C# Y .NET 10
Parte 8. Evaluador de expresiones algebraicas

2025-12

Rafael Alberto Moreno Parra
ramsoftware@gmail.com

Contenido

Tabla de ilustraciones

Acerca del autor

Rafael Alberto Moreno Parra

ramsoftware@gmail.com o enginelife@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 2026 ® Enlace: https://visualstudio.microsoft.com/es/vs/

Dedicatoria

En memoria de Tammy

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 4.1

Autor: Rafael Alberto Moreno Parra

El algoritmo

Esta es la versión 4.1 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:

```
public class Sintaxis4_1 {
  /* HashSet estático para caracteres válidos (se crea una sola vez) */
  private static readonly HashSet<char> CaracteresValidos =
    [.. "abcdefghijklmnopqrstuvwxyz0123456789.+-*/^()"];
```

```
  /* Paso 1: Convertir la expresión para poder analizarla */
  private void Convierte(string ExpresionOriginal) {
    /* Primero a minúsculas */
    string Minusculas = ExpresionOriginal.ToLower();

    /* Sólo los caracteres válidos (optimizado con HashSet estático) */
    StringBuilder ConLetrasValidas = new("(");
    foreach (char c in Minusculas) {
      if (CaracteresValidos.Contains(c))
        ConLetrasValidas.Append(c);
    }
    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: Convertir la expresión

Esta es la novedad de la versión 4.1 (la anterior fue la 4.0). Esto permite expresiones así:

1. Convierte de (- a (0- para dar apoyo al menos unario. Por ejemplo: 4+(-3*2) se convierte a 4+(0-3*2)
2. Convierte de)(a)*(. Por ejemplo: (7+g)(h/u) se convierte a (7+g)*(h/u)
3. Convierte de punto operador a punto cero operador. Ejemplo: 3.+1 se convierte a 3.0+1
4. Convierte de operador punto a operador cero punto. Ejemplo: 7-.9 se convierte a 7-0.9
5. Convierte de .(a .0*(. Ejemplo: 2.(8+1) se convierte a 2.0*(8+1)
6. Convierte de). a)*0. Ejemplo: (9 / 2) .5 se convierte a (9/2)*0.5
7. Convierte de .) a .0)
8. Convierte de (. a (0.
9. Convierte número letra en número * letra
10. Convierte número(en número*(
11. Convierte)numero en)*numero
12. Convierte de)letra en)*letra
13. Convierte de letra minúscula(a letra minúscula*(

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

Paso 3: Verificando la sintaxis de la expresión algebraica

Luego se verifica si la expresión cumple con las estrictas reglas sintácticas del algebra. Se hacen 14 validaciones que son:

1. Doble punto seguido. Ejemplo: 3..1
2. Un punto y sigue una letra. Ejemplo: 3+5.w-8
3. Dos operadores estén seguidos. Ejemplo: 2++4, 5-*3
4. Un operador seguido de un paréntesis que cierra. Ejemplo: 2-(4+)-7
5. Una letra seguida de número. Ejemplo: 7-a8-6
6. Una letra seguida de punto. Ejemplo: 7-a.-6
7. Una letra seguida de otra letra. Ejemplo: 5+zx*3
8. Un paréntesis que abre seguido de un operador. Ejemplo: 2-(*3)
9. Un paréntesis que abre seguido de un paréntesis que cierra. Ejemplo: 7-()-6
10. Inicia con operador. Ejemplo: +3*5
11. Finaliza con operador. Ejemplo: 3*5*
12. No hay correspondencia entre paréntesis. Ejemplo: 3+5)-(7*2
13. Los paréntesis están desbalanceados. Ejemplo: 3-(2*4))
14. Doble punto en un número de tipo real. Ejemplo: 7-6.46.1+2

Para hacer esto, se hace uso de una clase dedicada a la conversión y a la revisión de sintaxis.

H/Eval4_1/Constantes4_1.cs

```
public static class Constantes4_1 {
  /* Constantes de los diferentes tipos
   * de datos que tendrán las partes y piezas */
  public const int ESFUNCION = 1;
  public const int ESPARABRE = 2;
  public const int ESPARCIERRA = 3;
  public const int ESOPERADOR = 4;
  public const int ESNUMERO = 5;
  public const int ESVARIABLE = 6;
  public const int ESACUMULA = 7;

  public const int POTENCIA = 4;
  public const int DIVIDE = 3;
  public const int MULTIPLICA = 2;
  public const int RESTA = 1;
  public const int SUMA = 0;

  public const string SENO = "A";
  public const string COSENO = "B";
  public const string TANGENTE = "C";
  public const string VALORABSOLUTO = "D";
  public const string ARCOSENO = "E";
  public const string ARCOCOSENO = "F";
  public const string ARCOTANGENTE = "G";
  public const string LOGARITMONATURAL = "H";
  public const string EXPONENCIAL = "I";
  public const string RAIZCUADRADA = "J";

  public const int NO_FUNCION_COD = -1;
  public const int SENO_COD = 0;
  public const int COSENO_COD = 1;
  public const int TANGENTE_COD = 2;
  public const int VALORABSOLUTO_COD = 3;
  public const int ARCOSENO_COD = 4;
  public const int ARCOCOSENO_COD = 5;
  public const int ARCOTANGENTE_COD = 6;
  public const int LOGARITMONATURAL_COD = 7;
  public const int EXPONENCIAL_COD = 8;
  public const int RAIZCUADRADA_COD = 9;
}
```

```
  public class Sintaxis4_1 {
    /* HashSet estático para caracteres válidos (se crea una sola vez) */
    private static readonly HashSet<char> CaracteresValidos =
      [.. "abcdefghijklmnopqrstuvwxyz0123456789.+-*/^()"];

    /* Diccionario para reemplazos de funciones en expresión analizada */
    private static readonly Dictionary<string, string> ReemplazosFunciones
= new()
    {
      {"sen", Constantes4_1.SENO}, {"cos", Constantes4_1.COSENO}, {"tan",
Constantes4_1.TANGENTE},
      {"abs", Constantes4_1.VALORABSOLUTO}, {"asn", Constantes4_1.ARCOSENO},
{"acs", Constantes4_1.ARCOCOSENO},
      {"atn", Constantes4_1.ARCOTANGENTE}, {"log",
Constantes4_1.LOGARITMONATURAL}, {"exp", Constantes4_1.EXPONENCIAL},
      {"sqr", Constantes4_1.RAIZCUADRADA}
    };

    /* Diccionario estático para mensajes de error */
    private static readonly Dictionary<int, string> MensajesError = new()
    {
      {0, "0. Es correcta"},
      {1, "1. Doble punto seguido"},
      {2, "2. Punto y sigue una letra"},
      {3, "3. Dos operadores estén seguidos"},
      {4, "4. Operador seguido de paréntesis que cierra"},
      {5, "5. Letra seguida de número"},
      {6, "6. Letra seguida de punto"},
      {7, "7. Letra seguida de otra letra"},
      {8, "8. Paréntesis que abre y sigue operador"},
      {9, "9. Paréntesis que abre y luego cierra"},
      {10, "10. Inicia con operador"},
      {11, "11. Finaliza con operador"},
      {12, "12. No hay correspondencia entre paréntesis"},
      {13, "13. Paréntesis desbalanceados"},
      {14, "14. Dos o más puntos en número real"}
    };

    /* HashSet estático para operadores (más rápido que switch) */
    private static readonly HashSet<char> Operadores = new() { '+', '-',
'*', '/', '^' };

    /* Arreglo de mensajes de error */
    private int[] TipoError = new int[15];
```

```
  /* Expresión convertida y validada */
  public string ExpresionConvertida { get; private set; } = string.Empty;

  /* Convierte y chequea la expresión */
  public bool ChequeaSintaxis(string ExpresionOriginal) {
    /* Paso 1: Convierte la expresión */
    Convierte(ExpresionOriginal);

    /* Inicializa el arreglo de errores */
    Array.Fill(TipoError, 0);

    /* Paso 2: Chequea sintaxis */
    ValidaCaracteresSeguidos(ExpresionConvertida);
    ValidaInicioYFinDeLaExpresion(ExpresionConvertida);
    ValidaParentesisDesbalanceados(ExpresionConvertida);
    ValidaPuntosEnNumeroReal(ExpresionConvertida);

    /* Si una sola celda es 1, hay error */
    for (int Cont = 0; Cont < TipoError.Length; Cont++) {
      if (TipoError[Cont] == 1) {
        return false;
      }
    }
    return true;
  }

  public List<string> ObtenerMensajesErrores() {
    List<string> mensajes = new();
    for (int i = 1; i < TipoError.Length; i++) {
      if (TipoError[i] == 1) {
        mensajes.Add(MensajeError(i));
      }
    }
    return mensajes;
  }

  /* Paso 1: Convertir la expresión para poder analizarla */
  private void Convierte(string ExpresionOriginal) {
    /* Primero a minúsculas */
    string Minusculas = ExpresionOriginal.ToLower();

    /* Sólo los caracteres válidos (optimizado con HashSet estático) */
    StringBuilder ConLetrasValidas = new("(");
    foreach (char c in Minusculas) {
      if (CaracteresValidos.Contains(c))
        ConLetrasValidas.Append(c);
    }
    ConLetrasValidas.Append(')');
```

```
    /* Reemplazos de funciones usando diccionario */
    foreach (var reemplazo in ReemplazosFunciones) {
      ConLetrasValidas.Replace(reemplazo.Key, reemplazo.Value);
    }

    /* Agrega +0) donde exista )) porque es
     * necesario para crear las piezas */
    ConLetrasValidas.Replace("))", ")+0)");

    /* Convierte de (- a (0- para dar apoyo al menos unario */
    ConLetrasValidas.Replace("(-", "(0-");

    /* Convierte de )( a )*( */
    ConLetrasValidas.Replace")(", ")*(");

    /* Convierte de punto operador a punto cero operador */
    ConLetrasValidas.Replace(".+", ".0+");
    ConLetrasValidas.Replace(".-", ".0-");
    ConLetrasValidas.Replace(".*", ".0*");
    ConLetrasValidas.Replace("./", ".0/");
    ConLetrasValidas.Replace(".^", ".0^");

    /* Convierte de operador punto a operador cero punto */
    ConLetrasValidas.Replace("+.", "+0.");
    ConLetrasValidas.Replace("-.", "-0.");
    ConLetrasValidas.Replace("*.", "*0.");
    ConLetrasValidas.Replace("/.", "/0.");
    ConLetrasValidas.Replace("^.", "^0.");

    /* Convierte de .( a .0*( */
    ConLetrasValidas.Replace(".(", ".0*(");

    /* Convierte de ). a )*0. */
    ConLetrasValidas.Replace(").", ")*0.");

    /* Convierte de .) a .0) */
    ConLetrasValidas.Replace(".)", ".0)");

    /* Convierte de (. a (0. */
    ConLetrasValidas.Replace("(.", "(0.");

    /* Convierte a cadena para hacer uso del regex */
    ExpresionConvertida = ConLetrasValidas.ToString();

    /* Convierte número letra en número * letra */
    ExpresionConvertida = Regex.Replace(ExpresionConvertida, @"(\d)([a-
z])", "$1*$2");
```

```
      /* Convierte número( en número*( */
      ExpresionConvertida = Regex.Replace(ExpresionConvertida, @"(\d)\(",
"$1*(");

      /* Convierte )numero en )*numero */
      ExpresionConvertida = Regex.Replace(ExpresionConvertida, @"\)(\d+)",
@")*$1");

      /* Convierte de )letra en )*letra */
      ExpresionConvertida = Regex.Replace(ExpresionConvertida, @"\)([a-zA-
Z])", @")*$1");

      /* Convierte de letra minúscula( a letra minúscula*( */
      ExpresionConvertida = Regex.Replace(ExpresionConvertida, @"([a-z])\(",
@"$1*(");
    }

    /* Retorna mensaje de error sintáctico */
    private string MensajeError(int CodigoError) =>
      MensajesError.TryGetValue(CodigoError, out string? mensaje)
        ? mensaje
        : "Error desconocido";


    /* Retorna si el Caracter es un operador matemático */
    private static bool EsUnOperador(char Caracter) =>
Operadores.Contains(Caracter);

    /* Valida casos de dos caracteres seguidos */
    private void ValidaCaracteresSeguidos(string Expresion) {
      for (int Cont = 0; Cont < Expresion.Length - 1; Cont++) {
        char carA = Expresion[Cont];
        char carB = Expresion[Cont + 1];

        if (carA == '.' && carB == '.') TipoError[1] = 1;
        if (carA == '.' && char.IsLower(carB)) TipoError[2] = 1;
        if (EsUnOperador(carA) && EsUnOperador(carB)) TipoError[3] = 1;
        if (EsUnOperador(carA) && carB == ')') TipoError[4] = 1;
        if (char.IsLower(carA) && char.IsDigit(carB)) TipoError[5] = 1;
        if (char.IsLower(carA) && carB == '.') TipoError[6] = 1;
        if (char.IsLower(carA) && char.IsLower(carB)) TipoError[7] = 1;
        if (carA == '(' && EsUnOperador(carB)) TipoError[8] = 1;
        if (carA == '(' && carB == ')') TipoError[9] = 1;
      }
    }

    /* Valida el inicio y fin de la expresión */
```

```
    private void ValidaInicioYFinDeLaExpresion(string Expresion) {
      if (EsUnOperador(Expresion[1])) {
        TipoError[10] = 1;
      }
      if (EsUnOperador(Expresion[Expresion.Length - 2])) {
        TipoError[11] = 1;
      }
    }

    /* Valida si hay paréntesis desbalanceados */
    private void ValidaParentesisDesbalanceados(string Expresion) {
      int Balance = 0;
      for (int Cont = 0; Cont < Expresion.Length; Cont++) {
        if (Expresion[Cont] == '(') Balance++;
        if (Expresion[Cont] == ')') Balance--;
        if (Balance < 0) {
          TipoError[12] = 1;
          return;
        }
      }
      if (Balance != 0) {
        TipoError[13] = 1;
      }
    }

    /* Validar los puntos decimales de un número real */
    private void ValidaPuntosEnNumeroReal(string Expresion) {
      int TotalPuntos = 0;
      for (int Cont = 0; Cont < Expresion.Length; Cont++) {
        if (EsUnOperador(Expresion[Cont])) TotalPuntos = 0;
        if (Expresion[Cont] == '.') TotalPuntos++;
        if (TotalPuntos > 1) {
          TipoError[14] = 1;
          return;
        }
      }
    }
  }
}
```

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

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 4.1), sea más rápida que la versión anterior (3.0) 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 4.1, las constantes, piezas y variables guardan
      sus valores en la lista Valores.
      En partes, se almacena la posición en Valores */
   private List<ParteEvl4_1> Partes = new List<ParteEvl4_1>();
```

Y la clase de las partes es así:

H/Eval4_1/ParteEvl4_1.cs

```
 /* 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 4.1, las constantes, piezas y variables guardan
    sus valores en la lista Valores.
    En partes, se almacena la posición en Valores */
 public class ParteEvl4_1 {
   /* 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;

  // Diccionario estático para mapeo de operadores
  private static readonly Dictionary<char, int> OperadorMap = new()
  {
    {'+', 0}, {'-', 1}, {'*', 2}, {'/', 3}, {'^', 4}
  };

  public ParteEvl4_1(int TipoParte, int Valor) {
    Tipo = TipoParte;
    if (TipoParte == Constantes4_1.ESFUNCION) Funcion = Valor;
    else if (TipoParte == Constantes4_1.ESNUMERO) posNumero = Valor;
    else if (TipoParte == Constantes4_1.ESVARIABLE) posVariable = Valor;
    else if (TipoParte == Constantes4_1.ESPARABRE) Funcion = -1;
  }

  public ParteEvl4_1(char Operador) {
    Tipo = Constantes4_1.ESOPERADOR;
    this.Operador = OperadorMap[Operador];
  }
}
```

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:

H/Eval4_1/PiezaEvl4_1.cs

```
  public class PiezaEvl4_1 {
    /* 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;
  }
}
```

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_1 tmp;

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

        // Cacheo de operandos para menos indirections
        double a = Valores[tmp.pA];
        double b = Valores[tmp.pB];

        // Operación binaria (mantiene tu lógica original)
        Resulta = tmp.Operador switch {
          Constantes4_1.SUMA => a + b,
          Constantes4_1.RESTA => a - b,
          Constantes4_1.MULTIPLICA => a * b,
          Constantes4_1.DIVIDE => a / b,
          _ => Math.Pow(a, b),
        };

        switch (tmp.Funcion) {
          case Constantes4_1.SENO_COD: Resulta = Math.Sin(Resulta); break;
          case Constantes4_1.COSENO_COD: Resulta = Math.Cos(Resulta); break;
          case Constantes4_1.TANGENTE_COD: Resulta = Math.Tan(Resulta);
break;
          case Constantes4_1.VALORABSOLUTO_COD: Resulta = Math.Abs(Resulta);
break;
          case Constantes4_1.ARCOSENO_COD: Resulta = Math.Asin(Resulta);
break;
          case Constantes4_1.ARCOCOSENO_COD: Resulta = Math.Acos(Resulta);
break;
```

```
        case Constantes4_1.ARCOTANGENTE_COD: Resulta = Math.Atan(Resulta);
break;
        case Constantes4_1.LOGARITMONATURAL_COD: Resulta =
Math.Log(Resulta); break;
        case Constantes4_1.EXPONENCIAL_COD: Resulta = Math.Exp(Resulta);
break;
        case Constantes4_1.RAIZCUADRADA_COD: Resulta = Math.Sqrt(Resulta);
break;
        default:
          break;
       }

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

El código completo

Este es el código completo del evaluador de expresiones. La versión 4.1 refactorizó todo el código para hacerlo más comprensible. Luego tiene varias clases, cada clase en su propio archivo.

H/Eval4_1/Constantes4_1.cs

```
/* Evaluador de expresiones versión 4.1 (diciembre de 2025)
 * Autor: Rafael Alberto Moreno Parra
 * Correo: ramsoftware@gmail.com ;  enginelife@hotmail.com
 * URL: http://darwin.50webs.com
 * GitHub: https://github.com/ramsoftware
 * */

namespace Ejemplo {
  public static class Constantes4_1 {
    /* Constantes de los diferentes tipos
     * de datos que tendrán las partes y piezas */
    public const int ESFUNCION = 1;
    public const int ESPARABRE = 2;
    public const int ESPARCIERRA = 3;
    public const int ESOPERADOR = 4;
    public const int ESNUMERO = 5;
    public const int ESVARIABLE = 6;
    public const int ESACUMULA = 7;

    public const int POTENCIA = 4;
    public const int DIVIDE = 3;
    public const int MULTIPLICA = 2;
    public const int RESTA = 1;
    public const int SUMA = 0;

    public const string SENO = "A";
    public const string COSENO = "B";
    public const string TANGENTE = "C";
    public const string VALORABSOLUTO = "D";
    public const string ARCOSENO = "E";
    public const string ARCOCOSENO = "F";
    public const string ARCOTANGENTE = "G";
    public const string LOGARITMONATURAL = "H";
    public const string EXPONENCIAL = "I";
    public const string RAIZCUADRADA = "J";

    public const int NO_FUNCION_COD = -1;
    public const int SENO_COD = 0;
    public const int COSENO_COD = 1;
    public const int TANGENTE_COD = 2;
```

```
    public const int VALORABSOLUTO_COD = 3;
    public const int ARCOSENO_COD = 4;
    public const int ARCOCOSENO_COD = 5;
    public const int ARCOTANGENTE_COD = 6;
    public const int LOGARITMONATURAL_COD = 7;
    public const int EXPONENCIAL_COD = 8;
    public const int RAIZCUADRADA_COD = 9;
  }
}
```

H/Eval4_1/ParteEvl4_1.cs

```
/* Evaluador de expresiones versión 4.1 (diciembre de 2025)
 * Autor: Rafael Alberto Moreno Parra
 * Correo: ramsoftware@gmail.com ;  enginelife@hotmail.com
 * URL: http://darwin.50webs.com
 * GitHub: https://github.com/ramsoftware
 * */

namespace Ejemplo {
  /* 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 4.1, las constantes, piezas y variables guardan
     sus valores en la lista Valores.
     En partes, se almacena la posición en Valores */
  public class ParteEvl4_1 {
    /* 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;

    // Diccionario estático para mapeo de operadores
    private static readonly Dictionary<char, int> OperadorMap = new()
    {
      {'+', 0}, {'-', 1}, {'*', 2}, {'/', 3}, {'^', 4}
    };

    public ParteEvl4_1(int TipoParte, int Valor) {
      Tipo = TipoParte;
      if (TipoParte == Constantes4_1.ESFUNCION) Funcion = Valor;
      else if (TipoParte == Constantes4_1.ESNUMERO) posNumero = Valor;
      else if (TipoParte == Constantes4_1.ESVARIABLE) posVariable = Valor;
      else if (TipoParte == Constantes4_1.ESPARABRE) Funcion = -1;
    }

    public ParteEvl4_1(char Operador) {
      Tipo = Constantes4_1.ESOPERADOR;
      this.Operador = OperadorMap[Operador];
    }
  }
}
```

H/Eval4_1/PiezaEvl4_1.cs

```
/* Evaluador de expresiones versión 4.1 (diciembre de 2025)
 * Autor: Rafael Alberto Moreno Parra
 * Correo: ramsoftware@gmail.com ;  enginelife@hotmail.com
 * URL: http://darwin.50webs.com
 * GitHub: https://github.com/ramsoftware
 * */

namespace Ejemplo {
  public class PiezaEvl4_1 {
```

```
    /* 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;
  }
}
```

H/Eval4_1/Sintaxis4_1.cs

```
using System.Text;
using System.Text.RegularExpressions;

/* Evaluador de expresiones versión 4.1 (diciembre de 2025)
 * Autor: Rafael Alberto Moreno Parra
 * Correo: ramsoftware@gmail.com ;  enginelife@hotmail.com
 * URL: http://darwin.50webs.com
 * GitHub: https://github.com/ramsoftware
 * */

namespace Ejemplo {
  public class Sintaxis4_1 {
    /* HashSet estático para caracteres válidos (se crea una sola vez) */
    private static readonly HashSet<char> CaracteresValidos =
      [.. "abcdefghijklmnopqrstuvwxyz0123456789.+-*/^()"];

    /* Diccionario para reemplazos de funciones en expresión analizada */
    private static readonly Dictionary<string, string> ReemplazosFunciones
= new()
    {
      {"sen", Constantes4_1.SENO}, {"cos", Constantes4_1.COSENO}, {"tan",
Constantes4_1.TANGENTE},
      {"abs", Constantes4_1.VALORABSOLUTO}, {"asn", Constantes4_1.ARCOSENO},
{"acs", Constantes4_1.ARCOCOSENO},
```

```
    {"atn", Constantes4_1.ARCOTANGENTE}, {"log",
Constantes4_1.LOGARITMONATURAL}, {"exp", Constantes4_1.EXPONENCIAL},
    {"sqr", Constantes4_1.RAIZCUADRADA}
  };

  /* Diccionario estático para mensajes de error */
  private static readonly Dictionary<int, string> MensajesError = new()
  {
    {0, "0. Es correcta"},
    {1, "1. Doble punto seguido"},
    {2, "2. Punto y sigue una letra"},
    {3, "3. Dos operadores estén seguidos"},
    {4, "4. Operador seguido de paréntesis que cierra"},
    {5, "5. Letra seguida de número"},
    {6, "6. Letra seguida de punto"},
    {7, "7. Letra seguida de otra letra"},
    {8, "8. Paréntesis que abre y sigue operador"},
    {9, "9. Paréntesis que abre y luego cierra"},
    {10, "10. Inicia con operador"},
    {11, "11. Finaliza con operador"},
    {12, "12. No hay correspondencia entre paréntesis"},
    {13, "13. Paréntesis desbalanceados"},
    {14, "14. Dos o más puntos en número real"}
  };

  /* HashSet estático para operadores (más rápido que switch) */
  private static readonly HashSet<char> Operadores = new() { '+', '-',
'*', '/', '^' };

  /* Arreglo de mensajes de error */
  private int[] TipoError = new int[15];

  /* Expresión convertida y validada */
  public string ExpresionConvertida { get; private set; } = string.Empty;

  /* Convierte y chequea la expresión */
  public bool ChequeaSintaxis(string ExpresionOriginal) {
    /* Paso 1: Convierte la expresión */
    Convierte(ExpresionOriginal);

    /* Inicializa el arreglo de errores */
    Array.Fill(TipoError, 0);

    /* Paso 2: Chequea sintaxis */
    ValidaCaracteresSeguidos(ExpresionConvertida);
    ValidaInicioYFinDeLaExpresion(ExpresionConvertida);
    ValidaParentesisDesbalanceados(ExpresionConvertida);
    ValidaPuntosEnNumeroReal(ExpresionConvertida);
```

```
    /* Si una sola celda es 1, hay error */
    for (int Cont = 0; Cont < TipoError.Length; Cont++) {
      if (TipoError[Cont] == 1) {
        return false;
      }
    }
    return true;
  }

  public List<string> ObtenerMensajesErrores() {
    List<string> mensajes = new();
    for (int i = 1; i < TipoError.Length; i++) {
      if (TipoError[i] == 1) {
        mensajes.Add(MensajeError(i));
      }
    }
    return mensajes;
  }

  /* Paso 1: Convertir la expresión para poder analizarla */
  private void Convierte(string ExpresionOriginal) {
    /* Primero a minúsculas */
    string Minusculas = ExpresionOriginal.ToLower();

    /* Sólo los caracteres válidos (optimizado con HashSet estático) */
    StringBuilder ConLetrasValidas = new("(");
    foreach (char c in Minusculas) {
      if (CaracteresValidos.Contains(c))
        ConLetrasValidas.Append(c);
    }
    ConLetrasValidas.Append(')');

    /* Reemplazos de funciones usando diccionario */
    foreach (var reemplazo in ReemplazosFunciones) {
      ConLetrasValidas.Replace(reemplazo.Key, reemplazo.Value);
    }

    /* Agrega +0) donde exista )) porque es
     * necesario para crear las piezas */
    ConLetrasValidas.Replace("))", ")+0)");

    /* Convierte de (- a (0- para dar apoyo al menos unario */
    ConLetrasValidas.Replace("(-", "(0-");

    /* Convierte de )( a )*( */
    ConLetrasValidas.Replace")(", ")*(");
```

```
    /* Convierte de punto operador a punto cero operador */
    ConLetrasValidas.Replace(".+", ".0+");
    ConLetrasValidas.Replace(".-", ".0-");
    ConLetrasValidas.Replace(".*", ".0*");
    ConLetrasValidas.Replace("./", ".0/");
    ConLetrasValidas.Replace(".^", ".0^");

    /* Convierte de operador punto a operador cero punto */
    ConLetrasValidas.Replace("+.", "+0.");
    ConLetrasValidas.Replace("-.", "-0.");
    ConLetrasValidas.Replace("*.", "*0.");
    ConLetrasValidas.Replace("/.", "/0.");
    ConLetrasValidas.Replace("^.", "^0.");

    /* Convierte de .( a .0*( */
    ConLetrasValidas.Replace(".(", ".0*(");

    /* Convierte de ). a )*0. */
    ConLetrasValidas.Replace(").", ")*0.");

    /* Convierte de .) a .0) */
    ConLetrasValidas.Replace(".)", ".0)");

    /* Convierte de (. a (0. */
    ConLetrasValidas.Replace("(.", "(0.");

    /* Convierte a cadena para hacer uso del regex */
    ExpresionConvertida = ConLetrasValidas.ToString();

    /* Convierte número letra en número * letra */
    ExpresionConvertida = Regex.Replace(ExpresionConvertida, @"(\d)([a-
z])", "$1*$2");

    /* Convierte número( en número*( */
    ExpresionConvertida = Regex.Replace(ExpresionConvertida, @"(\d)\(",
"$1*(");

    /* Convierte )numero en )*numero */
    ExpresionConvertida = Regex.Replace(ExpresionConvertida, @"\)(\d+)",
@")*$1");

    /* Convierte de )letra en )*letra */
    ExpresionConvertida = Regex.Replace(ExpresionConvertida, @"\)([a-zA-
Z])", @")*$1");

    /* Convierte de letra minúscula( a letra minúscula*( */
    ExpresionConvertida = Regex.Replace(ExpresionConvertida, @"([a-z])\(",
@"$1*(");
```

```
    }

    /* Retorna mensaje de error sintáctico */
    private string MensajeError(int CodigoError) =>
      MensajesError.TryGetValue(CodigoError, out string? mensaje)
        ? mensaje
        : "Error desconocido";


    /* Retorna si el Caracter es un operador matemático */
    private static bool EsUnOperador(char Caracter) =>
Operadores.Contains(Caracter);

    /* Valida casos de dos caracteres seguidos */
    private void ValidaCaracteresSeguidos(string Expresion) {
      for (int Cont = 0; Cont < Expresion.Length - 1; Cont++) {
        char carA = Expresion[Cont];
        char carB = Expresion[Cont + 1];

        if (carA == '.' && carB == '.') TipoError[1] = 1;
        if (carA == '.' && char.IsLower(carB)) TipoError[2] = 1;
        if (EsUnOperador(carA) && EsUnOperador(carB)) TipoError[3] = 1;
        if (EsUnOperador(carA) && carB == ')') TipoError[4] = 1;
        if (char.IsLower(carA) && char.IsDigit(carB)) TipoError[5] = 1;
        if (char.IsLower(carA) && carB == '.') TipoError[6] = 1;
        if (char.IsLower(carA) && char.IsLower(carB)) TipoError[7] = 1;
        if (carA == '(' && EsUnOperador(carB)) TipoError[8] = 1;
        if (carA == '(' && carB == ')') TipoError[9] = 1;
      }
    }

    /* Valida el inicio y fin de la expresión */
    private void ValidaInicioYFinDeLaExpresion(string Expresion) {
      if (EsUnOperador(Expresion[1])) {
        TipoError[10] = 1;
      }
      if (EsUnOperador(Expresion[Expresion.Length - 2])) {
        TipoError[11] = 1;
      }
    }

    /* Valida si hay paréntesis desbalanceados */
    private void ValidaParentesisDesbalanceados(string Expresion) {
      int Balance = 0;
      for (int Cont = 0; Cont < Expresion.Length; Cont++) {
        if (Expresion[Cont] == '(') Balance++;
        if (Expresion[Cont] == ')') Balance--;
        if (Balance < 0) {
```

```
        TipoError[12] = 1;
        return;
      }
    }
    if (Balance != 0) {
      TipoError[13] = 1;
    }
  }

  /* Validar los puntos decimales de un número real */
  private void ValidaPuntosEnNumeroReal(string Expresion) {
    int TotalPuntos = 0;
    for (int Cont = 0; Cont < Expresion.Length; Cont++) {
      if (EsUnOperador(Expresion[Cont])) TotalPuntos = 0;
      if (Expresion[Cont] == '.') TotalPuntos++;
      if (TotalPuntos > 1) {
        TipoError[14] = 1;
        return;
      }
    }
  }
 }
}
```

