# Intérprete de Expresiones Matemáticas Integrable en el Software Evolución y en otras Herramientas Software

Lewing Andres Mendez Ortiz

Trabajo de Grado para Optar al Titulo de Ingeniero de Sistemas

Director

Hugo Hernando Andrade Sosa

Magíster en Informática

Codirector

Emiliano de Jesus Lince Mercado

Magíster en Informática

Universidad Industrial de Santander

Facultad de Ingeniería Fisicomecánicas

Escuela de Ingeniería de Sistemas e Informática

Ingeniería de Sistemas

Bucaramanga

2026

## Tabla de Contenido

**Pág.**

## Lista de Tablas

**Pág.**

## Lista de Figuras

**Pág.**

## Lista de Apéndices

**Pág.**

## Resumen

**Título:** Intérprete de Expresiones Matemáticas Integrable en el Software Evolución y en otras Herramientas Software.

**Autor:** Lewing Andres Mendez Ortiz.

**Palabras Clave:** Interprete, Expresiones Matemáticas, Evaluador, Componente de Código, Delphi, Integrable, Metodología Basada en Componentes.

**Descripción:** Evolución es un software de modelado y simulación basado en dinámica de sistemas, que permite construir modelos empleando expresiones matemáticas definidas por sus usuarios, cada expresión representa características observables y medibles, lo que posibilita generar simulaciones orientadas al estudio y comprensión de distintos fenómenos o sistemas, lo que promueve compartir, crear y fortalecer el conocimiento.

En los últimos años, el software ha presentado fallas de compatibilidad con Windows 11, asociadas al proceso de compilación de expresiones matemáticas, dicho proceso está estrechamente vinculado con la simulación de los modelos, por lo que se identificó la necesidad de diseñar un componente capaz de sustituir al compilador; de esta problemática surge el presente proyecto, cuyo propósito es diseñar e implementar un intérprete de expresiones matemáticas.

Un intérprete permite ejecutar en tiempo real instrucciones previamente programadas, basado en esta característica, se construye una herramienta de software que puede integrarse en Evolución y en otros desarrollos similares, ofreciendo a los usuarios la posibilidad de interpretar expresiones matemáticas y generar simulaciones a partir de sus modelos. Para alcanzar este objetivo se empleó la metodología de desarrollo en espiral y una arquitectura modular, garantizando flexibilidad y escalabilidad en la solución de software.

---

* Trabajo de Grado

** Facultad de Ingeniería Fisicomecánicas. Escuela de Ingeniería de Sistemas e Informática. Director: Hugo Hernando Andrade Sosa. Magíster en Informática. Codirector: Emiliano de Jesus Lince Mercado. Magíster en Informática.

## Abstract

**Title:** Interpreter of Mathematical Expressions Integrable into Evolution Software and Other Software Tools.

**Author:** Lewing Andrés Méndez Ortiz.

**Keywords:** Interpreter, Mathematical Expressions, Evaluator, Code Component, Delphi, Integrable, Component-Based Methodology.

**Description:** Evolution is a modeling and simulation software based on system dynamics that enables the construction of models through mathematical expressions defined by users. Each expression represents observable and measurable characteristics, which makes it possible to generate simulations oriented toward the study and understanding of different phenomena or systems. In this way, the software contributes to the sharing, creation, and strengthening of knowledge in various application domains.

In recent years, the software has exhibited compatibility issues with Windows 11, particularly associated with the compilation process of mathematical expressions. This process is closely linked to the execution and simulation of models, making it a critical component of the system's functionality. Consequently, the need was identified to design a new component capable of replacing the existing compiler and ensuring proper operation under current technological environments. From this problem, the present project emerged, whose main objective is to design and implement a mathematical expression interpreter.

An interpreter enables the real-time execution of previously defined instructions. Based on this capability, a software tool was developed that can be integrated into Evolution as well as other similar applications. This tool allows users to interpret mathematical expressions and generate simulations derived from their models. To achieve this objective, the spiral development methodology and a modular software architecture were employed, ensuring flexibility, maintainability, and scalability in the proposed solution.


---

* Degree Work

** Faculty of Physical-Mechanical Engineering. School of Systems and Computer Engineering. Director: Hugo Hernando Andrade Sosa. Master of Science in Computer Science. Co-director: Emiliano de Jesus Lince Mercado. Master of Science in Computer Science.

# 1. Objetivos

Este apartado presenta los objetivos planteados para el desarrollo del presente trabajo de grado.

## 1.1. Objetivo General

Desarrollar un componente de software en el lenguaje de programación Delphi, empleando el paradigma modular como arquitectura, con el fin de integrar el componente de código dentro del software Evolución, y colocarlo a disposición de otros desarrollos de software que requieran interpretar expresiones matemáticas que contengan operadores aritméticos, comparativos y funciones.

## 1.2. Objetivos Específicos

1. Estudiar y definir los requerimientos que debe tener un interprete de expresiones matemáticas integrable en desarrollos de software.

2. Diseñar un componente de software que interprete expresiones matemáticas, integrable en el software Evolución.

3. Implementar el componente de software con una interfaz que permita su integración en desarrollos de software.

4. Verificar que el componente desarrollado cumple con los requisitos funcionales y no funcionales; mediante pruebas unitarias, de integración del sistema, de aceptación, de rendimiento y de carga.

# 2. Planteamiento y Justificación

En el grupo de investigación SIMON, perteneciente a la Escuela de Ingeniería de Sistemas de la Universidad Industrial de Santander (UIS) y bajo la dirección del Profesor

Hugo Andrade Sosa, se ha desarrollado durante los últimos treinta años el software Evolución, una herramienta que permite aplicar la dinámica de sistemas (DS) y la sistémica en distintos niveles de complejidad; a lo largo de este tiempo ha permitido a sus usuarios crear modelos con simulaciones interactivas asociadas a diversas áreas del conocimiento.

Actualmente se han presentado problemas intermitentes en el uso del software Evolución, debido a incompatibilidades que se presenta en la definición de las expresiones matemáticas contenidas en los elementos del modelo, las cuales son indispensables en el proceso de simulación; la problemática puede estar relacionada al componente encargado de compilar las expresiones, este presenta las siguiente condiciones desfavorables:

1. Ausencia de soporte y actualizaciones.
2. Código fuente escrito en Ensamblador para las arquitecturas 32 y 16 bits.
3. Cuenta con algunas funciones en ensamblador que cumplieron su vida útil.

De esta manera se plantea el actual proyecto de pregrado, donde se espera poder desarrollar una herramienta de código que aborde las siguientes tareas:

1. Manejo de operadores: aritméticos, comparativos, lógicos y funciones matemáticas.
2. Almacenamiento de las expresiones, variables y constantes.
3. La herramienta deberá poder ser integrada en Evolución y otros software.

Lo anterior plantea un proceso de ingeniería que busca resolver una necesidad, ademas de desarrollar una herramienta que pueda integrarse en Evolución y por ende, en otros desarrollos de software que requieran evaluar y almacenar elementos matemáticos.

## 3. Marco de Referencia

Este apartado presenta los fundamentos que respaldan el desarrollo del componente de software propuesto, para este propósito se incluye una revisión y contextualización de

los antecedentes, conceptos clave, de las ideas principales relacionadas con la evaluación de expresiones matemáticas y del software Evolución, así como de otros aspectos técnicos.

## 3.1. Antecedentes

A continuación se dará el contexto del software Evolución, junto a la información que motivo el desarrollo del presente proyecto de grado.

### *3.1.1. Estructura General de Evolución*

Evolución se ha desarrollado a lo largo de los últimos 30 años dentro del grupo de investigación SIMON, mediante la modalidad de proyecto de grado, siendo el resultado de alrededor de diez proyectos de grado secuenciales. En este proceso, el Docente Hugo Andrade Sosa ha cumplido el papel de director de proyecto, estando presente en cada etapa del desarrollo de Evolución, junto a él, destaca el papel del profesor Emiliano Lince, quien desarrolla una versión estable y completa de Evolución, colabora activamente en el desarrollo de versiones posteriores y se hace cargo del mantenimiento del código hasta la actualidad. Evolución es el resultado de la colaboración de varios estudiantes pertenecientes a la Escuela de Ingeniería de Sistemas en la Universidad Industrial de Santander junto a docentes adscritos al grupo de investigación.

En sus inicios Evolución fue concebido por el profesor Hugo Andrade Sosa como una calculadora que permitía resolver ecuaciones de dinámica de sistemas, inspirándose en la metodología planteada en universidades de Estados Unidos y Europa, la cual, apenas comenzaba a desarrollarse; como referencia temporal, Stella sale al mercado oficialmente en 1985 (Isee Systems, Inc., s.f.), siendo el primer software comercial de modelado de sistemas con una interfaz grafica y Evolución es lanzando en su versión mas completa la 3.5 al rededor del año 2004.

Los software de DS aparecen como una necesidad de aplicar y probar el paradigma sistémico aparir del cual, surge la metodología del modelado basada en DS, con el objetivo

de poder crear, construir y compartir conocimiento, en base a modelos explicativos (modelos DS) siendo este el punto de partida al que se orienta Evolución; en el transcurso de 20 años se desarrollan un conjunto de proyectos como anteriormente se menciona, tanto como proyectos enfocados en extender las capacidades del software, así como, de sus posibles usos y aplicaciones que impacten en la sociedad contemporánea.

En el transcurso de los años se abordaron diferentes perspectivas de aquello que se quería lograr con cada versión de Evolución, lo cual se puede distribuir en tres secciones:

1. **Capacidades del software:** funciones básicas de simulación, fuzzy (lógica difusa), funciones complejas de simulación, vectores.

2. **Interfaz de usuario:** interfaz de modelado, de animadores y funcionalidades, modelado de diagrama de influencias, tablas y representaciones.

3. **Otras:** mejoras en el código, correcciones y pruebas o experimentos.

### *3.1.2. Restricciones Estructurales del Software*

El software al que se dirige este proyecto presenta una baja modularidad y un fuerte acoplamiento en su apartado central encargado de realizar evaluaciones matemáticas, lo cual responde a decisiones de diseño adoptadas en el año 2004 cuando las limitaciones de hardware y al componente de codigo encargado de compilar las expresiones matemáticas, en especial la reducida capacidad de memoria RAM que obligaba a implementar soluciones optimizadas que priorizaran el uso eficiente de los recursos.

En aquel contexto el sistema debía ejecutarse con un consumo cercano a tres megabytes de memoria y se optó por integrar procesos fuertemente acoplados para garantizar el rendimiento, lo que resultaba adecuado en ese momento pero hoy se ha convertido en un obstáculo para la evolución del software.

La metodología implementada ya no es necesaria en la actualidad y restringe la integración de nuevos componentes como el que se propone en este trabajo de grado, lo que hace

evidente la necesidad de emprender un proceso de reingeniería orientado a la modularización completa del sistema con el fin de mejorar su mantenibilidad, escalabilidad y capacidad de integración de nuevas capacidades.

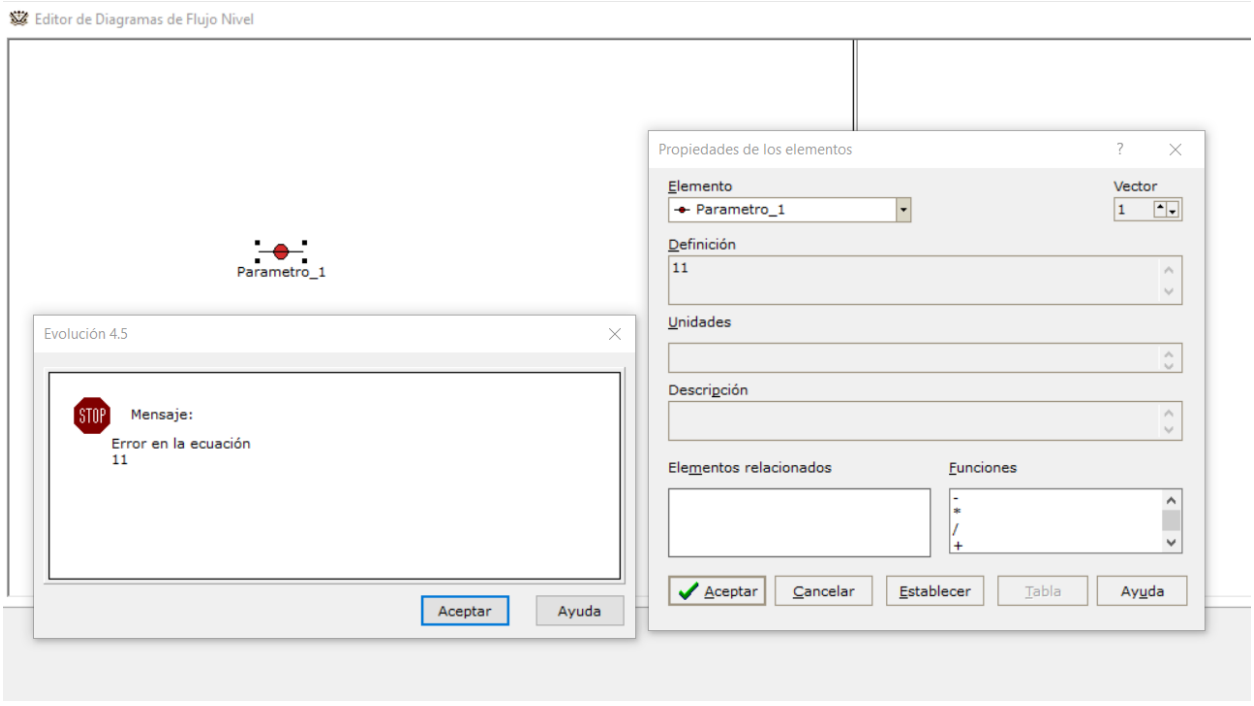
La solución planteada en el proyecto actual constituye un avance parcial que no puede integrarse plenamente en la arquitectura existente, aunque se contempla un proyecto complementario en el que se abordará la reestructuración general del sistema de manera que los desarrollos aquí presentados puedan integrarse de forma efectiva en una plataforma modular y sostenible en el tiempo.

