión para ser analizada.
     * Reemplaza las funciones de tres letras
     * por una letra mayúscula. Cambia los ))
     * por )+0) porque es requerido al crear las piezas */
    this.Analizada.Length = 0;
    this.Analizada.Append(ConLetrasValidas);
    this.Analizada.Replace("sen", "A");
    this.Analizada.Replace("cos", "B");
    this.Analizada.Replace("tan", "C");
    this.Analizada.Replace("abs", "D");
    this.Analizada.Replace("asn", "E");
    this.Analizada.Replace("acs", "F");
    this.Analizada.Replace("atn", "G");
    this.Analizada.Replace("log", "H");
    this.Analizada.Replace("exp", "I");
    this.Analizada.Replace("sqr", "J");

    /* Sintaxis correcta */
    return 0;
}

/* Retorna si el Caracter es un operador matemático */
private static bool EsUnOperador(char Caracter) {
    switch (Caracter) {
        case '+':
        case '-':

```

```
case '*' :  
case '/' :  
case '^' :  
    return true;  
default:  
    return false;
```

```
}
```

```
}
```

```
}
```

```
}
```

Como usar el evaluador de expresiones

En Visual Studio 2022, se añade la clase al proyecto:

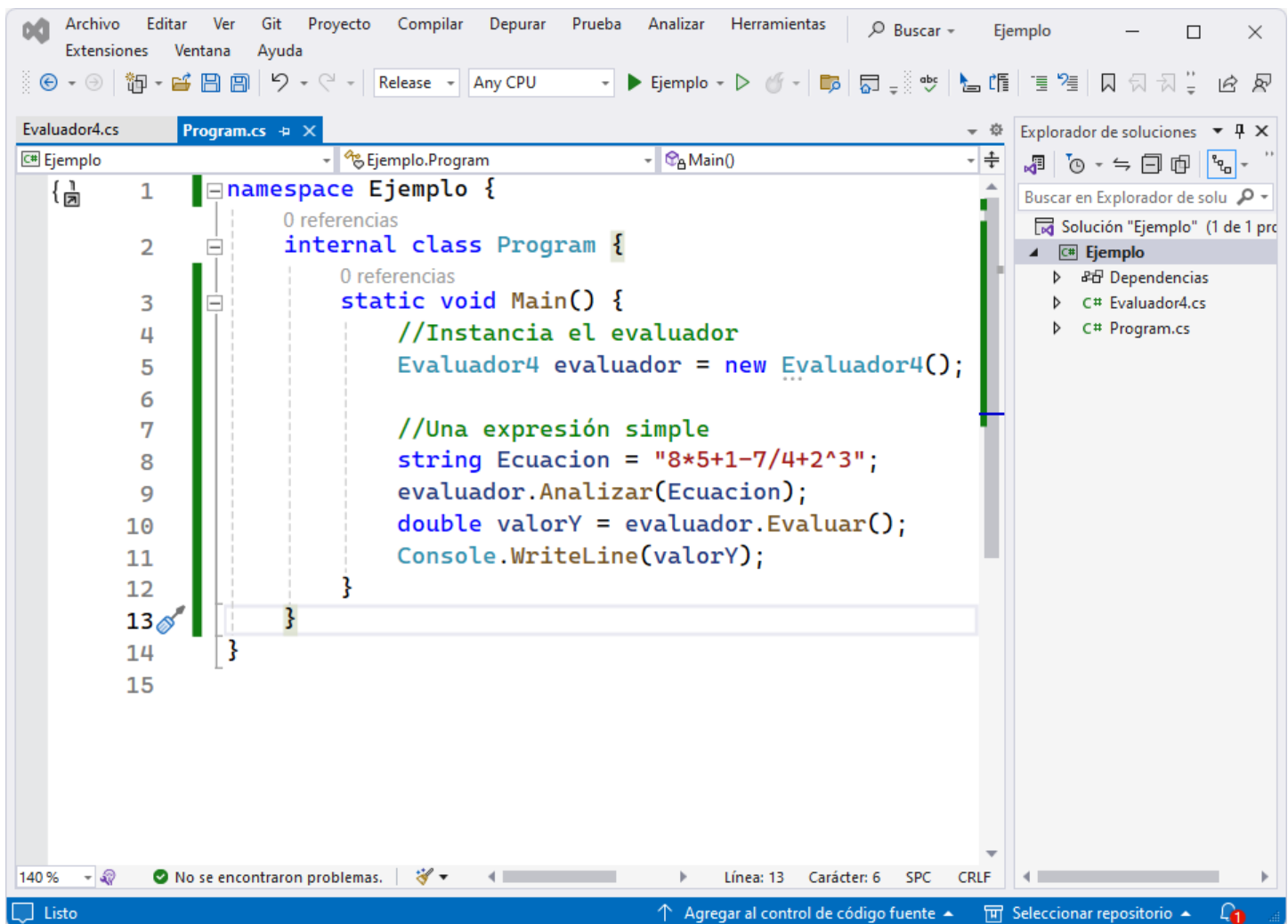


Ilustración 1: Uso del evaluador de expresiones

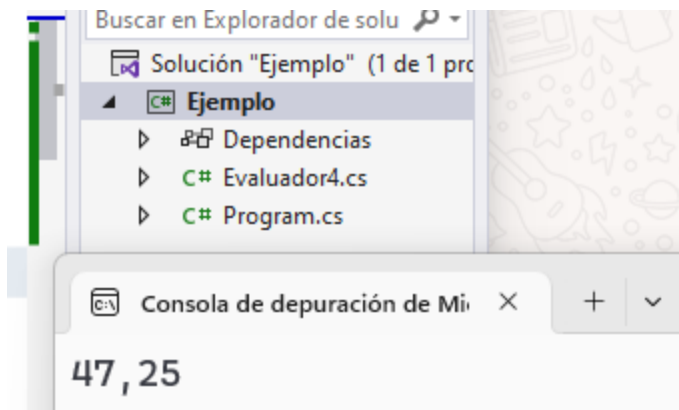


Ilustración 2: Resultado obtenido con el evaluador

H/Eval4/001.cs

```
namespace Ejemplo {  
    internal class Program {  
        static void Main() {  
            //Instancia el evaluador  
            Evaluador4 evaluador = new();  
  
            //Una expresión simple  
            string Ecuacion = "8*5+1-7/4+2^3";  
            evaluador.Analizar(Ecuacion);  
            double valorY = evaluador.Evaluar();  
            Console.WriteLine(valorY);  
        }  
    }  
}
```

```
namespace Ejemplo {  
    internal class Program {  
        static void Main() {  
            //Instancia el evaluador  
            Evaluador4 evaluador = new();  
  
            //Una expresión simple con números reales  
            string Ecuacion = "7.318+5.0045-9.071^2*8.04961";  
            evaluador.Analizar(Ecuacion);  
            double valorY = evaluador.Evaluar();  
            Console.WriteLine(valorY);  
        }  
    }  
}
```

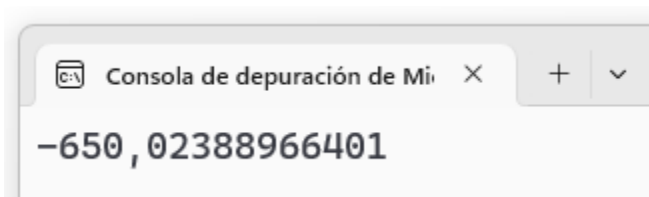


Ilustración 3: Operando números reales

```
namespace Ejemplo {  
    internal class Program {  
        static void Main() {  
            //Instancia el evaluador  
            Evaluador4 evaluador = new();  
  
            //Una expresión con paréntesis  
            string Ecuacion = "5*(3+2.5)";  
            Ecuacion += "-7/(8.03*2-5^3)";  
            Ecuacion += "+4.1*(3-(5.8*2.3))";  
            evaluador.Analizar(Ecuacion);  
            double valorY = evaluador.Evaluar();  
            Console.WriteLine(valorY);  
        }  
    }  
}
```

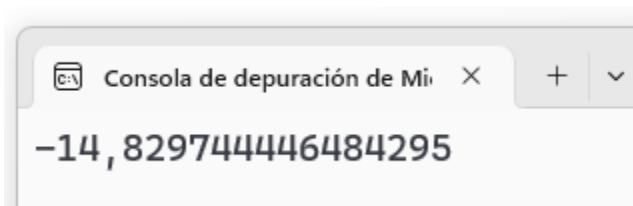


Ilustración 4: Operando con paréntesis

```
namespace Ejemplo {  
    internal class Program {  
        static void Main() {  
            //Instancia el evaluador  
            Evaluador4 evaluador = new();  
  
            //Una expresión con funciones  
            string Ecuacion = "sen(4.90+2.34)-cos(1.89)";  
            Ecuacion += "*tan(3)/abs(4-12)+asn(0.12)";  
            Ecuacion += "-acs(0-0.4)+atn(0.03)*log(1.3)";  
            Ecuacion += "+sqr(3.4)+exp(2.8)-sqr(9)";  
            evaluador.Analizar(Ecuacion);  
            double valorY = evaluador.Evaluar();  
            Console.WriteLine(valorY);  
        }  
    }  
}
```



Ilustración 5: Operando con funciones


```
namespace Ejemplo {  
    internal class Program {  
        static void Main() {  
            //Instancia el evaluador  
            Evaluador4 evaluador = new();  
  
            //Una expresión con uso de variables  
            //(deben estar en minúsculas)  
            string Ecuacion = "3*x+2*y";  
            evaluador.Analizar(Ecuacion);  
  
            //Le da valor a las variables  
            //(deben estar en minúsculas)  
            evaluador.DarValorVariable('x', 1.8);  
            evaluador.DarValorVariable('y', 3.5);  
            double valorY = evaluador.Evaluar();  
            Console.WriteLine(valorY);  
        }  
    }  
}
```

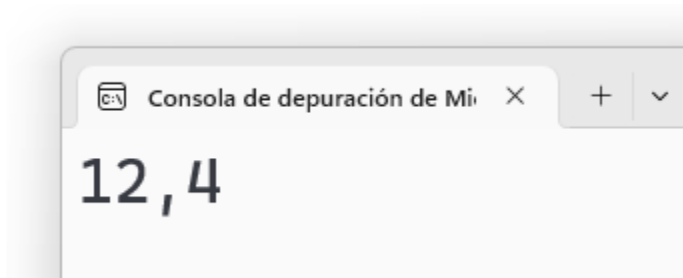


Ilustración 6: Operando con uso de variables

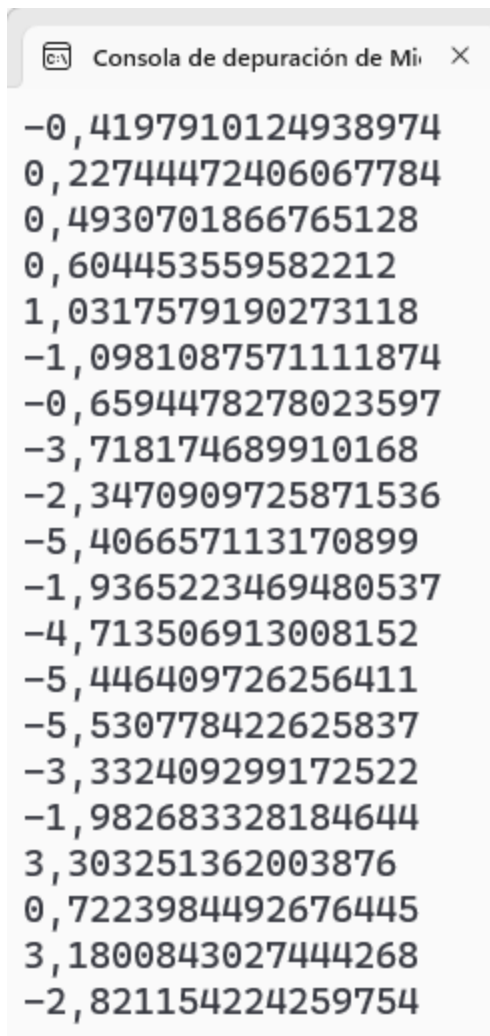
```
namespace Ejemplo {
    internal class Program {
        static void Main() {
            Random Azar = new();

            //Instancia el evaluador
            Evaluador4 evaluador = new();

            //Una expresión con uso de variables
            //(deben estar en minúsculas)
            string Ecuacion = "3*cos(2*x+4)-5*sen(4*y-7)";

            //Se analiza primero la ecuación
            evaluador.Analizar(Ecuacion);

            //Después de ser analizada, se le dan los
            //valores a las variables, esto hace que
            //el evaluador sea muy rápido
            for (int cont = 1; cont <= 20; cont++) {
                evaluador.DarValorVariable('x', Azar.NextDouble());
                evaluador.DarValorVariable('y', Azar.NextDouble());
                double valorY = evaluador.Evaluar();
                Console.WriteLine(valorY);
            }
        }
    }
}
```



Consola de depuración de Mi X

```
-0,4197910124938974  
0,22744472406067784  
0,4930701866765128  
0,604453559582212  
1,0317579190273118  
-1,0981087571111874  
-0,6594478278023597  
-3,718174689910168  
-2,3470909725871536  
-5,406657113170899  
-1,9365223469480537  
-4,713506913008152  
-5,446409726256411  
-5,530778422625837  
-3,332409299172522  
-1,982683328184644  
3,303251362003876  
0,7223984492676445  
3,1800843027444268  
-2,821154224259754
```

Ilustración 7: Generar múltiples valores de una ecuación al cambiar los valores.

Validando la sintaxis

El evaluador 4.0 verifica si la expresión algebraica es sintácticamente correcta al llamar al método Analizar(), si este método retorna el valor cero, significa que la expresión es correcta o no tiene errores de sintaxis, caso contrario, retorna un código de error.

H/Eval4/007.cs

```
namespace Ejemplo {
    internal class Program {
        static void Main() {
            string[] expAlg = new string[] {
                "2q-(*3)", //0
                "7-2(5-6)", //1
                "3..1", //2
                "3.*1", //3
                "3+5.w-8", //4
                "2-5.(4+1)*3", //5
                "2-(5.)*3", //6
                "2-(4+.1)-7", //7
                "5-*3", //8
                "2-(4+)-7", //9
                "7-a2-6", //10
                "7-a.4*3", //11
                "7-qw*9", //12
                "2-u(7-3)", //13
                "7-(.8+4)-6", //14
                "(+3-5)*7", //15
                "4+()*2", //16
                "(3-5)8", //17
                "(3-5).+2", //18
                "2-(7*3)k+7", //19
                "(4-3)(2+1)", //20
                "*3+5", //21
                "3*5*", //22
                "9*4)+(2-6", //23
                "((2+4)", //24
                "2.71*3.56.01" }; //25

            Evaluador4 evaluador = new();

            for (int num = 0; num < expAlg.Length; num++) {
                Console.WriteLine("\r\nExpr " + num + ": " + expAlg[num]);
                int Sintaxis = evaluador.Analizar(expAlg[num]);
                if (Sintaxis >= 0) {
                    Console.WriteLine(evaluador.MensajeError(Sintaxis));
                }
            }
        }
    }
}
```

```
}  
}  
}
```

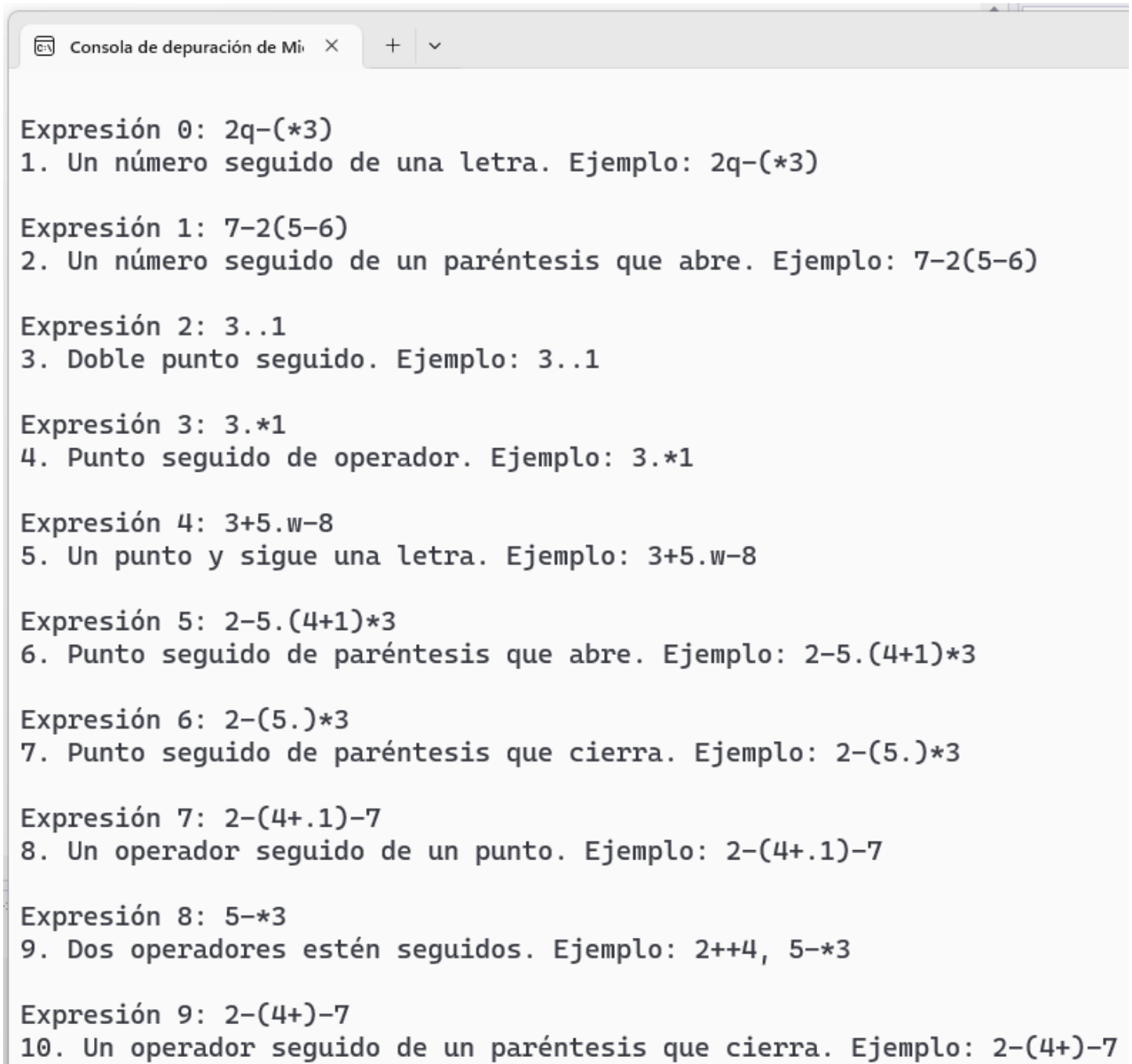


Ilustración 8: Evaluando la sintaxis

Expresión 10: $7-a^2-6$

11. Una letra seguida de número. Ejemplo: $7-2a-6$

Expresión 11: $7-a.4*3$

12. Una letra seguida de punto. Ejemplo: $7-a.-6$

Expresión 12: $7-qw*9$

13. Una letra seguida de otra letra. Ejemplo: $4-xy+3$

Expresión 13: $2-u(7-3)$

14. Una letra seguida de un paréntesis que abre. Ejemplo: $2-a(8*3)$

Expresión 14: $7-(.8+4)-6$

15. Un paréntesis que abre seguido de un punto. Ejemplo: $7-(.8+4)-6$

Expresión 15: $(+3-5)*7$

16. Un paréntesis que abre y sigue un operador. Ejemplo: $(+3-5)*7$

Expresión 16: $4+()*2$

17. Un paréntesis que abre y sigue un paréntesis que cierra. Ejemplo: $4+()*2$

Expresión 17: $(3-5)8$

18. Un paréntesis que cierra y sigue un número. Ejemplo: $(3-5)8$

Expresión 18: $(3-5).+2$

19. Un paréntesis que cierra y sigue un punto. Ejemplo: $(3-5).+2$

Expresión 19: $2-(7*3)k+7$

20. Un paréntesis que cierra y sigue una letra. Ejemplo: $2-(7*3)k+7$

Ilustración 9: Evaluando la sintaxis

Expresión 20: $(4-3)(2+1)$

21. Un paréntesis que cierra y sigue un paréntesis que abre. Ejemplo: $(4-3)(2+1)$

Expresión 21: $*3+5$

22. Inicia con un operador. Ejemplo: $*3+5$

Expresión 22: $3*5*$

23. Finaliza con un operador. Ejemplo: $7+9*$

Expresión 23: $9*4)+(2-6$

24. No hay correspondencia entre paréntesis que cierran y abren

Expresión 24: $((2+4)$

25. El número de paréntesis que cierran no es igual al número de paréntesis que abren

Expresión 25: $2.71*3.56.01$

26. Dos o más puntos en número real

Ilustración 10: Evaluando la sintaxis

Métricas: Desempeño con el evaluador interno de C#

.NET 8 tiene un evaluador de expresiones propio, así que una forma de probar que el evaluador está funcionando bien es generando ecuaciones al azar y comparar los resultados entre el evaluador propio de C# y el evaluador 4.0, si son iguales, significa que todo marcha bien. A continuación, el código que genera ecuaciones al azar y usa ambos evaluadores, compara los resultados y además el tiempo que tarda uno y otro.

H/Eval4/008.cs

```
using System.Data;
using System.Diagnostics;

namespace Ejemplo {
    internal class Program {
        static void Main() {
            Random Azar = new();

            //Versión 2024
            Evaluador4 evaluador2024 = new();

            //Arreglos que guardan valores de X, Y, Z
            double[] arregloX = new double[200];
            double[] arregloY = new double[200];
            double[] arregloZ = new double[200];
            for (int cont = 0; cont < arregloX.Length; cont++) {
                arregloX[cont] = Azar.NextDouble();
                arregloY[cont] = Azar.NextDouble();
                arregloZ[cont] = Azar.NextDouble();
            }