H/Eval4_1/Evaluador4_1.cs

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

/* Evaluador de expresiones versión 4.1 (diciembre de 2025)
 * Autor: Rafael Alberto Moreno Parra
 * Correo: ramsoftware@gmail.com ;  enginelife@hotmail.com
 * URL: http://darwin.50webs.com
 * GitHub: https://github.com/ramsoftware
 * */
namespace Ejemplo {

 public class Evaluador4_1 {
  /* Donde guarda los valores de variables, Constantes4_1 y piezas */
  public List<double> Valores;

  /* 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 4.1, las Constantes4_1, piezas y variables guardan
    sus valores en la lista Valores.
    En partes, se almacena la posición en Valores */
  public List<ParteEvl4_1> Partes;

  /* 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 */
  public List<PiezaEvl4_1> Piezas;

  /* Evaluar la sintaxis */
  public Sintaxis4_1 objSintaxis;

  /* Constructor */
  public Evaluador4_1() {
    this.Valores = [];
    this.Partes = [];
    this.Piezas = [];
    this.objSintaxis = new();
  }

  /* Analiza la expresión */
  public bool Analizar(string ExpresionOriginal) {
    if (objSintaxis.ChequeaSintaxis(ExpresionOriginal)) {
      CrearPartes(objSintaxis.ExpresionConvertida);
      CrearPiezas();
      return true;
    }
    return false;
  }

  /* 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_1 tmp;

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

        // Cacheo de operandos para menos indirections
        double a = Valores[tmp.pA];
        double b = Valores[tmp.pB];

        // Operación binaria (mantiene tu lógica original)
        Resulta = tmp.Operador switch {
          Constantes4_1.SUMA => a + b,
          Constantes4_1.RESTA => a - b,
          Constantes4_1.MULTIPLICA => a * b,
          Constantes4_1.DIVIDE => a / b,
          _ => Math.Pow(a, b),
        };

        switch (tmp.Funcion) {
          case Constantes4_1.SENO_COD: Resulta = Math.Sin(Resulta); break;
          case Constantes4_1.COSENO_COD: Resulta = Math.Cos(Resulta); break;
          case Constantes4_1.TANGENTE_COD: Resulta = Math.Tan(Resulta);
break;
          case Constantes4_1.VALORABSOLUTO_COD: Resulta = Math.Abs(Resulta);
break;
          case Constantes4_1.ARCOSENO_COD: Resulta = Math.Asin(Resulta);
break;
          case Constantes4_1.ARCOCOSENO_COD: Resulta = Math.Acos(Resulta);
break;
          case Constantes4_1.ARCOTANGENTE_COD: Resulta = Math.Atan(Resulta);
break;
          case Constantes4_1.LOGARITMONATURAL_COD: Resulta =
Math.Log(Resulta); break;
          case Constantes4_1.EXPONENCIAL_COD: Resulta = Math.Exp(Resulta);
break;
          case Constantes4_1.RAIZCUADRADA_COD: Resulta = Math.Sqrt(Resulta);
break;
          default:
```

```
          break;
        }

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

    /* Convierte las partes en las piezas finales de ejecución */
    private void CrearPiezas() {
      Piezas.Clear();

      int Contador = Partes.Count - 1;
      do {
        if (Partes[Contador].Tipo == Constantes4_1.ESPARABRE) {

          /* Evalúa las potencias de izquierda a derecha como Excel */
          GeneraPiezaOpera(Constantes4_1.POTENCIA, Constantes4_1.POTENCIA,
Contador);

          /* Luego evalúa multiplicar y dividir */
          GeneraPiezaOpera(Constantes4_1.MULTIPLICA, Constantes4_1.DIVIDE,
Contador);

          /* Finalmente evalúa sumar y restar */
          GeneraPiezaOpera(Constantes4_1.SUMA, Constantes4_1.RESTA,
Contador);

          /* Agrega la función a la última pieza */
          if (Contador > 0 && Partes[Contador - 1].Tipo ==
Constantes4_1.ESFUNCION) {
            Piezas[Piezas.Count - 1].Funcion = Partes[Contador - 1].Funcion;
            Partes.RemoveAt(Contador - 1);
            Partes.RemoveAt(Contador - 1);
            Partes.RemoveAt(Contador);
          }
          else {
            /* Quita el paréntesis 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_1 tmpParte = Partes[Contador];
    if (tmpParte.Tipo == Constantes4_1.ESOPERADOR &&
      (tmpParte.Operador == OperA || tmpParte.Operador == OperB)) {
      ParteEvl4_1 tmpParteIzq = Partes[Contador - 1];
      ParteEvl4_1 tmpParteDer = Partes[Contador + 1];

      /* Crea Pieza */
      PiezaEvl4_1 NuevaPieza = new();
      NuevaPieza.Funcion = -1;

      switch (tmpParteIzq.Tipo) {
        case Constantes4_1.ESNUMERO:
          NuevaPieza.pA = tmpParteIzq.posNumero;
          break;

        case Constantes4_1.ESVARIABLE:
          NuevaPieza.pA = tmpParteIzq.posVariable;
          break;

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

      NuevaPieza.Operador = tmpParte.Operador;

      switch (tmpParteDer.Tipo) {
        case Constantes4_1.ESNUMERO:
          NuevaPieza.pB = tmpParteDer.posNumero;
          break;

        case Constantes4_1.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.RemoveRange(Contador, 2);

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

      /* Cambia la parte anterior por parte que acumula */
      tmpParteIzq.Tipo = Constantes4_1.ESACUMULA;
      tmpParteIzq.posAcumula = NuevaPieza.PosResultado;
    }
    Contador++;
  } while (Partes[Contador].Tipo != Constantes4_1.ESPARCIERRA);
}

/* Divide la expresión en partes, yendo de caracter en caracter */
private void CrearPartes(string Expresion) {
  Partes.Clear();

  /* Hace espacio para las 26 variables que
   * puede manejar el evaluador */
  Valores = Enumerable.Repeat(0.0, 26).ToList();

  StringBuilder Numero = new();
  for (int Pos = 0; Pos < Expresion.Length; Pos++) {
    char Letra = Expresion[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 digito 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(double.Parse(Numero.ToString(),
CultureInfo.InvariantCulture));
            Partes.Add(new ParteEvl4_1(Constantes4_1.ESNUMERO,
Valores.Count - 1));
            Numero.Clear();
          }
          /* Agregar el operador matemático */
          Partes.Add(new ParteEvl4_1(Letra));
          break;

        case '(':   /* Es paréntesis que abre */
          Partes.Add(new ParteEvl4_1(Constantes4_1.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(double.Parse(Numero.ToString(),
CultureInfo.InvariantCulture));
            Partes.Add(new ParteEvl4_1(Constantes4_1.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 == Constantes4_1.ESPARABRE) {
            Partes.Add(new ParteEvl4_1(Constantes4_1.ESOPERADOR, 0));
            Valores.Add(0);
            Partes.Add(new ParteEvl4_1(Constantes4_1.ESNUMERO,
Valores.Count - 1));
          }

          /* Adiciona el paréntesis que cierra */
          Partes.Add(new ParteEvl4_1(Constantes4_1.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_1(Constantes4_1.ESFUNCION, Letra -
'A'));
          break;
        default:
          Partes.Add(new ParteEvl4_1(Constantes4_1.ESVARIABLE, Letra -
'a'));
          break;
       }
     }
   }
  }
}
```

Como usar el evaluador de expresiones

En Visual Studio 2026, se añaden las cinco(5) clases Constantes4_1.cs, ParteEvl4_1, PiezaEvl4_1. Sintaxis4_1, Evaluador4_1.cs al proyecto:

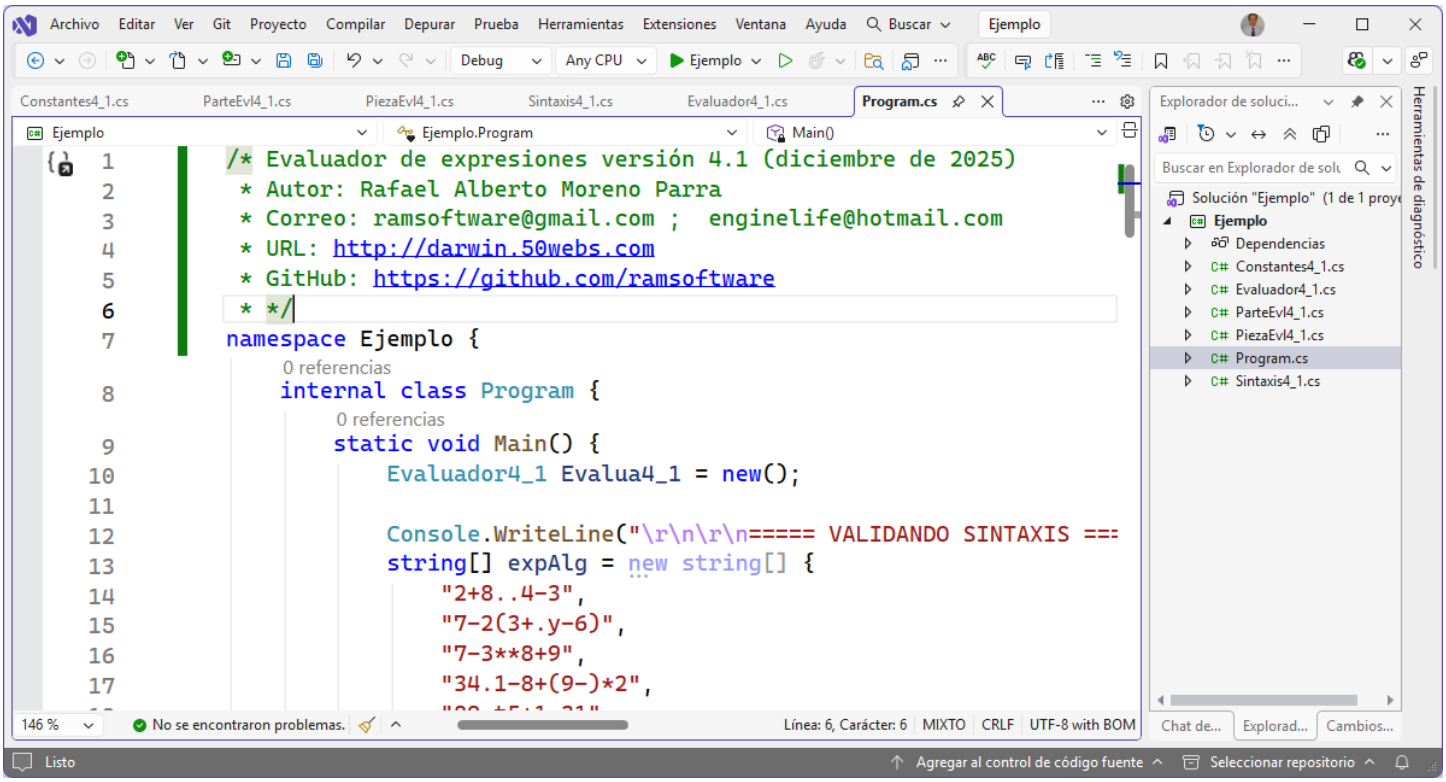


Ilustración 1: Uso del evaluador de expresiones

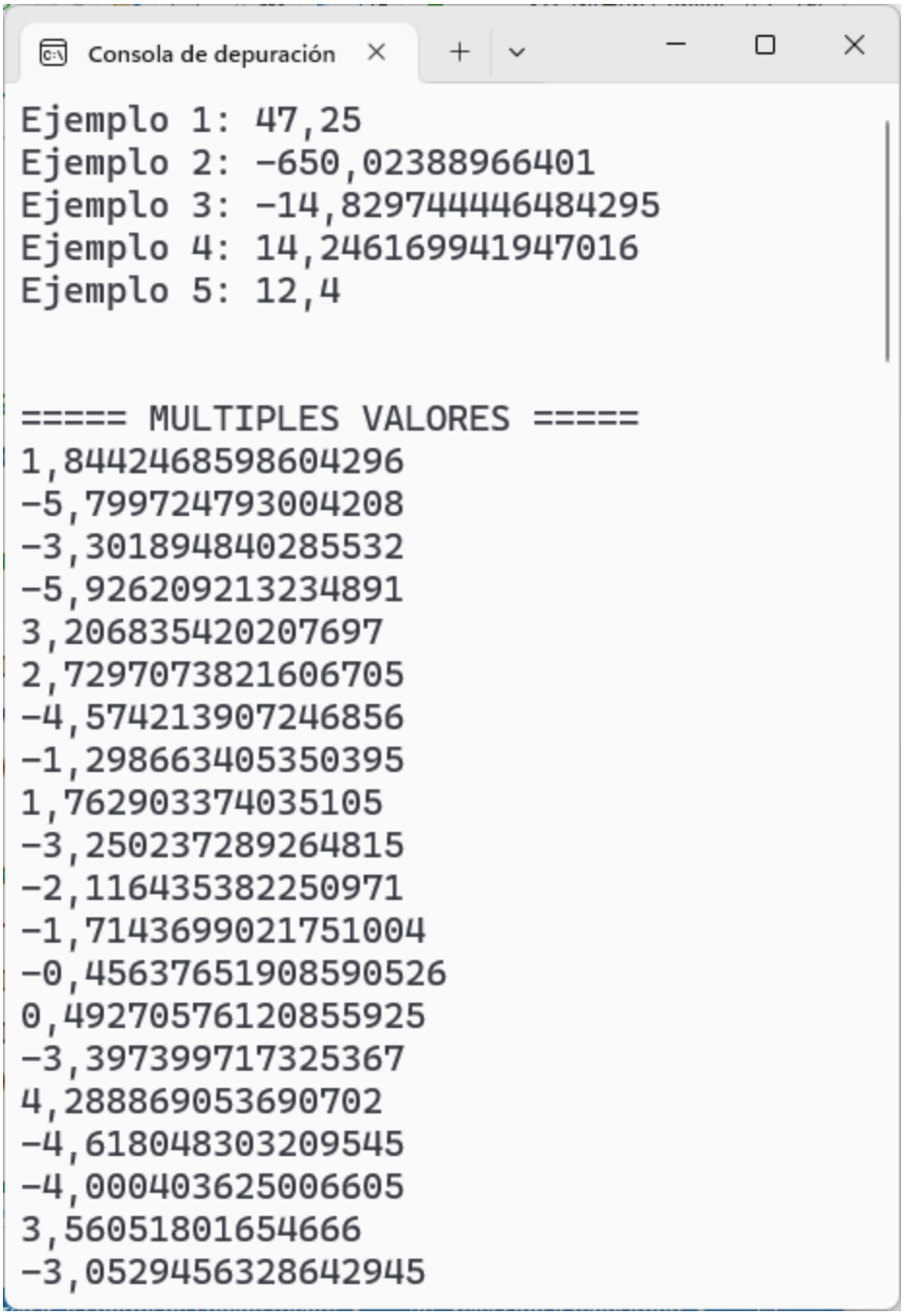


Ilustración 2: Resultado obtenido con el evaluador

Este es el código fuente y se ha puesto en un proyecto:

H/Eval4_1/Program.cs