### *3.1.3. Depuración del Código de Evolución*

Cuando se ejecuta Evolución en Windows 11 los usuarios se encuentran el siguiente error, el cual no permite utilizar de ninguna manera el software:

**Figura 1**

*Error al cargar expresión matemática*



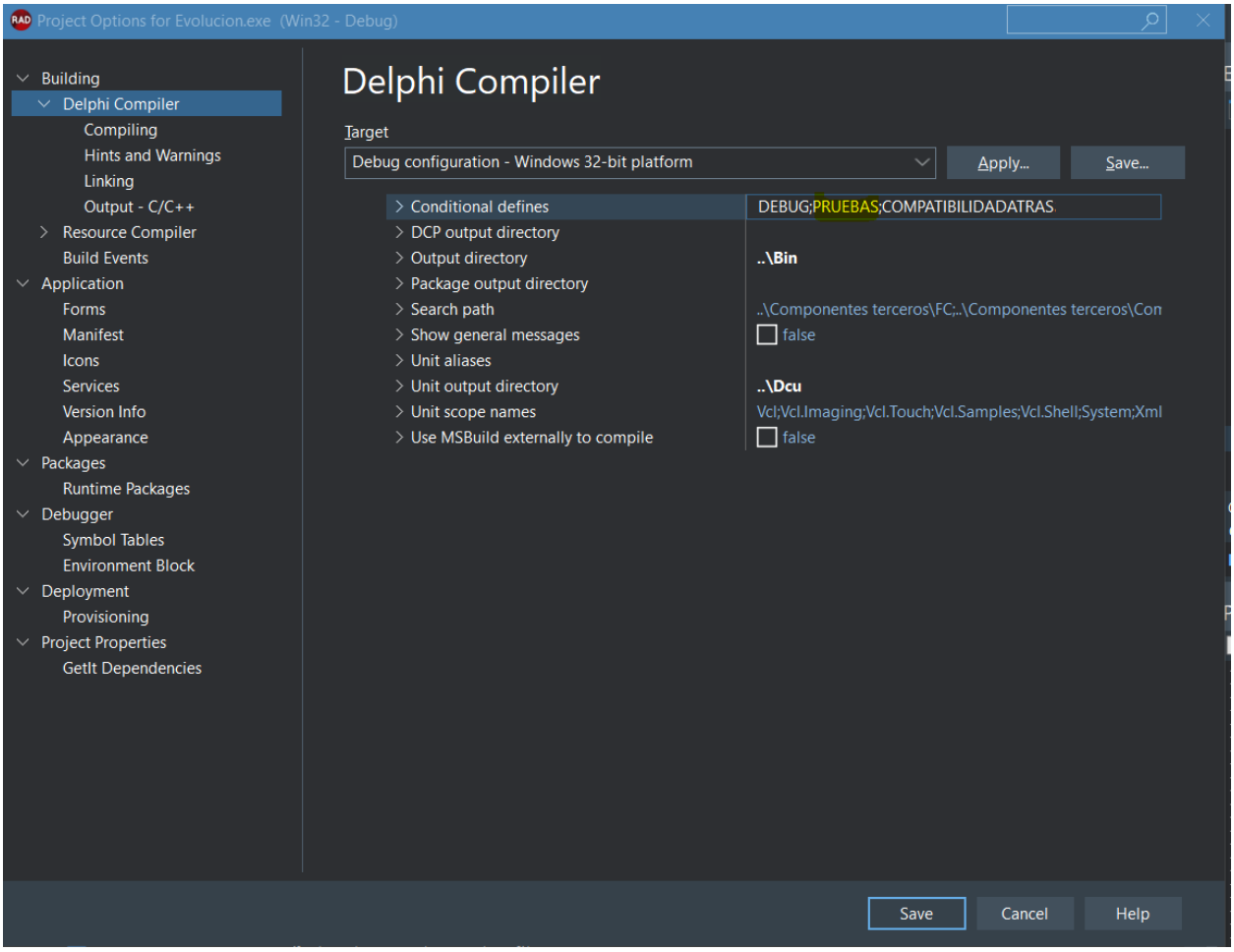

Este error no permite que ningún modelo se ejecute, ya que no es posible la asignación

de valores a los parámetros, niveles y en general al emplear cualquier elemento del modelo flujo - nivel. En base a este caso de error, se realizara un proceso de depuración del código, con el objetivo de encontrar el o los fragmentos de código responsables del fallo.

Para realizar las pruebas se debe activar la directiva *"PRUEBAS"* lo que facilita el proceso de depuración de errores, como se presenta en la siguiente imagen:

**Figura 2**

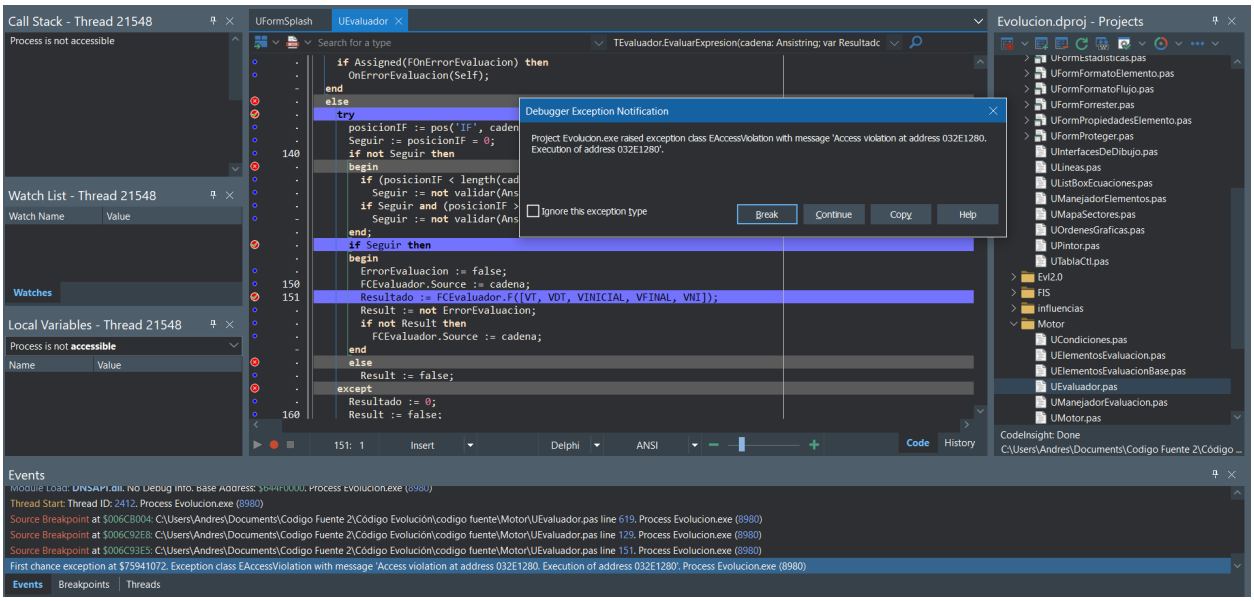
*Definición de directivas del IDE*




Una vez activada esta directiva, se deberá correr el programa en modo depuración, con el objetivo de realizar un seguimiento del proceso interno que el software lleva acabo. Al empezar a ejecutar el software en modo depuración, se encuentra el siguiente error:

**Figura 3**

*Depuración: Captura del error durante el proceso de depuración*




El error indica que no se puede acceder a un espacio en memoria, lo que apunta a los cambios que ha sufrido el compilador de Delphi en el manejo de punteros (manejo de memoria). El error aparece al evaluar la siguiente línea de código:

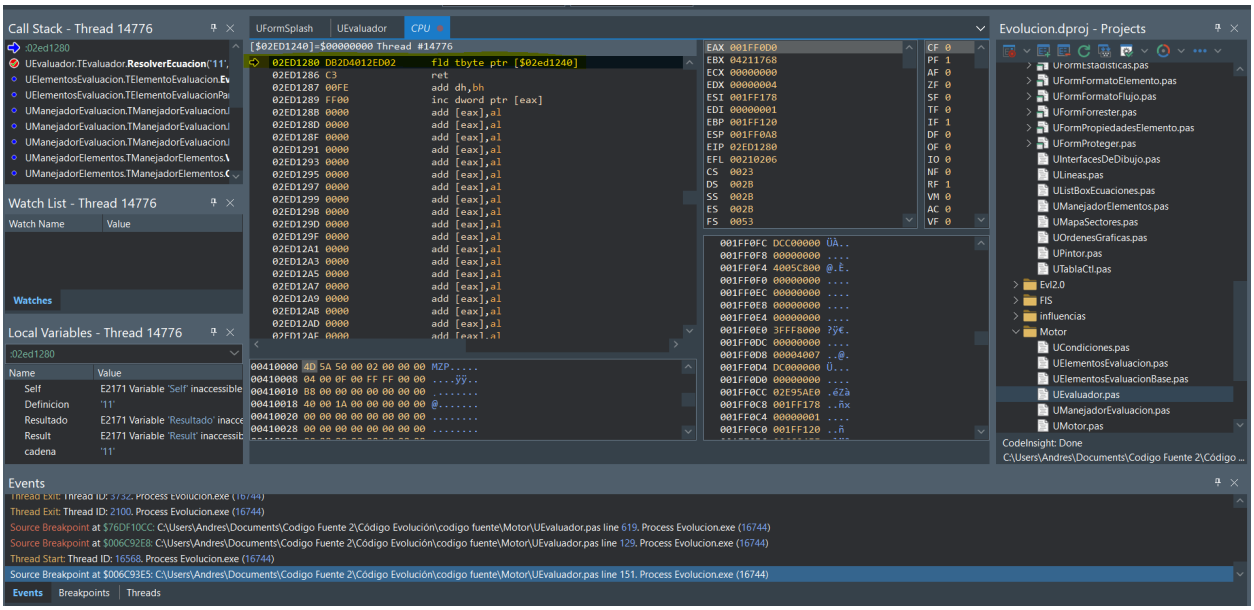
**Código 1.** *Línea responsable del error*

```
Resultado := FCEvaluador.F([VT, VDT, VINICIAL, VFINAL, VNI]);
```

A continuación se realiza el seguimiento del error en los puntos de quiebre del código, donde se puede observar la reserva de memoria y posterior intento de acceso o uso, del espacio en memoria.

**Figura 4**

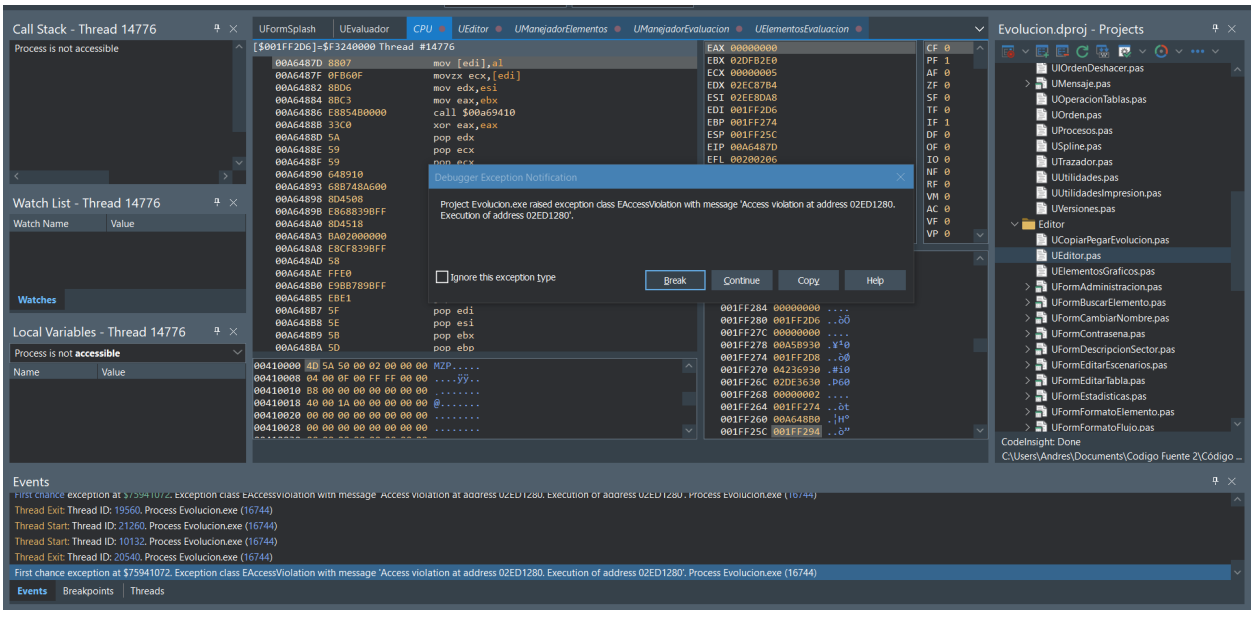
*Depuración: Espacio en memoria reservada y asignada*




Posteriormente a la reserva y asignación de memoria, no se puede acceder desde otras funciones que utilizan lo reservado en memoria:

**Figura 5**

*Depuración: Acceso invalido a espacio en memoria*



En base al anterior proceso descrito, se concluye que el error radica en como maneja la memoria el compilador de formulas, cuyas funciones se están viendo afectas de alguna manera por las actualizaciones mas recientes de Windows, esto podría empeora o solucionarse en un futuro, sin embargo, no es recomendable depender de código sin soporte por parte de sus autores y ademas, tan antiguo; cabe resaltar que no se cuenta con documentación de ningún tipo y el componente llega a emplear código ensamblador de alta complejidad (uso de direcciones especificas) con el objetivo de ser eficiente.