            //Prueba evaluador
            long TotalTiempo2024Evalua = 0, TotalTiempo2024Analiza = 0;
            double valor2024, AcumValor2024 = 0;

            //Evaluador interno
            long TotalTiempoInterno = 0;
            double valorInterno, AcumInterno = 0;

            //Toma el tiempo
            Stopwatch temporizador = new();

            for (int num = 1; num <= 1000; num++) {
                string ecuacion = EcuacionAzar(350, Azar);
                //Console.WriteLine(ecuacion);

                //Versión 2024. Análisis.
                temporizador.Reset();
```



```

temporizador.Start();
evaluador2024.Analizar(ecuacion);
temporizador.Stop();
TotalTiempo2024Analiza += temporizador.ElapsedTicks;

//Versión 2024. Evaluación
temporizador.Reset();
temporizador.Start();
valor2024 = 0;
for (int cont = 0; cont < arregloX.Length; cont++) {
    evaluador2024.DarValorVariable('x', arregloX[cont]);
    evaluador2024.DarValorVariable('y', arregloY[cont]);
    evaluador2024.DarValorVariable('z', arregloZ[cont]);
    valor2024 += Math.Abs(evaluador2024.Evaluar());
}
temporizador.Stop();
TotalTiempo2024Evalua += temporizador.ElapsedTicks;

//Compara contra el evaluador de
//expresiones propio que tiene C#
var EvaluadorInterno = new DataTable();
temporizador.Reset();
temporizador.Start();
valorInterno = 0;
for (int cont = 0; cont < arregloX.Length; cont++) {
    string ec1 = ecuacion;
    string ec2 = ec1.Replace("x", arregloX[cont].ToString());
    string ec3 = ec2.Replace("y", arregloY[cont].ToString());
    string ec4 = ec3.Replace("z", arregloZ[cont].ToString());
    string ec5 = ec4.Replace(",", ".");
    var Result = EvaluadorInterno.Compute(ec5, "");
    valorInterno += Math.Abs(Convert.ToDouble(Result));
}
temporizador.Stop();
TotalTiempoInterno += temporizador.ElapsedTicks;

if (Math.Abs(valor2024) > 10000000) continue;
if (double.IsNaN(valor2024) || double.IsInfinity(valor2024))
    continue;

AcumValor2024 += valor2024;
AcumInterno += valorInterno;
}
Console.WriteLine("Evaluador 2024 acumula: " + AcumValor2024);
Console.WriteLine("Interno C# acumula: " + AcumInterno);

Console.Write("\r\nEvaluador 2024 tiempo para evaluar: ");
Console.WriteLine(TotalTiempo2024Evalua);

```

```

Console.Write("Evaluador 2024 tiempo para analizar: ");
Console.WriteLine(TotalTiempo2024Analiza);

long TotalTiempo = TotalTiempo2024Evalua + TotalTiempo2024Analiza;

Console.Write("\r\nEvaluador 2024 tiempo: analizar y evaluar: ");
Console.WriteLine(TotalTiempo);

Console.Write("Interno tiempo: analizar y evaluar: ");
Console.WriteLine(TotalTiempoInterno);
}

public static string EcuacionAzar(int Longitud, Random Azar) {
    int cont = 0;
    int numParentesisAbre = 0;

    string Ecuacion = "";
    while (cont < Longitud) {

        //Función o paréntesis o nada
        if (Azar.NextDouble() < 0.5) {
            Ecuacion += "(";
            numParentesisAbre++;
            cont++;
        }

        //Variable o número
        cont++;
        switch (Azar.Next(4)) {
            case 0: Ecuacion += NumeroAzar(Azar); break;
            case 1: Ecuacion += "x"; break;
            case 2: Ecuacion += "y"; break;
            case 3: Ecuacion += "z"; break;
        }

        //Paréntesis que cierra
        int numParentesisCierra = Azar.Next(numParentesisAbre + 1);
        for (int num = 1; num <= numParentesisCierra; num++) {
            Ecuacion += ")";
            numParentesisAbre--;
            cont++;
        }

        //Operador
        cont++;
        Ecuacion += OperadorAzar(Azar);
    }
}

```

```

//Variable o número
switch (Azar.Next(4)) {
    case 0: Ecuacion += NumeroAzar(Azar); break;
    case 1: Ecuacion += "x"; break;
    case 2: Ecuacion += "y"; break;
    case 3: Ecuacion += "z"; break;
}

for (int num = 0; num < numParentesisAbre; num++)
    Ecuacion += ")";

return Ecuacion;
}

private static string OperadorAzar(Random azar) {
    string[] operadores = { "+", "-", "*" };
    return operadores[azar.Next(operadores.Length)];
}

private static string NumeroAzar(Random azar) {
    return "0." + Convert.ToString(azar.Next(1000000) + 1);
}
}
}

```

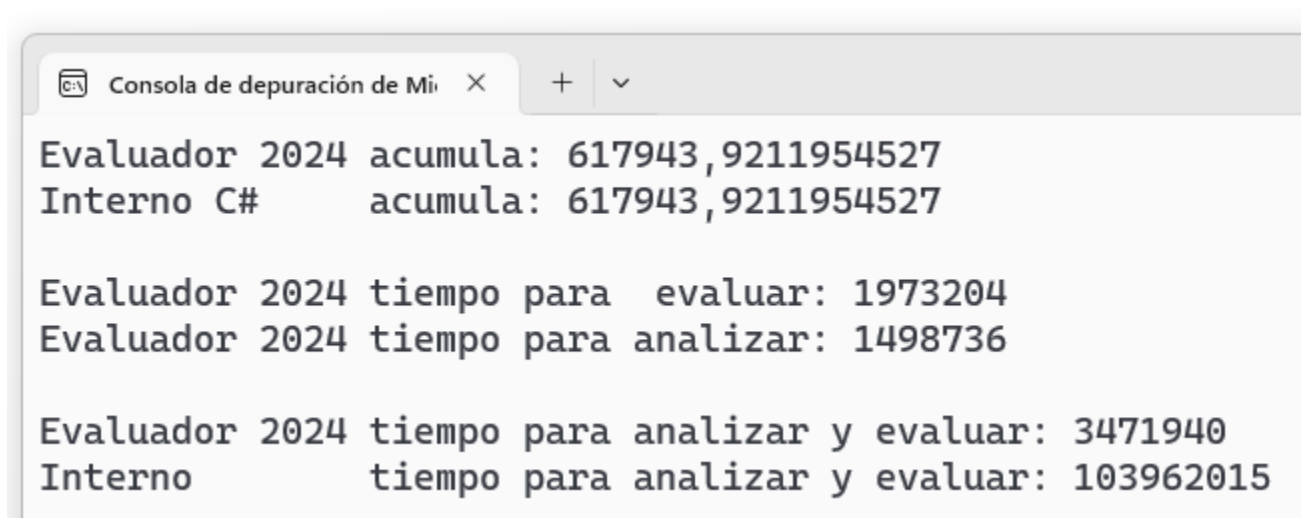


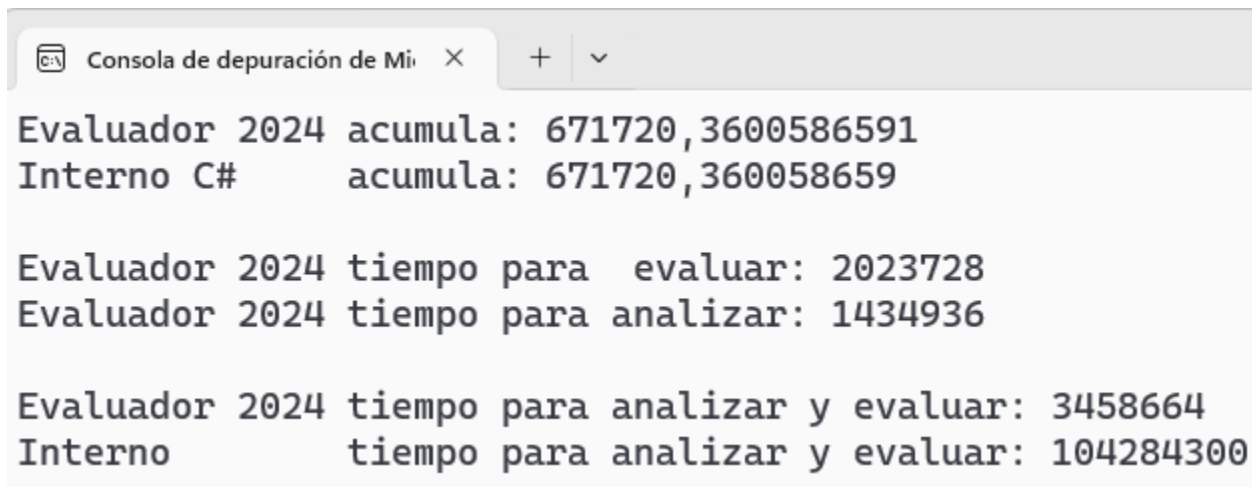
Ilustración 11: Métrica compara ambos evaluadores

Como se puede observar, ambos evaluadores obtienen los mismos resultados evaluando 1000 ecuaciones, cada una del tamaño de 350 caracteres y 300 valores distintos de variables por ecuación. Luego está correcto.

En el lado de desempeño, el Evaluador 4.0 es más rápido que el evaluador interno. La explicación a esta diferencia tan alta (a favor del Evaluador 4) es que el evaluador está

diseñado, escrito y optimizado sólo para ecuaciones algebraicas, mientras el evaluador interno de C# es más genérico para diversos tipos de expresiones. La especialización vence.

Otras pruebas:

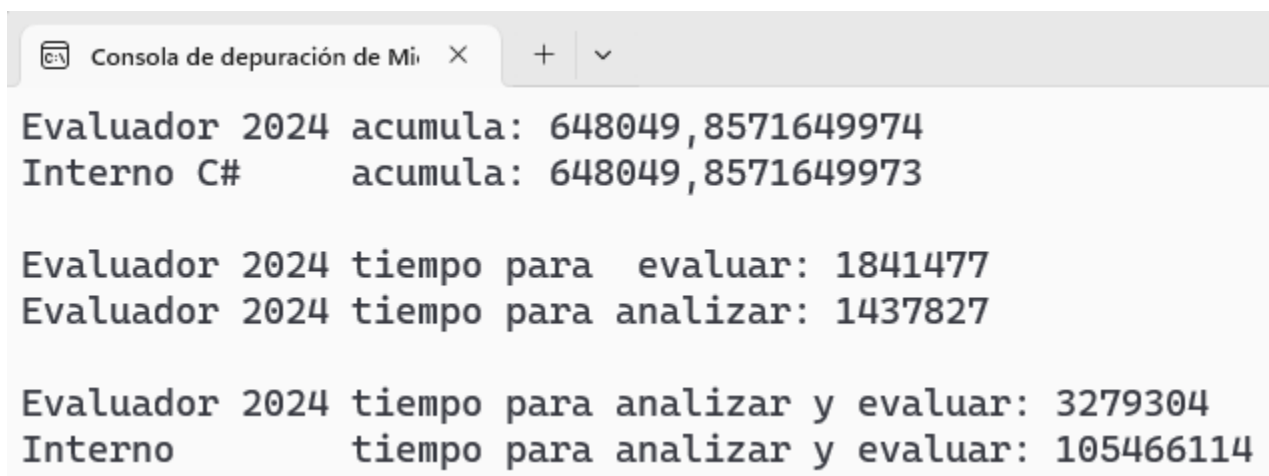


```
Consola de depuración de Mi... X + v
Evaluador 2024 acumula: 671720,3600586591
Interno C#      acumula: 671720,360058659

Evaluador 2024 tiempo para evaluar: 2023728
Evaluador 2024 tiempo para analizar: 1434936

Evaluador 2024 tiempo para analizar y evaluar: 3458664
Interno      tiempo para analizar y evaluar: 104284300
```

Ilustración 12: Métrica compara ambos evaluadores



```
Consola de depuración de Mi... X + v
Evaluador 2024 acumula: 648049,8571649974
Interno C#      acumula: 648049,8571649973

Evaluador 2024 tiempo para evaluar: 1841477
Evaluador 2024 tiempo para analizar: 1437827

Evaluador 2024 tiempo para analizar y evaluar: 3279304
Interno      tiempo para analizar y evaluar: 105466114
```

Ilustración 13: Métrica compara ambos evaluadores

Evaluador de expresiones usando un árbol binario. Programación recursiva

Fase 1. Sumas y restas

Se hace uso de la estructura de datos conocida como árbol binario. En este libro se aborda como fue construyéndose la solución hasta el programa final.

Se inicia con una expresión simple que tiene sólo números y los operadores son la suma y resta. Hay dos fases diferenciadas en cuanto a evaluar la expresión matemática:

1. El análisis que es la fase que construye el árbol binario.
2. La evaluación que es la fase en la que se recorre el árbol binario para dar con el resultado.

Como es una estructura de datos, el Nodo de un árbol binario tendrá esta estructura:

```
internal class Nodo {
    public int ID;
    public char Operador; //+ o -
    public double Numero;
    public Nodo Izquierda, Derecha;

    //Si el nodo tiene operador + o -
    public Nodo(char Operador, int identifica) {
        Numero = 0;
        this.Operador = Operador;
        Izquierda = null;
        Derecha = null;
        ID = identifica;
    }

    //Si el Nodo sólo tiene número
    public Nodo(double Numero, int identifica) {
        this.Numero = Numero;
        Operador = '#';
        Izquierda = null;
        Derecha = null;
        ID = identifica;
    }

    public void Imprime() {
        if (Operador != '#')
            Console.WriteLine("\"[" + ID + "] " + Operador + "\"");
        else
            Console.WriteLine("\"[" + ID + "] " + Numero + "\"");
    }
}
```

El Nodo maneja dos tipos de datos: el número o el operador (suma o resta). Para este libro se añade un tercer atributo que es el ID que va a ser usado para poder dibujar el árbol usando la herramienta en línea <http://viz-js.com> pero más adelante se retirará. Como se trabaja con árboles binarios, entonces el Nodo maneja dos apuntadores: Izquierda y Derecha.

Una expresión matemática como:

1+3-7

Debe formar un árbol binario así:

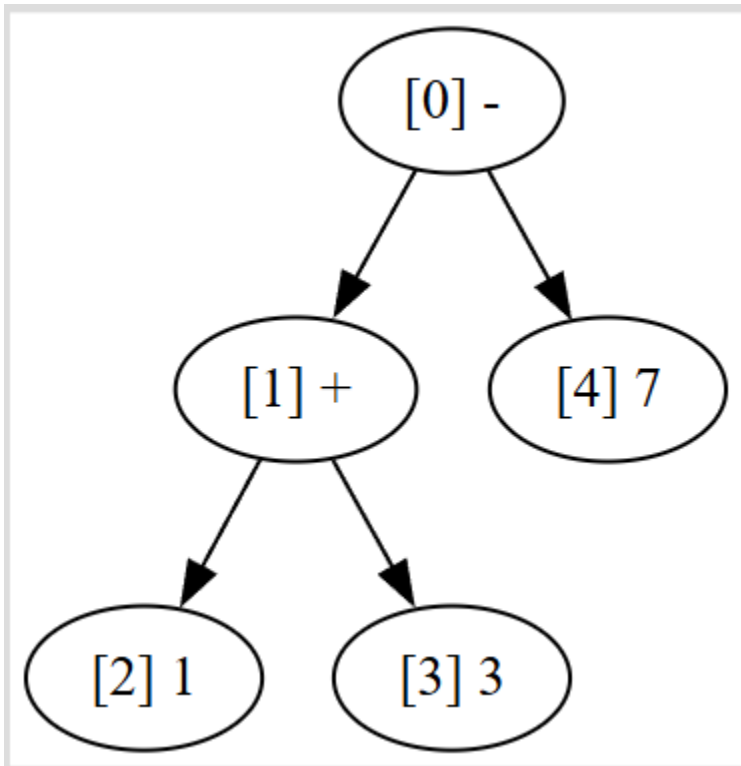


Ilustración 14: Árbol binario generado

Un segundo ejemplo:

$2.7-3.6+4.5-1.8$

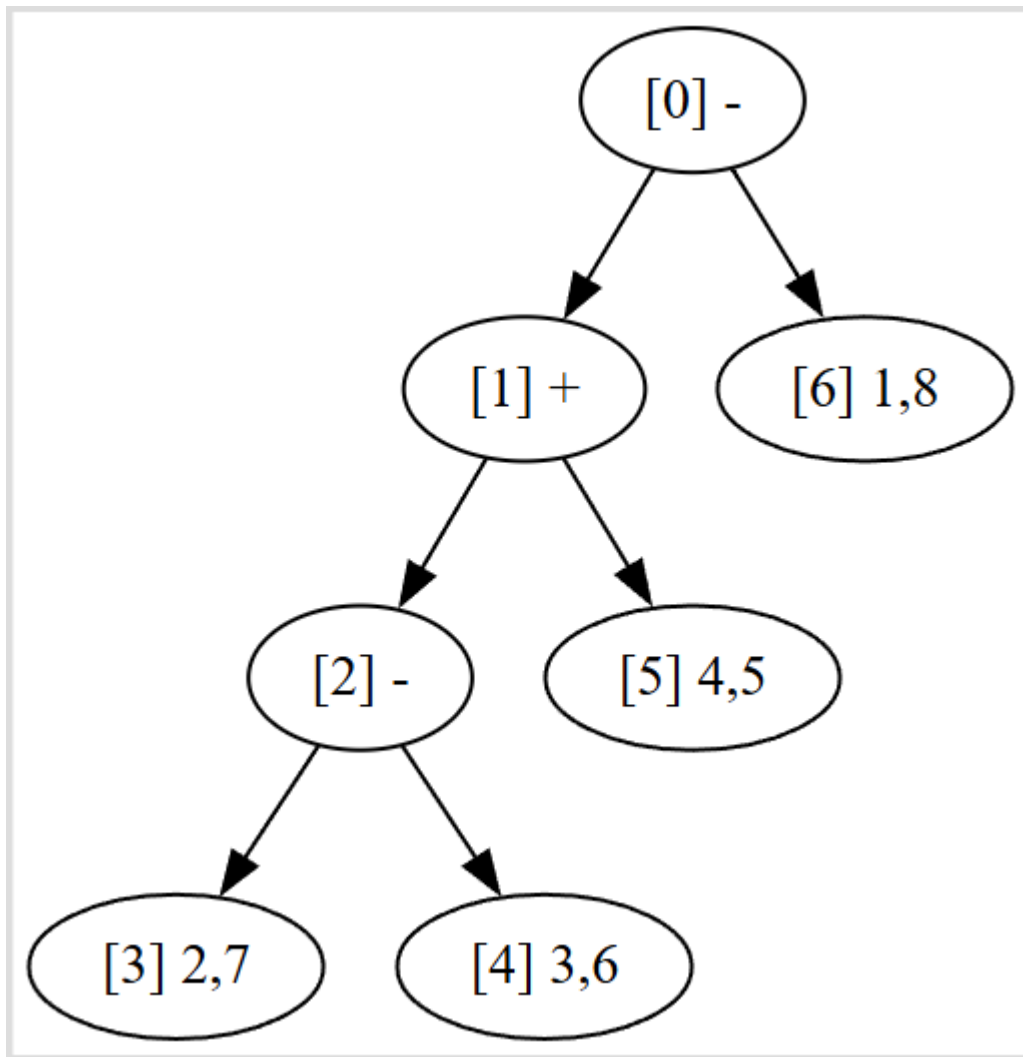


Ilustración 15: Árbol binario generado


```
using System.Globalization;

//Forma el árbol binario dada una expresión matemática
//de números y operadores suma y resta

namespace ArbolBinarioEvaluador {
    internal class Nodo {
        public int ID;
        public char Operador; //+ o -
        public double Numero;
        public Nodo Izquierda, Derecha;

        //Si el nodo tiene operador + o -
        public Nodo(char Operador, int identifica) {
            Numero = 0;
            this.Operador = Operador;
            Izquierda = null;
            Derecha = null;
            ID = identifica;
        }

        //Si el Nodo sólo tiene número
        public Nodo(double Numero, int identifica) {
            this.Numero = Numero;
            Operador = '#';
            Izquierda = null;
            Derecha = null;
            ID = identifica;
        }

        public void Imprime() {
            if (Operador != '#')
                Console.WriteLine("\[" + ID + "] " + Operador + "\");
            else
                Console.WriteLine("\[" + ID + "] " + Numero + "\");
        }
    }

    internal class Program {
        static int Identifica = 0;

        static void Main(string[] args) {
            //Ecuación con sólo sumas y restas
            string Ecuacion = "8+3-4+1-9";
            Nodo MiArbol = null;
        }
    }
}
```

```

MiArbol = CreaArbol(Ecuacion, MiArbol);

//Probarlo en: http://viz-js.com
Console.WriteLine("digraph testgraph{");
Dibujar(MiArbol);
Console.WriteLine("}");
}

public static Nodo CreaArbol(string Cadena, Nodo Arbol) {
    //Busca +, -
    for (int Cont = Cadena.Length - 1; Cont >= 0; Cont--) {
        if (Cadena[Cont] == '+' || Cadena[Cont] == '-') {
            string Izquierda = Cadena.Substring(0, Cont);
            string Derecha = Cadena.Substring(Cont + 1);
            Arbol = new Nodo(Cadena[Cont], Identifica++);
            Arbol.Izquierda = CreaArbol(Izquierda, Arbol.Izquierda);
            Arbol.Derecha = CreaArbol(Derecha, Arbol.Derecha);
            return Arbol;
        }
    }