```
/* Evaluador de expresiones versión 4.1 (diciembre de 2025)
 * Autor: Rafael Alberto Moreno Parra
 * Correo: ramsoftware@gmail.com ;  enginelife@hotmail.com
 * URL: http://darwin.50webs.com
 * GitHub: https://github.com/ramsoftware
 * */
namespace Ejemplo {
  internal class Program {
    static void Main() {
      Evaluador4_1 Evalua4_1 = new();

      Console.WriteLine("\r\n\r\n===== VALIDANDO SINTAXIS =====");
      string[] expAlg = new string[] {
        "2+8..4-3",
        "7-2(3+.y-6)",
        "7-3**8+9",
        "34.1-8+(9-)*2",
        "89-t5+1.31",
        "4/p.12+9",
        "2+hu-3.4",
        "9-(*2+4)",
        "9.02-7*()+4",
        "/3*7",
        "9-61/",
        "9+3))-((2+9",
        "(7-8)(3+1+(3*4)",
        "9.03.2+8.0121",
        "0.4-cos(3*x+2)/(2*x)-tan(7/0.8*x+3)+9.012/x^2+3*x-7.02*x"
      };

      for (int num = 0; num < expAlg.Length; num++) {
        Console.WriteLine("\r\nExpr " + num + ": " + expAlg[num]);

        if (Evalua4_1.Analizar(expAlg[num])) {
          Console.WriteLine("Sintaxis correcta");
        }
        else {
          List<string> Errores = Evalua4_1.objSintaxis.ObtenerMensajesErrores();
          for (int i = 0; i < Errores.Count; i++) {
            Console.WriteLine("Error: " + Errores[i]);
          }
        }
      }

      //Ejemplo 1: Una expresión simple
```

```
        string Ecuacion = "3+(4)-(5)+(6)-(7)";
        Evalua4_1.Analizar(Ecuacion);
        double valorY = Evalua4_1.Evaluar();
        Console.WriteLine("Ejemplo 1: " + valorY);

        //Ejemplo 2: Una expresión simple con números reales
        Ecuacion = "7.318/5.0045-9.071^2*8.04961";
        Evalua4_1.Analizar(Ecuacion);
        valorY = Evalua4_1.Evaluar();
        Console.WriteLine("Ejemplo 2: " + valorY);

        //Ejemplo 3: Una expresión con paréntesis
        Ecuacion = "(3+2)-7";
        Evalua4_1.Analizar(Ecuacion);
        valorY = Evalua4_1.Evaluar();
        Console.WriteLine("Ejemplo 3: " + valorY);

        //Ejemplo 4: Una expresión con funciones
        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)";
        Evalua4_1.Analizar(Ecuacion);
        valorY = Evalua4_1.Evaluar();
        Console.WriteLine("Ejemplo 4: " + valorY);

        //Ejemplo 5: Una expresión con uso de variables
        //(deben estar en minúsculas)
        Ecuacion = "3*x+2*y";
        Evalua4_1.Analizar(Ecuacion);

        //Le da valor a las variables
        Evalua4_1.DarValorVariable('x', 1.8);
        Evalua4_1.DarValorVariable('y', 3.5);
        valorY = Evalua4_1.Evaluar();
        Console.WriteLine("Ejemplo 5: " + valorY);

        //Una expresión con uso de variables
        Console.WriteLine("\r\n\r\n===== MULTIPLES VALORES =====");

        Ecuacion = "3*cos(2*x+4)-5*sen(4*y-7)";
        Evalua4_1.Analizar(Ecuacion);

        //Después de ser analizada, se le dan los
        //valores a las variables, esto hace que
        //el evaluador sea muy rápido
        Random Azar = new();
        for (int cont = 1; cont <= 20; cont++) {
```

```
        double X = Azar.NextDouble();
        double Y = Azar.NextDouble();
        Evalua4_1.DarValorVariable('x', X);
        Evalua4_1.DarValorVariable('y', Y);
        valorY = Evalua4_1.Evaluar();
        Console.WriteLine(valorY);
      }
    }
  }
}
```

Chequear: Desempeño con el evaluador interno de C#

.NET 10 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 4.1, 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/Chequear.zip

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

namespace Chequear {
  internal class Program {

    const int ANCHO_ANALISIS = 10; // ancho de la columna "Análisis"
    const int ANCHO_EVALUA = 10; // ancho de la columna "Evalúa"
    static void Main() {
      Random Azar = new();

      //Versión 2025
      Evaluador4_1 evaluador2025 = 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 TotalTiempo2025Evalua = 0, TotalTiempo2025Analiza = 0;
      double valor2025, AcumValor2025 = 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 2025. Análisis.
    temporizador.Reset();
    temporizador.Start();
    evaluador2025.Analizar(ecuacion);
    temporizador.Stop();
    TotalTiempo2025Analiza += temporizador.ElapsedTicks;

    //Versión 2025. Evaluación
    temporizador.Reset();
    temporizador.Start();
    valor2025 = 0;
    for (int cont = 0; cont < arregloX.Length; cont++) {
      evaluador2025.DarValorVariable('x', arregloX[cont]);
      evaluador2025.DarValorVariable('y', arregloY[cont]);
      evaluador2025.DarValorVariable('z', arregloZ[cont]);
      valor2025 += Math.Abs(evaluador2025.Evaluar());
    }
    temporizador.Stop();
    TotalTiempo2025Evalua += 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(valor2025) > 10000000) continue;
    if (double.IsNaN(valor2025) || double.IsInfinity(valor2025))
      continue;

    AcumValor2025 += valor2025;
    AcumInterno += valorInterno;
  }

  Console.WriteLine("[Eval4_1  ] Acumula: {0," + ANCHO_ANALISIS + "}",
```

```
      AcumValor2025);

    Console.WriteLine("[Interno  ] Acumula: {0," + ANCHO_ANALISIS + "}",
      AcumInterno);

    Console.WriteLine("\r\n\r\n[Eval4_1  ] Análisis: {0," + ANCHO_ANALISIS
+ "} Evalúa: {1," + ANCHO_EVALUA + "}",
      TotalTiempo2025Analiza, TotalTiempo2025Evalua);

    Console.WriteLine("[Eval4_1  ] Análisis y evaluación: {0," +
ANCHO_ANALISIS + "}",
      TotalTiempo2025Analiza + TotalTiempo2025Evalua);

    Console.WriteLine("[Interno  ] Análisis y evaluación: {0," +
ANCHO_ANALISIS + "}", 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);
    }
  }
}
```

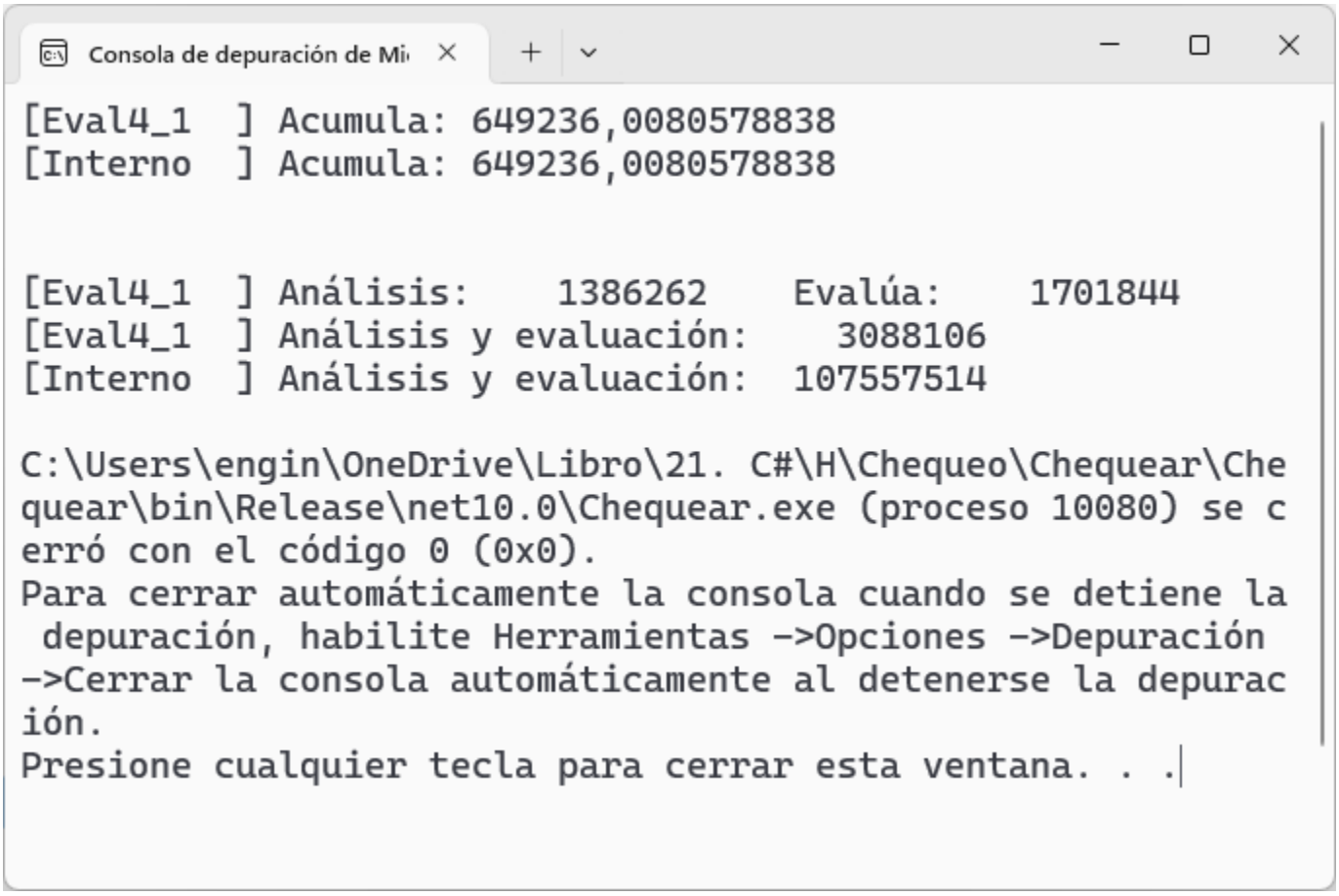


Ilustración 3: Métrica compara ambos evaluadores

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 4.1 es más rápido que el evaluador interno. La explicación a esta diferencia tan alta (a favor del Evaluador 4.1) 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.

Evaluador de expresiones usando un árbol binario. Programación recursiva

Autor: Rafael Alberto Moreno Parra

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.Write("\"[" + ID + "] " + Operador + "\"");
    else
      Console.Write("\"[" + 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:

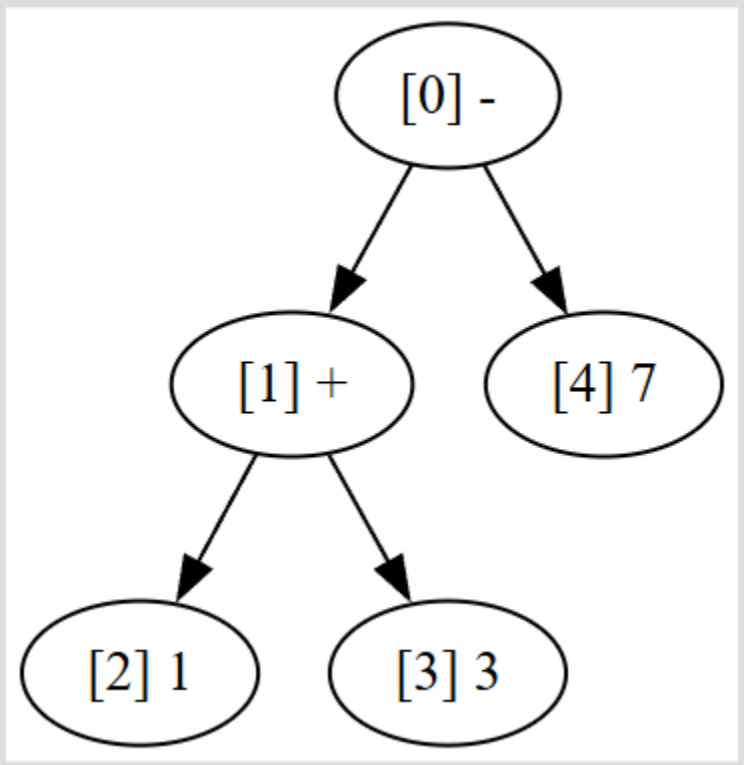


Ilustración 4: Árbol binario generado

Un segundo ejemplo:

2.7-3.6+4.5-1.8

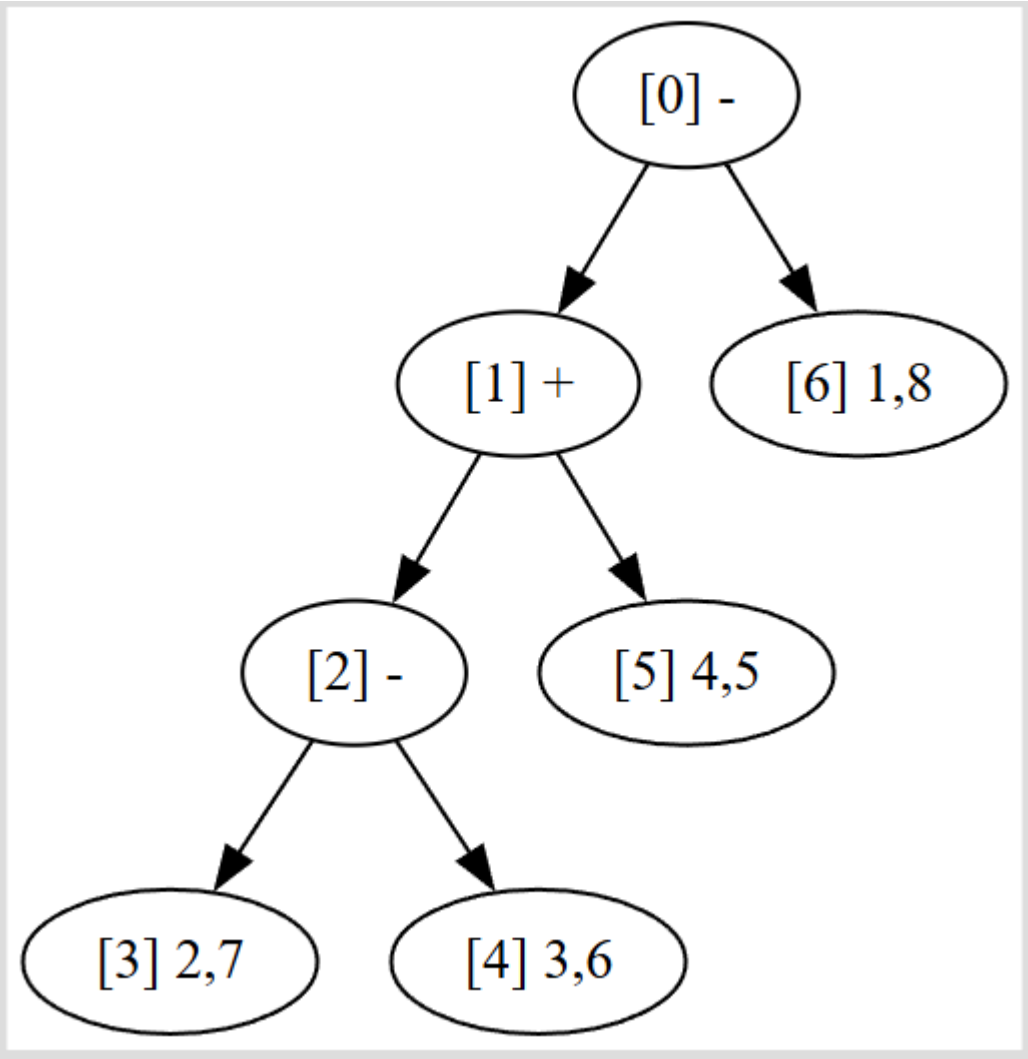


Ilustración 5: Árbol binario generado

El programa en C# que aplica el algoritmo:

H/Árbol/001.zip

```
using System.Globalization;

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

namespace Ejemplo {
  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.Write("\"[" + ID + "] " + Operador + "\"");
      else
        Console.Write("\"[" + 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);
        }
      }
    }
  }
}
```

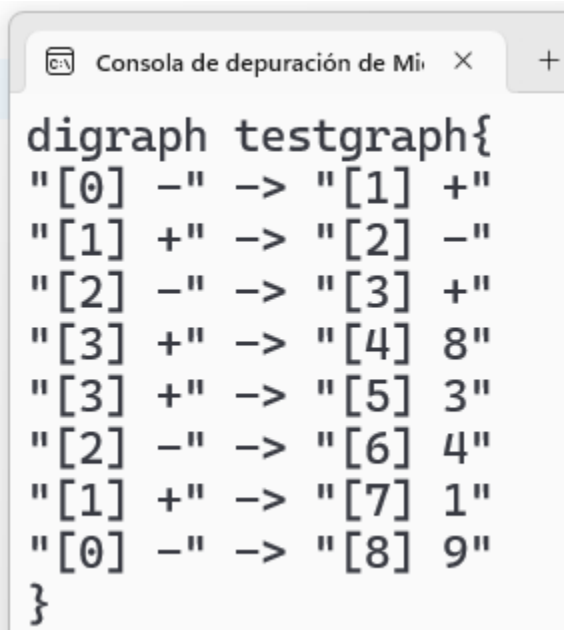


Ilustración 6: Ejemplo de ejecución del programa

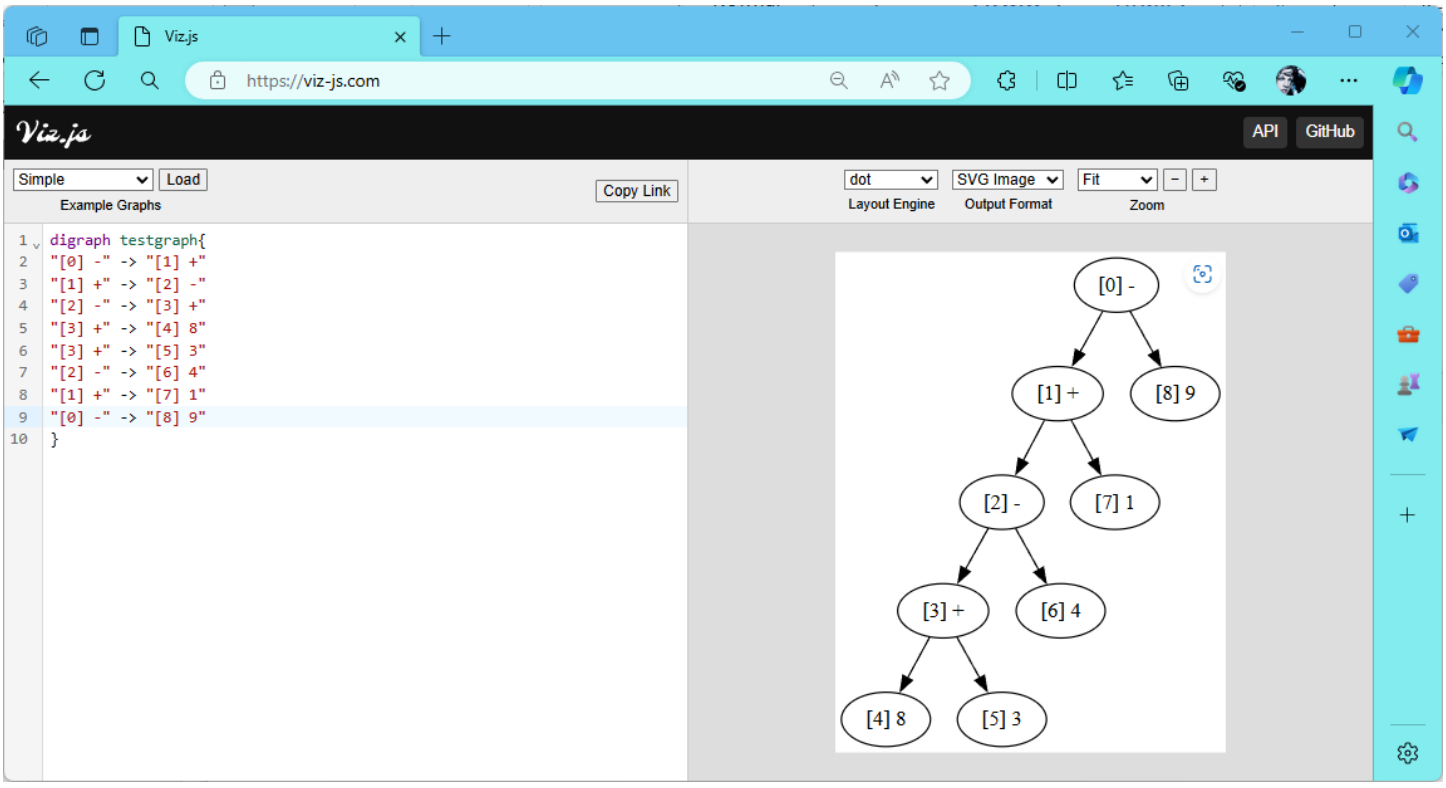


Ilustración 7: Interpretación en viz-js.com

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

H/Árbol/001.zip

```
using System.Globalization;

//Forma el árbol binario dada una expresión matemática
//de números y operadores suma y resta
//Luego evalúa la expresión
namespace Ejemplo {
  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.Write("\"[" + ID + "] " + Operador + "\"");
      else
        Console.Write("\"[" + ID + "] " + Numero + "\"");
    }
  }

  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;
    }
  }
}
```

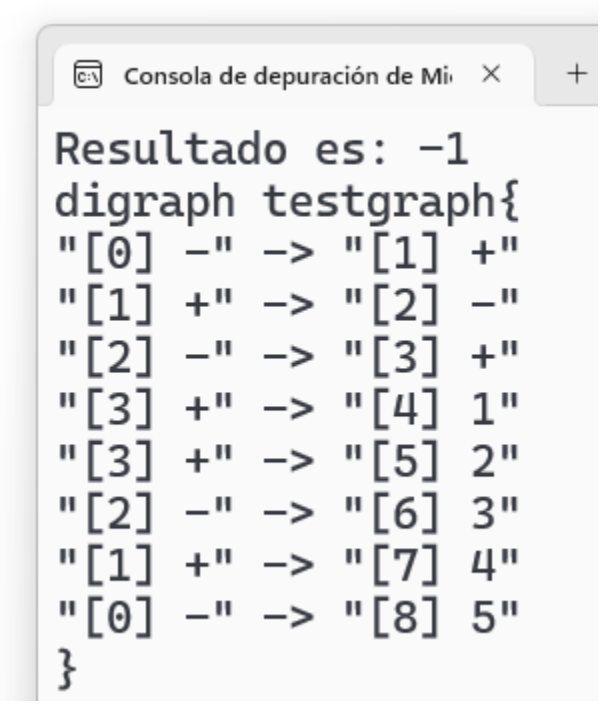


Ilustración 8: Resultado al evaluar

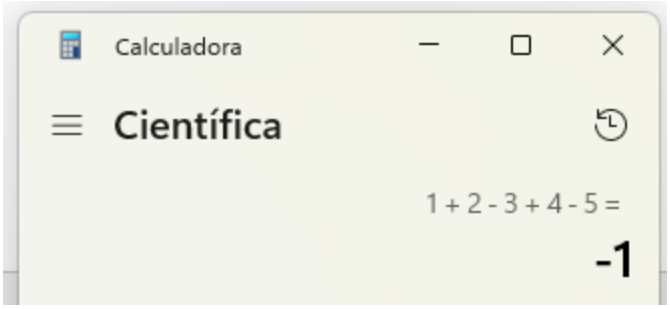


Ilustración 9: Validado con la calculadora

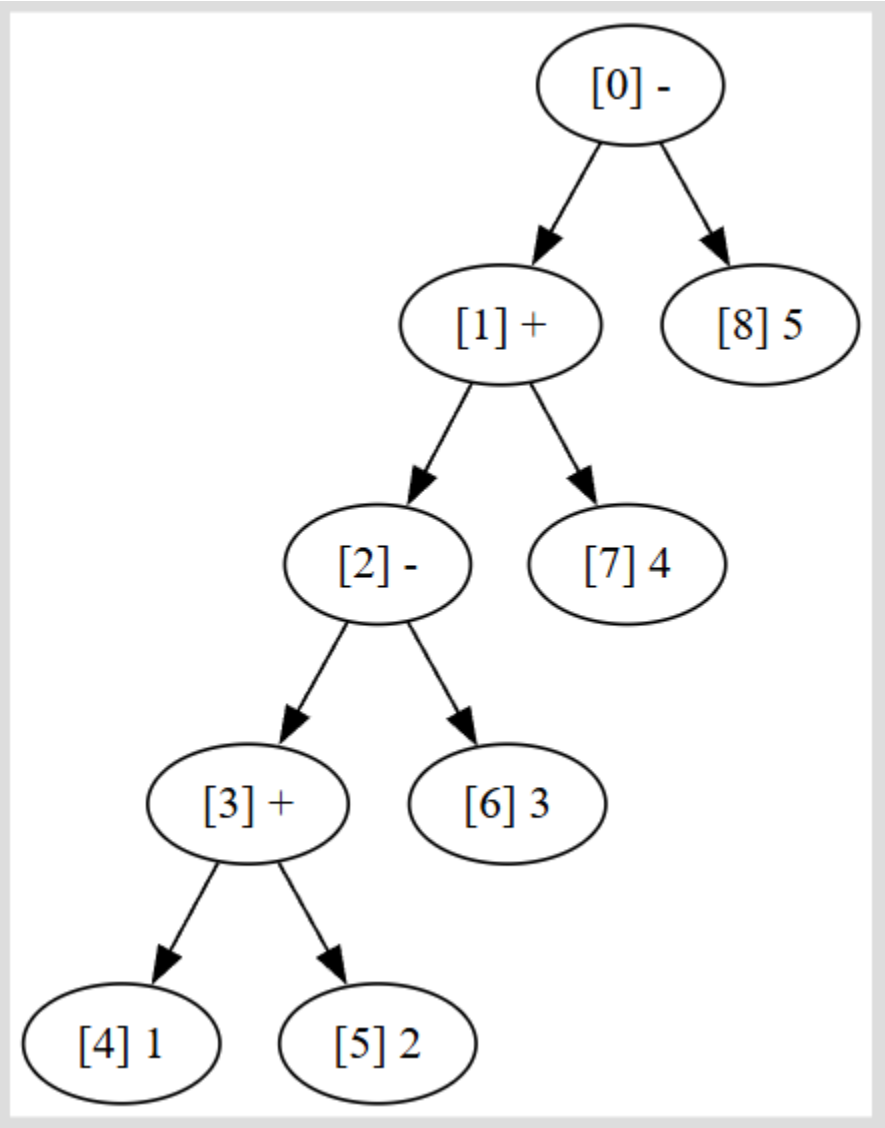


Ilustración 10: Árbol binario generado

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:

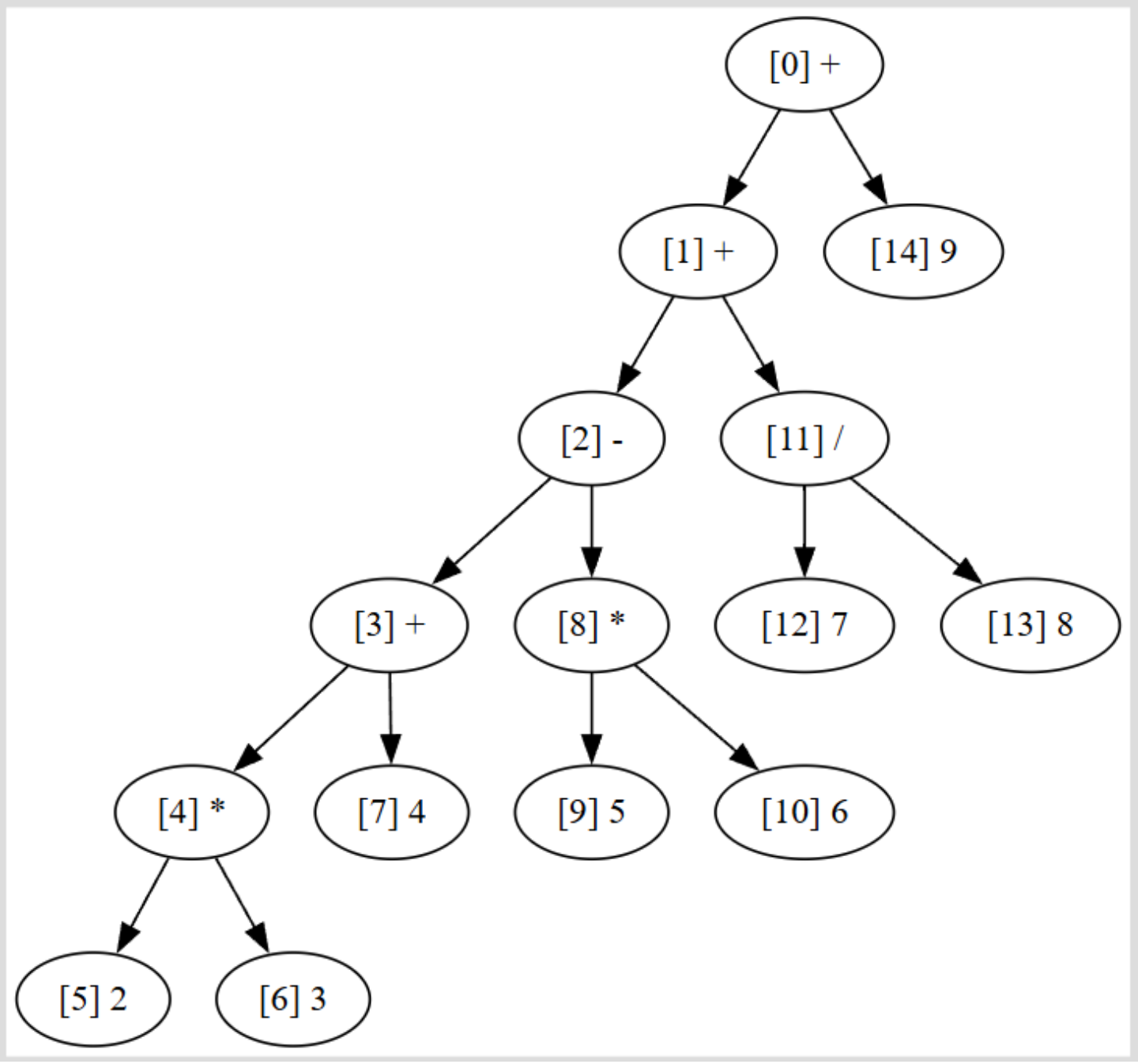


Ilustración 11: Árbol binario generado

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 Ejemplo {
  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.Write("\"[" + ID + "] " + Operador + "\"");
      else
        Console.Write("\"[" + ID + "] " + Numero + "\"");
    }
  }

  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:

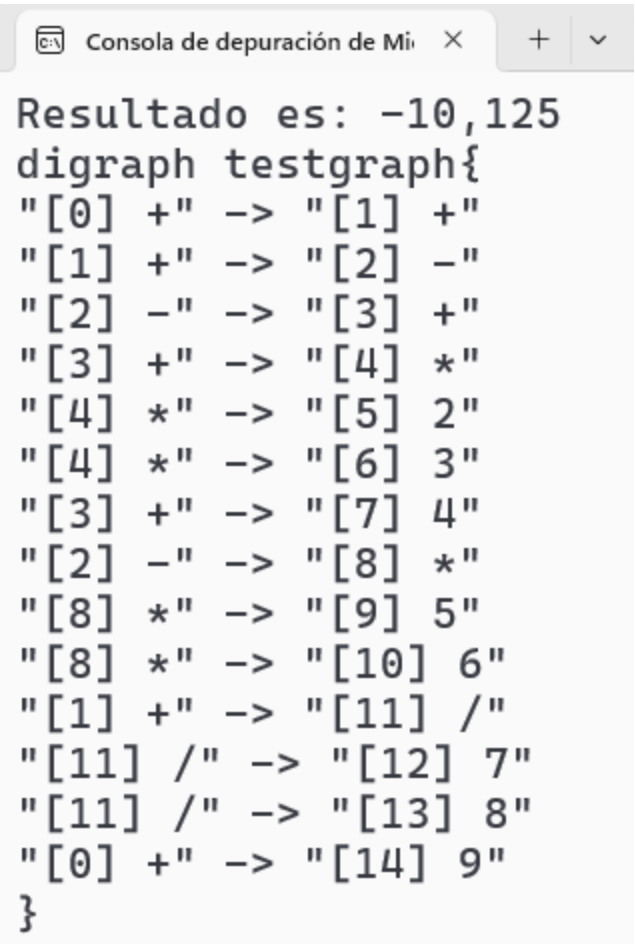


Ilustración 12: Ejecución del evaluador

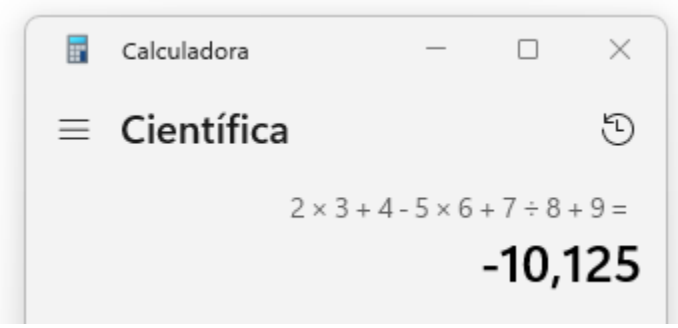


Ilustración 13: Prueba con calculadora

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:

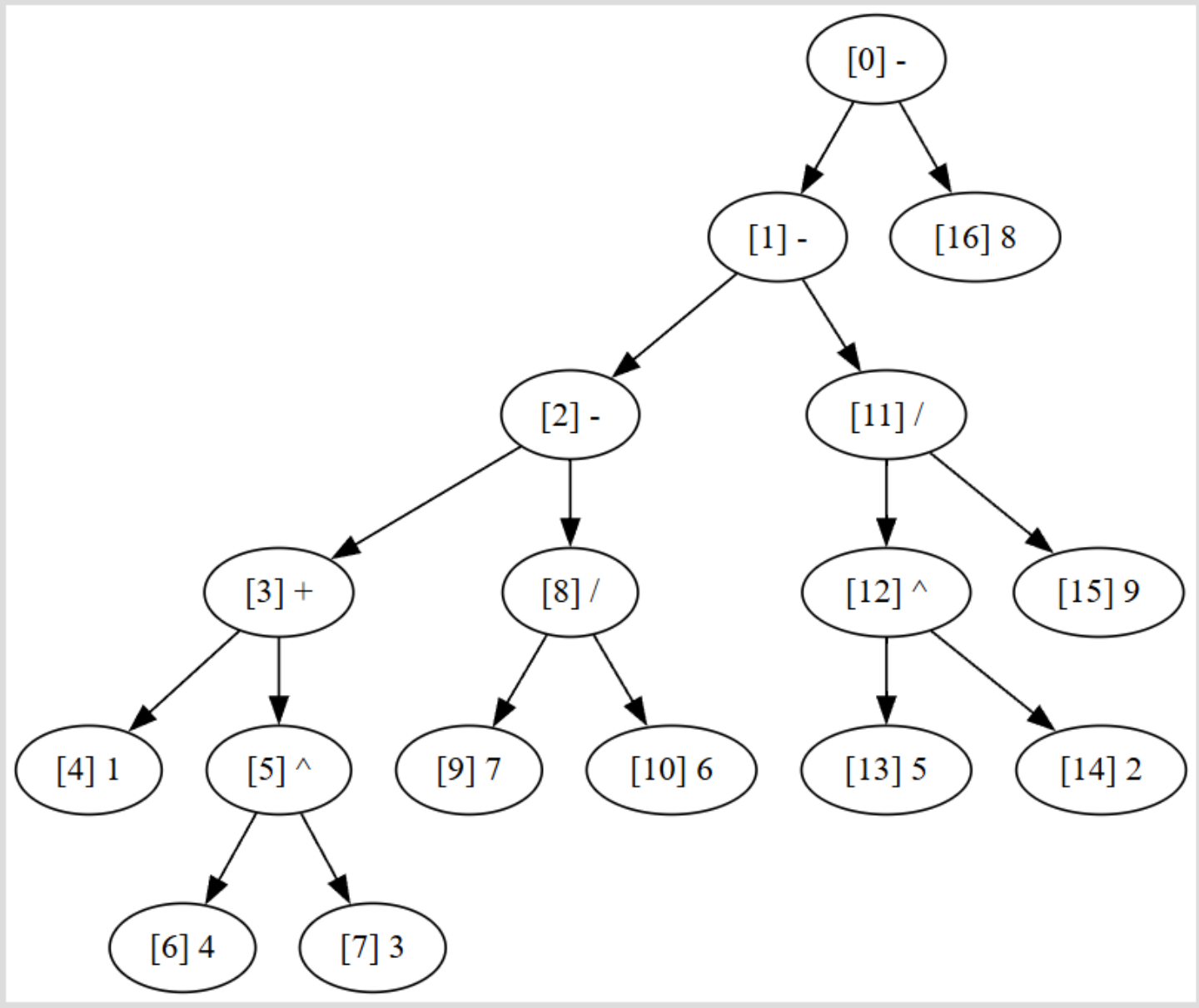


Ilustración 14: Árbol binario generado

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 Ejemplo {
  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.Write("\"[" + ID + "] " + Operador + "\"");
      else
        Console.Write("\"[" + ID + "] " + Numero + "\"");
    }
  }

  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:

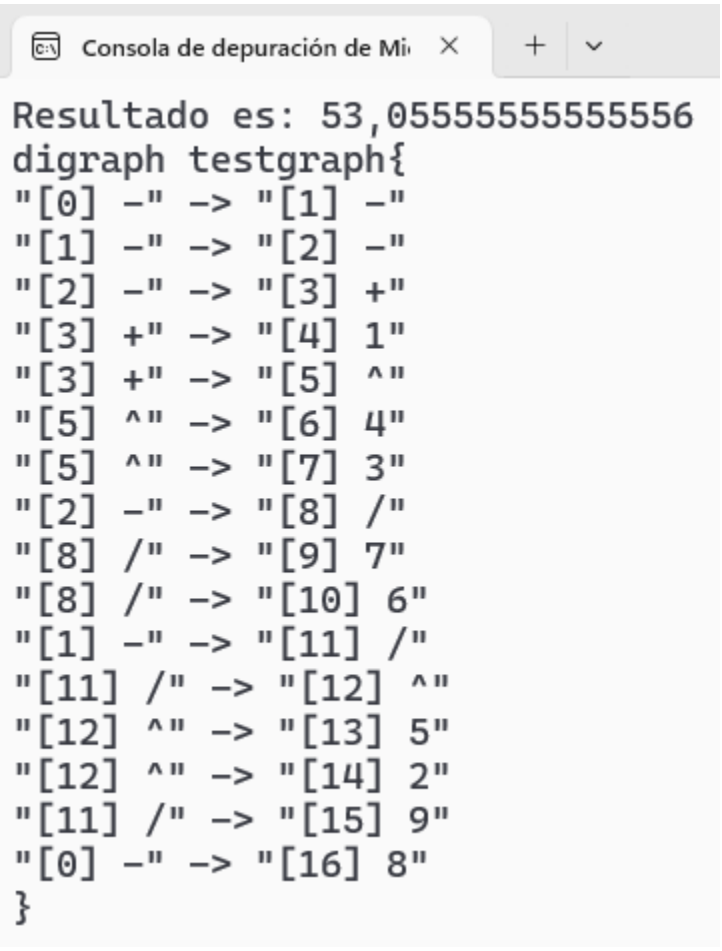


Ilustración 15: Ejemplo de ejecución

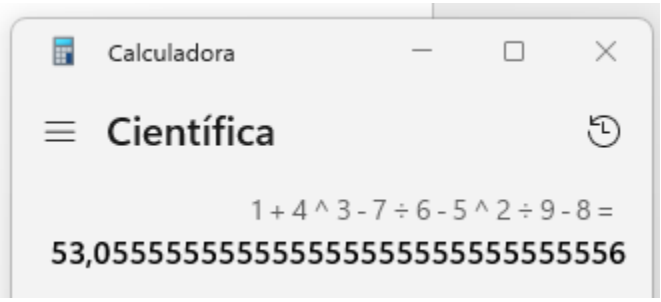


Ilustración 16: Prueba con la calculadora

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:

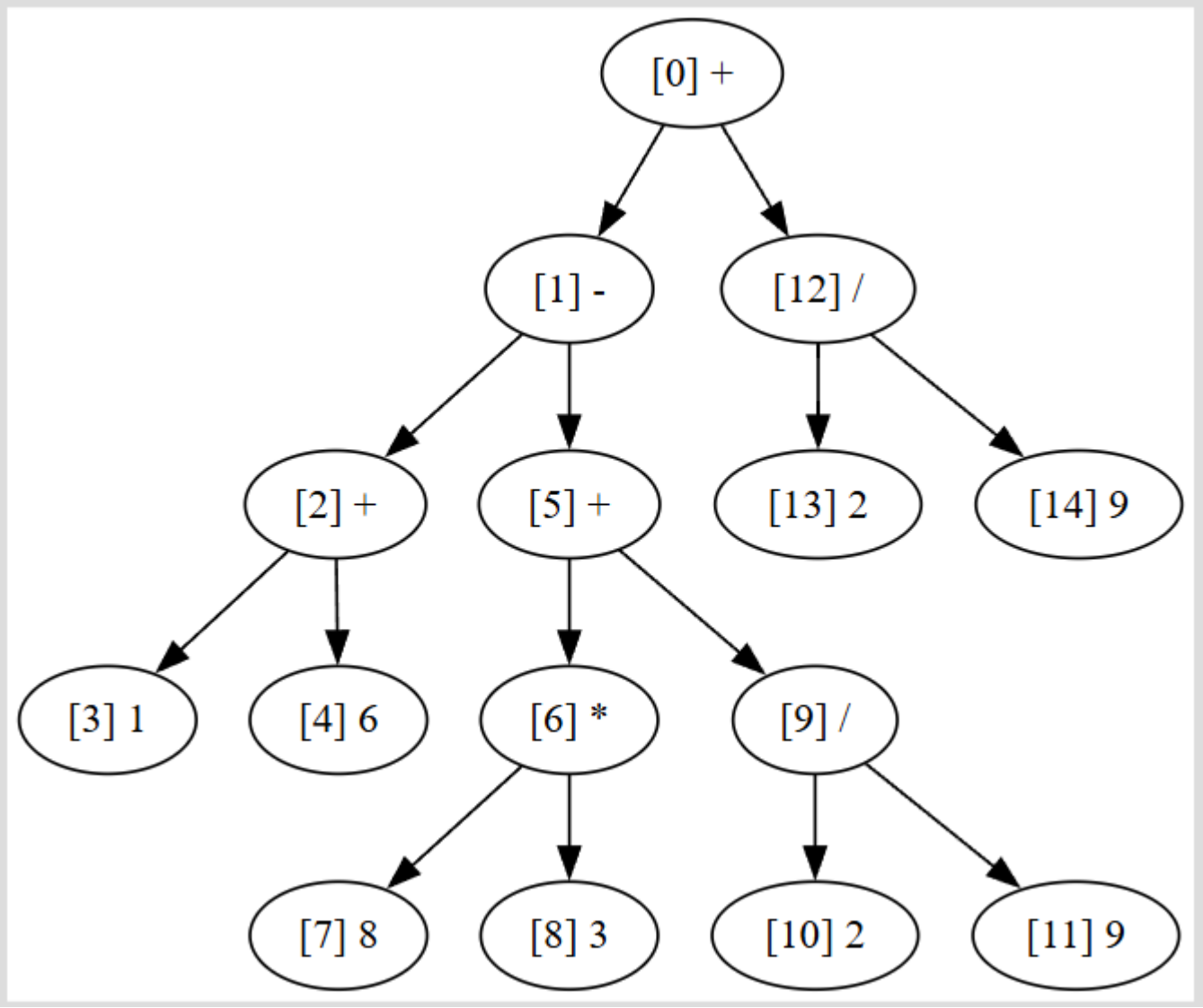


Ilustración 17: Árbol binario generado

Este es el programa:

H/Árbol/004.zip

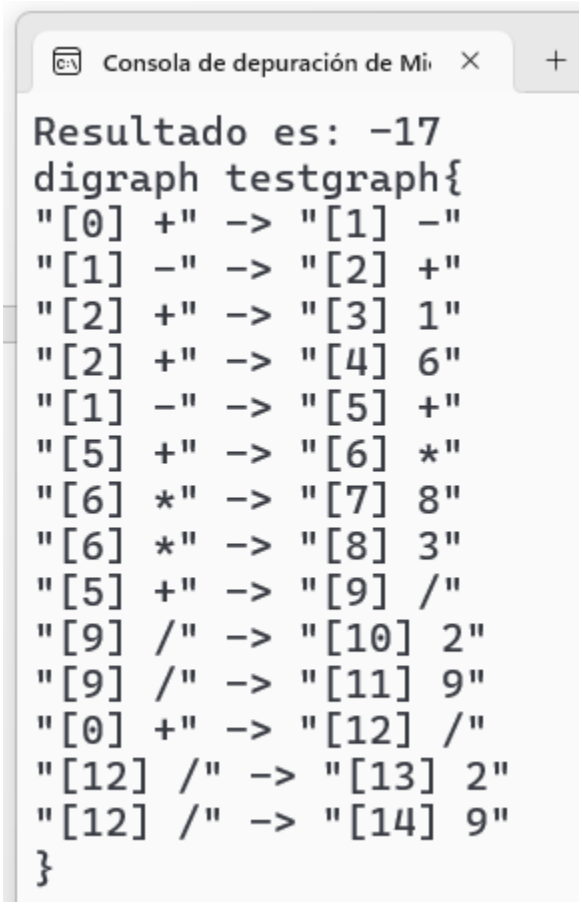
```
using System.Globalization;

//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 Ejemplo {
  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.Write("\"[" + ID + "] " + Operador + "\"");
      else
        Console.Write("\"[" + ID + "] " + Numero + "\"");
    }
  }

  internal class Program {
    static int Identifica = 0;

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

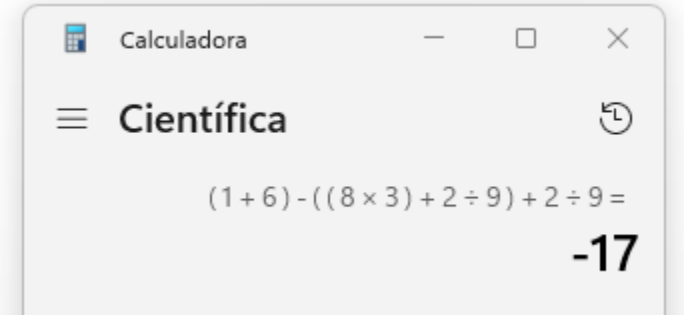
```
    //Ecuación los cuatro operadores
    string Ecuacion = "(1+6)-((8*3)+2/9)+2/9";
    Nodo MiArbol = null;
    MiArbol = CreaArbol(Ecuacion, MiArbol);

    //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
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"
}


Calculadora
Científica
(1+6)-((8×3)+2÷9)+2÷9=
-17

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 4.1

```
     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/005.zip

```
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 Ejemplo {
  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.Write("\"[" + ID + "] " + Operador + "\"");
    else if (Variable != -1) {
      int Ascii = 'a' + Variable;
      char Letra = (char)Ascii;
      Console.Write("\"[" + ID + "] " + Letra + "\"");
    }
    else
      Console.Write("\"[" + ID + "] " + Numero + "\"");
  }
}

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--;
```

```
        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:

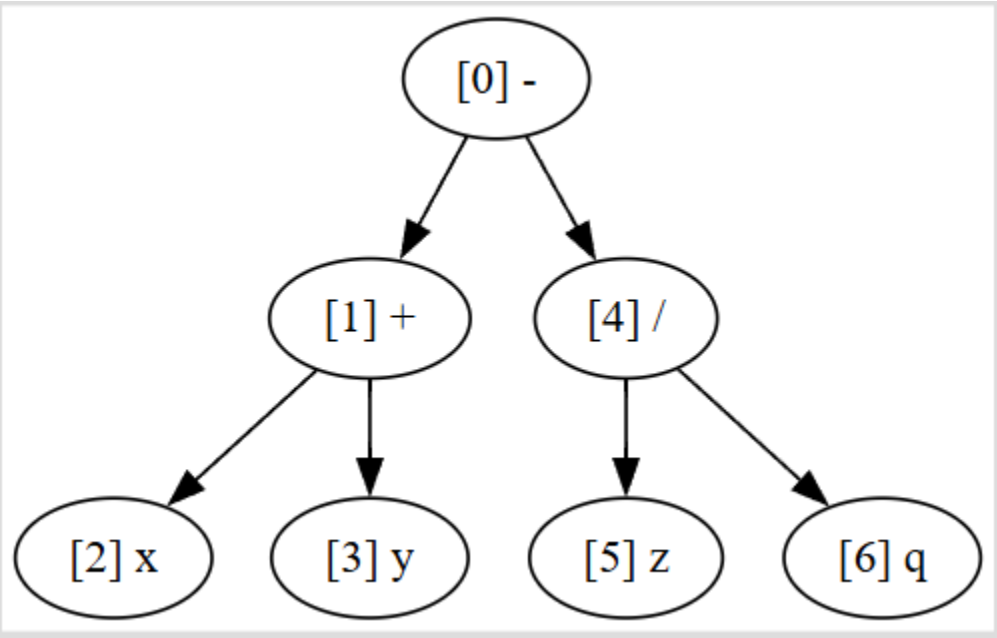


Ilustración 18: Árbol binario generado

Y esta es su ejecución:

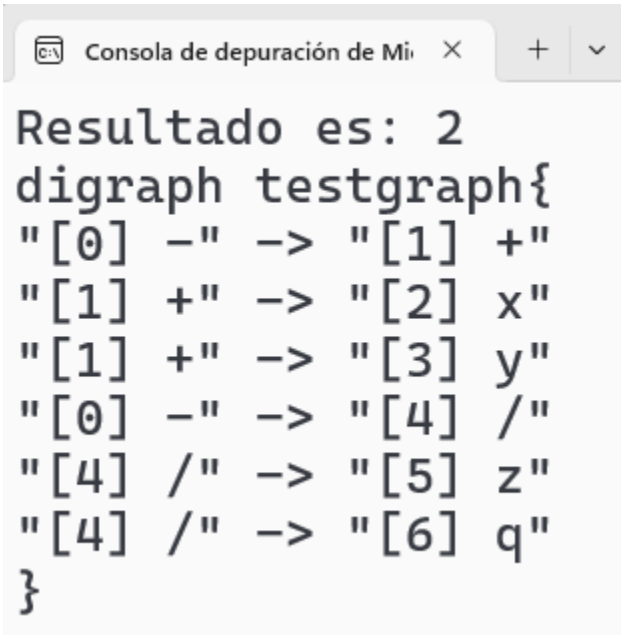


Ilustración 19: Ejemplo de ejecución

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-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

Este es el código:

H/Árbol/006.zip

```
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 Ejemplo {
  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.Write("\"[" + ID + "] " + Operador + "\"");
    else if (Variable != -1) {
      int Ascii = 'a' + Variable;
      char Letra = (char)Ascii;
      Console.Write("\"[" + ID + "] " + Letra + "\"");
    }
    else if (Funcion != -1) {
      Console.Write("\"[" + 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.Write("seno \""); break;
        case 1: Console.Write("coseno \""); break;
        case 2: Console.Write("tangente \""); break;
        case 3: Console.Write("absoluto \""); break;
        case 4: Console.Write("arcoseno \""); break;
        case 5: Console.Write("arcocoseno \""); break;
        case 6: Console.Write("arcotangente \""); break;
        case 7: Console.Write("log \""); break;
        case 8: Console.Write("ceil \""); break;
        case 9: Console.Write("exp \""); break;
        case 10: Console.Write("sqrt \""); break;
      }
    }
    else
      Console.Write("\"[" + ID + "] " + Numero + "\"");
  }
}
```

```
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;
  }
 }
}
```

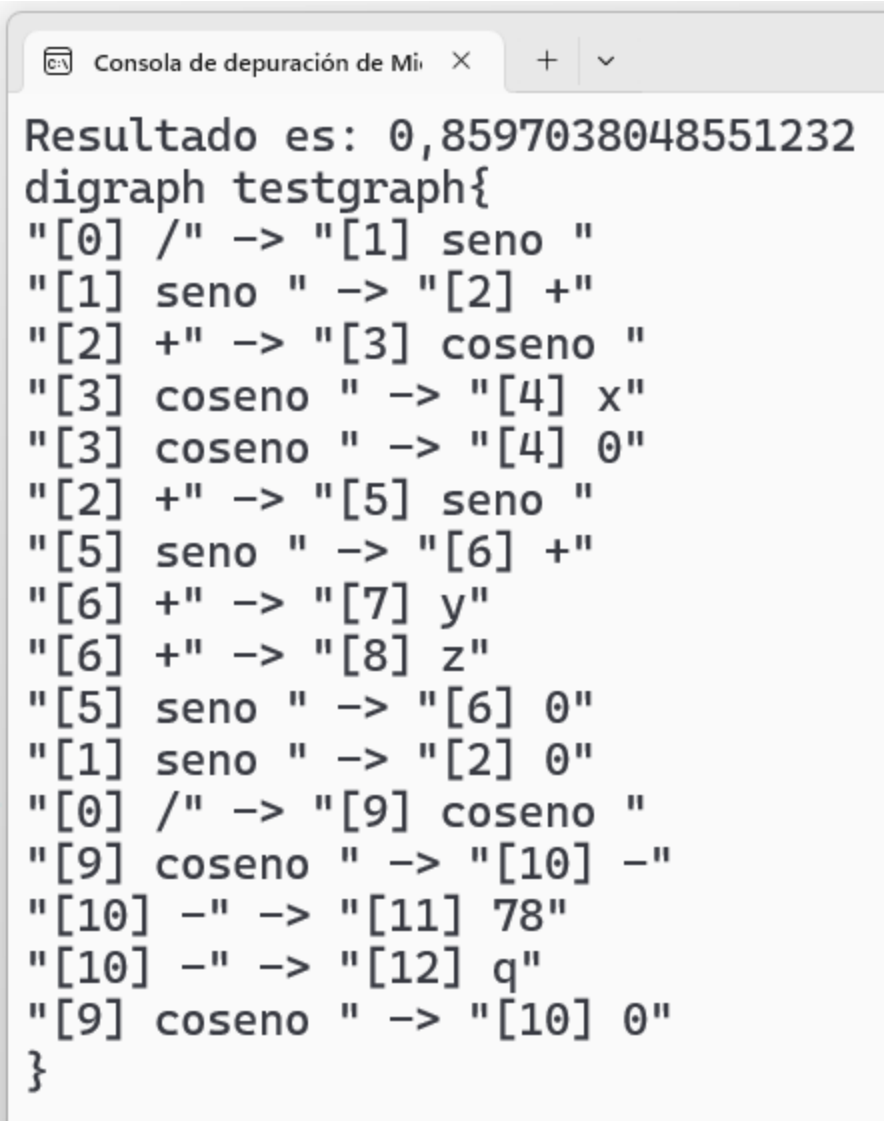


Ilustración 20: Ejemplo de ejecución

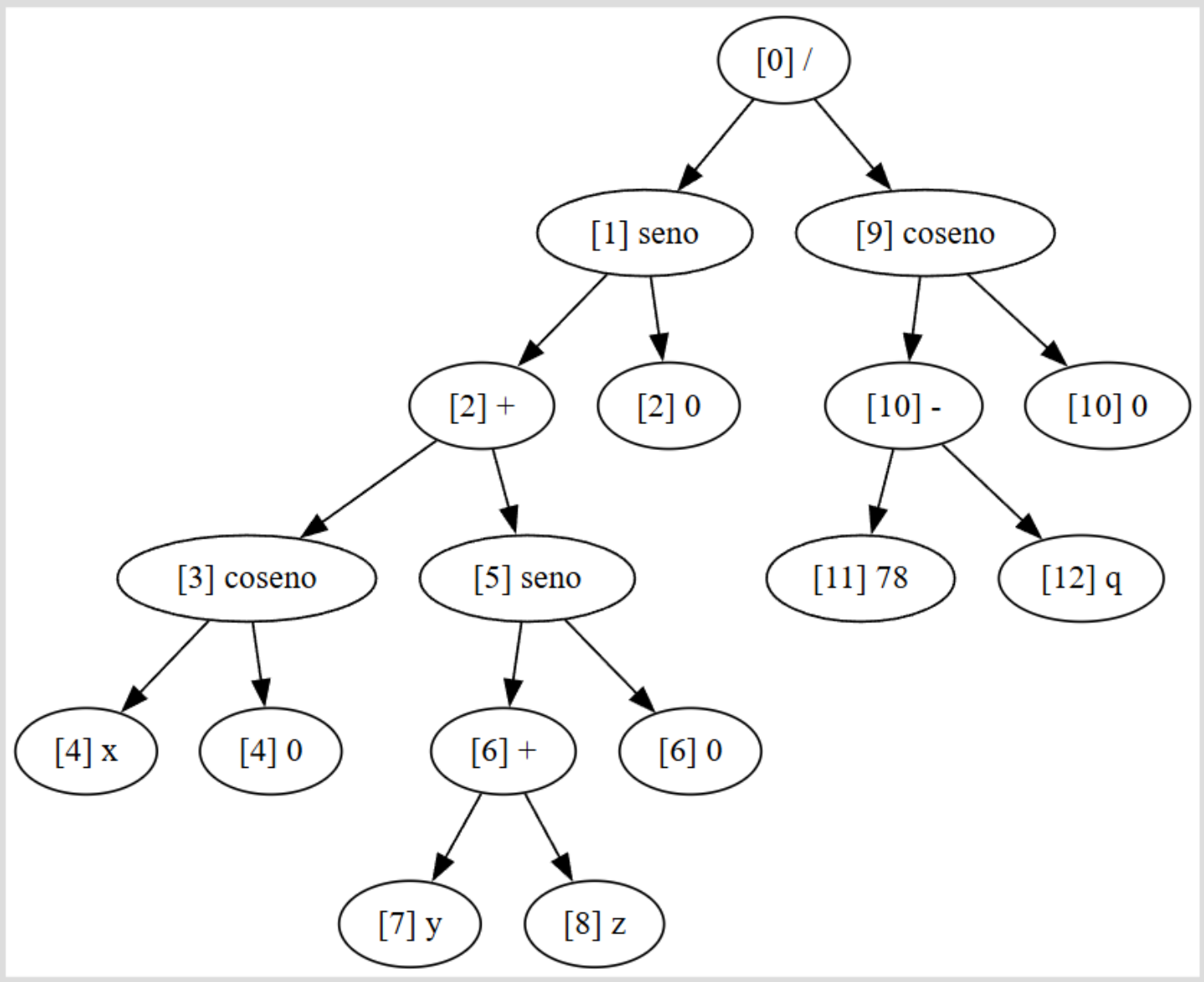


Ilustración 21: Árbol binario generado

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/007.zip

```
namespace Ejemplo {
  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.Write("\"[" + ID + "] " + Operador + "\"");
      else if (Variable != -1) {
        int Ascii = 'a' + Variable;
        char Letra = (char)Ascii;
        Console.Write("\"[" + ID + "] " + Letra + "\"");
      }
      else if (Funcion != -1) {
        Console.Write("\"[" + 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.Write("seno \""); break;
          case 1: Console.Write("coseno \""); break;
          case 2: Console.Write("tangente \""); break;
          case 3: Console.Write("absoluto \""); break;
          case 4: Console.Write("arcoseno \""); break;
          case 5: Console.Write("arcocoseno \""); break;
          case 6: Console.Write("arcotangente \""); break;
          case 7: Console.Write("log \""); break;
          case 8: Console.Write("ceil \""); break;
          case 9: Console.Write("exp \""); break;
          case 10: Console.Write("sqrt \""); break;
        }
      }
      else
        Console.Write("\"[" + ID + "] " + Numero + "\"");
    }
  }
}
```

```
using System.Globalization;

/* Evaluador de expresiones usando un árbol binario */
namespace Ejemplo {
  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("}");
    }

    /* Evalía 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/007.zip