### *3.1.4. Causas del Error en el Código de Evolución*

El software Evolución presenta un error que afecta su capacidad de operar correctamente sobre el SO Windows 11 (sobre algunas actualizaciones del sistema operativo, se desconoce cuales), al realizar un proceso de investigación y depuración, se concluye que el componente encargado de compilar las expresiones matemáticas, emplea código ensamblador donde se utilizan funciones de manejo de memoria obsoletas o en desuso (Código fuente desarrollado en 1998), las cuales serian las responsables del error fatal encontrado. Como soporte a esto, se explicaran los siguientes dos puntos:

1. Implementación de nuevas tecnologías de protección en el acceso a memoria RAM por parte de Microsoft.

2. Actualizaciones y modernización del compilador de Delphi, dentro de Rad Studio con el objetivo de contar con software multiplataforma basado en el mismo código.

En el primer punto tenemos las siguientes tecnologías que pueden estar relacionadas con los fallos presentados en el funcionamiento:

- Prevención de Ejecución de Datos (DEP): La Prevención de ejecución de datos (DEP) es una tecnología integrada en Windows que ayuda a proteger del inicio de código ejecutable desde lugares donde no se supone que sea necesario. DEP lo hace al marcar

algunas áreas de la memoria del equipo como solo para datos, no se permitirá que se ejecute ningún código ejecutable o aplicaciones desde esas áreas de memoria. Esto está diseñado para dificultar que los ataques que intentan usar desbordamientos de búfer u otras técnicas que ejecuten su malware desde aquellas partes de la memoria que normalmente solo contendrán datos (Microsoft Support, s.f.).

- Aleatorización del Diseño del Espacio de Direcciones (ASLR): El principio denominado ASLR (Address Space Layout Randomization en inglés) es un mecanismo de seguridad digital empleado para obstaculizar la efectividad de amenazas de software maligno. Este principio centra su enfoque en la variación arbitraria en la distribución del espacio de dirección de un procedimiento, generando así dificultades para prever la posición de las áreas de memoria por aquellos que intenten un ataque (Wallarm Learning Center, 2025).

En el segundo punto encontramos que se realizan diferentes cambios y generalizaciones para crear un proceso único por el cual, se puede desarrollar aplicaciones multiplataforma basado en el mismo proyecto, lo que plantea agilizar en gran medida el desarrollo de este tipo de aplicaciones, esto tiene tres implicaciones importantes (Embarcadero Technologies, 2025):

1. Cambios en los *Strings*, ahora los *Strings* por defecto son Unicode *Strings*.
2. Cambio en la estructura interna de los *Strings*, enfocándose en los Unicode y en los *AnsiStrings*.
3. El manejo de punteros cambia, para adaptarse a los nuevos modelos de manejo de memoria implementados por los sistemas operativos. Esta actualización plantea problemas en cuanto al manejo de punteros que se realizaba en versiones pasadas, ya que estas contaban con diferentes vulnerabilidades.

Sumado a esto Embarcadero deja a disposición de sus usuarios un manual de casos de migración de proyecto, donde se dan algunos consejos en proyectos que requieran de un proceso de recompilación empleando compiladores modernos, como los ofrecidos en la versión de Rad Studio 12 Athens (Cantù, 2008).

De acuerdo a lo anteriormente expuesto, se concluye que el código ha quedado obsoleto debido a cambios importantes en las ultimas versiones del SO Windows y por consiguiente, a los cambios que ha sufrido el compilador de Delphi que buscan mantener sus herramientas de desarrollo a la vanguardia tecnológica, ofreciendo un producto seguro y de calidad.

## 3.2. Fundamentos Teóricos

En esta sección se presentan las bases conceptuales que dan sentido al proyecto, para este propósito se hablará del modelado de DS mediante Evolución, se expondrán las herramientas y tecnologías que se plantean utilizar para el desarrollo de la solución descrita en el actual proyecto.

### *3.2.1. Evolución, Modelado de Dinámica de Sistemas*

La dinámica de sistemas que emplea EVOLUCIÓN se ha desarrollado como producto del impacto de la sistémica, adquiriendo gran importancia en lo académico y en general dentro de cualquier sector de la industria que quiera modelar sus procesos. El conocimiento que surge al intentar modelar un fenómeno es tan amplio y preciso como se requiera, logrando contener todos los aspectos diferentes que un individuo pueda observar del fenómeno, aunque de ser necesario, ciertos aspectos podrían ignorarse buscando una representación más sencilla (Hitchins, s.f.).

El modelado de sistemas es una tarea importante en la labor de crear, construir y compartir de forma transparente los conocimientos que vamos adquiriendo mediante el estudio, lo que se traduce en la capacidad de almacenar conocimiento de manera que otros lo pueda entender mediante su observación y estudio, algo que los software de modelado de

DS permiten que sea sencillo y práctico. Los sectores industriales a mediana y gran escala son muy complejos, conllevando demasiados procesos que ocurren simultáneamente, además de que capacitar nuevos empleados para adaptarse a esta tarea no es sencillo, razón por la cual la dinámica de sistemas tuvo un gran impacto en el mundo como lo conocemos ya que facilita compartir el conocimiento (Hitchins, s.f.).

Los sistemas diseñados se han empleado para el desarrollo de herramientas que interactúen con operarios o usuarios y les permitan gestionar un proceso tanto en su totalidad, como de forma parcial o específica. Esto dio como resultado que se implementaran sistemas de gestión y control, servicios de emergencia, gestión del tráfico aéreo, etc. Procesos que no se podrían diseñar de forma sencilla y elegante sin los software de modelado y simulación de DS (Hitchins, s.f.).

Con el propósito de diseñar y simular modelos de dinámica de sistemas, EVOLUCIÓN cuenta con las siguientes tres herramientas:

1. **Diagrama de flujo y nivel:** Les permite a los usuarios representar relaciones de material, donde exista como mínimo un nivel y un flujo asociados. Los modelos pueden ser tan complejos como el problema lo demande, ya que el software cuenta con una variedad de herramientas que permite generar simulaciones más allá de flujos de material, como lo son variables exógenas, retardos, lógica difusa, entre otras.

2. **Diagrama de influencias:** El objetivo de esta herramienta con la que cuenta Evolución es permitirle al usuario observar un modelo donde puedan representar cómo los elementos interactúan entre sí, para este propósito, se cuenta con “Elementos”, “Relaciones de Información” y “Relaciones de Material”.

3. **Animadores (Visor de resultados):** Los Animadores cumplen la función de un visor de simulaciones contemplando el análisis por escenarios. Adicionalmente, en el software se implementa la idea de interactuar con el modelo o con la simulación, por lo cual, los usuarios pueden cambiar ciertos tipos de valores de la simulación mientras esta se

desarrolla. Además, se cuenta con herramientas visuales que facilitan esto, permitiendo observar con mayor detalle el comportamiento del modelo y su respuesta ante distintas perturbaciones del sistema.

### *3.2.2. Delphi*

Delphi es un entorno integrado de desarrollo (IDE) que utiliza una versión moderna de Turbo Pascal, siendo el lenguaje empleado en la construcción de Evolución. Desarrollado por Embarcadero, Delphi forma parte de un conjunto de soluciones para el desarrollo de software multiplataforma con alto nivel de seguridad. Además de su propio lenguaje, Delphi soporta otros como C++ y ofrece productos orientados a diversas áreas de programación, distribuidos bajo una licencia privativa.

Entre las características más destacadas de Delphi se encuentra el hecho de ser un lenguaje compilado, lo que permite un control preciso sobre el consumo de recursos y el manejo de procesos, garantizando así eficiencia y estabilidad en las plataformas seleccionadas. Asimismo, Delphi proporciona un alto nivel de seguridad en los desarrollos, uno de los objetivos principales de Embarcadero, y se caracteriza por su estabilidad y mantenimiento continuo. Aunque su uso no es tan extendido en el ámbito general, grandes corporaciones lo emplean, lo que asegura actualizaciones de seguridad, correcciones de errores y el lanzamiento de nuevas versiones, convirtiéndolo en una opción confiable para proyectos a largo plazo y para la escalabilidad de aplicaciones (Embarcadero Technologies, s.f.).

### *3.2.3. Arquitectura Basada en Componentes*

Una arquitectura basada en componentes ofrece múltiples ventajas, además de ser la arquitectura empleada para la construcción de Evolución, esta se caracteriza por dividir el diseño en componentes independientes y con muy pocas dependencias, los cuales realizan un proceso de trabajo focalizado a costa de la definición de un contrato de uso (Interface) el cual lo hace apto para la reutilización ya sea dentro de un software en específico o en otros

para los cuales, quizás no esté pensando.

Se puede definir un componente como: “Un componente es una unidad de composición de aplicaciones software, que posee un conjunto de interfaces y un conjunto de requisito, y que ha de poder ser desarrollado, adquirido, incorporado al sistema y compuesto con otros componentes de forma independiente, en tiempo y espacio.” (Morales, 2011)

La arquitectura basada en componentes permite la construcción de software reutilizable e integrable en software de mayor complejidad y tamaño.

### *3.2.4. Integrabilidad*

La arquitectura y metodología basada en componentes son estrategias adoptadas con el fin agilizar el proceso de desarrollo, esto mediante la reutilización de componentes de código de propósito específico, esta idea, noción o principio se emplea en lenguajes de programación como Delphi, Java y frameworks como .NET. Entornos de desarrollo como RAD Studio buscan e incentivan el desarrollo ágil de aplicaciones (Ball, 2020), ya que de esta manera se crea un mercado de componentes de código (gratuitos y de pago) que pueden ser empleados en multitud de desarrollo de software (Technologies, 2025), fortaleciendo de esta manera el concepto de desarrollo ágil (DevOps). Algunos ejemplos de componentes de código desarrollados en Delphi son:

- **TeeChart**: Biblioteca de gráficos y visualización de datos; permite crear diagramas de barras, líneas, pastel, mapas y 3D.
- **SDL (Simple DirectMedia Layer)**: Librería multiplataforma para multimedia; facilita trabajar con gráficos, sonido y entradas de usuario. Muy usada en videojuegos y simulaciones.
- **Combos**: Conjunto de componentes visuales para listas desplegables (ComboBox) que mejoran la selección de opciones en la interfaz.