    //Sólo queda el número y se le crea el nodo
    double Numero;
    Numero = double.Parse(Cadena, CultureInfo.InvariantCulture);
    Arbol = new Nodo(Numero, Identifica++);
    return Arbol;
}

static void Dibujar(Nodo Arbol) {
    if (Arbol != null) {
        if (Arbol.Izquierda != null) {
            Arbol.Imprime();
            Console.Write(" -> ");
            Arbol.Izquierda.Imprime();
            Console.WriteLine(" ");
            Dibujar(Arbol.Izquierda);
        }
        if (Arbol.Derecha != null) {
            Arbol.Imprime();
            Console.Write(" -> ");
            Arbol.Derecha.Imprime();
            Console.WriteLine(" ");
            Dibujar(Arbol.Derecha);
        }
    }
}
}
}
}
}

```

```
Consola de depuración de Mi × +
digraph testgraph{
  "[0] -" -> "[1] +"
  "[1] +" -> "[2] -"
  "[2] -" -> "[3] +"
  "[3] +" -> "[4] 8"
  "[3] +" -> "[5] 3"
  "[2] -" -> "[6] 4"
  "[1] +" -> "[7] 1"
  "[0] -" -> "[8] 9"
}
```

Ilustración 16: Ejemplo de ejecución del programa

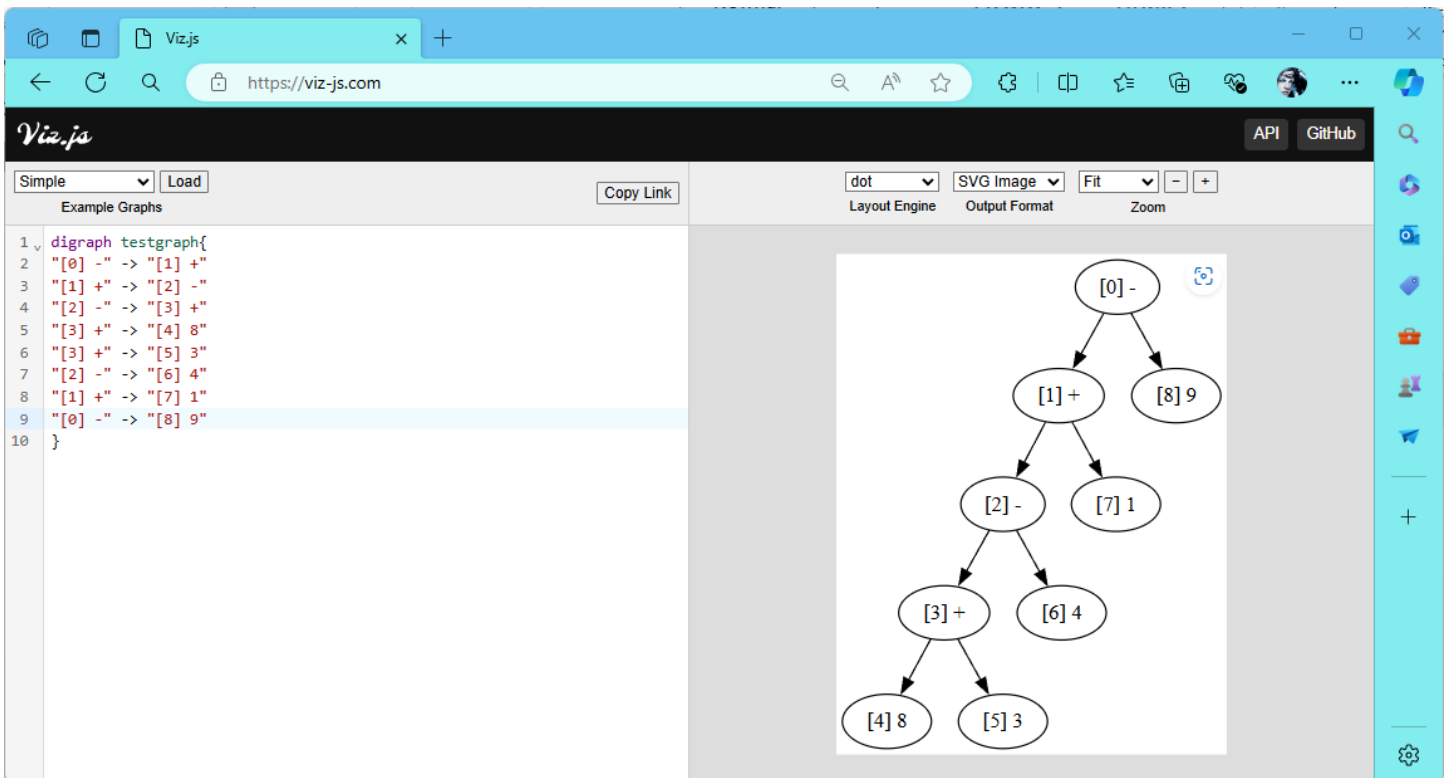


Ilustración 17: Interpretación en viz-js.com

Para evaluar la expresión, ya generado el árbol binario, es recorrerlo en post-orden (izquierda, derecha, raíz).

H/Árbol/002.cs

```
using System.Globalization;

//Forma el árbol binario dada una expresión matemática
//de números y operadores suma y resta
//Luego evalúa la expresión
namespace ArbolBinarioEvaluador {
    internal class Nodo {
        public int ID;
        public char Operador; //+ o -
        public double Numero;
        public Nodo Izquierda, Derecha;

        //Si el nodo tiene operador + o -
        public Nodo(char Operador, int identifica) {
            Numero = 0;
            this.Operador = Operador;
            Izquierda = null;
            Derecha = null;
            ID = identifica;
        }

        //Si el Nodo sólo tiene número
        public Nodo(double Numero, int identifica) {
            this.Numero = Numero;
            Operador = '#';
            Izquierda = null;
            Derecha = null;
            ID = identifica;
        }

        public void Imprime() {
            if (Operador != '#')
                Console.WriteLine("\n[" + ID + "] " + Operador + "\n");
            else
                Console.WriteLine("\n[" + ID + "] " + Numero + "\n");
        }
    }

    internal class Program {
        static int Identifica = 0;

        static void Main(string[] args) {
            //Ecuación con sólo sumas y restas
            string Ecuacion = "1+2-3+4-5";
        }
    }
}
```

```

    Nodo MiArbol = null;
    MiArbol = CreaArbol(Ecuacion, MiArbol);

    //Evalúa la expresión
    double Resultado = EvaluaArbol(MiArbol);
    Console.WriteLine("Resultado es: " + Resultado);

    //Probarlo en: http://viz-js.com
    Console.WriteLine("digraph testgraph{");
    Dibujar(MiArbol);
    Console.WriteLine("}");
}

public static Nodo CreaArbol(string Cadena, Nodo Arbol) {
    //Busca +, -
    for (int Cont = Cadena.Length - 1; Cont >= 0; Cont--) {
        if (Cadena[Cont] == '+' || Cadena[Cont] == '-') {
            string Izquierda = Cadena.Substring(0, Cont);
            string Derecha = Cadena.Substring(Cont + 1);
            Arbol = new Nodo(Cadena[Cont], Identifica++);
            Arbol.Izquierda = CreaArbol(Izquierda, Arbol.Izquierda);
            Arbol.Derecha = CreaArbol(Derecha, Arbol.Derecha);
            return Arbol;
        }
    }

    //Sólo queda el número y se le crea el nodo
    double Numero;
    Numero = double.Parse(Cadena, CultureInfo.InvariantCulture);
    Arbol = new Nodo(Numero, Identifica++);
    return Arbol;
}

static void Dibujar(Nodo Arbol) {
    if (Arbol != null) {
        if (Arbol.Izquierda != null) {
            Arbol.Imprime();
            Console.Write(" -> ");
            Arbol.Izquierda.Imprime();
            Console.WriteLine(" ");
            Dibujar(Arbol.Izquierda);
        }
        if (Arbol.Derecha != null) {
            Arbol.Imprime();
            Console.Write(" -> ");
            Arbol.Derecha.Imprime();
            Console.WriteLine(" ");
            Dibujar(Arbol.Derecha);
        }
    }
}

```

```

    }
}

// Recorrido en Post-Orden para
//evaluar el árbol binario
public static double EvaluaArbol(Nodo Arbol) {
    //No hay ramas hijas, entonces es un número
    if (Arbol.Izquierda == null)
        return Arbol.Numero;

    //Recorrido Post-Orden
    double ValIzquierda = EvaluaArbol(Arbol.Izquierda);
    double ValDerecha = EvaluaArbol(Arbol.Derecha);

    //Aplica operador
    switch (Arbol.Operador) {
        case '+': return ValIzquierda + ValDerecha;
        case '-': return ValIzquierda - ValDerecha;
    }
    return 0;
}
}
}

```

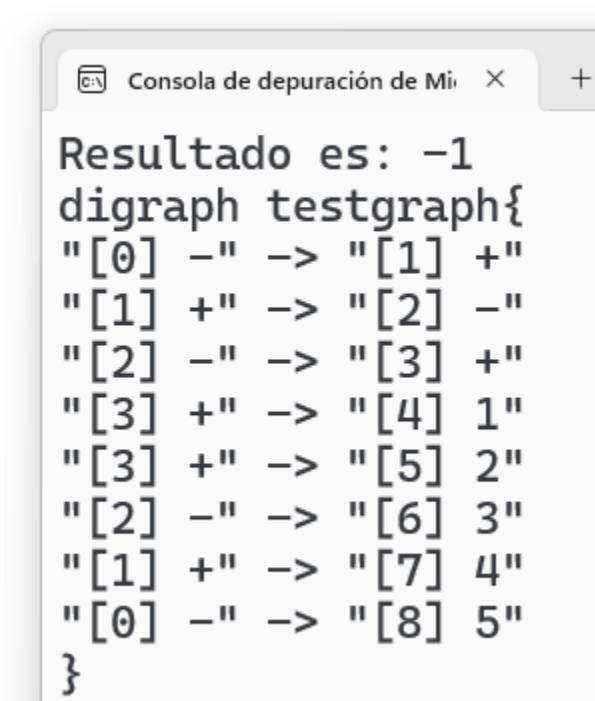


Ilustración 18: Resultado al evaluar

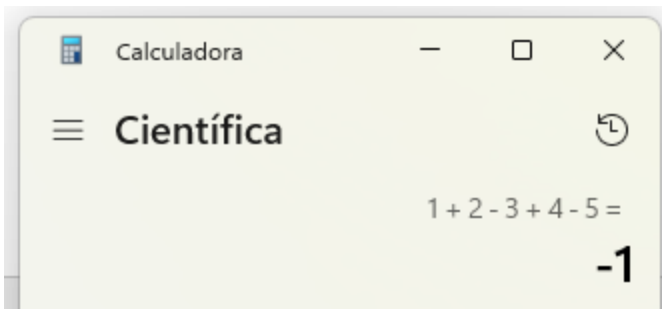


Ilustración 19: Validado con la calculadora

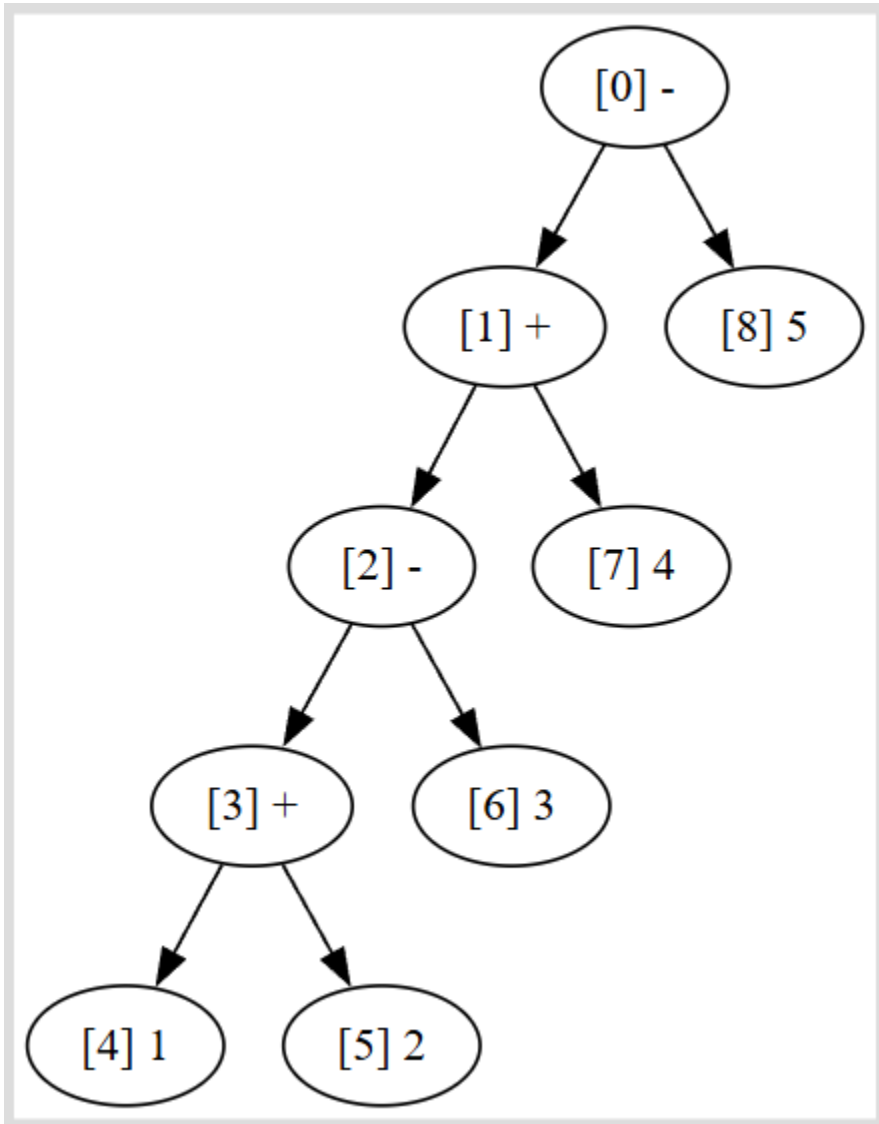


Ilustración 20: Árbol binario generado

Fase 2. Multiplicación y división

El siguiente paso es implementar los operadores de multiplicación y división. Según las reglas matemáticas, primero deben evaluarse estos operadores. Luego ante una expresión así:

$$2*3+4-5*6+7/8+9$$

El árbol binario generado debe ser este:

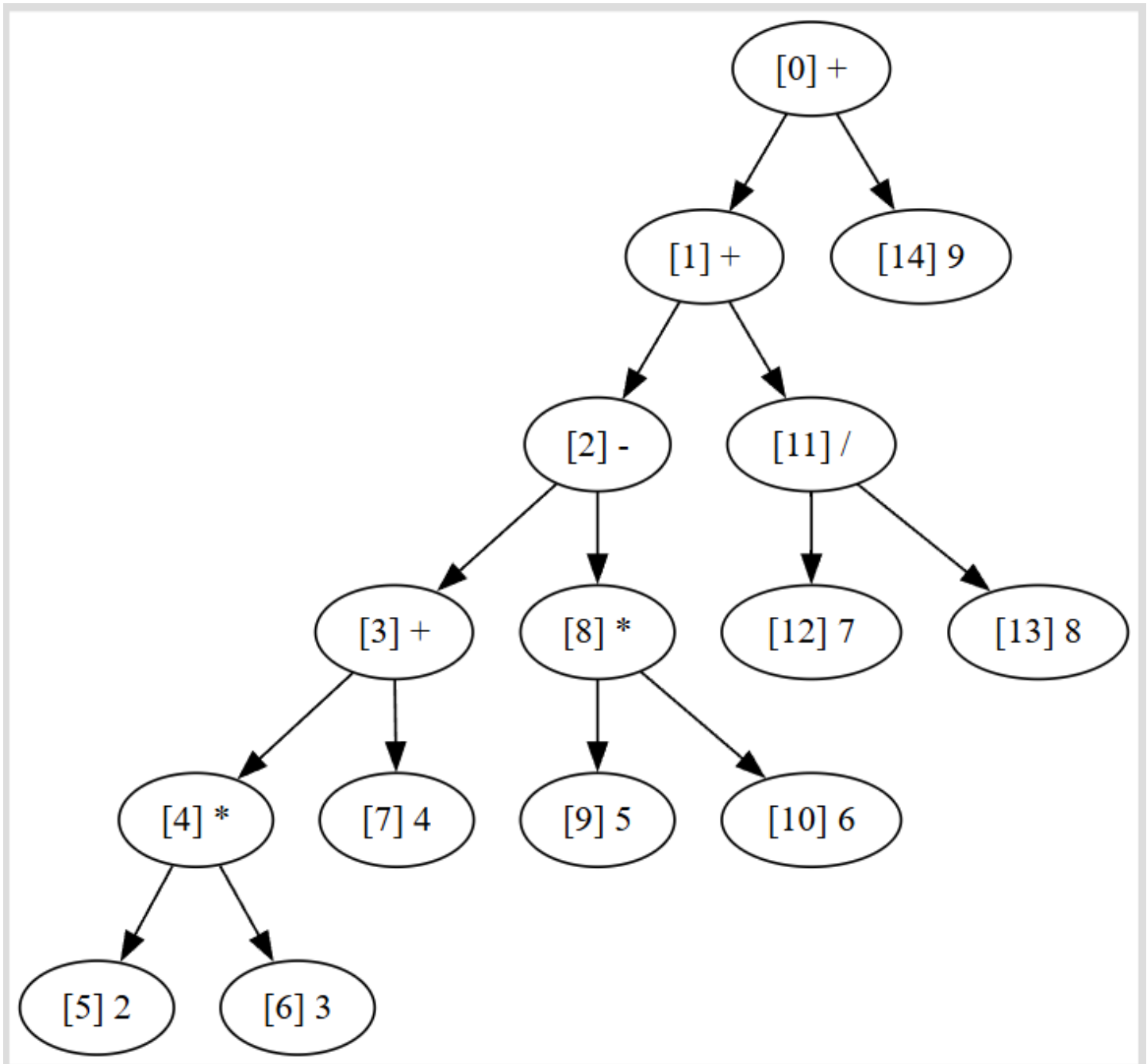


Ilustración 21: Árbol binario generado

Al evaluarse en post-orden se obtiene el resultado. Este es el programa:


```

using System.Globalization;

//Forma el árbol binario dada una expresión matemática
//de números y los operadores:
//Suma, resta, multiplicación y división
//Luego evalúa la expresión
namespace ArbolBinarioEvaluador {
    internal class Nodo {
        public int ID;
        public char Operador; //+ - * /
        public double Numero;
        public Nodo Izquierda, Derecha;

        //Si el nodo tiene operador + - * /
        public Nodo(char Operador, int identifica) {
            Numero = 0;
            this.Operador = Operador;
            Izquierda = null;
            Derecha = null;
            ID = identifica;
        }

        //Si el Nodo sólo tiene número
        public Nodo(double Numero, int identifica) {
            this.Numero = Numero;
            Operador = '#';
            Izquierda = null;
            Derecha = null;
            ID = identifica;
        }

        public void Imprime() {
            if (Operador != '#')
                Console.WriteLine("\n[" + ID + "] " + Operador + "\n");
            else
                Console.WriteLine("\n[" + ID + "] " + Numero + "\n");
        }
    }

    internal class Program {
        static int Identifica = 0;

        static void Main(string[] args) {
            //Ecuación los cuatro operadores
            string Ecuacion = "1+2*3+4/5-6";
            Nodo MiArbol = null;
            MiArbol = CreaArbol(Ecuacion, MiArbol);
        }
    }
}

```

```

//Evalúa la expresión
double Resultado = EvaluaArbol(MiArbol);
Console.WriteLine("Resultado es: " + Resultado);

//Probarlo en: http://viz-js.com
Console.WriteLine("digraph testgraph{");
Dibujar(MiArbol);
Console.WriteLine("}");
}

public static Nodo CreaArbol(string Cadena, Nodo Arbol) {
    //Busca +, -
    for (int Cont = Cadena.Length - 1; Cont >= 0; Cont--) {
        if (Cadena[Cont] == '+' || Cadena[Cont] == '-') {
            string Izquierda = Cadena.Substring(0, Cont);
            string Derecha = Cadena.Substring(Cont + 1);
            Arbol = new Nodo(Cadena[Cont], Identifica++);
            Arbol.Izquierda = CreaArbol(Izquierda, Arbol.Izquierda);
            Arbol.Derecha = CreaArbol(Derecha, Arbol.Derecha);
            return Arbol;
        }
    }

    //Busca *, /
    for (int Cont = Cadena.Length - 1; Cont >= 0; Cont--) {
        if (Cadena[Cont] == '*' || Cadena[Cont] == '/') {
            string Izquierda = Cadena.Substring(0, Cont);
            string Derecha = Cadena.Substring(Cont + 1);
            Arbol = new Nodo(Cadena[Cont], Identifica++);
            Arbol.Izquierda = CreaArbol(Izquierda, Arbol.Izquierda);
            Arbol.Derecha = CreaArbol(Derecha, Arbol.Derecha);
            return Arbol;
        }
    }

    //Sólo queda el número y se le crea el nodo
    double Numero;
    Numero = double.Parse(Cadena, CultureInfo.InvariantCulture);
    Arbol = new Nodo(Numero, Identifica++);
    return Arbol;
}

static void Dibujar(Nodo Arbol) {
    if (Arbol != null) {
        if (Arbol.Izquierda != null) {
            Arbol.Imprime();
            Console.Write(" -> ");
        }
    }
}

```

```

        Arbol.Izquierda.Imprime();
        Console.WriteLine(" ");
        Dibujar(Arbol.Izquierda);
    }
    if (Arbol.Derecha != null) {
        Arbol.Imprime();
        Console.Write(" -> ");
        Arbol.Derecha.Imprime();
        Console.WriteLine(" ");
        Dibujar(Arbol.Derecha);
    }
}

// Recorrido en Post-Orden para
//evaluar el árbol binario
public static double EvaluaArbol(Nodo Arbol) {
    //No hay ramas hijas, luego es un número
    if (Arbol.Izquierda == null)
        return Arbol.Numero;