```
namespace Ejemplo {
//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:

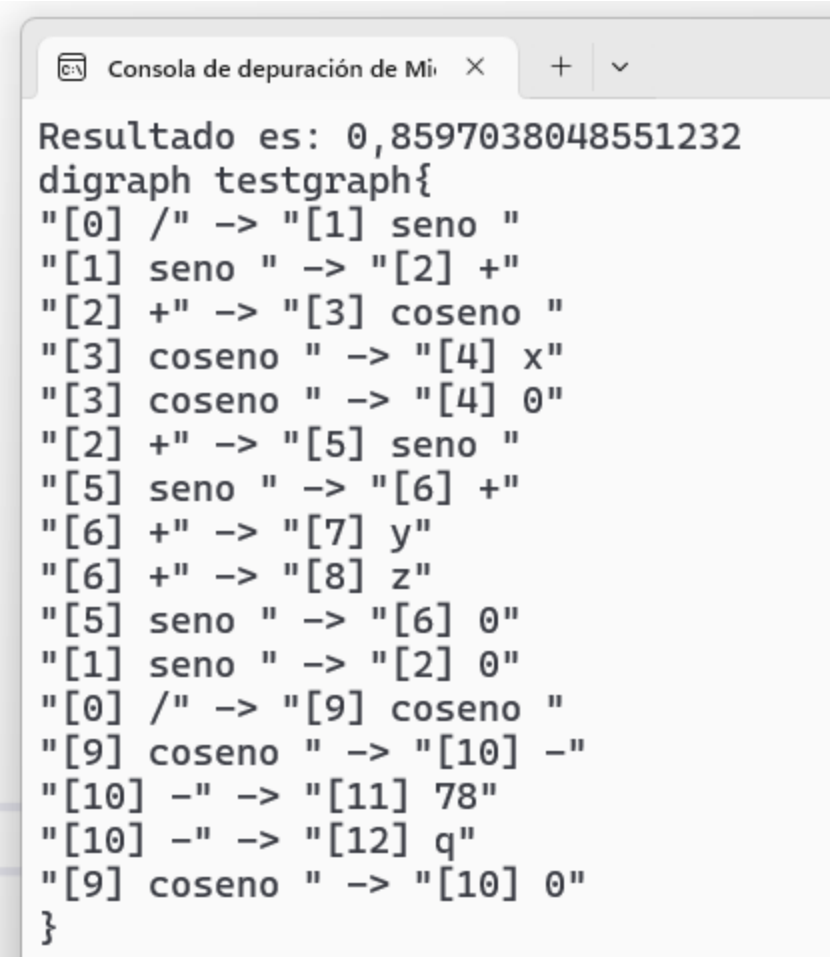


Ilustración 22: Ejemplo de ejecución

Fase 8. Evaluación de sintaxis y optimización

Finalmente se le agrega la evaluación de sintaxis que es muy similar a la del evaluador 4.1. Se retira el atributo ID, porque no es necesario dibujar el árbol binario (además le resta velocidad).

H/Árbol/008.zip

```
namespace Ejemplo {
  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/008.zip

```
using System.Globalization;

namespace Ejemplo {
  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;
    }

    /* Evalía 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 < sbSintax.Length; 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;
    }

    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/008.zip

```
namespace Ejemplo {
  //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));
      }
    }
  }
}
```

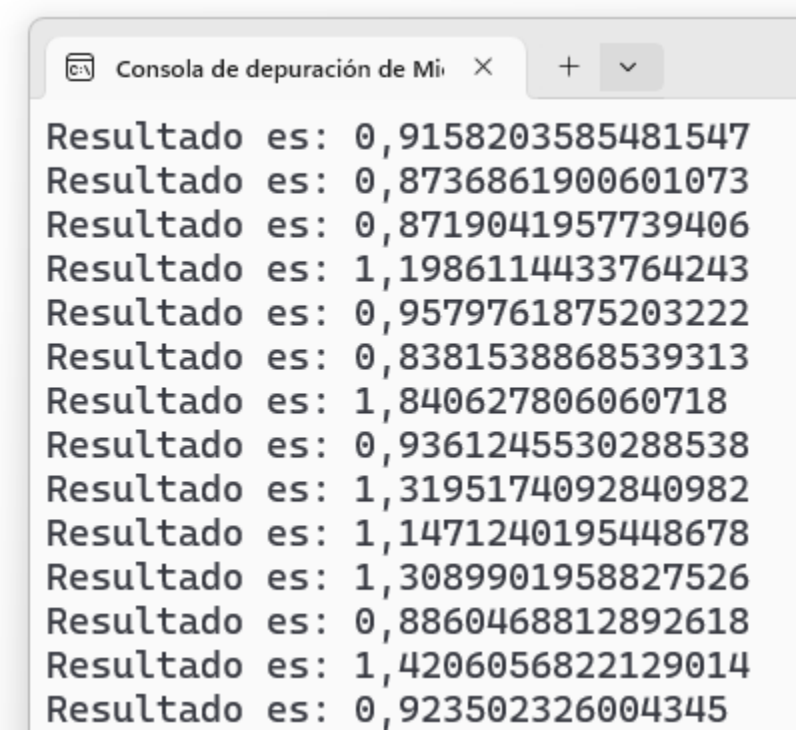


Ilustración 23: Ejemplo de ejecución

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 sí 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/Comparativa 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 4.1
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/Comparativa.zip

```
using System.Diagnostics;

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

    const int ANCHO_ANALISIS = 10; // ancho de la columna "Análisis"
    const int ANCHO_EVALUA = 10; // ancho de la columna "Evalúa"

    static void Main(string[] args) {
      long TAnalisisEval4_1 = 0, TEvaluaEval4_1 = 0;
      long TAnalisisEval4 = 0, TEvaluaEval4 = 0;
      long TAnalisisArbol = 0, TEvaluaArbol = 0;

      EvalArbolBin arbol = new();
      Evaluador4 eval4 = new();
      Evaluador4_1 eval4_1 = new();
      EcuacionAzar ecu = new();

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

      //Parámetros
      int TotalEcuaciones = 700;
      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;

      //Analiza la expresión con el evaluador 4.1
      temporizador.Reset();
      temporizador.Start();
      eval4_1.Analizar(Ecuacion);
      temporizador.Stop();
      TAnalisisEval4_1 += 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;

       //Evalúa la expresión con el evaluador 4.0
       temporizador.Reset();
       temporizador.Start();
       double ResEval4_1 = 0;
       for (int xvalor = 0; xvalor < valX.Length; xvalor++) {
        eval4_1.DarValorVariable('x', valX[xvalor]);
        ResEval4_1 += eval4_1.Evaluar();
       }
       temporizador.Stop();
       TEvaluaEval4_1 += temporizador.ElapsedTicks;

       //Chequea si hay una diferencia entre ambos evaluadores
       if (Math.Abs(ResArbol - ResEval4) > 0.01 || Math.Abs(ResArbol -
ResEval4_1) > 0.01) {
        Console.WriteLine(Ecuacion);
        Console.WriteLine("Arbol:   " + ResArbol);
        Console.WriteLine("Eval4:   " + ResEval4);
        Console.WriteLine("Eval4_1: " + ResEval4_1);
        break;
       }
     }

     Console.WriteLine("Tamaño ecuación: " + Tamano);
     Console.WriteLine("Total ecuaciones: " + TotalEcuaciones);
     Console.WriteLine("Veces evalúa: " + VecesEvalua);
     Console.WriteLine("\r\n[Eval4_1  ] Análisis: {0," + ANCHO_ANALISIS +
"} Evalúa: {1," + ANCHO_EVALUA + "}",
       TAnalisisEval4_1, TEvaluaEval4_1);
     Console.WriteLine("\r\n[Eval4 ] Análisis: {0," + ANCHO_ANALISIS + "}
  Evalúa: {1," + ANCHO_EVALUA + "}",
       TAnalisisEval4, TEvaluaEval4);
     Console.WriteLine("\r\n[Árbol ] Análisis: {0," + ANCHO_ANALISIS + "}
  Evalúa: {1," + ANCHO_EVALUA + "}",
       TAnalisisArbol, TEvaluaArbol);
     Console.WriteLine("FIN");
    }
  }
}
```

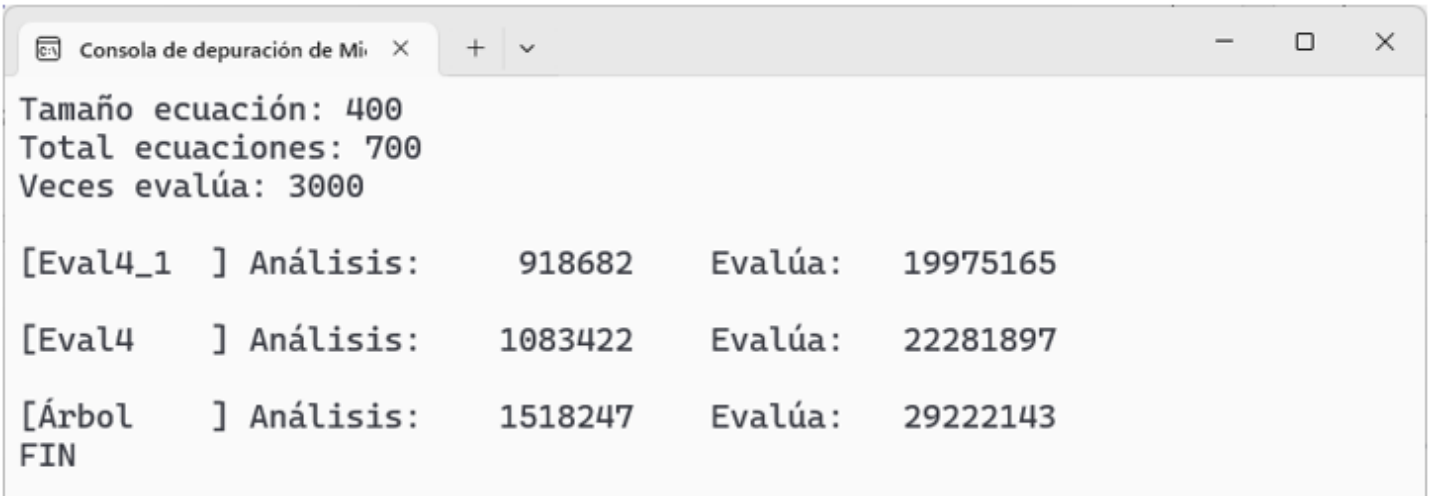


Ilustración 24: Comparativa de desempeño

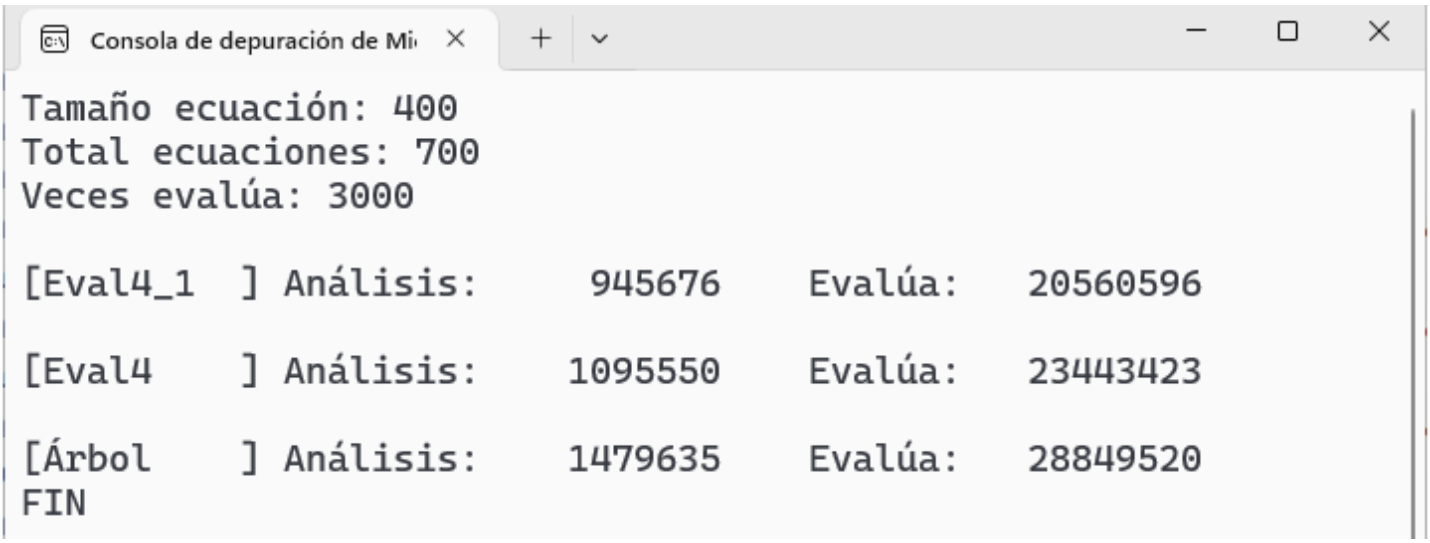


Ilustración 25: Comparativa de desempeño

El evaluador más rápido en la fase de análisis y evaluación es el 4.1.