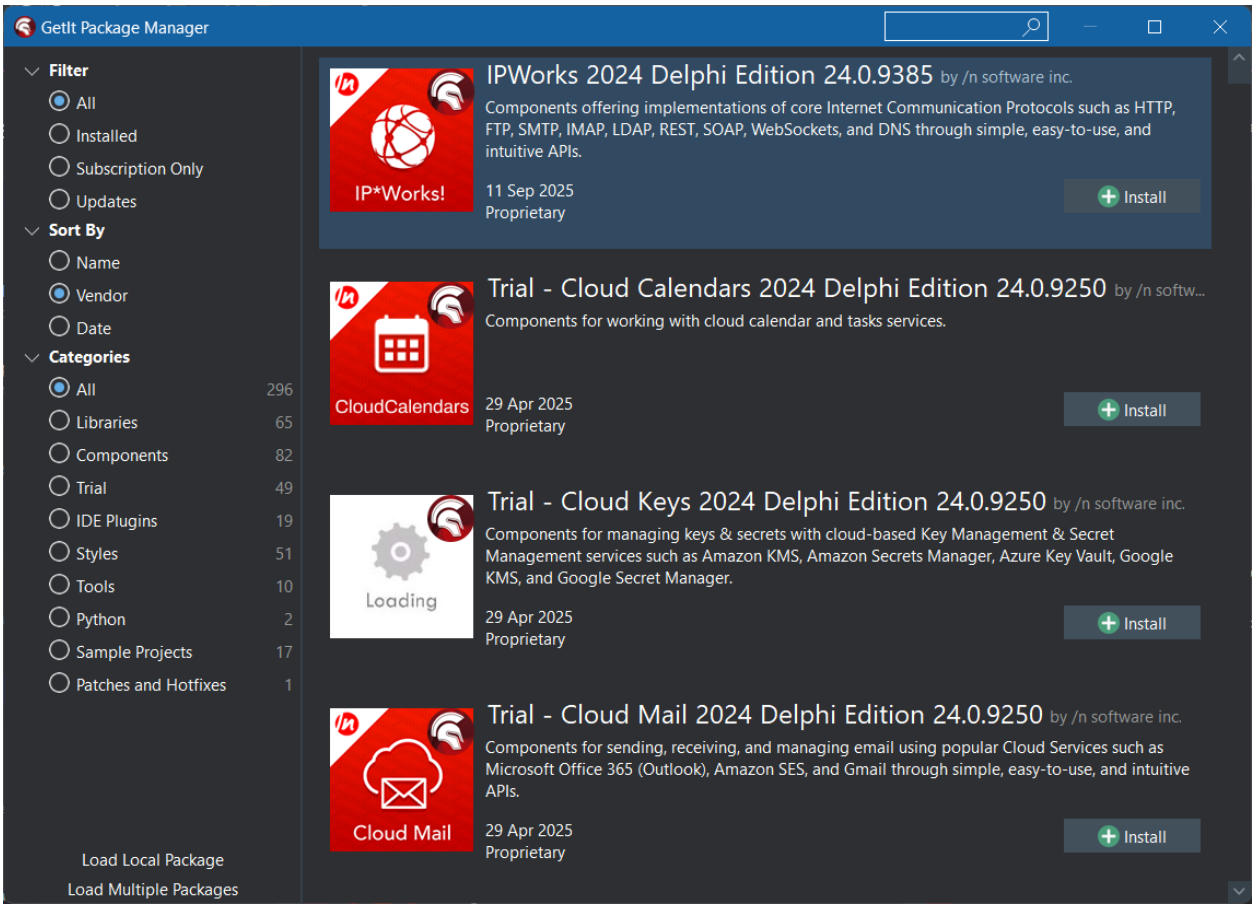
- **Abbrevia**: Librería para compresión y descompresión de archivos (ZIP, TAR, CAB, etc.), integrando empaquetado en las aplicaciones.
- **FlatStyle**: Paquete de componentes visuales con estilo plano y minimalista que moderniza la apariencia de botones, menús y formularios.
- **RegExpr**: Componente para trabajar con expresiones regulares, útil en búsquedas y validación de patrones de texto.
- **SpinEdit**: Control de entrada numérica con botones de incremento/decremento, usado para valores dentro de un rango.
- **Tracker**: Componente para seguimiento de datos o eventos, como progreso de procesos, logging o monitoreo de variables.
- **ZIP**: Biblioteca para el manejo directo del formato ZIP, que permite comprimir, descomprimir y manipular archivos ZIP.

En este sentido, el proyecto propuesto se concibe como un posible componente integrable en el software Evolución y en otros desarrollos, lo que garantiza su reutilización, escalabilidad y aporte a comunidades de desarrollo académica, ajustándose a la arquitectura y propósito bajo el cual se diseña el software Evolución.

Para facilitar este propósito RAD Studio permite exportar los componentes de código bajo un formato especifico, el cual es denominado bpl, este tipo de archivos encapsula la lógica (mediante la creación de un archivo binario) permitiendo la integración en otros proyectos sin exponer la lógica interna (Technologies, 2015), la cual representa el valor del producto software. Ademas de esto cuenta con una tienda propia integrada en su IDE (RAD Studio) en la cual, podemos adquirir e integrar componente de manera segura y rápida:

**Figura 6**

*Tienda de componente integrada en RAD Studio*




### *3.2.5. Compiladores*

Un compilador es un programa informático que traduce el código fuente escrito en un lenguaje de programación de alto nivel a un lenguaje de bajo nivel, como el lenguaje máquina o un lenguaje intermedio. Este proceso se lleva a cabo en varias fases, entre las que destacan la tokenización, el análisis sintáctico y la generación de código, cada una de las cuales juega un papel fundamental en la correcta transformación y ejecución del programa (Nystrom, 2021).

1. **Tokenización:** También conocida como análisis léxico, es la primera etapa en el proceso de compilación. Durante esta fase, el compilador examina el código fuente y lo

divide en unidades básicas llamadas tokens. Cada token representa una secuencia de caracteres que tiene un significado específico en el lenguaje, como pueden ser palabras reservadas, identificadores, números, operadores y delimitadores. Este proceso simplifica el análisis posterior al convertir el código en una secuencia estructurada de elementos, facilitando la detección de errores y la aplicación de las reglas sintácticas (Nystrom, 2021).

2. **Gramáticas:** Las gramáticas son un conjunto de reglas formales que definen la estructura y la sintaxis de un lenguaje de programación. Estas reglas especifican cómo deben organizarse los tokens para formar sentencias y expresiones válidas. A diferencia de otros conceptos en la teoría de lenguajes formales, las gramáticas en este contexto se utilizan exclusivamente para describir las reglas de formación del lenguaje, sin hacer referencia directa a la teoría de autómatas. Esto permite que el análisis sintáctico se enfoque en verificar que el código fuente cumpla con la estructura esperada, facilitando la identificación de errores de sintaxis y garantizando la correcta interpretación del lenguaje (Nystrom, 2021).

3. **Árboles de Sintaxis:** Una vez que el código ha sido tokenizado y verificado conforme a las reglas de gramática, se procede al análisis sintáctico o parsing, durante el cual se construye un árbol de sintaxis. Este árbol, también conocido como árbol de derivación o árbol de análisis, es una representación jerárquica de la estructura del código fuente. En él, cada nodo interno representa una construcción sintáctica (como expresiones, declaraciones o bloques de código) y las hojas corresponden a los tokens individuales. El árbol de sintaxis facilita la comprensión de la estructura del programa y sirve como base para etapas posteriores, como el análisis semántico y la generación de código, permitiendo optimizar y transformar el código fuente en un formato que pueda ser ejecutado de manera eficiente (Nystrom, 2021).

### *3.2.6. Algoritmo Shuting Yard - Edsger Dijkstra*

Este algoritmo fue diseñado por el científico en computación Edsger Dijkstra (Casales, 2019), este algoritmo se inspira en el proceso de aparcar y desaparcar vehículos, en español se traduce como patio de maniobras. Fue diseñado para poder interpretar una expresión matemática simple por parte de la computadora, pero puede ser adaptada para operar con operaciones más complejas como funciones y lógica booleana (Wolf, s.f.).

Se ha modificado y ampliado su alcance y uso en el diseño y construcción de compiladores, ya que es un método eficiente para realizar diferentes tipos de operaciones en base a un tipado y proceso de tokenizacion. La estructura de este algoritmo se puede observar dentro de cualquier software al emplear herramientas de depuración que nos permita observar el código en ensamblador o maquina, donde las operaciones se organizan y manejan de forma semejante, esto debido a su orden de complejidad lineal.

Este algoritmo depende de la notación polaca reversa la cual transforma la notación que empleamos los humanos, notación infija, a notación posfija o prefija, esto ya que es una manera de ordenar de izquierda a derecha o viceversa la precedencia de las operaciones, lo cual transforma la cadena en elementos ordenados en fila esperando a ser atendidos (Rastogi, 2015).

### *3.2.7. Metodología de Desarrollo de Software en Espiral*

Para el proceso de desarrollo del proyecto se implementará la metodología en espiral, la cual presenta características interesantes ante el desafío (Pressman, 2010):

1. **Gestión de riesgos efectiva:** Se identifica y gestionan riesgos desde etapas tempranas, permitiendo mitigar problemas potenciales antes de que se conviertan en un gran obstáculo.

2. **Desarrollo iterativo:** Facilita la mejora continua del software mediante ciclos repetitivos, lo que permite incorporar retroalimentación y mejoras graduales.

3. **Flexibilidad y adaptabilidad:** Se adapta bien a proyectos grandes, complejos o con requisitos que pueden cambiar durante el desarrollo.

4. **Integración temprana de calidad:** Permite realizar pruebas y evaluaciones constantes, asegurando que la calidad del producto sea alta desde las primeras fases.

5. **Documentación constante:** La planificación y análisis de riesgos en cada iteración generan documentación útil y detallada.

6. **Retroalimentación continua:** Facilita la comunicación constante con el cliente, asegurando que el producto final cumpla con las expectativas y necesidades.

7. **Mejora progresiva:** Cada iteración agrega funcionalidades y mejoras, lo cual facilita detectar y corregir errores antes de que se vuelvan críticos.

## 4. Metodología

Para el desarrollo del proyecto bajo la metodología en espiral, se plantea una primera fase donde se desarrollará un marco general basado en:

1. Comunicación con el cliente acerca de los requerimientos y resultados esperados.

2. Análisis del problema.

3. Diseño de una arquitectura basada en componentes que sirva de exoesqueleto para la construcción del software mediante la metodología en espiral.

Posterior a esta etapa se continua con el proceso de desarrollo dando inicio a la segunda fase, la cual se desarrollará bajo la metodología en espiral, descrita a continuación, concluyendo con una ultima fase, en la que se redactara un informe final, en el que se describe todo el trabajo realizado durante el proceso de desarrollo. A continuación se describe el proceso planteado para el desarrollo de cada ciclo o iteración bajo la metodología en espiral:

### 4.1. Planeación

En la etapa de planeación se define tratar los siguientes puntos, con el objetivo de definir la hoja de ruta de la iteración ademas de evaluar diferentes aspectos del proceso:

1. **Estimación:** Consistirá en dos etapas.
    a. **Identificación de componentes:** Define el componente fundamental para la iteración basado en el exoesqueleto planteado en la etapa previa al inicio de la metodología en espiral. Se establecen sus responsabilidades, interfaces y relaciones.
    b. **Establecimiento de objetivos del ciclo:** Determina qué funcionalidades o mejoras se abordarán en la iteración actual, tanto a nivel de cada componente como del sistema en conjunto.
2. **Programación:** Se realiza un proceso de estudio en el alcance real que se puede obtenerse durante la iteración, con el objetivo de establecer un tiempo adecuado a la iteración.
3. **Análisis de riesgo:** Evalúa riesgos técnicos y de integración entre componentes, considerando dependencias, compatibilidad y posibles cuellos de botella.

### 4.2. Modelado

La etapa de modelado se centra en el análisis y diseño respectivo de los puntos que se esperan abordar, debe centrarse unicamente en lo definido en la etapa de planeación, los puntos a tratar son:

1. **Análisis:** Se realiza el estudio y análisis de los requerimientos de la iteración.
2. **Diseño:** Se realiza el diseño o rediseño de las partes que estarán involucradas en la iteración.

### 4.3. Construcción

La proceso de construcción llevado a cabo en cada iteración deberá apegarse a los anteriores puntos, cumpliendo con los siguientes dos elementos:

1. **Construcción de componentes individuales:** Desarrolla cada componente de forma iterativa. Crea prototipos de cada módulo para validar que cumplen su función de manera aislada.

2. **Pruebas unitarias y de integración:** A medida que se desarrollan los componentes, realizables pruebas independientes y posteriormente pruebas de integración en un entorno controlado.

### 4.4. Despliegue

En esta etapa se espera ir integrando los componentes o sus diferentes evoluciones con las que pueda contar cada uno de estos, ya sea por motivos de actualización, mejora o ampliación de las capacidades del componente, esto según se haya defino en la etapa de planeación en cada iteración. Se plantea cumplir con lo siguiente:

1. **Revisión del prototipo integrado**: Una vez integrados los componentes desarrollados en la iteración, valida el sistema global con usuarios o expertos (Pruebas o Tests que de soporte del despliegue). Se recopilará un seguimiento sobre el rendimiento, la interacción entre módulos y el cumplimiento de los requisitos.

2. **Análisis de desviaciones:** Evalúa qué componentes necesitan ajustes o mejoras, y actualiza la documentación y los requisitos según lo observado.

### 4.5. Comunicación

1. **Revisión y actualización de objetivos:** Con la retroalimentación obtenida, revalúa los riesgos y redefine los objetivos de ser necesario para el siguiente ciclo.

2. **Ajustes en la arquitectura:** Si es necesario, modifica las interfaces o la interacción entre componentes para optimizar la integración y la escalabilidad del sistema.

### 4.6. Informe Final

En la etapa final se llevará a cabo un informe en el que se recopile toda la información sobre el desarrollo del proyecto, este incluirá la descripción de cada etapa de desarrollo, así como los soportes que validen lo descrito, esto incluye diagramas, modelos y otros recursos que se lleguen a usar. El objetivo de este informe es presentar el desarrollo tanto a los calificadores en una primera instancia, como a estudiantes que deseen continuar con el desarrollo.