    //Recorrido Post-Orden
    double ValIzquierda = EvaluaArbol(Arbol.Izquierda);
    double ValDerecha = EvaluaArbol(Arbol.Derecha);

    //Aplica operador
    switch (Arbol.Operador) {
        case '+': return ValIzquierda + ValDerecha;
        case '-': return ValIzquierda - ValDerecha;
        case '*': return ValIzquierda * ValDerecha;
        case '/': return ValIzquierda / ValDerecha;
    }
    return 0;
}
}
}

```

Así ejecuta:

```
Consola de depuración de Mi  X + v
Resultado es: -10,125
digraph testgraph{
"[0] +" -> "[1] +"
"[1] +" -> "[2] -"
"[2] -" -> "[3] +"
"[3] +" -> "[4] *"
"[4] *" -> "[5] 2"
"[4] *" -> "[6] 3"
"[3] +" -> "[7] 4"
"[2] -" -> "[8] *"
"[8] *" -> "[9] 5"
"[8] *" -> "[10] 6"
"[1] +" -> "[11] /"
"[11] /" -> "[12] 7"
"[11] /" -> "[13] 8"
"[0] +" -> "[14] 9"
}
```

Ilustración 22: Ejecución del evaluador

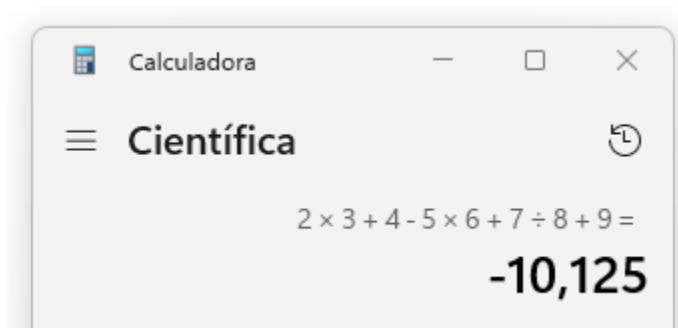


Ilustración 23: Prueba con calculadora

Fase 3. Potencia

El último operador es la potencia, que es con el símbolo $^$. Este operador debe ser evaluado primero, luego multiplicación y división, por último suma y resta.

Una expresión como:

$$2^3 - 8 + 7 * 6 - 1 + 7 / 5$$

Genera este árbol binario:

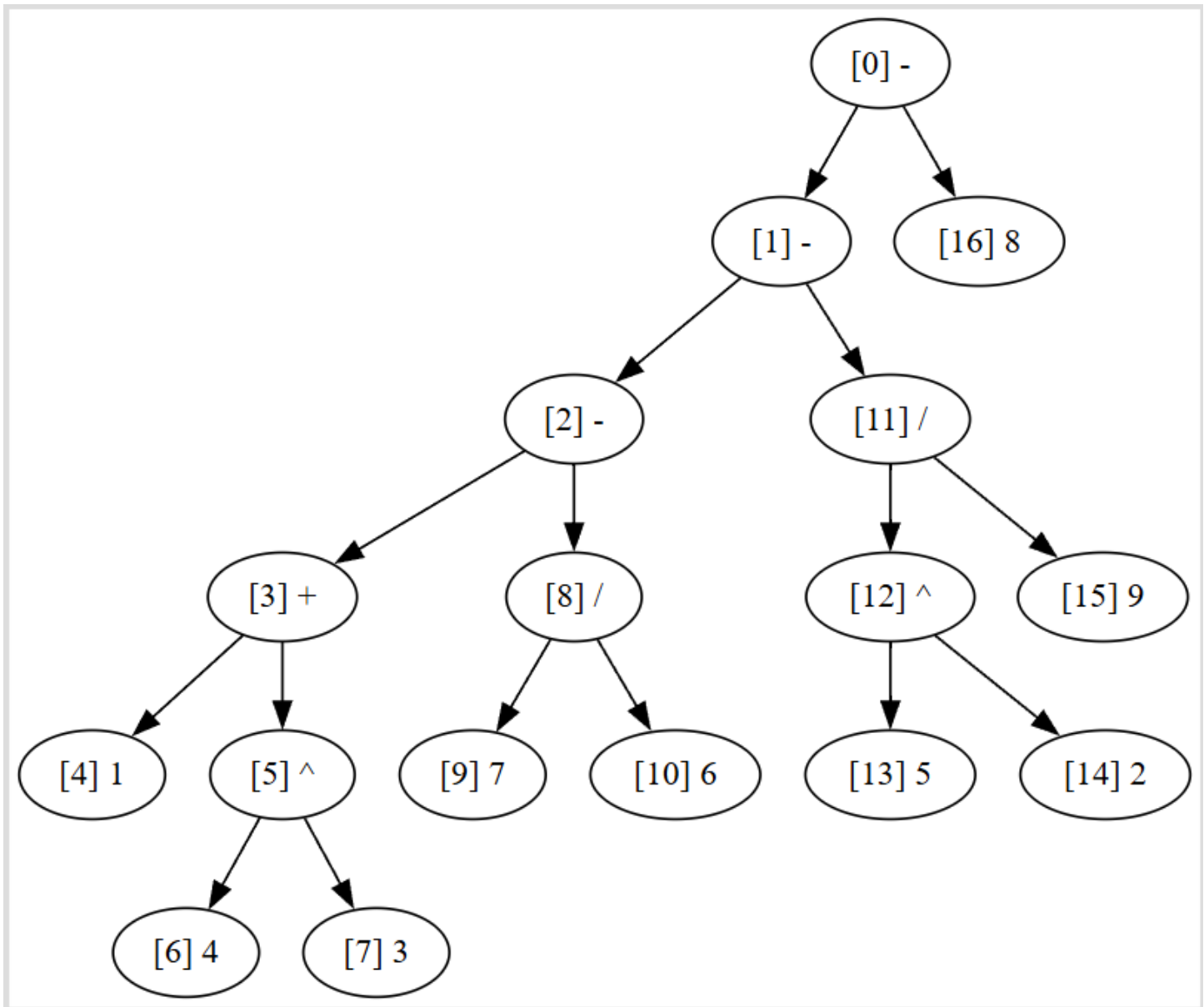


Ilustración 24: Árbol binario generado

Al recorrerlo en Post-Orden se evalúa la expresión.

```

using System.Globalization;

//Forma el árbol binario dada una expresión matemática
//de números y los operadores:
//Suma, resta, multiplicación, división y potencia
//Luego evalúa la expresión
namespace ArbolBinarioEvaluador {
    internal class Nodo {
        public int ID;
        public char Operador; //+ - * / ^
        public double Numero;
        public Nodo Izquierda, Derecha;

        //Si el nodo tiene operador + - * / ^
        public Nodo(char Operador, int identifica) {
            Numero = 0;
            this.Operador = Operador;
            Izquierda = null;
            Derecha = null;
            ID = identifica;
        }

        //Si el Nodo sólo tiene número
        public Nodo(double Numero, int identifica) {
            this.Numero = Numero;
            Operador = '#';
            Izquierda = null;
            Derecha = null;
            ID = identifica;
        }

        public void Imprime() {
            if (Operador != '#')
                Console.WriteLine("\n[" + ID + "] " + Operador + "\n");
            else
                Console.WriteLine("\n[" + ID + "] " + Numero + "\n");
        }
    }

    internal class Program {
        static int Identifica = 0;

        static void Main(string[] args) {
            //Ecuación los cuatro operadores
            string Ecuacion = "1+4^3-7/6-5^2/9-8";

```

```

    Nodo MiArbol = null;
    MiArbol = CreaArbol(Ecuacion, MiArbol);

    //Evalúa la expresión
    double Resultado = EvaluaArbol(MiArbol);
    Console.WriteLine("Resultado es: " + Resultado);

    //Probarlo en: http://viz-js.com
    Console.WriteLine("digraph testgraph{");
    Dibujar(MiArbol);
    Console.WriteLine("}");
}

public static Nodo CreaArbol(string Cadena, Nodo Arbol) {
    //Busca +, -
    for (int Cont = Cadena.Length - 1; Cont >= 0; Cont--) {
        if (Cadena[Cont] == '+' || Cadena[Cont] == '-') {
            string Izquierda = Cadena.Substring(0, Cont);
            string Derecha = Cadena.Substring(Cont + 1);
            Arbol = new Nodo(Cadena[Cont], Identifica++);
            Arbol.Izquierda = CreaArbol(Izquierda, Arbol.Izquierda);
            Arbol.Derecha = CreaArbol(Derecha, Arbol.Derecha);
            return Arbol;
        }
    }

    //Busca *, /
    for (int Cont = Cadena.Length - 1; Cont >= 0; Cont--) {
        if (Cadena[Cont] == '*' || Cadena[Cont] == '/') {
            string Izquierda = Cadena.Substring(0, Cont);
            string Derecha = Cadena.Substring(Cont + 1);
            Arbol = new Nodo(Cadena[Cont], Identifica++);
            Arbol.Izquierda = CreaArbol(Izquierda, Arbol.Izquierda);
            Arbol.Derecha = CreaArbol(Derecha, Arbol.Derecha);
            return Arbol;
        }
    }

    //Busca ^
    for (int Cont = Cadena.Length - 1; Cont >= 0; Cont--) {
        if (Cadena[Cont] == '^') {
            string Izquierda = Cadena.Substring(0, Cont);
            string Derecha = Cadena.Substring(Cont + 1);
            Arbol = new Nodo(Cadena[Cont], Identifica++);
            Arbol.Izquierda = CreaArbol(Izquierda, Arbol.Izquierda);
            Arbol.Derecha = CreaArbol(Derecha, Arbol.Derecha);
            return Arbol;
        }
    }
}

```

```

    }

    //Sólo queda el número y se le crea el nodo
    double Numero;
    Numero = double.Parse(Cadena, CultureInfo.InvariantCulture);
    Arbol = new Nodo(Numero, Identifica++);
    return Arbol;
}

static void Dibujar(Nodo Arbol) {
    if (Arbol != null) {
        if (Arbol.Izquierda != null) {
            Arbol.Imprime();
            Console.Write(" -> ");
            Arbol.Izquierda.Imprime();
            Console.WriteLine(" ");
            Dibujar(Arbol.Izquierda);
        }
        if (Arbol.Derecha != null) {
            Arbol.Imprime();
            Console.Write(" -> ");
            Arbol.Derecha.Imprime();
            Console.WriteLine(" ");
            Dibujar(Arbol.Derecha);
        }
    }
}

// Recorrido en Post-Orden para
//evaluar el árbol binario
public static double EvaluaArbol(Nodo Arbol) {
    //No hay ramas hijas entonces es un número
    if (Arbol.Izquierda == null)
        return Arbol.Numero;

    //Recorrido Post-Orden
    double ValIzquierda = EvaluaArbol(Arbol.Izquierda);
    double ValDerecha = EvaluaArbol(Arbol.Derecha);

    //Aplica operador
    switch (Arbol.Operador) {
        case '+': return ValIzquierda + ValDerecha;
        case '-': return ValIzquierda - ValDerecha;
        case '*': return ValIzquierda * ValDerecha;
        case '/': return ValIzquierda / ValDerecha;
        case '^': return Math.Pow(ValIzquierda, ValDerecha);
    }
    return 0;
}

```

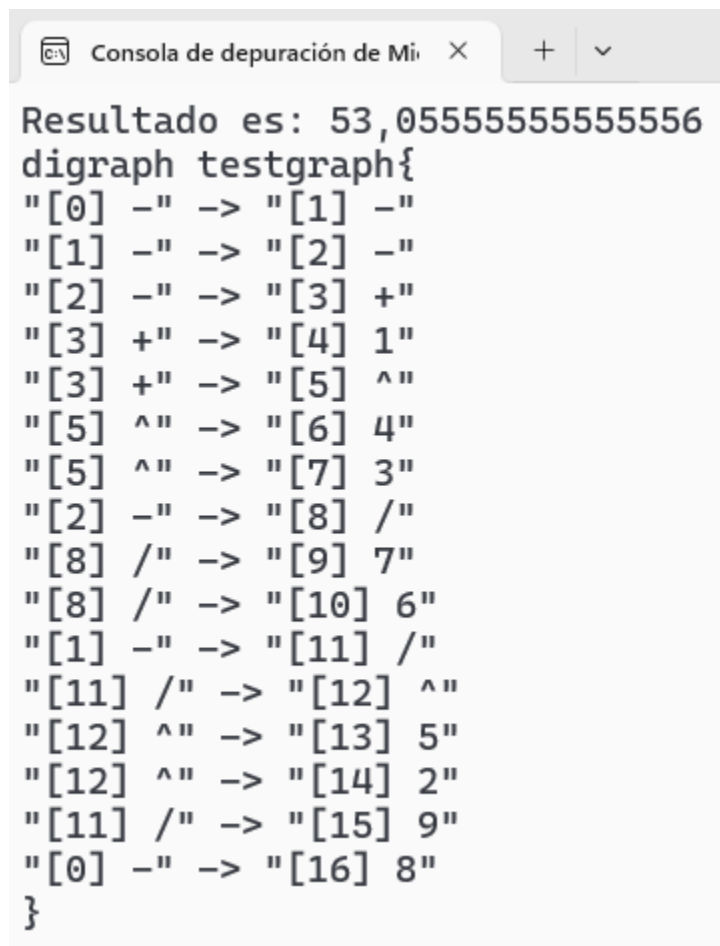


```

    }
}
}

```

Ejemplo de ejecución:



Consola de depuración de Mi

Resultado es: 53,05555555555556

```

digraph testgraph{
"[0] -" -> "[1] -"
"[1] -" -> "[2] -"
"[2] -" -> "[3] +"
"[3] +" -> "[4] 1"
"[3] +" -> "[5] ^"
"[5] ^" -> "[6] 4"
"[5] ^" -> "[7] 3"
"[2] -" -> "[8] /"
"[8] /" -> "[9] 7"
"[8] /" -> "[10] 6"
"[1] -" -> "[11] /"
"[11] /" -> "[12] ^"
"[12] ^" -> "[13] 5"
"[12] ^" -> "[14] 2"
"[11] /" -> "[15] 9"
"[0] -" -> "[16] 8"
}

```

Ilustración 25: Ejemplo de ejecución

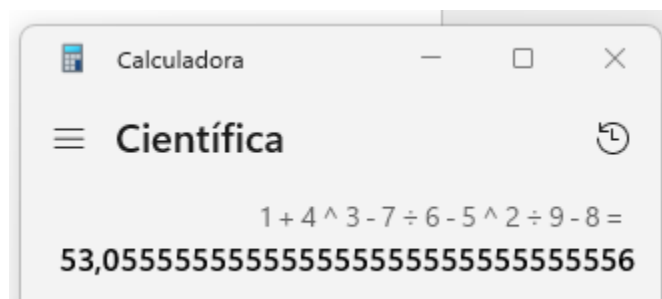


Ilustración 26: Prueba con la calculadora

Fase 4. Paréntesis

Los paréntesis cambian el orden de interpretación de una expresión matemática, hay que considerar que puede haber paréntesis internos. Incluso hay casos como que hay paréntesis internos sin ninguna función, ejemplo:

$46-(((792))) + (((125)))$

Puede convertirse a:

$46-792+125$

Luego el software debe eliminar ese tipo de paréntesis.

Una expresión como:

$(1+6)-((8*3)+2/9)+2/9$

Genera un árbol binario así:

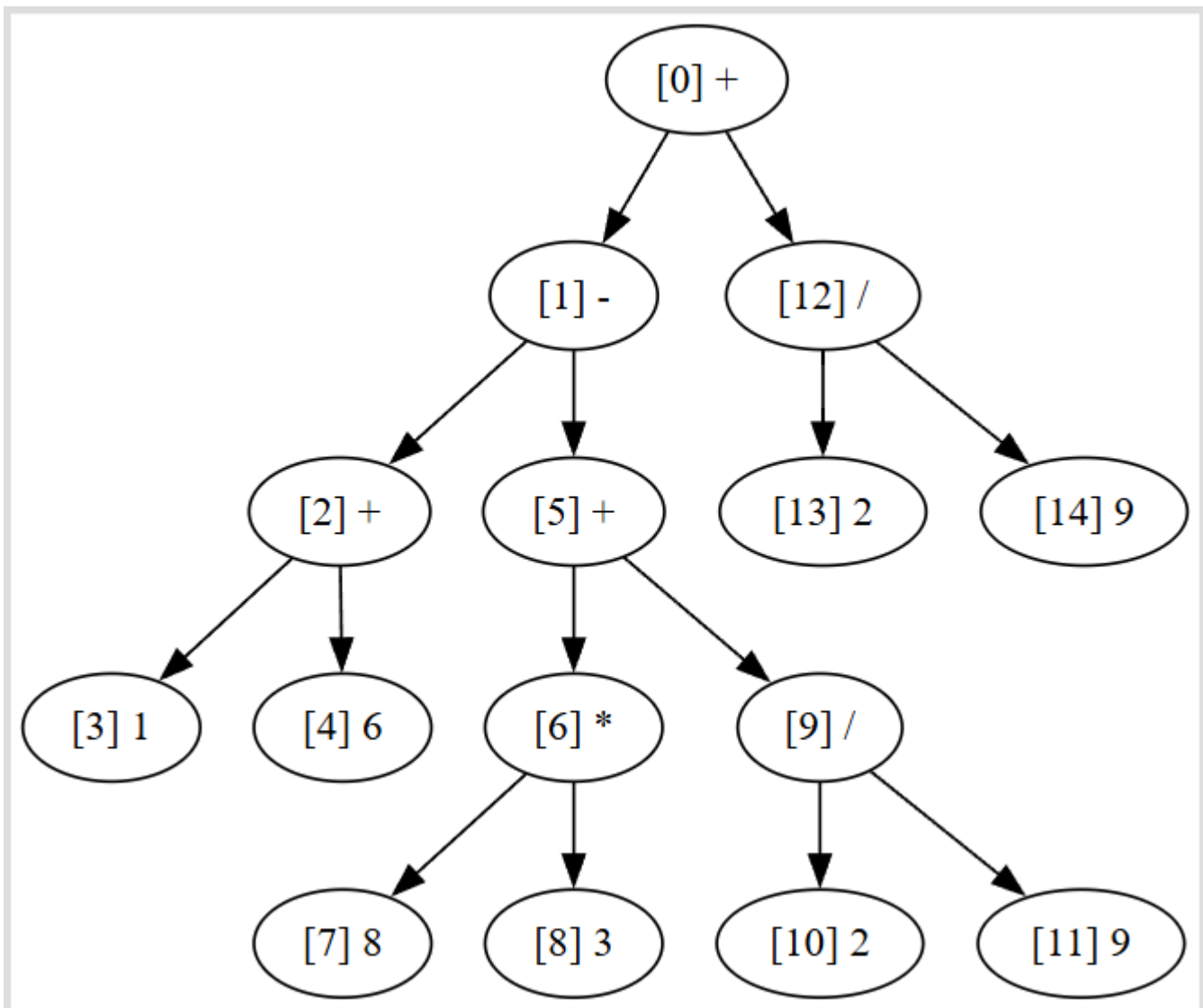


Ilustración 27: Árbol binario generado

Este es el programa:

H/Árbol/005.cs

```
using System.Globalization;

//Forma el árbol binario dada una expresión matemática
//de paréntesis, números y los operadores:
//Suma, resta, multiplicación, división y potencia
//Luego evalúa la expresión
namespace ArbolBinarioEvaluador {
    internal class Nodo {
        public int ID;
        public char Operador; //+ - * / ^
        public double Numero;
        public Nodo Izquierda, Derecha;

        //Si el nodo tiene operador + - * / ^
        public Nodo(char Operador, int identifica) {
            Numero = 0;
            this.Operador = Operador;
            Izquierda = null;
            Derecha = null;
            ID = identifica;
        }

        //Si el Nodo sólo tiene número
        public Nodo(double Numero, int identifica) {
            this.Numero = Numero;
            Operador = '#';
            Izquierda = null;
            Derecha = null;
            ID = identifica;
        }

        public void Imprime() {
            if (Operador != '#')
                Console.WriteLine("\[" + ID + "] " + Operador + "\");
            else
                Console.WriteLine("\[" + ID + "] " + Numero + "\");
        }
    }

    internal class Program {
        static int Identifica = 0;

        static void Main(string[] args) {
```

```

//Ecuación los cuatro operadores
string Ecuacion = "(1+6)-((8*3)+2/9)+2/9";
Nodo MiArbol = null;
MiArbol = CreaArbol(Ecuacion, MiArbol);

//Evalúa la expresión
double Resultado = EvaluaArbol(MiArbol);
Console.WriteLine("Resultado es: " + Resultado);

//Probarlo en: http://viz-js.com
Console.WriteLine("digraph testgraph{");
Dibujar(MiArbol);
Console.WriteLine("}");
}

public static Nodo CreaArbol(string Cad, Nodo Arbol) {
    /* Elimina paréntesis al inicio y al final si es
       una expresión del tipo:
           (expresión de números y operadores)
       la convierte en:
           expresión de números y operadores
       Ejemplo:
           (2^3-8+7*2-14+7/2)
       Se vuelve:
           2^3-8+7*2-14+7/2

       Pero si encuentra algo así:
           (2+4) * (5-2)
       No aplica tal conversión porque no son
       paréntesis que cubren toda la expresión
    */
    int Prntss;
    bool EsFinal;
    do {
        EsFinal = false;
        if (Cad[0] == '(') {
            EsFinal = true;
            Prntss = 0;
            for (int Cont = 1; Cont < Cad.Length - 1; Cont++) {
                if (Cad[Cont] == '(') Prntss++;
                if (Cad[Cont] == ')') Prntss--;
                if (Prntss < 0) {
                    EsFinal = false;
                    break;
                }
            }
        }
    }
    if (EsFinal)

```

```

        Cad = Cad.Substring(1, Cad.Length - 2);
    } while (EsFinal == true);

    //Busca +, -
    Prntss = 0;
    for (int Cont = Cad.Length - 1; Cont >= 0; Cont--) {
        if (Cad[Cont] == '(') Prntss++;
        if (Cad[Cont] == ')') Prntss--;
        if ((Cad[Cont] == '+' || Cad[Cont] == '-') && Prntss == 0) {
            string Izquierda = Cad.Substring(0, Cont);
            string Derecha = Cad.Substring(Cont + 1);
            Arbol = new Nodo(Cad[Cont], Identifica++);
            Arbol.Izquierda = CreaArbol(Izquierda, Arbol.Izquierda);
            Arbol.Derecha = CreaArbol(Derecha, Arbol.Derecha);
            return Arbol;
        }
    }

    //Busca *, /
    for (int Cont = Cad.Length - 1; Cont >= 0; Cont--) {
        if (Cad[Cont] == '(') Prntss++;
        if (Cad[Cont] == ')') Prntss--;
        if ((Cad[Cont] == '*' || Cad[Cont] == '/') && Prntss == 0) {
            string Izquierda = Cad.Substring(0, Cont);
            string Derecha = Cad.Substring(Cont + 1);
            Arbol = new Nodo(Cad[Cont], Identifica++);
            Arbol.Izquierda = CreaArbol(Izquierda, Arbol.Izquierda);
            Arbol.Derecha = CreaArbol(Derecha, Arbol.Derecha);
            return Arbol;
        }
    }

    //Busca ^
    for (int Cont = Cad.Length - 1; Cont >= 0; Cont--) {
        if (Cad[Cont] == '(') Prntss++;
        if (Cad[Cont] == ')') Prntss--;
        if (Cad[Cont] == '^' && Prntss == 0) {
            string Izquierda = Cad.Substring(0, Cont);
            string Derecha = Cad.Substring(Cont + 1);
            Arbol = new Nodo(Cad[Cont], Identifica++);
            Arbol.Izquierda = CreaArbol(Izquierda, Arbol.Izquierda);
            Arbol.Derecha = CreaArbol(Derecha, Arbol.Derecha);
            return Arbol;
        }
    }

    //Sólo queda el número y se le crea el nodo
    double Numero = double.Parse(Cad, CultureInfo.InvariantCulture);

```

```

        Arbol = new Nodo(Numero, Identifica++);
        return Arbol;
    }

    static void Dibujar(Nodo Arbol) {
        if (Arbol != null) {
            if (Arbol.Izquierda != null) {
                Arbol.Imprime();
                Console.Write(" -> ");
                Arbol.Izquierda.Imprime();
                Console.WriteLine(" ");
                Dibujar(Arbol.Izquierda);
            }
            if (Arbol.Derecha != null) {
                Arbol.Imprime();
                Console.Write(" -> ");
                Arbol.Derecha.Imprime();
                Console.WriteLine(" ");
                Dibujar(Arbol.Derecha);
            }
        }
    }

    //Recorrido en Post-Orden para
    //evaluar el árbol binario
    public static double EvaluaArbol(Nodo Arbol) {
        //No hay ramas hijas entonces es un número
        if (Arbol.Izquierda == null)
            return Arbol.Numero;

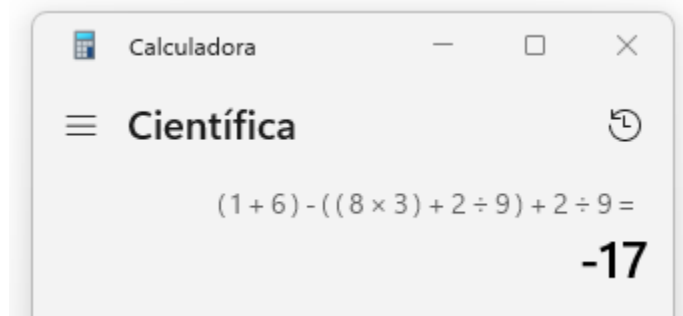
        //Recorrido Post-Orden
        double ValIzquierda = EvaluaArbol(Arbol.Izquierda);
        double ValDerecha = EvaluaArbol(Arbol.Derecha);

        //Aplica operador
        switch (Arbol.Operador) {
            case '+': return ValIzquierda + ValDerecha;
            case '-': return ValIzquierda - ValDerecha;
            case '*': return ValIzquierda * ValDerecha;
            case '/': return ValIzquierda / ValDerecha;
            case '^': return Math.Pow(ValIzquierda, ValDerecha);
        }
        return 0;
    }
}

```

Así ejecuta:

```
Consola de depuración de Mi X +
Resultado es: -17
digraph testgraph{
"[0] +" -> "[1] -"
"[1] -" -> "[2] +"
"[2] +" -> "[3] 1"
"[2] +" -> "[4] 6"
"[1] -" -> "[5] +"
"[5] +" -> "[6] *"
"[6] *" -> "[7] 8"
"[6] *" -> "[8] 3"
"[5] +" -> "[9] /"
"[9] /" -> "[10] 2"
"[9] /" -> "[11] 9"
"[0] +" -> "[12] /"
"[12] /" -> "[13] 2"
"[12] /" -> "[14] 9"
}
```



Fase 5. Variables

Una expresión matemática puede manejar variables, en este caso permite variables de la 'a' hasta la 'z'.

En un arreglo unidimensional se tiene el valor de las variables:

```
static double[] Valores = new double[26];
```

Y las variables se tratan igual que con el evaluador 4.0

```
string Ecuacion = "x+y-z/q";  
DarValorVariable('x', 2);  
DarValorVariable('y', 3);  
DarValorVariable('z', 9);  
DarValorVariable('q', 3);
```

Y la función DarValorVariable

```
/* Da valor a las variables que tendrá  
 * la expresión algebraica */  
public static void DarValorVariable(char varAlgebra, double Valor) {  
    Valores[varAlgebra - 'a'] = Valor;  
}
```

Este sería el código:

H/Árbol/006.cs

```
using System.Globalization;  
  
//Forma el árbol binario dada una expresión matemática  
//de números, variables, paréntesis y los operadores:  
//Suma, resta, multiplicación, división y potencia  
//Luego evalúa la expresión  
namespace ArbolBinarioEvaluador {  
    internal class Nodo {  
        public int ID;  
        public char Operador; //+ - * / ^  
        public double Numero;  
        public int Variable;
```



```

public Nodo Izquierda, Derecha;

//Si el nodo tiene operador + - * / ^
public Nodo(char Operador, int identifica) {
    Variable = -1;
    Numero = 0;
    this.Operador = Operador;
    Izquierda = null;
    Derecha = null;
    ID = identifica;
}

//Si el Nodo sólo tiene número
public Nodo(double Numero, int identifica) {
    Variable = -1;
    this.Numero = Numero;
    Operador = '#';
    Izquierda = null;
    Derecha = null;
    ID = identifica;
}

//Si el Nodo sólo tiene variable
public Nodo(int Variable, int identifica) {
    this.Variable = Variable;
    Numero = 0;
    Operador = '#';
    Izquierda = null;
    Derecha = null;
    ID = identifica;
}

public void Imprime() {
    if (Operador != '#')
        Console.WriteLine("\n[" + ID + "] " + Operador + "\n");
    else if (Variable != -1) {
        int Ascii = 'a' + Variable;
        char Letra = (char)Ascii;
        Console.WriteLine("\n[" + ID + "] " + Letra + "\n");
    }
    else
        Console.WriteLine("\n[" + ID + "] " + Numero + "\n");
}

}

internal class Program {
    static int Identifica = 0;
    static double[] Valores = new double[26];
}

```

```

static void Main(string[] args) {
    //Ecuación los cuatro operadores
    string Ecuacion = "x+y-z/q";
    DarValorVariable('x', 2);
    DarValorVariable('y', 3);
    DarValorVariable('z', 9);
    DarValorVariable('q', 3);

    Nodo MiArbol = null;
    MiArbol = CreaArbol(Ecuacion, MiArbol);

    //Evalúa la expresión
    double Resultado = EvaluaArbol(MiArbol);
    Console.WriteLine("Resultado es: " + Resultado);

    //Probarlo en: http://viz-js.com
    Console.WriteLine("digraph testgraph{");
    Dibujar(MiArbol);
    Console.WriteLine("}");
}

public static Nodo CreaArbol(string Cad, Nodo Arbol) {
    /* Elimina paréntesis al inicio y al final si es
       una expresión del tipo:
           (expresión de números y operadores)
       la convierte en:
           expresión de números y operadores
       Ejemplo:
           (2^3-8+7*2-14+7/2)
       Se vuelve:
           2^3-8+7*2-14+7/2

       Pero si encuentra algo así:
           (2+4) * (5-2)
       No aplica tal conversión porque no son
       paréntesis que cubren toda la expresión
    */
    int Prntss;
    bool EsFinal;
    do {
        EsFinal = false;
        if (Cad[0] == '(') {
            EsFinal = true;
            Prntss = 0;
            for (int Cont = 1; Cont < Cad.Length - 1; Cont++) {
                if (Cad[Cont] == '(') Prntss++;
                if (Cad[Cont] == ')') Prntss--;
            }
        }
    } while (!EsFinal);
}

```

```

        if (Prntss < 0) {
            EsFinal = false;
            break;
        }
    }
}
if (EsFinal)
    Cad = Cad.Substring(1, Cad.Length - 2);
} while (EsFinal == true);

//Busca +, -
Prntss = 0;
for (int Cont = Cad.Length - 1; Cont >= 0; Cont--) {
    if (Cad[Cont] == '(') Prntss++;
    if (Cad[Cont] == ')') Prntss--;
    if ((Cad[Cont] == '+' || Cad[Cont] == '-') && Prntss == 0) {
        string Izquierda = Cad.Substring(0, Cont);
        string Derecha = Cad.Substring(Cont + 1);
        Arbol = new Nodo(Cad[Cont], Identifica++);
        Arbol.Izquierda = CreaArbol(Izquierda, Arbol.Izquierda);
        Arbol.Derecha = CreaArbol(Derecha, Arbol.Derecha);
        return Arbol;
    }
}

//Busca *, /
for (int Cont = Cad.Length - 1; Cont >= 0; Cont--) {
    if (Cad[Cont] == '(') Prntss++;
    if (Cad[Cont] == ')') Prntss--;
    if ((Cad[Cont] == '*' || Cad[Cont] == '/') && Prntss == 0) {
        string Izquierda = Cad.Substring(0, Cont);
        string Derecha = Cad.Substring(Cont + 1);
        Arbol = new Nodo(Cad[Cont], Identifica++);
        Arbol.Izquierda = CreaArbol(Izquierda, Arbol.Izquierda);
        Arbol.Derecha = CreaArbol(Derecha, Arbol.Derecha);
        return Arbol;
    }
}

//Busca ^
for (int Cont = Cad.Length - 1; Cont >= 0; Cont--) {
    if (Cad[Cont] == '(') Prntss++;
    if (Cad[Cont] == ')') Prntss--;
    if (Cad[Cont] == '^' && Prntss == 0) {
        string Izquierda = Cad.Substring(0, Cont);
        string Derecha = Cad.Substring(Cont + 1);
        Arbol = new Nodo(Cad[Cont], Identifica++);
        Arbol.Izquierda = CreaArbol(Izquierda, Arbol.Izquierda);

```

```

        Arbol.Derecha = CreaArbol(Derecha, Arbol.Derecha);
        return Arbol;
    }
}

//Sólo queda el número o la variable y se le crea el nodo
if (Cad[0] >= 'a' && Cad[0] <= 'z') {
    int Variable = (int)Cad[0] - 'a';
    Arbol = new Nodo(Variable, Identifica++);
}
else {
    double Numero;
    Numero = double.Parse(Cad, CultureInfo.InvariantCulture);
    Arbol = new Nodo(Numero, Identifica++);
}

return Arbol;
}

/* Da valor a las variables que tendrá
 * la expresión algebraica */
public static void DarValorVariable(char varAlgebra, double Valor) {
    Valores[varAlgebra - 'a'] = Valor;
}

static void Dibujar(Nodo Arbol) {
    if (Arbol != null) {
        if (Arbol.Izquierda != null) {
            Arbol.Imprime();
            Console.Write(" -> ");
            Arbol.Izquierda.Imprime();
            Console.WriteLine(" ");
            Dibujar(Arbol.Izquierda);
        }
        if (Arbol.Derecha != null) {
            Arbol.Imprime();
            Console.Write(" -> ");
            Arbol.Derecha.Imprime();
            Console.WriteLine(" ");
            Dibujar(Arbol.Derecha);
        }
    }
}

//Recorrido en Post-Orden para
//evaluar el árbol binario
public static double EvaluaArbol(Nodo Arbol) {
    //No hay rama hija entonces es número o variable

```

```

    if (Arbol.Izquierda == null)
        if (Arbol.Variable != -1)
            return Valores[Arbol.Variable];
        else
            return Arbol.Numero;

    //Recorrido Post-Orden
    double ValIzquierda = EvaluaArbol(Arbol.Izquierda);
    double ValDerecha = EvaluaArbol(Arbol.Derecha);

    //Aplica operador
    switch (Arbol.Operador) {
        case '+': return ValIzquierda + ValDerecha;
        case '-': return ValIzquierda - ValDerecha;
        case '*': return ValIzquierda * ValDerecha;
        case '/': return ValIzquierda / ValDerecha;
        case '^': return Math.Pow(ValIzquierda, ValDerecha);
    }
    return 0;
}
}
}

```

Una expresión como:

$x+y-z/q$

Genera un árbol binario así:

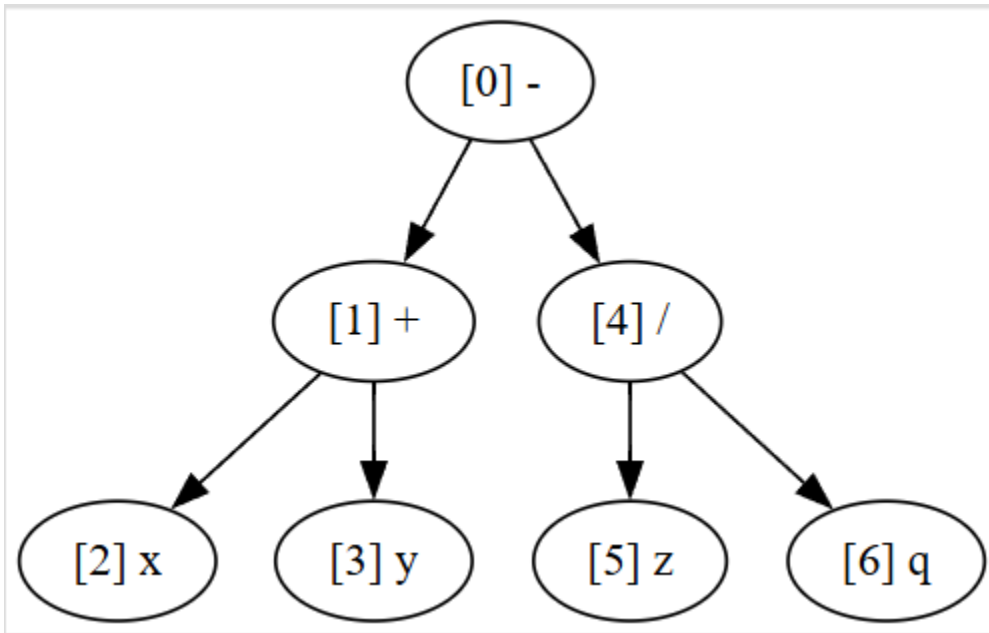
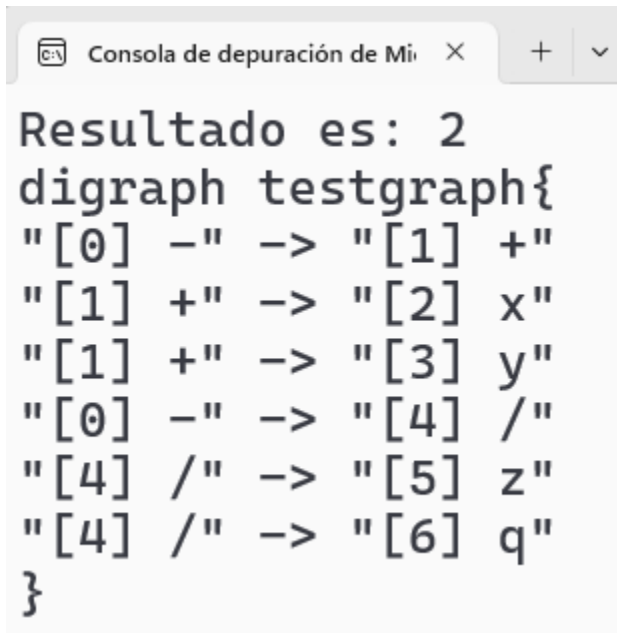


Ilustración 28: Árbol binario generado

Y esta es su ejecución:

A screenshot of a web browser's developer console. The console tab is titled 'Consola de depuración de Mi...'. It displays the output of a JavaScript execution. The first line shows 'Resultado es: 2'. The second line shows a function definition for 'digraph testgraph'. The function body contains seven lines of code that map array indices to mathematical symbols: index 0 to '-', index 1 to '+', index 2 to 'x', index 3 to 'y', index 4 to '/', index 5 to 'z', and index 6 to 'q'. The function is enclosed in curly braces.

```
Resultado es: 2
digraph testgraph{
"[0] -" -> "[1] +"
"[1] +" -> "[2] x"
"[1] +" -> "[3] y"
"[0] -" -> "[4] /"
"[4] /" -> "[5] z"
"[4] /" -> "[6] q"
}
```

Ilustración 29: Ejemplo de ejecución

Fase 6. Funciones matemáticas

Finalmente vienen las funciones matemáticas como seno, coseno, tangente. En este caso, la función tiene su propio nodo, la rama izquierda es la que tiene toda la operación interna que va a ser luego evaluada por la función, como requiere una rama derecha, entonces esta rama derecha es simplemente un número cero.

Las funciones son de tres letras. Una expresión como:

$$3-\text{sen}(x*y+2)$$

Se reemplaza a:

$$3-A(x*y+2)$$

Así facilita evaluar la expresión.

Esta es la tabla de equivalencias:

Función	Descripción	Letra con que se reemplaza
Sen	Seno	A
Cos	Coseno	B
Tan	Tangente	C
Abs	Valor absoluto	D
Asn	Arcoseno	E
Acs	Arcocoseno	F
Atn	Arcotangente	G
Log	Logaritmo Natural	H
Exp	Exponencial	I
Sqr	Raíz cuadrada	J

```
using System.Globalization;

//Forma el árbol binario dada una expresión matemática
//de números, variables, paréntesis, funciones y los operadores:
//Suma, resta, multiplicación, división y potencia
//Luego evalúa la expresión
namespace ArbolBinarioEvaluador {
    internal class Nodo {
        public int ID;
        public char Operador; //+ - * / ^
        public double Numero;
        public int Variable;
        public int Funcion;
        public Nodo Izquierda, Derecha;

        //Si el nodo tiene operador + - * / ^
        public Nodo(char Operador, int identifica) {
            Variable = -1;
            Numero = 0;
            this.Operador = Operador;
            Funcion = -1;
            Izquierda = null;
            Derecha = null;
            ID = identifica;
        }

        //Si el Nodo sólo tiene número
        public Nodo(double Numero, int identifica) {
            Variable = -1;
            this.Numero = Numero;
            Operador = '#';
            Funcion = -1;
            Izquierda = null;
            Derecha = null;
            ID = identifica;
        }

        //Si el Nodo sólo tiene variable
        public Nodo(int Variable, int identifica) {
            this.Variable = Variable;
            Numero = 0;
            Operador = '#';
            Funcion = -1;
            Izquierda = null;
            Derecha = null;
        }
    }
}
```



```

        ID = identifica;
    }

    //Si el Nodo es de una función
    public Nodo(int Funcion, int identifica, bool EsFuncion) {
        Variable = -1;
        Numero = 0;
        Operador = '#';
        this.Funcion = Funcion;
        Izquierda = null;
        Derecha = null;
        ID = identifica;
    }

    public void Imprime() {
        if (Operador != '#')
            Console.WriteLine("\n[" + ID + "] " + Operador + "\n");
        else if (Variable != -1) {
            int Ascii = 'a' + Variable;
            char Letra = (char)Ascii;
            Console.WriteLine("\n[" + ID + "] " + Letra + "\n");
        }
        else if (Funcion != -1) {
            Console.WriteLine("\n[" + ID + "] ");
            /* Código de la función 0:seno, 1:coseno, 2:tangente,
             * 3: valor absoluto, 4: arcoseno, 5: arcocoseno,
             * 6: arcotangente, 7: logaritmo natural, 8: valor tope,
             * 9: exponencial, 10: raíz cuadrada */
            switch (Funcion) {
                case 0: Console.WriteLine("seno \n"); break;
                case 1: Console.WriteLine("coseno \n"); break;
                case 2: Console.WriteLine("tangente \n"); break;
                case 3: Console.WriteLine("absoluto \n"); break;
                case 4: Console.WriteLine("arcoseno \n"); break;
                case 5: Console.WriteLine("arcocoseno \n"); break;
                case 6: Console.WriteLine("arcotangente \n"); break;
                case 7: Console.WriteLine("log \n"); break;
                case 8: Console.WriteLine("ceil \n"); break;
                case 9: Console.WriteLine("exp \n"); break;
                case 10: Console.WriteLine("sqrt \n"); break;
            }
        }
        else
            Console.WriteLine("\n[" + ID + "] " + Numero + "\n");
    }
}

```

```

internal class Program {
    static int Identifica = 0;
    static double[] Valores = new double[26];

    static void Main(string[] args) {
        //Ecuación operadores, variables, funciones, paréntesis
        string Ecuacion = "sen(cos(x)+sen(y+z))/cos(78-q)";
        string Convertir = Convierte(Ecuacion);
        DarValorVariable('x', 120);
        DarValorVariable('y', 150);
        DarValorVariable('z', 45);
        DarValorVariable('q', -10);