## 5. Desarrollo del Proyecto

Para el desarrollo del proyecto de grado, se cuenta con dos fases de desarrollo definidas; en una primera fase se realiza la definición de requerimientos y el alcance del proyecto, así como un diseño base de las partes del componente de código; posteriormente se iniciará con el desarrollo bajo la metodología en espiral, ajustándose a los requerimientos, diseño y alcance definidos en la primera fase, se debe resaltar que en esta ultima etapa se esperan tener múltiples cambios y ajustes sin sobre pasar el alcance del proyecto alineándose con la complejidad del proceso.

### 5.1. Fase Uno: Identificación y Diseño de la Estructura General del Componente

En esta primera fase se aborda el proceso de identificación de requerimientos funcionales, no funcionales, alcance y diseño base del componente; se finaliza con la presentación del diseño de componentes refinado durante el proceso de desarrollo.

### *5.1.1. Identificación de Requerimientos*

Inicialmente se realiza una reunión con el director del proyecto de grado, el Docente Hugo Andrade donde se definen los siguientes puntos:

1. Alcance del proyecto: El componente de código desarrollado debe poder realizar como mínimo, lo mismo que puede y no hacer el actual compilador de formulas.

2. Requerimientos: Están dados por las capacidades del compilador de formulas con el que cuenta actualmente Evolución, por lo que se debe realizar un estudio de sus capacidades.

3. Alternativas: Se discuten opciones posibles para el reemplazo del actual compilador de formulas, evaluándose pros y contras de tres opciones posibles:

   *a*) Desarrollar un componente de código de reemplazo.

   *b*) Corrección de errores en el actual compilador de formulas.

   *c*) Adquisición de un componente de terceros (privativo o gratuito) que pueda reemplazarlo.

En esta reunión se concluye que en el presente proyecto de grado, se realizara el análisis, diseño y desarrollo de un componente de código, que deberá contar con las mismas capacidades que el actual compilador de formulas integrado en Evolución, con el objetivo de integrarse en el software.

### *5.1.2. Definición de Requerimientos*

La información presentada a continuacion es el resultado del estudio de las capacidades en el manejo de expresiones matemáticas, con las que cuenta el software Evolución actualmente, ya que el componente encargado de realizar este proceso es el compilador de

formulas, es importante resaltar que este componente fue extendido para poder procesar cadenas con una mayor complejidad, admitiendo otro tipo de funciones.

A partir del análisis de requerimientos se obtiene todos los requerimientos funcionales y no funcionales presentados con prioridad uno, todos los demás surgen como complemento o mejora a la capacidad de suplir la necesidad de evaluar dinámicamente expresiones matemáticas.

**Tabla 1**

*Requerimientos funcionales*

| ID | Nombre | Descripción | Prioridad |
|---|---|---|---|
| RF1 | Caracteres permitidos | El componente debe aceptar expresiones que contengan letras, números, operadores matemáticos y símbolos definidos como válidos. | 1 |
| RF2 | Reconocimiento de números | El componente debe reconocer y procesar valores numéricos en las expresiones. | 1 |
| RF3 | Manejo de variables | El componente debe reconocer y procesar variables o elementos previamente declarados en las expresiones. | 1 |
| RF4 | Manejo de funciones | El componente debe reconocer y ejecutar funciones matemáticas predefinidas, así como otras funciones útiles en simulación. | 1 |
| RF5 | Reconocimiento de delimitadores | El componente debe reconocer delimitadores (como ,[ ] y ( )) y respetar su jerarquía para el análisis sintáctico. | 1 |
| RF6 | Operaciones matemáticas | El componente debe reconocer y ejecutar operaciones matemáticas básicas: suma, resta, multiplicación, división, potencia y módulo. | 1 |
| RF7 | Operadores de comparación | El componente debe reconocer y aplicar operadores de comparación (>, <, =, <>, >= y <=). | 1 |
| RF8 | Operadores lógicos | El componente debe reconocer y aplicar operadores lógicos (& y \|). | 2 |
| RF9 | Retorno de valores | El componente debe permitir el retorno del valor de una variable o número como resultado de una operación o expresión. | 1 |
| RF10 | Definición de elementos | El componente debe permitir definir, almacenar y eliminar elementos como constantes, variables y expresiones. Las variables pueden contener el resultado de evaluaciones. | 1 |
| RF11 | Detección de errores léxicos y sintácticos | El componente debe manejar errores léxicos y sintácticos en las expresiones. | 1 |
| RF12 | Mensajes de error en expresiones | El componente debe devolver mensajes de error claros indicando la causa y ubicación del problema en la expresión matemática ingresada. | 2 |
| RF13 | Mensajes de error generales | El componente debe devolver mensajes de error cuando se realicen operaciones incorrectas en el uso del componente. | 3 |
| RF14 | Carga de funciones en tiempo de ejecución | El componente debe permitir declarar y cargar funciones escritas en código Delphi para su ejecución dentro del componente. | 2 |
| RF15 | Carga de funciones desde DLL | El componente debe permitir cargar funciones desde bibliotecas dinámicas (DLL) para su uso en el componente. | 2 |

**Tabla 2**

*Requerimientos no funcionales*

| ID | Nombre | Descripción | Prioridad |
|---|---|---|---|
| RNF1 | Lenguaje de implementación | El componente debe estar implementado en el lenguaje de programación Delphi. | 1 |
| RNF2 | Capacidad de almacenamiento | El componente debe estar diseñado para almacenar eficientemente al menos 2000 elementos sin degradar el rendimiento. | 1 |
| RNF3 | Desempeño | El componente debe ofrecer un rendimiento comparable al del compilador de fórmulas actual al evaluar expresiones matemáticas. | 1 |
| RNF4 | Modularidad | El componente debe estar construido bajo una arquitectura modular que facilite la separación de responsabilidades y el mantenimiento. | 2 |
| RNF5 | Interfaz desacoplada | El componente debe contar con una interfaz de control que desacople al usuario del funcionamiento interno de compilación o interpretación. | 3 |
| RNF6 | Extensibilidad | El componente debe ser diseñado para facilitar futuras ampliaciones sin necesidad de modificaciones significativas. | 4 |

### *5.1.3. Diseño Base del Componente de Código*

Para la etapa de diseño fue necesario revisar la teoría de compiladores, donde este material aportaría diversas técnicas y formas de diseño, cuyo fin equivale al de los interpretes, difiriendo en la representación final, ya que el propósito central de estos conceptos es comunicar instrucciones basadas en algoritmos a las computadoras; bajo esta idea los compiladores se centran en la usabilidad determinista del algoritmo, mientras que un interprete permite construir instrucciones dinamicamente definidas por un usuario final, donde la complejidad del lenguaje puede variar, llegando acercarse al lenguaje humano o natural.

De esta forma se encuentra que los compiladores e interpretes comparten el proceso interno, que se componen de un elemento denominado tokenizador ó generador de tokens, el cual se encarga de transformar una cadena de texto en una lista de tokens tipados, que son empleados en la construcción de un árbol de sintaxis abstracto ó AST. Los compiladores e interpretes difieren en su resultado final, ya que los compiladores representan el AST en forma de código maquina, mientras los interpretes cuentan con la capacidad de representar

y ejecutar el código directamente.

Una idea inicial planteaba un símil entre lo que realiza una calculadora con lo que evolución puede hacer, que recae en evaluar expresiones matemáticas que puedan contener un conjunto de elementos matemáticos previamente declarados, como constantes y otras expresiones almacenadas bajo un nombre (semejante a una variable), bajo esta idea se plantea inicialmente utilizar el algoritmo de Shunting Yard de Edsger Dijkstra para evaluar la expresión matemática en base a la lista de tokens y posteriormente compilarse empleando algún compilador portable de Delphi. En relación al anterior planteamiento, se construye el siguiente diseño componentes base:

**Figura 7**

*Diseño de componentes base para el desarrollo en espiral*


Cabe destacar que se planteo realizar prototipos funcionales de cada componente con el objetivo de probar la viabilidad del diseño base, de esta manera durante el desarrollo de la metodología en espiral, se encuentra que el algoritmo de Shunting Yard de Edsger Dijkstra se centra en evaluar mas no ofrece alguna ventaja en el análisis de las expresiones, dificultando el proceso de depuración de las expresiones matemáticas. Es por esto que es necesario reajustar el diseño de componentes base al siguiente modelo:

**Figura 8**

*Diseño modificado de los componentes base para el desarrollo en espiral*


El anterior diseño cuenta con los elementos primordiales en el proceso de compilación o interpretación, a partir de los cuales, como se mencionó anteriormente, se construye código maquina o se interpreta el código, empleando en este caso un proceso de transpilación o representación de la información contenida en los diferentes tipos de expresiones, concluyendo en un proceso de interpretación. A continuación se abordaran los componentes necesarios en el proceso de interpretativo de una expresión matemática:

1. **Análisis Léxico:** Descompone la cadena de texto en tokens tipados, en esta etapa del proceso de interpretación se reconocen varios tipos de errores de léxico en la expresión.

2. **Análisis Sintáctico:** Construye un árbol de sintaxis en base a la lista de tokens y tipos, para posteriormente realizar una revisión de las reglas de gramaticales, reconociendo de esta forma errores relacionados en la expresión.

3. **Proceso de Transpilación:** Utilizando el árbol de sintaxis se crea una orden de evaluación de los diferentes elementos contenidos en cada expresión matemática, esta lista representa junto al árbol de sintaxis, la información contenida en la expresión matemática, la cual sera utilizada para la creación de una estructura de datos que pueda contener dicha información, para posteriormente, ser almacenada y evaluada según se requiera.

4. **Hoja de Expresiones:** Este componente se encarga de gestionar y orquestar el proceso de interpretación, encapsulando todos los elementos como las constantes y expresiones que el usuario puede definir. Cumple ademas la función de interface de uso aislando al usuario del proceso interno.

## 5.2. Fase Dos: Desarrollo Bajo la Metodología en Espiral

El desarrollo en espiral obtuvo como resultado el diseño de componente presentado en la Figura 9, donde cada componente se subdividió en clases (cumpliendo con el principio de responsabilidad única, SOLID); a continuacion se presenta los componentes y sus partes:

1. **Análisis Léxico:** Este proceso es realizado por los componentes FTokenParser, FStates, FTransitions. Esta tarea contó con un conjunto de arreglos que almacenan la información necesaria en la operatividad del proceso, siendo: Caracteres validos por transición y su tipado.

2. **Análisis Sintáctico:** Los responsables de llevar acabo esta tarea, son las clases BuilderTre, SyntaxTree y Grammars, orquestados por la clase ExpressionBuilder. Este proceso requiere de un orden de procedencia en base al tipado de cada elemento en la expresión.

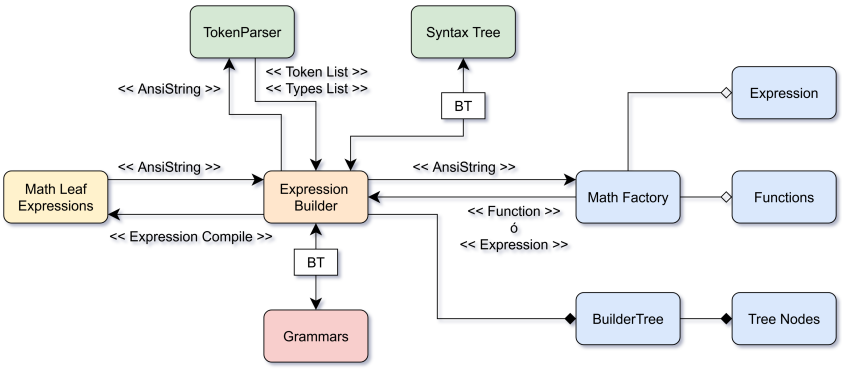
3. **Proceso de Transpilación:** La clase ExpressionBuilder es la encargada de representar la expresión por medio de una estructura de datos, que es contenida en la clase FExpression. FMathFactory emplea el patrón de diseño Factory para generalizar la construcción de las partes de la expresión, esto incluye: estructura de la expresión, representación de las operaciones, funciones y representación de los elementos de la expresión (Valores numéricos, variables y constantes).

4. **Hoja de Expresiones:** MathLeafExpressions corresponde al componente que permite al usuario emplear todas las funcionalidades definidas: Manejo de expresiones, variables, constantes, escenarios, agregación de funciones (puntero y DLL) y evaluación de

las expresiones. Se encarga de contener unicamente los elementos, lo que implica la liberación de las estructuras empleadas en la construcción de las expresiones.