        Nodo MiArbol = null;
        MiArbol = CreaArbol(Convertir, MiArbol);

        //Evalúa la expresión
        double Resultado = EvaluaArbol(MiArbol);
        Console.WriteLine("Resultado es: " + Resultado);

        //Probarlo en: http://viz-js.com
        Console.WriteLine("digraph testgraph{");
        Dibujar(MiArbol);
        Console.WriteLine("}");
    }

    public static Nodo CreaArbol(string Cad, Nodo Arbol) {
        int Prntss;
        bool EsFinal;

        //Detecta si es función
        if (Cad[0] >= 'A' && Cad[0] <= 'J') {
            //Busca el paréntesis que cierra la función
            EsFinal = true;
            Prntss = 0;
            for (int Cont = 2; Cont < Cad.Length - 1; Cont++) {
                if (Cad[Cont] == '(') Prntss++;
                if (Cad[Cont] == ')') Prntss--;
                if (Prntss < 0) {
                    EsFinal = false;
                    break;
                }
            }
        }

        //¿Es una función en solitario?
        if (EsFinal) {
            int Ascii = Cad[0] - 'A';
            Arbol = new Nodo(Ascii, Identifica++, true);
        }
    }
}

```

```

        Arbol.Derecha = new Nodo(0.0, Identifica);

        //Retira la letra A, el primer y último paréntesis
        Cad = Cad.Substring(2, Cad.Length - 3);
        Arbol.Izquierda = CreaArbol(Cad, Arbol.Izquierda);
        return Arbol;
    }
}

/* Elimina paréntesis al inicio y al final si es
una expresión del tipo:
    (expresión de números y operadores)
la convierte en:
    expresión de números y operadores
Ejemplo:
    (2^3-8+7*2-14+7/2)
Se vuelve:
    2^3-8+7*2-14+7/2

Pero si encuentra algo así:
    (2+4) * (5-2)
No aplica tal conversión porque no son
paréntesis que cubren toda la expresión
*/
do {
    EsFinal = false;
    if (Cad[0] == '(') {
        EsFinal = true;
        Prntss = 0;
        for (int Cont = 1; Cont < Cad.Length - 1; Cont++) {
            if (Cad[Cont] == '(') Prntss++;
            if (Cad[Cont] == ')') Prntss--;
            if (Prntss < 0) {
                EsFinal = false;
                break;
            }
        }
    }
    if (EsFinal)
        Cad = Cad.Substring(1, Cad.Length - 2);
} while (EsFinal == true);

//Busca +, -
Prntss = 0;
for (int Cont = Cad.Length - 1; Cont >= 0; Cont--) {
    if (Cad[Cont] == '(') Prntss++;
    if (Cad[Cont] == ')') Prntss--;
    if ((Cad[Cont] == '+' || Cad[Cont] == '-') && Prntss == 0) {

```

```

        string Izquierda = Cad.Substring(0, Cont);
        string Derecha = Cad.Substring(Cont + 1);
        Arbol = new Nodo(Cad[Cont], Identifica++);
        Arbol.Izquierda = CreaArbol(Izquierda, Arbol.Izquierda);
        Arbol.Derecha = CreaArbol(Derecha, Arbol.Derecha);
        return Arbol;
    }
}

//Busca *, /
for (int Cont = Cad.Length - 1; Cont >= 0; Cont--) {
    if (Cad[Cont] == '(') Prntss++;
    if (Cad[Cont] == ')') Prntss--;
    if ((Cad[Cont] == '*' || Cad[Cont] == '/') && Prntss == 0) {
        string Izquierda = Cad.Substring(0, Cont);
        string Derecha = Cad.Substring(Cont + 1);
        Arbol = new Nodo(Cad[Cont], Identifica++);
        Arbol.Izquierda = CreaArbol(Izquierda, Arbol.Izquierda);
        Arbol.Derecha = CreaArbol(Derecha, Arbol.Derecha);
        return Arbol;
    }
}

//Busca ^
for (int Cont = Cad.Length - 1; Cont >= 0; Cont--) {
    if (Cad[Cont] == '(') Prntss++;
    if (Cad[Cont] == ')') Prntss--;
    if (Cad[Cont] == '^' && Prntss == 0) {
        string Izquierda = Cad.Substring(0, Cont);
        string Derecha = Cad.Substring(Cont + 1);
        Arbol = new Nodo(Cad[Cont], Identifica++);
        Arbol.Izquierda = CreaArbol(Izquierda, Arbol.Izquierda);
        Arbol.Derecha = CreaArbol(Derecha, Arbol.Derecha);
        return Arbol;
    }
}

//Sólo queda el número o la variable y se le crea el nodo
if (Cad[0] >= 'a' && Cad[0] <= 'z') {
    int Variable = Cad[0] - 'a';
    Arbol = new Nodo(Variable, Identifica++);
}
else {
    double Numero;
    Numero = double.Parse(Cad, CultureInfo.InvariantCulture);
    Arbol = new Nodo(Numero, Identifica++);
}
}

```

```

    return Arbol;
}

/* Da valor a las variables que tendrá
 * la expresión algebraica */
public static void DarValorVariable(char varAlgebra, double Valor) {
    Valores[varAlgebra - 'a'] = Valor;
}

static void Dibujar(Nodo Arbol) {
    if (Arbol != null) {
        if (Arbol.Izquierda != null) {
            Arbol.Imprime();
            Console.Write(" -> ");
            Arbol.Izquierda.Imprime();
            Console.WriteLine(" ");
            Dibujar(Arbol.Izquierda);
        }
        if (Arbol.Derecha != null) {
            Arbol.Imprime();
            Console.Write(" -> ");
            Arbol.Derecha.Imprime();
            Console.WriteLine(" ");
            Dibujar(Arbol.Derecha);
        }
    }
}

//Recorrido en Post-Orden para
//evaluar el árbol binario
public static double EvaluaArbol(Nodo Arbol) {
    //No hay rama hija entonces es número o variable
    if (Arbol.Izquierda == null)
        if (Arbol.Variable != -1)
            return Valores[Arbol.Variable];
        else
            return Arbol.Numero;

    //Recorrido Post-Orden
    double ValIzquierda = EvaluaArbol(Arbol.Izquierda);
    double ValDerecha = EvaluaArbol(Arbol.Derecha);

    /* Código de la función 0:seno, 1:coseno, 2:tangente,
     * 3: valor absoluto, 4: arcoseno, 5: arcocoseno,
     * 6: arcotangente, 7: logaritmo natural, 8: valor tope,
     * 9: exponencial, 10: raíz cuadrada */
    if (Arbol.Funcion != -1) {
        switch (Arbol.Funcion) {

```

```

        case 0: return Math.Sin(ValIzquierda);
        case 1: return Math.Cos(ValIzquierda);
        case 2: return Math.Tan(ValIzquierda);
        case 3: return Math.Abs(ValIzquierda);
        case 4: return Math.Asin(ValIzquierda);
        case 5: return Math.Acos(ValIzquierda);
        case 6: return Math.Atan(ValIzquierda);
        case 7: return Math.Log(ValIzquierda);
        case 8: return Math.Ceiling(ValIzquierda);
        case 9: return Math.Exp(ValIzquierda);
        case 10: return Math.Sqrt(ValIzquierda);
    }
}

//Aplica operador
switch (Arbol.Operador) {
    case '+': return ValIzquierda + ValDerecha;
    case '-': return ValIzquierda - ValDerecha;
    case '*': return ValIzquierda * ValDerecha;
    case '/': return ValIzquierda / ValDerecha;
    case '^': return Math.Pow(ValIzquierda, ValDerecha);
}
return 0;
}

//Convierte la expresión algebraica, escrita por el usuario,
//a un formato que pueda ser interpretado por el evaluador
static string Convierte(string ExpOrig) {
    /* Primero a minúsculas */
    string Minusculas = ExpOrig.ToLower();

    /* Cadena a evaluar */
    string Cadena = Minusculas.Replace("sen", "A");
    Cadena = Cadena.Replace("cos", "B");
    Cadena = Cadena.Replace("tan", "C");
    Cadena = Cadena.Replace("abs", "D");
    Cadena = Cadena.Replace("asn", "E");
    Cadena = Cadena.Replace("acs", "F");
    Cadena = Cadena.Replace("atn", "G");
    Cadena = Cadena.Replace("log", "H");
    Cadena = Cadena.Replace("exp", "I");
    Cadena = Cadena.Replace("sqr", "J");

    return Cadena;
}
}
}

```

```
Consola de depuración de Mi X + v
Resultado es: 0,8597038048551232
digraph testgraph{
"[0] /" -> "[1] seno "
"[1] seno " -> "[2] +"
"[2] +" -> "[3] coseno "
"[3] coseno " -> "[4] x"
"[3] coseno " -> "[4] 0"
"[2] +" -> "[5] seno "
"[5] seno " -> "[6] +"
"[6] +" -> "[7] y"
"[6] +" -> "[8] z"
"[5] seno " -> "[6] 0"
"[1] seno " -> "[2] 0"
"[0] /" -> "[9] coseno "
"[9] coseno " -> "[10] -"
"[10] -" -> "[11] 78"
"[10] -" -> "[12] q"
"[9] coseno " -> "[10] 0"
}
```

Ilustración 30: Ejemplo de ejecución

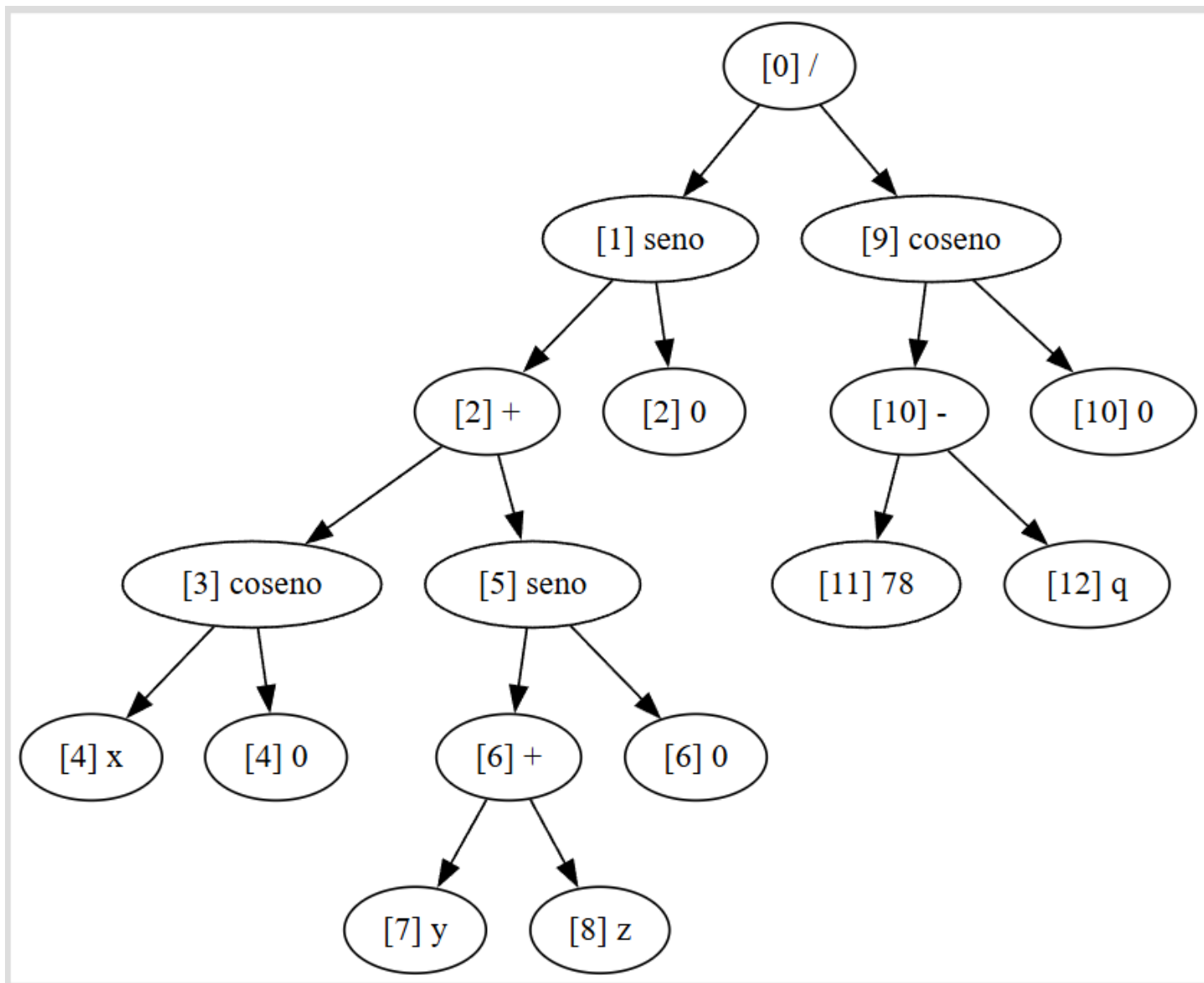


Ilustración 31: Árbol binario generado

Fase 7. Orientado a objetos

Ahora es volver el evaluador como clases (aplicar la programación orientada a objetos). La primera clase es el Nodo.

H/Árbol/008/Nodo.cs

```
namespace ArbolBinarioEvaluador {
    internal class Nodo {
        public int ID;
        public char Operador; //+ - * / ^
        public double Numero;
        public int Variable;
        public int Funcion;
        public Nodo Izquierda, Derecha;

        //Si el nodo tiene operador + - * / ^
        public Nodo(char Operador, int identifica) {
            Variable = -1;
            Numero = 0;
            this.Operador = Operador;
            Funcion = -1;
            Izquierda = null;
            Derecha = null;
            ID = identifica;
        }

        //Si el Nodo sólo tiene número
        public Nodo(double Numero, int identifica) {
            Variable = -1;
            this.Numero = Numero;
            Operador = '#';
            Funcion = -1;
            Izquierda = null;
            Derecha = null;
            ID = identifica;
        }

        //Si el Nodo sólo tiene variable
        public Nodo(int Variable, int identifica) {
            this.Variable = Variable;
            Numero = 0;
            Operador = '#';
            Funcion = -1;
            Izquierda = null;
            Derecha = null;
            ID = identifica;
        }
    }
}
```

```

}

//Si el Nodo es de una función
public Nodo(int Funcion, int identifica, bool EsFuncion) {
    Variable = -1;
    Numero = 0;
    Operador = '#';
    this.Funcion = Funcion;
    Izquierda = null;
    Derecha = null;
    ID = identifica;
}

public void Imprime() {
    if (Operador != '#')
        Console.WriteLine("\n[" + ID + "] " + Operador + "\n");
    else if (Variable != -1) {
        int Ascii = 'a' + Variable;
        char Letra = (char)Ascii;
        Console.WriteLine("\n[" + ID + "] " + Letra + "\n");
    }
    else if (Funcion != -1) {
        Console.WriteLine("\n[" + ID + "] ");
        /* Código de la función 0:seno, 1:coseno, 2:tangente,
        * 3: valor absoluto, 4: arcoseno, 5: arcocoseno,
        * 6: arcotangente, 7: logaritmo natural, 8: valor tope,
        * 9: exponencial, 10: raíz cuadrada */
        switch (Funcion) {
            case 0: Console.WriteLine("seno \n"); break;
            case 1: Console.WriteLine("coseno \n"); break;
            case 2: Console.WriteLine("tangente \n"); break;
            case 3: Console.WriteLine("absoluto \n"); break;
            case 4: Console.WriteLine("arcoseno \n"); break;
            case 5: Console.WriteLine("arcocoseno \n"); break;
            case 6: Console.WriteLine("arcotangente \n"); break;
            case 7: Console.WriteLine("log \n"); break;
            case 8: Console.WriteLine("ceil \n"); break;
            case 9: Console.WriteLine("exp \n"); break;
            case 10: Console.WriteLine("sqrt \n"); break;
        }
    }
    else
        Console.WriteLine("\n[" + ID + "] " + Numero + "\n");
}
}
}

```

```

using System.Globalization;

/* Evaluador de expresiones usando un árbol binario */
namespace ArbolBinarioEvaluador {
    internal class EvalArbolBin {

        /* Usado para dibujar el árbol */
        private int Identifica = 0;

        /* Las 26 variables que puede tener la expresión */
        private double[] Valores = new double[26];

        /* Árbol binario */
        private Nodo MiArbol;

        public void Analizar(string Ecuacion) {
            string Convertir = Convierte(Ecuacion);

            MiArbol = null;
            MiArbol = CreaArbol(Convertir, MiArbol);
        }

        /* Da valor a las variables que tendrá
         * la expresión algebraica */
        public void DarValorVariable(char varAlgebra, double Valor) {
            Valores[varAlgebra - 'a'] = Valor;
        }

        /* Dibuja el árbol generado */
        public void Dibujar() {
            //Probarlo en: http://viz-js.com
            Console.WriteLine("digraph testgraph{");
            DibujaArbol(MiArbol);
            Console.WriteLine("}");
        }

        /* Evalúa la expresión ya analizada */
        public double Evaluar() {
            return EvaluaArbol(MiArbol);
        }

        /* Analiza la expresión generando un árbol binario */
        private Nodo CreaArbol(string Cad, Nodo Arbol) {
            int Prntss;
            bool EsFinal;

```

```

//Detecta si es función
if (Cad[0] >= 'A' && Cad[0] <= 'J') {
    //Busca el paréntesis que cierra la función
    EsFinal = true;
    Prntss = 0;
    for (int Cont = 2; Cont < Cad.Length - 1; Cont++) {
        if (Cad[Cont] == '(') Prntss++;
        if (Cad[Cont] == ')') Prntss--;
        if (Prntss < 0) {
            EsFinal = false;
            break;
        }
    }
}

//¿Es una función en solitario?
if (EsFinal) {
    int Ascii = Cad[0] - 'A';
    Arbol = new Nodo(Ascii, Identifica++, true);
    Arbol.Derecha = new Nodo(0.0, Identifica);

    //Retira la letra A, el primer y último paréntesis
    Cad = Cad.Substring(2, Cad.Length - 3);
    Arbol.Izquierda = CreaArbol(Cad, Arbol.Izquierda);
    return Arbol;
}
}

/* Elimina paréntesis al inicio y al final si es
una expresión del tipo:
    (expresión de números y operadores)
la convierte en:
    expresión de números y operadores
Ejemplo:
    (2^3-8+7*2-14+7/2)
Se vuelve:
    2^3-8+7*2-14+7/2

Pero si encuentra algo así:
    (2+4) * (5-2)
No aplica tal conversión porque no son
paréntesis que cubren toda la expresión
*/
do {
    EsFinal = false;
    if (Cad[0] == '(') {
        EsFinal = true;
        Prntss = 0;
    }
}

```

```

        for (int Cont = 1; Cont < Cad.Length - 1; Cont++) {
            if (Cad[Cont] == '(') Prntss++;
            if (Cad[Cont] == ')') Prntss--;
            if (Prntss < 0) {
                EsFinal = false;
                break;
            }
        }
    }
    if (EsFinal)
        Cad = Cad.Substring(1, Cad.Length - 2);
} while (EsFinal == true);

//Busca +, -
Prntss = 0;
for (int Cont = Cad.Length - 1; Cont >= 0; Cont--) {
    if (Cad[Cont] == '(') Prntss++;
    if (Cad[Cont] == ')') Prntss--;
    if ((Cad[Cont] == '+' || Cad[Cont] == '-') && Prntss == 0) {
        string Izquierda = Cad.Substring(0, Cont);
        string Derecha = Cad.Substring(Cont + 1);
        Arbol = new Nodo(Cad[Cont], Identifica++);
        Arbol.Izquierda = CreaArbol(Izquierda, Arbol.Izquierda);
        Arbol.Derecha = CreaArbol(Derecha, Arbol.Derecha);
        return Arbol;
    }
}

//Busca *, /
for (int Cont = Cad.Length - 1; Cont >= 0; Cont--) {
    if (Cad[Cont] == '(') Prntss++;
    if (Cad[Cont] == ')') Prntss--;
    if ((Cad[Cont] == '*' || Cad[Cont] == '/') && Prntss == 0) {
        string Izquierda = Cad.Substring(0, Cont);
        string Derecha = Cad.Substring(Cont + 1);
        Arbol = new Nodo(Cad[Cont], Identifica++);
        Arbol.Izquierda = CreaArbol(Izquierda, Arbol.Izquierda);
        Arbol.Derecha = CreaArbol(Derecha, Arbol.Derecha);
        return Arbol;
    }
}

//Busca ^
for (int Cont = Cad.Length - 1; Cont >= 0; Cont--) {
    if (Cad[Cont] == '(') Prntss++;
    if (Cad[Cont] == ')') Prntss--;
    if (Cad[Cont] == '^' && Prntss == 0) {
        string Izquierda = Cad.Substring(0, Cont);

```

```

        string Derecha = Cad.Substring(Cont + 1);
        Arbol = new Nodo(Cad[Cont], Identifica++);
        Arbol.Izquierda = CreaArbol(Izquierda, Arbol.Izquierda);
        Arbol.Derecha = CreaArbol(Derecha, Arbol.Derecha);
        return Arbol;
    }
}

//Sólo queda el número o la variable y se le crea el nodo
if (Cad[0] >= 'a' && Cad[0] <= 'z') {
    int Variable = Cad[0] - 'a';
    Arbol = new Nodo(Variable, Identifica++);
}
else {
    double Numero;
    Numero = double.Parse(Cad, CultureInfo.InvariantCulture);
    Arbol = new Nodo(Numero, Identifica++);
}

return Arbol;
}

/* Dibuja el árbol binario */
private void DibujaArbol(Nodo Arbol) {
    if (Arbol != null) {
        if (Arbol.Izquierda != null) {
            Arbol.Imprime();
            Console.Write(" -> ");
            Arbol.Izquierda.Imprime();
            Console.WriteLine(" ");
            DibujaArbol(Arbol.Izquierda);
        }
        if (Arbol.Derecha != null) {
            Arbol.Imprime();
            Console.Write(" -> ");
            Arbol.Derecha.Imprime();
            Console.WriteLine(" ");
            DibujaArbol(Arbol.Derecha);
        }
    }
}

/* Recorrido en Post-Orden para
evaluar el árbol binario */
private double EvaluaArbol(Nodo Arbol) {
    //No hay rama hija entonces es número o variable
    if (Arbol.Izquierda == null)
        if (Arbol.Variable != -1)

```

```

        return Valores[Arbol.Variable];
    else
        return Arbol.Numero;

//Recorrido Post-Orden
double ValIzquierda = EvaluaArbol(Arbol.Izquierda);
double ValDerecha = EvaluaArbol(Arbol.Derecha);

/* Código de la función 0:seno, 1:coseno, 2:tangente,
 * 3: valor absoluto, 4: arcoseno, 5: arcocoseno,
 * 6: arcotangente, 7: logaritmo natural, 8: valor tope,
 * 9: exponencial, 10: raíz cuadrada */
if (Arbol.Funcion != -1) {
    switch (Arbol.Funcion) {
        case 0: return Math.Sin(ValIzquierda);
        case 1: return Math.Cos(ValIzquierda);
        case 2: return Math.Tan(ValIzquierda);
        case 3: return Math.Abs(ValIzquierda);
        case 4: return Math.Asin(ValIzquierda);
        case 5: return Math.Acos(ValIzquierda);
        case 6: return Math.Atan(ValIzquierda);
        case 7: return Math.Log(ValIzquierda);
        case 8: return Math.Ceiling(ValIzquierda);
        case 9: return Math.Exp(ValIzquierda);
        case 10: return Math.Sqrt(ValIzquierda);
    }
}

//Aplica operador
switch (Arbol.Operador) {
    case '+': return ValIzquierda + ValDerecha;
    case '-': return ValIzquierda - ValDerecha;
    case '*': return ValIzquierda * ValDerecha;
    case '/': return ValIzquierda / ValDerecha;
    case '^': return Math.Pow(ValIzquierda, ValDerecha);
}
return 0;
}

/* Convierte la expresión algebraica, escrita por el usuario,
a un formato que pueda ser interpretado por el evaluador */
private string Convierte(string ExpOrig) {
    /* Primero a minúsculas */
    string Minusculas = ExpOrig.ToLower();

    /* Cadena a evaluar */
    string Cadena = Minusculas.Replace("sen", "A");
    Cadena = Cadena.Replace("cos", "B");
}

```

```

        Cadena = Cadena.Replace("tan", "C");
        Cadena = Cadena.Replace("abs", "D");
        Cadena = Cadena.Replace("asn", "E");
        Cadena = Cadena.Replace("acs", "F");
        Cadena = Cadena.Replace("atn", "G");
        Cadena = Cadena.Replace("log", "H");
        Cadena = Cadena.Replace("exp", "I");
        Cadena = Cadena.Replace("sqr", "J");

        return Cadena;
    }
}

```

Y el que lo utiliza

H/Árbol/008/Program.cs

```

namespace ArbolBinarioEvaluador {
//Forma el árbol binario dada una expresión matemática
//de números, variables, paréntesis, funciones y los operadores:
//Suma, resta, multiplicación, división y potencia
//Luego evalúa la expresión
    internal class Program {

        static void Main(string[] args) {
            //Ecuación operadores, variables, funciones, paréntesis
            string Ecuacion = "sen(cos(x)+sen(y+z))/cos(78-q) ";

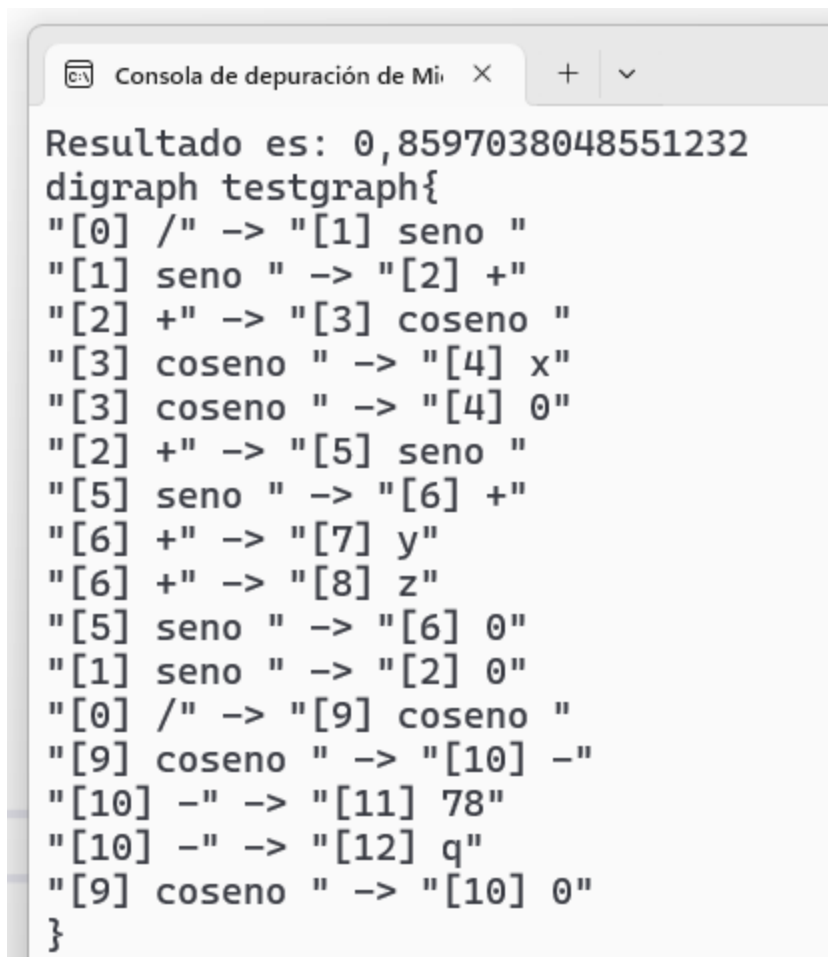
            EvalArbolBin obj = new();
            obj.Analizar(Ecuacion);
            obj.DarValorVariable('x', 120);
            obj.DarValorVariable('y', 150);
            obj.DarValorVariable('z', 45);
            obj.DarValorVariable('q', -10);

            //Evalúa la expresión
            double Resultado = obj.Evaluar();
            Console.WriteLine("Resultado es: " + Resultado);

            //Probarlo en: http://viz-js.com
            obj.Dibujar();
        }
    }
}

```

Ejemplo de ejecución:



The image shows a web browser window with a developer console open. The console title is "Consola de depuración de Mi". The output of the console is as follows:

```
Resultado es: 0,8597038048551232
digraph testgraph{
"[0] /" -> "[1] seno "
"[1] seno " -> "[2] +"
"[2] +" -> "[3] coseno "
"[3] coseno " -> "[4] x"
"[3] coseno " -> "[4] 0"
"[2] +" -> "[5] seno "
"[5] seno " -> "[6] +"
"[6] +" -> "[7] y"
"[6] +" -> "[8] z"
"[5] seno " -> "[6] 0"
"[1] seno " -> "[2] 0"
"[0] /" -> "[9] coseno "
"[9] coseno " -> "[10] -"
"[10] -" -> "[11] 78"
"[10] -" -> "[12] q"
"[9] coseno " -> "[10] 0"
}
```

Ilustración 32: Ejemplo de ejecución

Fase 8. Evaluación de sintaxis y optimización

Finalmente se le agrega la evaluación de sintaxis que es muy similar a la del evaluador 4.0. Se retira el atributo ID, porque no es necesario dibujar el árbol binario (además le resta velocidad).

H/Árbol/009/Nodo.cs

```
namespace ArbolBinarioEvaluador {
    internal class Nodo {
        public char Operador; //+ - * / ^
        public double Numero;
        public int Variable;
        public int Funcion;
        public Nodo Izquierda, Derecha;

        //Si el nodo tiene operador + - * / ^
        public Nodo(char Operador) {
            Variable = -1;
            Numero = 0;
            this.Operador = Operador;
            Funcion = -1;
            Izquierda = null;
            Derecha = null;
        }

        //Si el Nodo sólo tiene número
        public Nodo(double Numero) {
            Variable = -1;
            this.Numero = Numero;
            Operador = '#';
            Funcion = -1;
            Izquierda = null;
            Derecha = null;
        }

        //Si el Nodo sólo tiene variable
        public Nodo(int Variable) {
            this.Variable = Variable;
            Numero = 0;
            Operador = '#';
            Funcion = -1;
            Izquierda = null;
            Derecha = null;
        }

        //Si el Nodo es de una función
        public Nodo(int Funcion, bool EsFuncion) {
```

```

        Variable = -1;
        Numero = 0;
        Operador = '#';
        this.Funcion = Funcion;
        Izquierda = null;
        Derecha = null;
    }
}
}

```

H/Árbol/009/EvalArbolBin.cs

```

using System.Globalization;

namespace ArbolBinarioEvaluador {
    internal class EvalArbolBin {
        /* Las 26 variables que puede tener la expresión */
        private double[] Valores = new double[26];

        /* Árbol binario */
        private Nodo MiArbol;

        public int Analizar(string ExpresionOriginal) {
            int Sintaxis = ChequeaSintaxis(ExpresionOriginal);
            if (Sintaxis == 0) {
                for (int cont = 0; cont < Valores.Length; cont++)
                    Valores[cont] = 0;
                string Convertir = Convierte(ExpresionOriginal);
                MiArbol = null;
                MiArbol = CreaArbol(Convertir, MiArbol);
            }
            return Sintaxis;
        }

        /* Da valor a las variables que tendrá
         * la expresión algebraica */
        public void DarValorVariable(char varAlgebra, double Valor) {
            Valores[varAlgebra - 'a'] = Valor;
        }

        /* Evalúa la expresión ya analizada */
        public double Evaluar() {
            return EvaluaArbol(MiArbol);
        }

        /* Retorna mensaje de error sintáctico */
    }
}

```

```
public string MensajeError(int CodigoError) {
    string Msj = "";
    switch (CodigoError) {
        case 1:
            Msj = "1. Número seguido de letra";
            break;

        case 2:
            Msj = "2. Número seguido de paréntesis que abre";
            break;

        case 3:
            Msj = "3. Doble punto seguido";
            break;

        case 4:
            Msj = "4. Punto seguido de operador";
            break;

        case 5:
            Msj = "5. Punto y sigue una letra";
            break;

        case 6:
            Msj = "6. Punto seguido de paréntesis que abre";
            break;

        case 7:
            Msj = "7. Punto seguido de paréntesis que cierra";
            break;

        case 8:
            Msj = "8. Operador seguido de un punto";
            break;

        case 9:
            Msj = "9. Dos operadores estén seguidos";
            break;

        case 10:
            Msj = "10. Operador seguido de paréntesis que cierra";
            break;

        case 11:
            Msj = "11. Letra seguida de número";
            break;

        case 12:
```

```
    Msj = "12. Letra seguida de punto";  
    break;  
  
case 13:  
    Msj = "13. Letra seguida de otra letra";  
    break;  
  
case 14:  
    Msj = "14. Letra seguida de paréntesis que abre";  
    break;  
  
case 15:  
    Msj = "15. Paréntesis que abre seguido de punto";  
    break;  
  
case 16:  
    Msj = "16. Paréntesis que abre y sigue operador";  
    break;  
  
case 17:  
    Msj = "17. Paréntesis que abre y luego cierra";  
    break;  
  
case 18:  
    Msj = "18. Paréntesis que cierra y sigue número";  
    break;  
  
case 19:  
    Msj = "19. Paréntesis que cierra y sigue punto";  
    break;  
  
case 20:  
    Msj = "20. Paréntesis que cierra y sigue letra";  
    break;  
  
case 21:  
    Msj = "21. Paréntesis que cierra y luego abre";  
    break;  
  
case 22:  
    Msj = "22. Inicia con operador";  
    break;  
  
case 23:  
    Msj = "23. Finaliza con operador";  
    break;  
  
case 24:
```

```

        Msj = "24. No hay correspondencia entre paréntesis";
        break;

    case 25:
        Msj = "25. Paréntesis desbalanceados";
        break;

    case 26:
        Msj = "26. Dos o más puntos en número real";
        break;
    }
    return Msj;
}

/* Analiza la expresión generando un árbol binario */
private Nodo CreaArbol(string Cad, Nodo Arbol) {
    int Prntss;
    bool EsFinal;

    //Detecta si es función
    if (Cad[0] >= 'A' && Cad[0] <= 'J') {
        //Busca el paréntesis que cierra la función
        EsFinal = true;
        Prntss = 0;
        for (int Cont = 2; Cont < Cad.Length - 1; Cont++) {
            if (Cad[Cont] == '(') Prntss++;
            if (Cad[Cont] == ')') Prntss--;
            if (Prntss < 0) {
                EsFinal = false;
                break;
            }
        }

        //¿Es una función en solitario?
        if (EsFinal) {
            int Ascii = Cad[0] - 'A';
            Arbol = new Nodo(Ascii, true);
            Arbol.Derecha = new Nodo(0.0);

            //Retira la letra A, el primer y último paréntesis
            Cad = Cad.Substring(2, Cad.Length - 3);
            Arbol.Izquierda = CreaArbol(Cad, Arbol.Izquierda);
            return Arbol;
        }
    }
}

/* Elimina paréntesis al inicio y al final si es
una expresión del tipo:

```

(expresión de números y operadores)

la convierte en:

expresión de números y operadores

Ejemplo:

$(2^3 - 8 + 7 * 2 - 14 + 7 / 2)$

Se vuelve:

$2^3 - 8 + 7 * 2 - 14 + 7 / 2$

Pero si encuentra algo así:

$(2 + 4) * (5 - 2)$

No aplica tal conversión porque no son
paréntesis que cubren toda la expresión

```
*/  
do {  
    EsFinal = false;  
    if (Cad[0] == '(') {  
        EsFinal = true;  
        Prntss = 0;  
        for (int Cont = 1; Cont < Cad.Length - 1; Cont++) {  
            if (Cad[Cont] == '(') Prntss++;  
            if (Cad[Cont] == ')') Prntss--;  
            if (Prntss < 0) {  
                EsFinal = false;  
                break;  
            }  
        }  
    }  
    if (EsFinal)  
        Cad = Cad.Substring(1, Cad.Length - 2);  
} while (EsFinal == true);  
  
//Busca +, -  
Prntss = 0;  
for (int Cont = Cad.Length - 1; Cont >= 0; Cont--) {  
    if (Cad[Cont] == '(') Prntss++;  
    if (Cad[Cont] == ')') Prntss--;  
    if ((Cad[Cont] == '+' || Cad[Cont] == '-') && Prntss == 0) {  
        string Izquierda = Cad.Substring(0, Cont);  
        string Derecha = Cad.Substring(Cont + 1);  
        Arbol = new Nodo(Cad[Cont]);  
        Arbol.Izquierda = CreaArbol(Izquierda, Arbol.Izquierda);  
        Arbol.Derecha = CreaArbol(Derecha, Arbol.Derecha);  
        return Arbol;  
    }  
}  
  
//Busca *, /  
for (int Cont = Cad.Length - 1; Cont >= 0; Cont--) {
```

```

        if (Cad[Cont] == '(') Prntss++;
        if (Cad[Cont] == ')') Prntss--;
        if ((Cad[Cont] == '*' || Cad[Cont] == '/') && Prntss == 0) {
            string Izquierda = Cad.Substring(0, Cont);
            string Derecha = Cad.Substring(Cont + 1);
            Arbol = new Nodo(Cad[Cont]);
            Arbol.Izquierda = CreaArbol(Izquierda, Arbol.Izquierda);
            Arbol.Derecha = CreaArbol(Derecha, Arbol.Derecha);
            return Arbol;
        }
    }

    //Busca ^
    for (int Cont = Cad.Length - 1; Cont >= 0; Cont--) {
        if (Cad[Cont] == '(') Prntss++;
        if (Cad[Cont] == ')') Prntss--;
        if (Cad[Cont] == '^' && Prntss == 0) {
            string Izquierda = Cad.Substring(0, Cont);
            string Derecha = Cad.Substring(Cont + 1);
            Arbol = new Nodo(Cad[Cont]);
            Arbol.Izquierda = CreaArbol(Izquierda, Arbol.Izquierda);
            Arbol.Derecha = CreaArbol(Derecha, Arbol.Derecha);
            return Arbol;
        }
    }

    //Sólo queda el número o la variable y se le crea el nodo
    if (Cad[0] >= 'a' && Cad[0] <= 'z') {
        int Variable = Cad[0] - 'a';
        Arbol = new Nodo(Variable);
    }
    else {
        double Numero;
        Numero = double.Parse(Cad, CultureInfo.InvariantCulture);
        Arbol = new Nodo(Numero);
    }

    return Arbol;
}

/* Recorrido en Post-Orden para
evaluar el árbol binario */
private double EvaluaArbol(Nodo Arbol) {
    //No hay nodos hijos, entonces
    //es un número o una variable
    if (Arbol.Izquierda == null)
        if (Arbol.Variable != -1)
            return Valores[Arbol.Variable];

```



```

        else
            return Arbol.Numero;

//Recorrido Post-Orden
double ValIzquierda = EvaluaArbol(Arbol.Izquierda);
double ValDerecha = EvaluaArbol(Arbol.Derecha);

/* Código de la función 0:seno, 1:coseno, 2:tangente,
 * 3: valor absoluto, 4: arcoseno, 5: arcocoseno,
 * 6: arcotangente, 7: logaritmo natural, 8: valor tope,
 * 9: exponencial, 10: raíz cuadrada, -1 no es función */
switch (Arbol.Funcion) {
    case 0: return Math.Sin(ValIzquierda);
    case 1: return Math.Cos(ValIzquierda);
    case 2: return Math.Tan(ValIzquierda);
    case 3: return Math.Abs(ValIzquierda);
    case 4: return Math.Asin(ValIzquierda);
    case 5: return Math.Acos(ValIzquierda);
    case 6: return Math.Atan(ValIzquierda);
    case 7: return Math.Log(ValIzquierda);
    case 8: return Math.Exp(ValIzquierda);
    case 9: return Math.Sqrt(ValIzquierda);
}

//Aplica operador
return Arbol.Operador switch {
    '+' => ValIzquierda + ValDerecha,
    '-' => ValIzquierda - ValDerecha,
    '*' => ValIzquierda * ValDerecha,
    '/' => ValIzquierda / ValDerecha,
    '^' => Math.Pow(ValIzquierda, ValDerecha),
    _ => 0,
};
}

/* Chequea la sintaxis de la expresión algebraica */
private int ChequeaSintaxis(string ExpOrig) {
    /* Primero a minúsculas */
    string Minusculas = ExpOrig.ToLower();

    /* Sólo los caracteres válidos */
    string Valido = "abcdefghijklmnopqrstuvwxyz0123456789.+*/^()";
    HashSet<char> Permite = new HashSet<char>(Valido);
    string ConLetrasValidas = "";
    for (int Cont = 0; Cont < Minusculas.Length; Cont++)
        if (Permite.Contains(Minusculas[Cont]))
            ConLetrasValidas += Minusculas[Cont].ToString();
}

```

```

/* Validación de sintaxis, se genera una copia
 * y allí se reemplaza las funciones por a+( */
string sbSintax = ConLetrasValidas;
sbSintax = sbSintax.Replace("sen(", "a+(");
sbSintax = sbSintax.Replace("cos(", "a+(");
sbSintax = sbSintax.Replace("tan(", "a+(");
sbSintax = sbSintax.Replace("abs(", "a+(");
sbSintax = sbSintax.Replace("asn(", "a+(");
sbSintax = sbSintax.Replace("acs(", "a+(");
sbSintax = sbSintax.Replace("atn(", "a+(");
sbSintax = sbSintax.Replace("log(", "a+(");
sbSintax = sbSintax.Replace("exp(", "a+(");
sbSintax = sbSintax.Replace("sqr(", "a+(");

for (int Cont = 0; Cont < sbSintax.Length - 1; Cont++) {
    char cA = sbSintax[Cont];
    char cB = sbSintax[Cont + 1];

    if (Char.IsDigit(cA) && Char.IsLower(cB)) return 1;
    if (Char.IsDigit(cA) && cB == '(') return 2;
    if (cA == '.' && cB == '.') return 3;
    if (cA == '.' && EsUnOperador(cB)) return 4;
    if (cA == '.' && Char.IsLower(cB)) return 5;
    if (cA == '.' && cB == '(') return 6;
    if (cA == '.' && cB == ')') return 7;
    if (EsUnOperador(cA) && cB == '.') return 8;
    if (EsUnOperador(cA) && EsUnOperador(cB)) return 9;
    if (EsUnOperador(cA) && cB == ')') return 10;
    if (Char.IsLower(cA) && Char.IsDigit(cB)) return 11;
    if (Char.IsLower(cA) && cB == '.') return 12;
    if (Char.IsLower(cA) && Char.IsLower(cB)) return 13;
    if (Char.IsLower(cA) && cB == '(') return 14;
    if (cA == '(' && cB == '.') return 15;
    if (cA == '(' && EsUnOperador(cB)) return 16;
    if (cA == '(' && cB == ')') return 17;
    if (cA == ')' && Char.IsDigit(cB)) return 18;
    if (cA == ')' && cB == '.') return 19;
    if (cA == ')' && Char.IsLower(cB)) return 20;
    if (cA == ')' && cB == '(') return 21;
}

/* Valida el inicio y fin de la expresión */
if (EsUnOperador(sbSintax[0])) return 22;
if (EsUnOperador(sbSintax[sbSintax.Length - 1])) return 23;

/* Valida balance de paréntesis */
int ParentesisAbre = 0; /* Contador de paréntesis que abre */
int ParentesisCierra = 0; /* Contador de paréntesis que cierra */

```