**Figura 9**

*Diseño final de los componentes desarrollados*



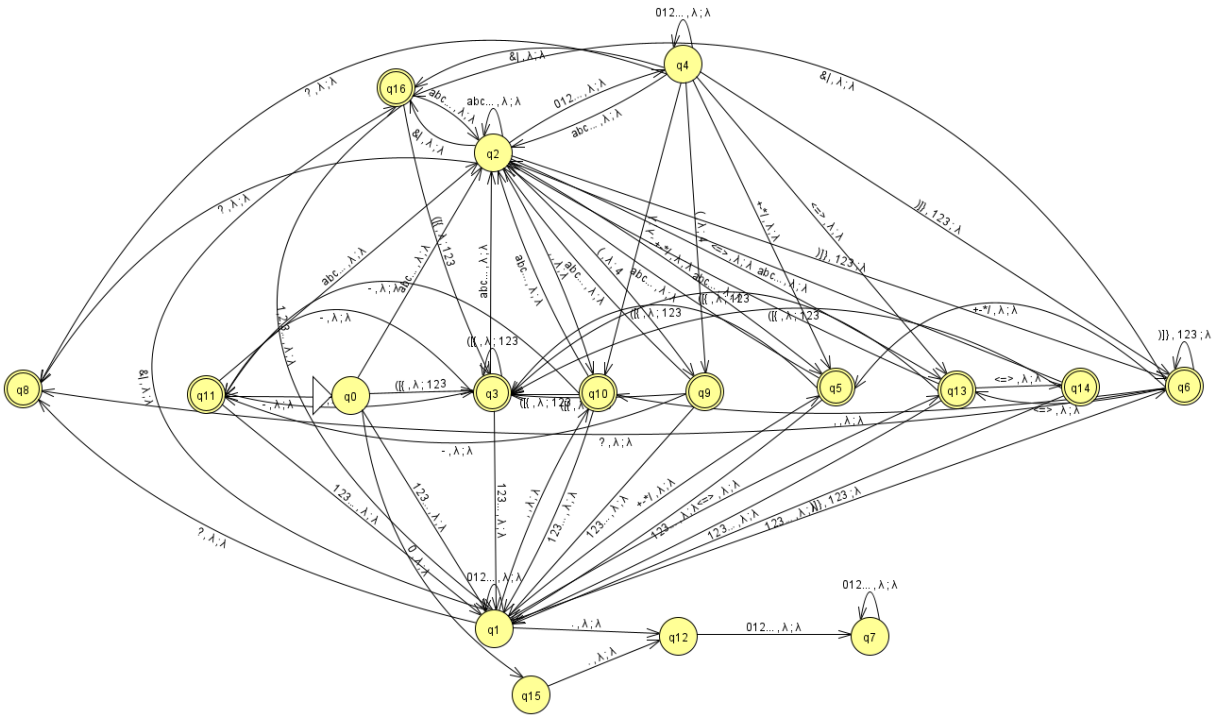

En las subsecciones siguientes se abordará cada etapa del desarrollo en espiral, describiendo en detalle como funciona y se relacionan las diferentes partes necesarias en el proceso de interpretación y evaluación de las expresiones matemáticas, así como todo lo relacionado al proceso de gestión definido en la hoja de expresiones.

### *5.2.1. Análisis Léxico*

El problema que plantea realizar un análisis léxico de una cadena de texto o expresión matemática, se puede realizar mediante el diseño de un autómata con pila no determinista y transiciones epsilon, ya que estos se especializan en reconocer lenguajes libres de contexto, como las expresiones matemáticas y lenguajes de programación sencillos, ya que este tipo de lenguajes posee estructuras recursivas en su interior, recayendo de esta manera, en los lenguajes de segundo tipo según la clasificación de Chomsky (Moral, s.f.), a pesar de esto, se construye un autómata con pila determinista con transiciones epsilon, ya que los elementos dentro de las expresiones cuentan con un orden de precedencia que se debe cumplir en cada caso, esto lo vuelve determinista. La siguiente figura presenta el diseño final del autómata:

**Figura 10**

*Diseño final del autómata finito no determinista con pila y transiciones epsilon*




A continuación se presenta de forma resumida la definición del autómata:

$$M = (Q, \Sigma, \Gamma, \delta, q_0, F)$$

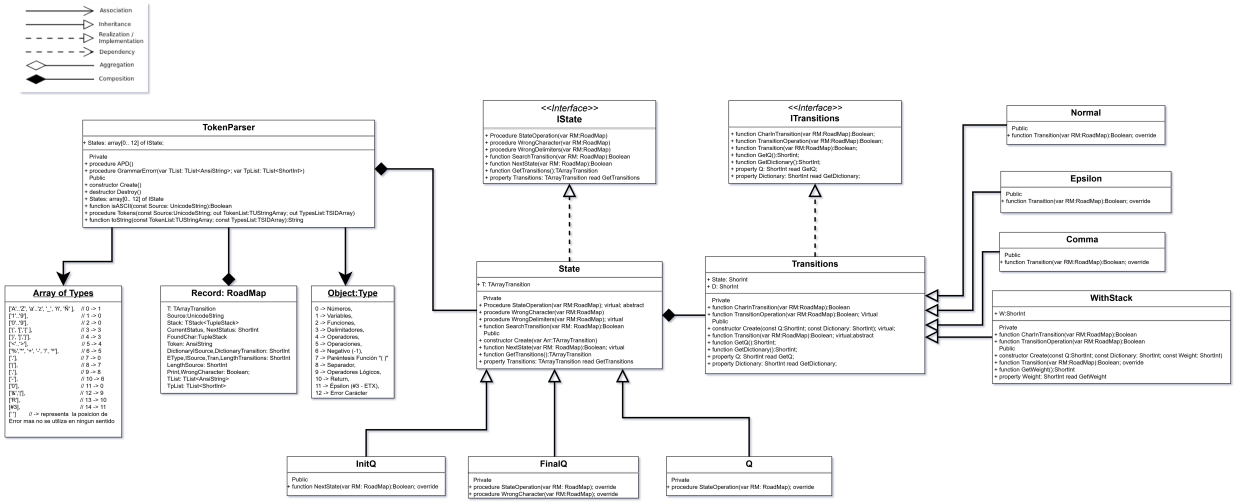
Donde:

- $Q = \{q_0, q_1, q_2, \ldots, q_{16}\}$ es el conjunto de estados.
- $\Sigma = \{a, b, \ldots, Z, +, \ldots, *, (, [, \{, \ldots, ), \varepsilon\}$ es el alfabeto de entrada.
- $\Gamma = \{1, 2, 3, 4\}$ es el alfabeto de la pila.
- $q_0$ es el estado inicial.
- $F = \{q_3, q_5, q_6, q_8, q_9, q_{10}, q_{11}, q_{13}, q_{14}, q_{16}\}$ es el conjunto de estados de aceptación.
- $\delta$ es la función de transición.

Para poder probar y diseñar el autómata, se diseña y desarrolla un conjunto de clases capaces de soportar la lógica requerida por el autómata, este componente corresponde al siguiente diagrama de clases:

**Figura 11**

*Diagrama de clases del autómata finito no determinista con pila y transiciones epsilon*




El diseño y concepto bajo el que se construye el autómata difiere un poco de la teoría, ya que de esta manera, se centraría en reconocer si una palabra pertenece a un lenguaje, sin la opción de desglosar en partes o tokens la expresión, permitiendo bajo este diseño la definición del tipo de cada elemento encontrado. El autómata se diseña de la siguiente manera:

1. **Estados:** El autómata puede contar con n estados según se requiera, cada estado tendrá un conjunto de transiciones. Se cuentan con tres tipos de estados:
   - **Estados Normales:** Son los responsables de construir cadenas que representan números, nombres de variables y funciones.
   - **Estado Inicial:** Es el punto de arranque del autómata, difiere de los demás tipos de estados al no realizar alguna tarea, en cambio, es el encargado de reconocer si una cadena se inicia correctamente.

- **Estado Final:** Este estado identifica elementos y sus tipos (para esto emplea operadores, delimitadores y otros símbolos como punto de corte en la cadena), a la vez que encuentra errores de léxico en la cadena, al encontrar un error, vuelve al autómata al estado inicial para continuar buscando errores.

2. **Transiciones:** Cuentan con un alfabeto propio, que es un subconjunto de elementos del alfabeto del autómata, junto a un estado al que apuntan, las transiciones verifican si un carácter pertenece a su alfabeto, permitiendo el paso o no al siguiente estado, si el carácter no pertenece ninguna de las transiciones es un indicador de error. Se contemplan tres tipos de transiciones:

   - **Transiciones Normales:** Realizan unicamente la revisión de pertenencia del carácter al sub-alfabeto asignado.
   - **Transiciones Epsilon:** Permiten realizar una transición especial ya que no realizan ninguna actividad, su objetivo principal es encontrar errores relacionados con el cómo debe terminar una expresión matemática, solo se puede pasar a través de ella mediante un carácter epsilon (Se agrega automáticamente a todas las cadenas al final de la misma se emplea el carácter especial ETX).
   - **Transiciones Comma:** Se encarga de manejar este carácter especial empleado para separar los parámetros dentro de una función, por lo que su utilidad recae en la identificación del uso correcto o incorrecto del símbolo.
   - **Transiciones con Pila:** El manejo correcto de los paréntesis y su jerarquía representa un caso especial en el análisis léxico, es por esto que se emplea una pila general, la cual se encarga de abrir y cerrar cada paréntesis empleado en la expresión, permitiendo saber si se están usando correctamente dichos elementos, identificando y tipando cada error, así como cada tipo de paréntesis. Cabe resaltar que se reconocen dos tipos de delimitadores: agrupación y de función.

Con este enfoque es posible iterar sobre cada elemento de la cadena identificando sus

partes. Es importante resaltar que el autómata carece de estado alguno, es decir, el autómata se comporta como un autopista complejo, donde los estados se comportan de forma similar a los peajes, permitiendo o no el paso y realizando un proceso de clasificación definido, respecto al tipo de peaje o estado en el que se encuentre, de esta manera, el autómata no contiene datos y es posible usar al mismo para reconocer múltiples cadenas en paralelo, debido a que cada cadena tendrá asociado un Record que almacena la información del proceso llevado acabo por el autómata.

De esta manera el autómata reconoce un conjunto de palabras en base a un alfabeto, cada palabra tiene un tipo que es definido por la estructura de la propia palabra. La capacidad de identificar elementos matemáticos sobre pasa las posibilidades ofrecidas por el compilador de formulas integrado en Evolución, ya que el proceso de identificación de funciones es limitado junto al manejo de operadores lógicos, el cual carece de soporte directo y depende de funciones (AND(a,b,..., n) y OR(a,b,..., n)). El conjunto de elementos y tipos que reconoce el autómata es el siguiente:

**Tabla 3**

*Elementos y sus tipos*

| **Elemento** | **Tipo** |
|---|---|
| Números | 0 |
| Variables | 1 |
| Funciones | 2 |
| Delimitadores ( ”(”, ”)”, ”[”, ”]”, ”{”, ”}”) | 3 |
| Operadores (>, >=, <, <=, <>, =) | 4 |
| Operaciones (+, -, *, /, ^ , %) | 5 |
| Negativo (-) | 6 |
| Paréntesis Función ”( )” | 7 |
| Separador ( ”,”) | 8 |
| Operadores Lógicos (&, \| ) | 9 |
| Return (Cuando la expresión contiene unicamente un numero o variable) | 10 |
| Epsilon (#3 - ETX - ?) | 11 |
| Error | 12 |

El proceso del análisis léxico identifica diversos tipos de errores en la estructura de la

expresión, como lo son: la forma correcta de terminar una expresión matemática, el correcto uso de los operadores, la estructura correcta de los nombres de las variables y números, entre otros. Los errores manejados por la etapa del análisis léxico son descritos en la siguiente tabla:

**Tabla 4**

*Errores encontrados durante el análisis léxico*

| **Responsable** | **Descripción** |
|---|---|
| Caracteres | Caracteres inválidos detectados. |
| Números | Falta el punto (.) esperado.<br>Punto (.) sin elemento precedente. |
| Operaciones | Sintaxis de operación inválida. |
| Operadores | Operador de comparación inválido. |
| Funciones | Falta el paréntesis de cierre en la función.<br>Faltan parámetros en la función.<br>Coma sin valor posterior.<br>Coma no está entre valores. |
| Delimitadores | Se esperaba un paréntesis de apertura '('.<br>Se esperaba un paréntesis de cierre ')'. |
| Operadores lógicos | Operador lógico inválido: se esperaba '&' o \|.<br>Operador lógico en la posición incorrecta. |
| Operadores | Operaciones lógicas anidadas no permitidas.<br>Uso inválido de operador lógico. |

### *5.2.2. Análisis Sintáctico*

Una vez se ha realizado el análisis léxico y se ha comprobado que la cadena no contiene errores, es necesario construir un árbol de sintaxis basado en las lista de tokens y tipos, con el objetivo de lograr:

1. **Árbol de sintaxis:** Se revisa que no se cometan errores relacionados a la sintaxis evaluando un conjunto de reglas gramaticales por medio del árbol de sintaxis.

2. **Evaluación y optimizaciones del árbol de sintaxis:** Se realiza una evaluación de la reglas gramaticales en la expresión, sino se encuentran errores gramaticales, se continua la optimizar las estructuras dentro del grafo correspondientes a las operaciones suma y resta, un proceso semejante a la factorización, ahorrando pasos de ejecución durante la evaluación de las expresiones.