```

    for (int Cont = 0; Cont < sbSyntax.Length; Cont++) {
        switch (sbSyntax[Cont]) {
            case '(': ParentesisAbre++; break;
            case ')': ParentesisCierra++; break;
        }
        if (ParentesisCierra > ParentesisAbre) return 24;
    }
    if (ParentesisAbre != ParentesisCierra) return 25;

    /* Validar los puntos decimales de un número real */
    int TotalPuntos = 0;
    for (int Cont = 0; Cont < sbSyntax.Length; Cont++) {
        if (EsUnOperador(sbSyntax[Cont])) TotalPuntos = 0;
        if (sbSyntax[Cont] == '.') TotalPuntos++;
        if (TotalPuntos > 1) return 26;
    }

    return 0;
}

/* Convierte la expresión algebraica, escrita por el usuario,
a un formato que pueda ser interpretado por el evaluador */
private string Convierte(string ExpOrig) {
    /* Primero a minúsculas */
    string Minusculas = ExpOrig.ToLower();

    /* Sólo los caracteres válidos */
    string Valido = "abcdefghijklmnopqrstuvwxyz0123456789.+*/^()";
    HashSet<char> Permite = new HashSet<char>(Valido);
    string ConLetrasValidas = "";
    for (int Cont = 0; Cont < Minusculas.Length; Cont++)
        if (Permite.Contains(Minusculas[Cont]))
            ConLetrasValidas += Minusculas[Cont].ToString();

    /* Cadena a evaluar */
    string Cadena = ConLetrasValidas;
    Cadena = Cadena.Replace("sen", "A");
    Cadena = Cadena.Replace("cos", "B");
    Cadena = Cadena.Replace("tan", "C");
    Cadena = Cadena.Replace("abs", "D");
    Cadena = Cadena.Replace("asn", "E");
    Cadena = Cadena.Replace("acs", "F");
    Cadena = Cadena.Replace("atn", "G");
    Cadena = Cadena.Replace("log", "H");
    Cadena = Cadena.Replace("exp", "I");
    Cadena = Cadena.Replace("sqr", "J");

    return Cadena;
}

```

```

    }

    /* Retorna true si el Caracter es
    * un operador matemático */
    private bool EsUnOperador(char Caracter) {
        return Caracter switch {
            '+' or '-' or '*' or '/' or '^' => true,
            _ => false,
        };
    }
}
}
}

```

Y el programa que lo usa:

H/Árbol/009/Program.cs

```

namespace ArbolBinarioEvaluador {
    //Forma el árbol binario dada una expresión matemática
    //de números, variables, paréntesis, funciones y los operadores:
    //Suma, resta, multiplicación, división y potencia
    //Luego evalúa la expresión
    internal class Program {

        static void Main(string[] args) {
            //Ecuación operadores, variables, funciones, paréntesis
            string Ecuacion = "sen(cos(x)+sen(y*z))/cos(q^3-z)";

            EvalArbolBin obj = new();
            int Sintaxis = obj.Analizar(Ecuacion);

            if (Sintaxis == 0) {

                //En un ciclo se cambian los valores de las
                //variables
                Random Azar = new();
                for (int Cont = 1; Cont <= 30; Cont++) {
                    obj.DarValorVariable('x', Azar.NextDouble());
                    obj.DarValorVariable('y', Azar.NextDouble());
                    obj.DarValorVariable('z', Azar.NextDouble());
                    obj.DarValorVariable('q', Azar.NextDouble());

                    //Evalúa la expresión
                    double Resultado = obj.Evaluar();
                    Console.WriteLine("Resultado es: " + Resultado);
                }
            }
            else {

```

```
        //Hay un error de sintaxis
        Console.WriteLine(obj.MensajeError(Sintaxis));
    }
}
}
```

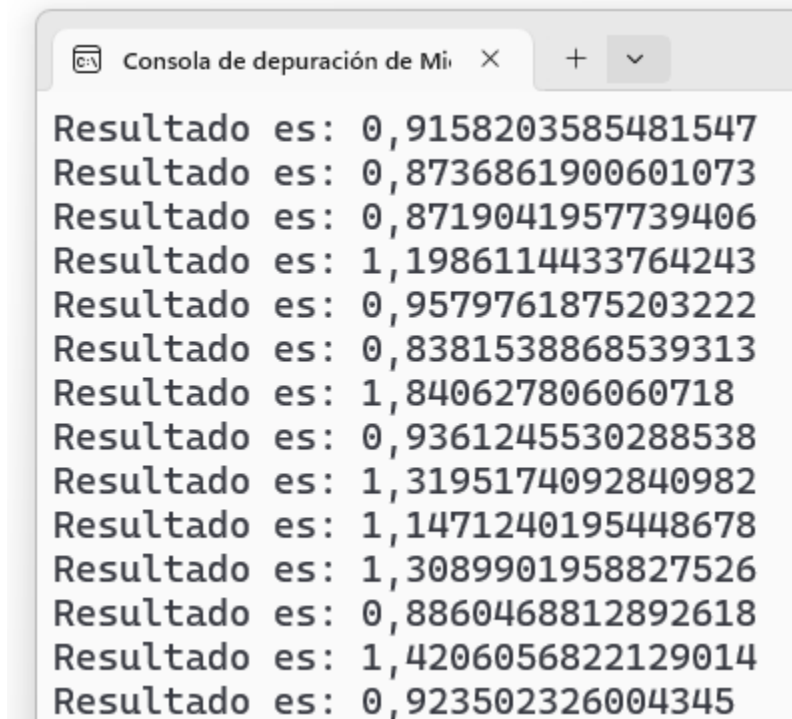


Ilustración 33: Ejemplo de ejecución

Comparativa de desempeño entre evaluadores

¿Cuál de los dos evaluadores es más rápido? Ambos evaluadores tienen dos fases frente a una expresión matemática: el análisis y la evaluación. Si sólo requiere tan sólo un valor de la expresión (raro), entonces hay que fijarse en el tiempo de análisis, pero si de la misma expresión se requiere generar miles de valores al cambiar los valores de las variables (algo muy común), hay que fijarse en el tiempo de la evaluación.

En H/Árbol/010 está la comparativa entre ambos evaluadores. Los pasos que se hacen son:

1. Generar una ecuación al azar con variable X, operadores, paréntesis y funciones.
2. Analizar y evaluar con el Evaluador 4.0
3. Analizar y evaluar con el evaluador de árbol binario.

Se varía el tamaño de la expresión matemática, desde 50 hasta 400. Se generan unas 10000 ecuaciones y cada una se evalúa unas 3000 veces. Para validar que ambos evaluadores están generando valores correctos, se comparan los resultados.

Estos son los programas usados para hacer la comparativa:

H/Árbol/010/EcuacionAzar.cs

```
namespace ArbolBinarioEvaluador {  
  
    //Genera una ecuación al azar para que sea  
    //evaluada  
    internal class EcuacionAzar {  
        Random azar;  
  
        public EcuacionAzar() {  
            azar = new Random();  
        }  
  
        public string Ecuacion(int longitud) {  
            int TamanoEcuacion = 0;  
            int numParentesisAbre = 0;  
  
            string ecuacion = "";  
            while (TamanoEcuacion < longitud) {  
  
                if (azar.NextDouble() > 0.5) {  
                    if (azar.NextDouble() < 0.5) {  
                        ecuacion += FuncionAzar() + "(";  
                    }  
                    else {  
                        ecuacion += "(";  
                    }  
                    numParentesisAbre++;  
                    TamanoEcuacion++;  
                }  
            }  
        }  
    }  
}
```

```

        //Variable o número
        if (azar.NextDouble() < 0.5)
            ecuacion += NumeroAzar();
        else
            ecuacion += "x";
        TamanoEcuacion++;

        //Paréntesis que cierra
        int numParentesisCierra = azar.Next(numParentesisAbre + 1);
        for (int num = 1; num <= numParentesisCierra; num++) {
            ecuacion += ")";
            numParentesisAbre--;
            TamanoEcuacion++;
        }

        //Operador
        TamanoEcuacion++;
        ecuacion += OperadorAzar();
    }

    //Variable o número
    if (azar.NextDouble() < 0.5)
        ecuacion += NumeroAzar();
    else
        ecuacion += "x";

    //Cierra los paréntesis
    for (int num = 0; num < numParentesisAbre; num++) ecuacion += ")";

    return ecuacion;
}

private string FuncionAzar() {
    string[] funciones = { "sen", "cos", "tan", "abs", "asn", "acs",
"atn", "log", "exp", "sqr" };
    return funciones[azar.Next(funciones.Length)];
}

private string OperadorAzar() {
    string[] operadores = { "+", "-", "*", "/" };
    return operadores[azar.Next(operadores.Length)];
}

private string NumeroAzar() {
    string undecimal = Convert.ToString(azar.Next(100) + 1);
    return "1." + undecimal;
}
}

```



```
}
```

```
using System.Diagnostics;

//Comparar los dos evaluadores
namespace ArbolBinarioEvaluador {
    internal class Program {

        static void Main(string[] args) {
            long TAnalisisEval4 = 0, TEvaluaEval4 = 0;
            long TAnalisisArbol = 0, TEvaluaArbol = 0;

            EvalArbolBin arbol = new();
            Evaluador4 eval4 = new();
            EcuacionAzar ecu = new();

            //Toma el tiempo
            Stopwatch temporizador = new();

            //Parámetros
            int TotalEcuaciones = 10000;
            int VecesEvalua = 3000;
            int Tamano = 400;

            //Valores de X al azar
            double[] valX = new double[VecesEvalua];
            Random azar = new();

            for (int cont = 1; cont <= TotalEcuaciones; cont++) {
                //Ecuación al azar
                string Ecuacion = ecu.Ecuacion(Tamano);

                //Valores de X al azar
                for (int xvalor = 0; xvalor < valX.Length; xvalor++) {
                    valX[xvalor] = azar.NextDouble() - azar.NextDouble();
                }

                //Analiza la expresión con el árbol binario
                temporizador.Reset();
                temporizador.Start();
                arbol.Analizar(Ecuacion);
                temporizador.Stop();
                TAnalisisArbol += temporizador.ElapsedTicks;

                //Analiza la expresión con el evaluador 4.0
```

```

    temporizador.Reset();
    temporizador.Start();
    eval4.Analizar(Ecuacion);
    temporizador.Stop();
    TAnalisisEval4 += temporizador.ElapsedTicks;

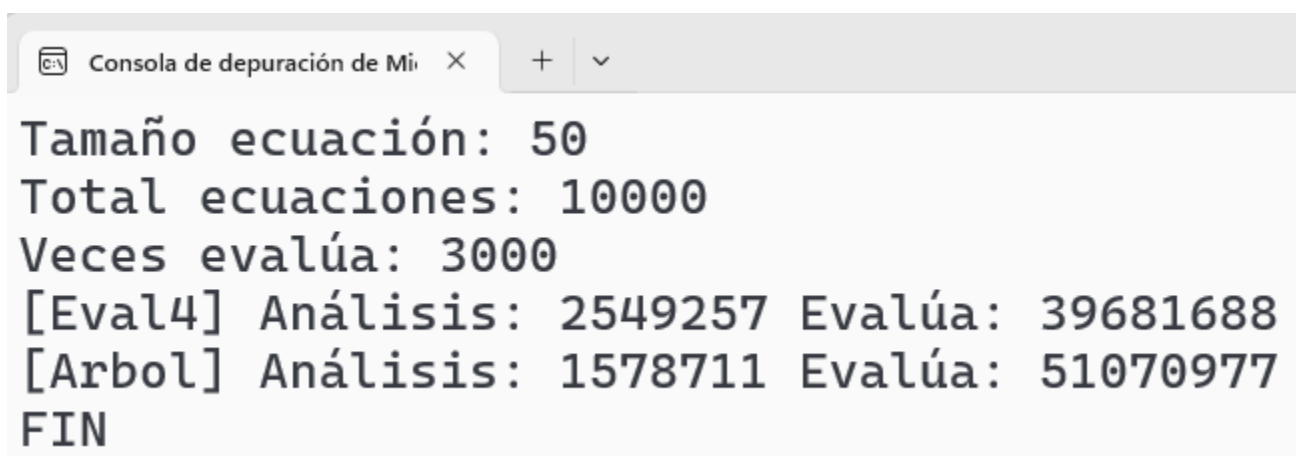
    //Evalúa la expresión con el árbol binario
    temporizador.Reset();
    temporizador.Start();
    double ResArbol = 0;
    for (int xvalor = 0; xvalor < valX.Length; xvalor++) {
        arbol.DarValorVariable('x', valX[xvalor]);
        ResArbol += arbol.Evaluar();
    }
    temporizador.Stop();
    TEvaluaArbol += temporizador.ElapsedTicks;

    //Evalúa la expresión con el evaluador 4.0
    temporizador.Reset();
    temporizador.Start();
    double ResEval4 = 0;
    for (int xvalor = 0; xvalor < valX.Length; xvalor++) {
        eval4.DarValorVariable('x', valX[xvalor]);
        ResEval4 += eval4.Evaluar();
    }
    temporizador.Stop();
    TEvaluaEval4 += temporizador.ElapsedTicks;

    //Chequea si hay una diferencia entre ambos evaluadores
    if (Math.Abs(ResArbol - ResEval4) > 0.01) {
        Console.WriteLine(Ecuacion);
        Console.WriteLine("Arbol: " + ResArbol);
        Console.WriteLine("Eval4: " + ResEval4);
        break;
    }
}

Console.WriteLine("Tamaño ecuación: " + Tamano);
Console.WriteLine("Total ecuaciones: " + TotalEcuaciones);
Console.WriteLine("Veces evalúa: " + VecesEvalua);
Console.WriteLine("[Eval4] Análisis: " + TAnalisisEval4 + "
Evalúa: " + TEvaluaEval4);
Console.WriteLine("[Arbol] Análisis: " + TAnalisisArbol + "
Evalúa: " + TEvaluaArbol);
Console.WriteLine("FIN");
}
}
}

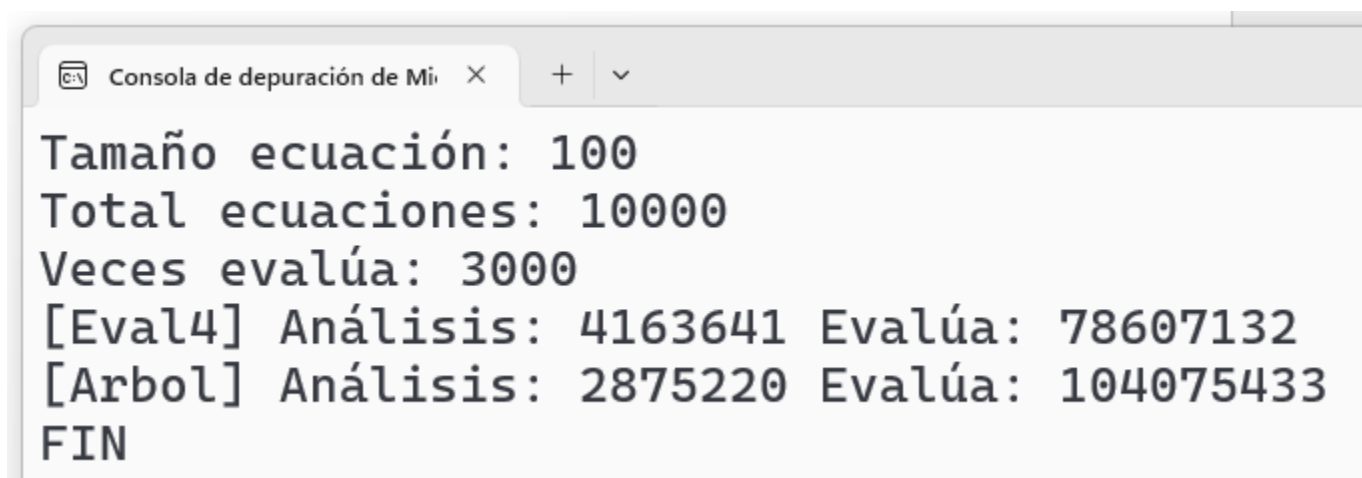
```



Consola de depuración de Mi X + v

```
Tamaño ecuación: 50
Total ecuaciones: 10000
Veces evalúa: 3000
[Eval4] Análisis: 2549257 Evalúa: 39681688
[Arbol] Análisis: 1578711 Evalúa: 51070977
FIN
```

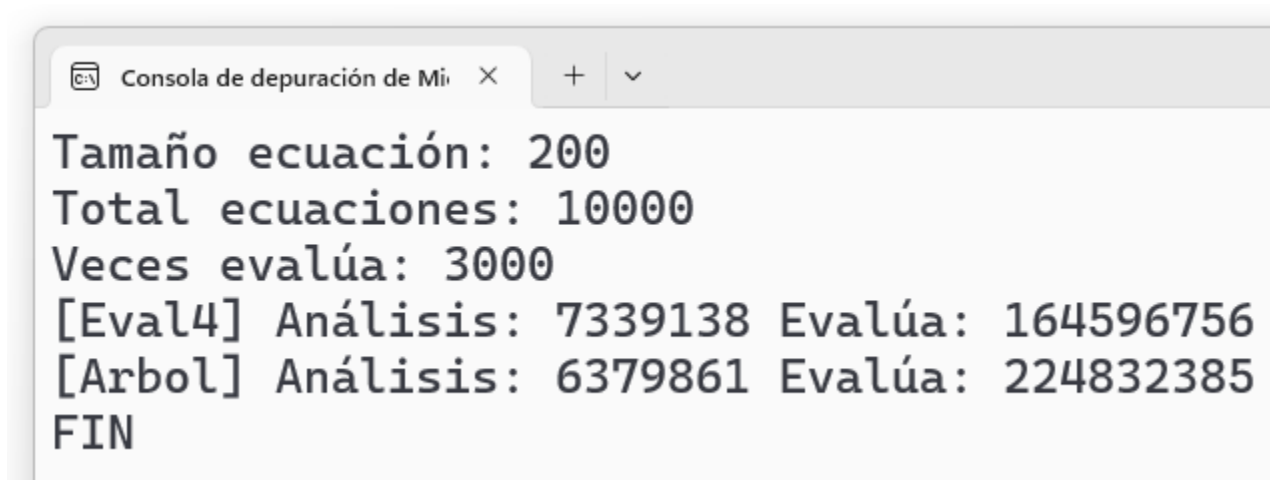
Ilustración 34: Comparativa de desempeño



Consola de depuración de Mi X + v

```
Tamaño ecuación: 100
Total ecuaciones: 10000
Veces evalúa: 3000
[Eval4] Análisis: 4163641 Evalúa: 78607132
[Arbol] Análisis: 2875220 Evalúa: 104075433
FIN
```

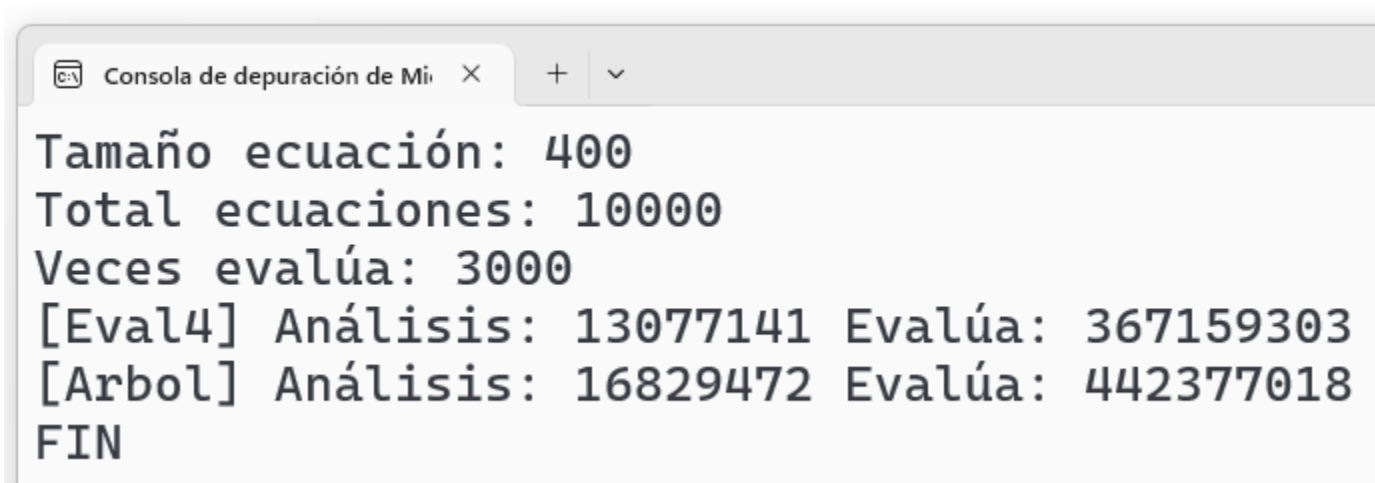
Ilustración 35: Comparativa de desempeño



Consola de depuración de Mi X + v

```
Tamaño ecuación: 200
Total ecuaciones: 10000
Veces evalúa: 3000
[Eval4] Análisis: 7339138 Evalúa: 164596756
[Arbol] Análisis: 6379861 Evalúa: 224832385
FIN
```

Ilustración 36: Comparativa de desempeño



The image shows a screenshot of a debug console window titled 'Consola de depuración de Mi'. The console displays the following text:

```
Tamaño ecuación: 400
Total ecuaciones: 10000
Veces evalúa: 3000
[Eval4] Análisis: 13077141 Evalúa: 367159303
[Arbol] Análisis: 16829472 Evalúa: 442377018
FIN
```

Ilustración 37: Comparativa de desempeño

El evaluador más rápido en la fase de evaluación es el Evaluador 4.0 en todas las pruebas. El evaluador que usa el árbol binario sólo es más rápido en análisis cuando la expresión es corta.