3. **Construcción del orden de evaluación:** A partir del árbol de sintaxis se crea un orden de evaluación de las operaciones contenidas en la expresión.

**5.2.2.1 Árbol de Sintaxis.** Conforme a lo planteado anteriormente, es necesario definir los elementos requeridos en la construcción del árbol de sintaxis, por ello se diseñaron los siguiente procesos consecutivos:

1. **Estructura de datos que soporte el árbol de sintaxis:** Corresponde al componente de código encargado de representar mediante la programación orientada a objetos la información contenida en la expresión matemática bajo la notación infija.

2. **Constructor del árbol de sintaxis:** Es necesario construir la representación correspondiente a cada expresión matemática en la estructura de datos diseñada, este proceso permite ademas evaluar diferentes tipos de errores gramaticales a partir de la propia estructura del árbol.

Bajo este orden de ideas, se diseña un grafo con los elementos necesarios para construir un árbol de sintaxis que represente de forma adecuada cualquier expresión matemática, por ello, se diseñan un conjunto de clases que emplean la herencia para optimizar la estructura de datos, como se presenta en la Figura 12, cada árbol de sintaxis puede contar con:

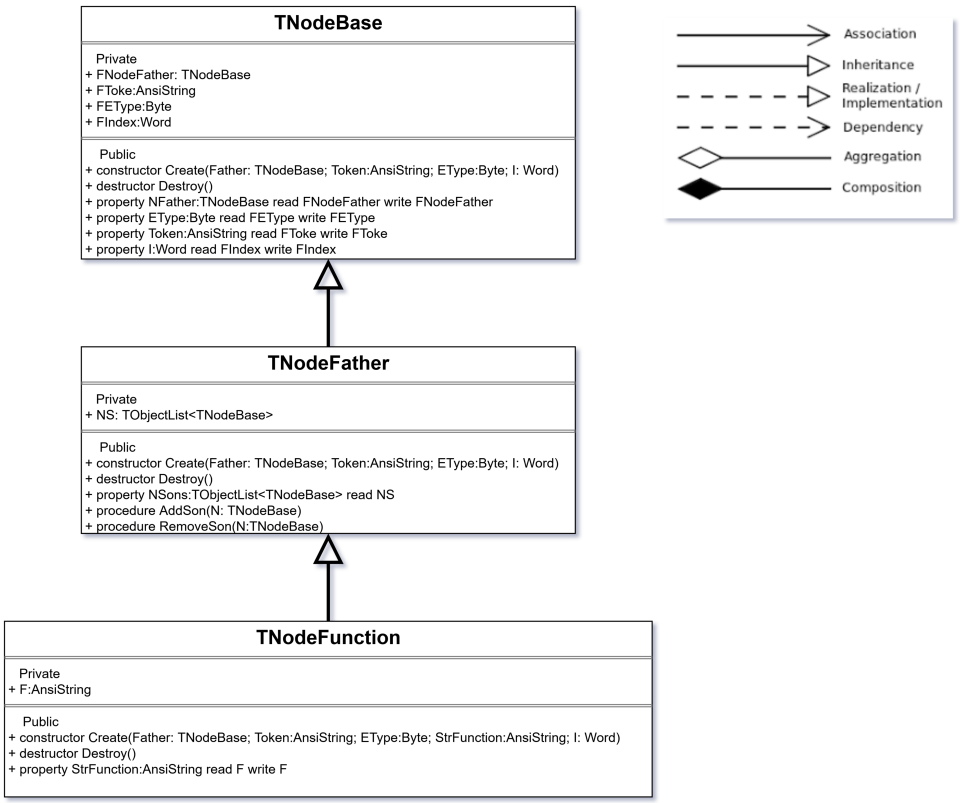
- **Nodo Base:** Clase base de las que heredan las demás clases de nodos. El nodo base contiene una cadena que representa el token, el tipo del token, y la posición del token en la cadena de tokens. Se emplea para almacenar números y variables.

- **Nodo Padre:** Clase que hereda directamente de la clase Nodo Base. Representa los elementos capaces de tener hijos como: operaciones, operadores comparativos, operadores lógicos y paréntesis. Almacenan elementos que sean ó hereden de la clase Nodo Base.

- **Nodo Función:** Clase que hereda de Nodo Padre y por consecuencia de la clase Nodo Base. Como su nombre lo indica, permite representar los elementos de tipo función, ya que este tipo contiene mas datos que los anteriores, lo que facilita el proceso de evaluación de las gramáticas (la función plantea el problema del inicio y cierre de la misma, semejante a los paréntesis, permitiendo evaluar el proceso como correcto o incorrecto).

**Figura 12**

*Diagrama de clases del grafo o árbol de sintaxis*

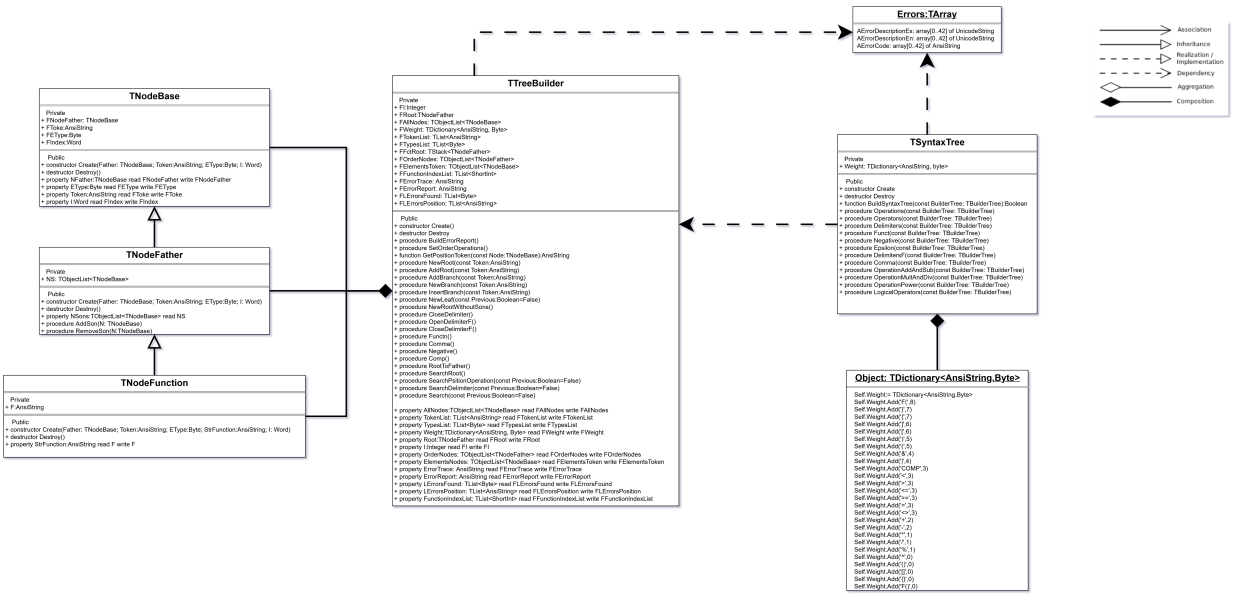


Esta jerarquía de herencia permiten construir un grafo o árbol de sintaxis, que emplea los recursos necesarios para representar una expresión matemática, de otra manera existirán elementos con reservas de memoria vacías, lo que genera un aumento en el uso de espacio en memoria innecesario y en general una sobre carga.

**5.2.2.2 Evaluación y Optimizaciones del Árbol de Sintaxis.** Teniendo esta estructura de datos, se da comienzo a la construcción del árbol de sintaxis, para lo cual se diseñan y desarrollan un conjunto de clases presentadas en la siguiente figura:

**Figura 13**

*Diagrama de clases del proceso de construcción del árbol de sintaxis*




Esta estructura responde a la necesidad de resolver fugas de memoria y a la extensibilidad de las capacidades del componente, para lo cual se hace necesario separar las operaciones de construcción del árbol de la reglas gramaticales. De esta manera contamos con las siguientes tareas realizadas por las clases presentadas en la anterior imagen:

1. **Constructor del árbol:** Se encarga de almacenar el grafo y toda la información pertinente, como los errores encontrados; ademas construye el orden de evaluación y lo contiene en una lista de nodos. Cuenta ademas con todas las operaciones generales en

el proceso de agregación de nodos al árbol (construcción del árbol), esto se entenderá mejor al revisar la sección de ejemplos.

2. **Árbol de sintaxis:** Contiene la lógica de programación bajo la cual se construye una representación de las reglas gramaticales sobre el grafo o árbol de sintaxis, para esto se emplea un pivote (almacenado en el constructor del árbol) que cumple la función de conocer donde se esta construyendo o agregando elementos; se mueve el pivote en el árbol según las reglas gramaticales de las expresiones matemáticas. Cabe destacar que las reglas de gramaticales bajo las cuales se construye el árbol no implican el estado de aceptación del proceso, por este motivo es necesario realizar una revisión de otro conjunto de reglas gramaticales que terminan de definir si es correcta su estructura o no.

3. **Optimización del árbol:** Se realizan dos proceso, uno se relaciona con la agrupación de operaciones de suma y resta anidadas, de esta manera se consigue eliminar elementos anidados reduciendo el numero de operaciones necesario en la evaluación; por otro lado se eliminan los paréntesis contenidos en el árbol, estos no cumplen ninguna función en el proceso de construcción del orden de evaluación, lo que recae en pasos de ejecución desperdiciados. Esta tarea es contenida en las gramáticas (Figura 14).

4. **Estructuras de datos requeridas:** La construcción del árbol depende de un orden de precedencia asignado a los diferentes elementos que pueden ser Nodos Padre Función, para esto se crea un diccionario que contiene pesos que se emplean para mover el pivote y tomar diferentes tipos de decisiones dentro del algoritmo. Los valores cero anulan el peso en el manejo de los delimitadores y las funciones, ya que una vez se encuentra el elemento de cierre, estos se comportan como un numero o variable.

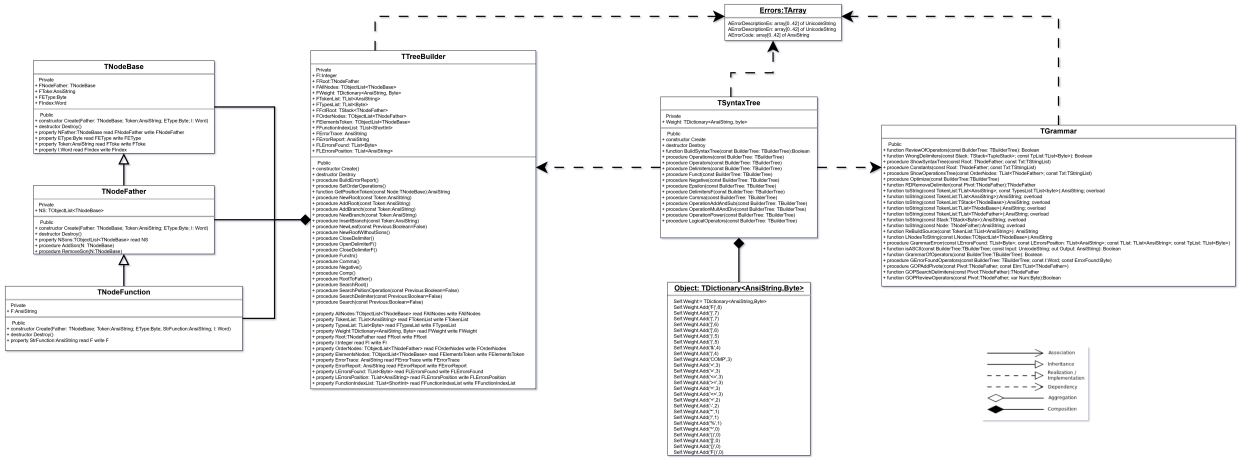
5. **Manejo de errores:** Están contenidos en una lista de códigos y definiciones, el algoritmo según se incumpla una regla de gramática asigna el error correspondiente, esto ya que, cada operador aritmético, de comparación, lógico y función cuentan con un

método destinado a su manejo, permitiendo de esta manera métodos especializados y por ende capaces de definir errores gramaticales con precisión.

Por ultimo se integra una clase centrada en contener todas las gramáticas y otras operaciones que impliquen recorrer el árbol de sintaxis, de esta manera se obtiene el componente encargado de analizar las gramáticas de la expresión matemática:

**Figura 14**

*Diagrama de clases del proceso de análisis sintáctico*




Las tareas realizadas por la clase encargada de las gramáticas son:

- Revisión de operadores de comparación.
- Definición de errores relacionados a los delimitadores.
- Optimización.
- Eliminación de los paréntesis.
- Reconstrucción de la expresión a partir del árbol y sus errores de gramática.
- Revisión de caracteres *Ansi* según el conjunto especifico permitido.
- Revisión de las gramáticas relacionadas a los operadores de comparación.

- Contiene ademas métodos auxiliares a otros procesos dentro la propia clase.
- Funciones auxiliares en los procesos de depuración, las cuales permitían reconstruir la expresión a partir del grafo de diferentes maneras, con el objetivo de comprobar su veracidad.

Este componente maneja los siguientes errores:

**Tabla 5**

*Errores gramaticales*

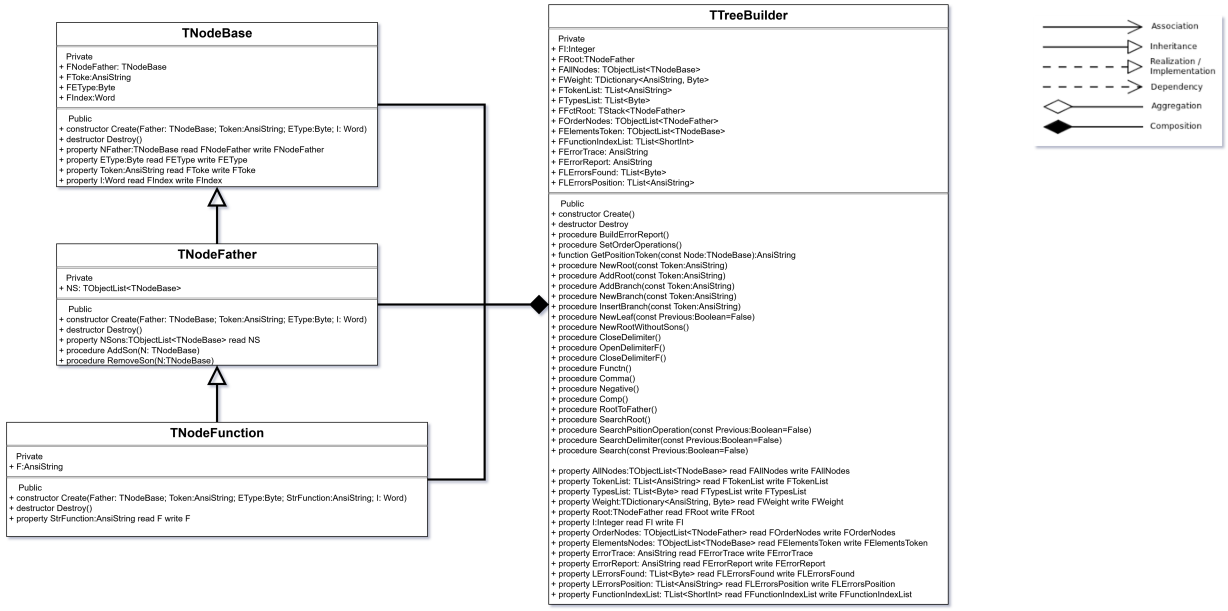
| **Categoría** | **Descripción** |
|---|---|
| Operadores | Operaciones lógicas anidadas no permitidas.<br>Uso inválido de operador lógico. |

**5.2.2.3 Construcción del Orden de Evaluación.** El orden de evaluación es construido solo si en todo el proceso no se encuentra error de ningún tipo, para este proceso se recorre en profundidad el árbol, ya que los elementos deben ser evaluados desde los nodos padres de ultimo nivel o profundidad, este proceso cobrara sentido en la sección de ejemplos.

El resultado de este proceso es almacenado dentro de una lista contenida en la clase encargada de construir el árbol (BuilderTree). El constructor del árbol se presenta de forma asilada en la siguiente figura:

**Figura 15**

*Diagrama de clases del constructor del árbol de sintaxis*




### *5.2.3. Transpilación de la Expresión*

El proceso de transpilación de la expresión depende completamente del orden de evaluación, obtenido como resultado en el proceso anterior; se emplea el orden de evaluación para codificar la expresión sobre un conjunto de elementos, los cuales son:

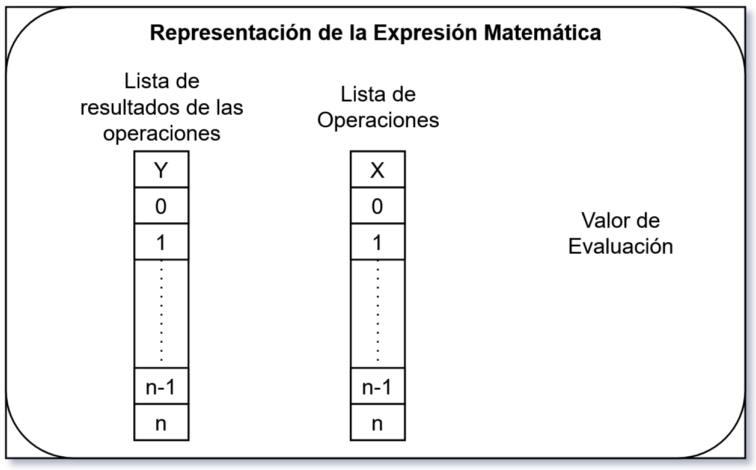
1. **Operaciones:** son representaciones de funciones matemáticas codificadas bajo una estructura, estas reciben una lista que contiene punteros, lo que permite generalizar el proceso de construcción y evaluación de las expresiones; cada operación cuenta con una función programada en Delphi, por lo que cada expresión es en ultima instancia, un conjunto ordenado de funciones que reciben una lista con punteros apuntando a los valores correspondientes a la operación.

2. **Expresión:** Las expresiones son la representación del orden de evaluación, que se constituye de las operaciones y funciones matemáticas que serán evaluadas junto a

sus parámetros, semejante a lo que un ser humano hace para evaluar una expresión matemática.

**Figura 16**

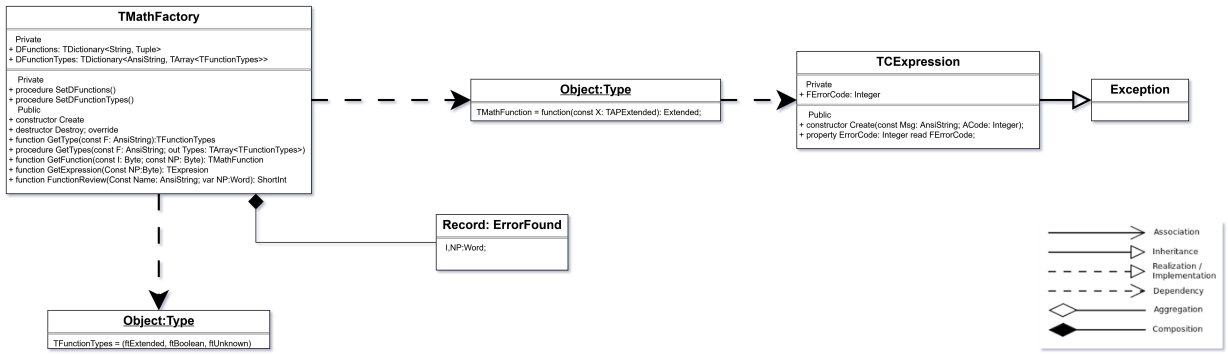
*Diseño de la estructura de datos que soporta las expresiones*




Gracias al anterior planteamiento, se alcanza el diseño de un proceso que generaliza la construcción de las expresiones, para esto se diseñan un conjunto de clases con el fin de construir esta estructura mediante una metodología (generalización) empleando para este propósito el patrón de diseño *Fabrica*, con esto se consigue un proceso de construcción fluido y extensible, ya que basta con escribir la función bajo la estructura definida para estas, agregarla en la lista de funciones disponibles en la *Fabrica* para dejarla disponible a los usuarios finales. De esta manera se logra una representación codificada de las expresiones, el resultado final se presenta en la siguiente figura:

**Figura 17**

*Diagrama de clases de la fabrica de los elementos matemáticos contenidos en las expresiones interpretadas*




Con estos elementos juntos y orquestados por una clase, se logra construir de forma limpia y ágil, una representación exacta de las diferentes expresiones matemáticas que se puedan querer interpretar o compilar. Bajo esta idea se lleva acabo el siguiente proceso en la clase encargada de construir la representación de una expresión:

1. **Orquesta del proceso de construcción del orden de evaluación:** Este proceso involucra a los componentes encargados del análisis léxico y sintáctico, tendiendo en cuenta los posibles errores o fallos que puedan tener cada uno de los proceso, y devolviendo el correspondiente mensaje de error.

2. **Construcción de la expresión interpretada:** Se emplea la clase *Fabrica* que devuelve los elementos correspondientes y bajo el formato adecuado, es así, que se construyen la representación correspondientes al orden de evaluación obtenido de los diferentes tipos de expresiones.

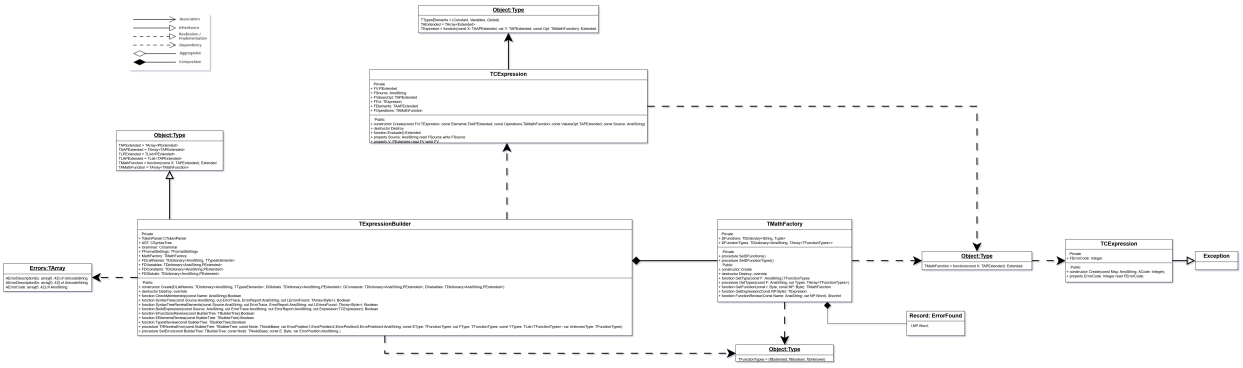
3. **Creación y retorno del objeto:** Se instancia un objeto que representa a la expresión, con el propósito de encapsular la información descrita por la expresión matemática y generalizar el proceso evaluativo, por medio del cual, se obtiene un valor numérico correspondiente a los valores y elementos contenidos en la expresión, dando como resultado exactamente el valor que se obtendría al realizar el proceso de forma manual

o usando otro método evaluativo (software de calculo como Python, Matlab o calculadoras).

Por ultimo se presenta el diseño completo del componente encargado de construir las expresiones en la siguiente figura:

**Figura 18**

*Diagrama de clases del componente encargada de la transpilación*




El anterior componente de código maneja los siguiente errores relacionados:

**Tabla 6**

*Errores semánticos*

| **Categoría** | **Descripción** |
|---|---|
| Funciones | La llamada a la función excede el número de parámetros esperados.<br>Función desconocida: `<NombreDeLaFunción>`. |
| Variables y constantes | La expresión contiene elementos no declarados. |
| Tipos | Valor Booleano no permitido, se esperaba un valor Numérico.<br>Valor Numérico no permitido, se esperaba un valor Booleano.<br>La función `if` debe retornar el mismo tipo de valores.<br>El operador lógico debe comparar un único tipo de valores. |

### *5.2.4. Interface de Uso*

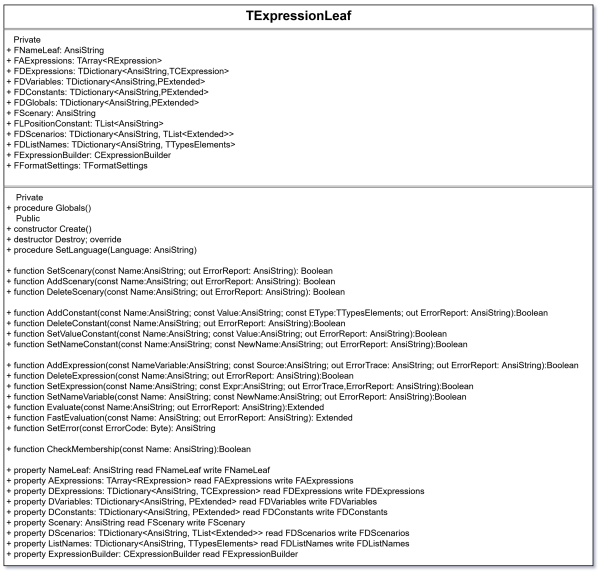
Para facilitar la implementación y asilar al usuario del proceso interno que involucra la interpretación de expresiones matemáticas, se diseña una clase (Figura 19) encargada de

la gestión y uso de los elementos matemáticos contenidos en las expresiones; con esta clase se consigue ademas contar con un entorno de evaluación de expresiones matemáticas muy útil en modelos de simulación, semejante a una hoja de Excel, pero pensada para satisfacer las necesidades que implica construir un software de simulación basada en dinámica de sistemas (DS) como Evolución.

- Constantes Globales (se relacionan con valores matemáticos como pi y e).
- Constantes Normales (su valor depende del escenario).
- Expresiones Interpretadas.
- Resultados de Evaluar una Expresión.
- Escenarios de Simulación.

**Figura 19**

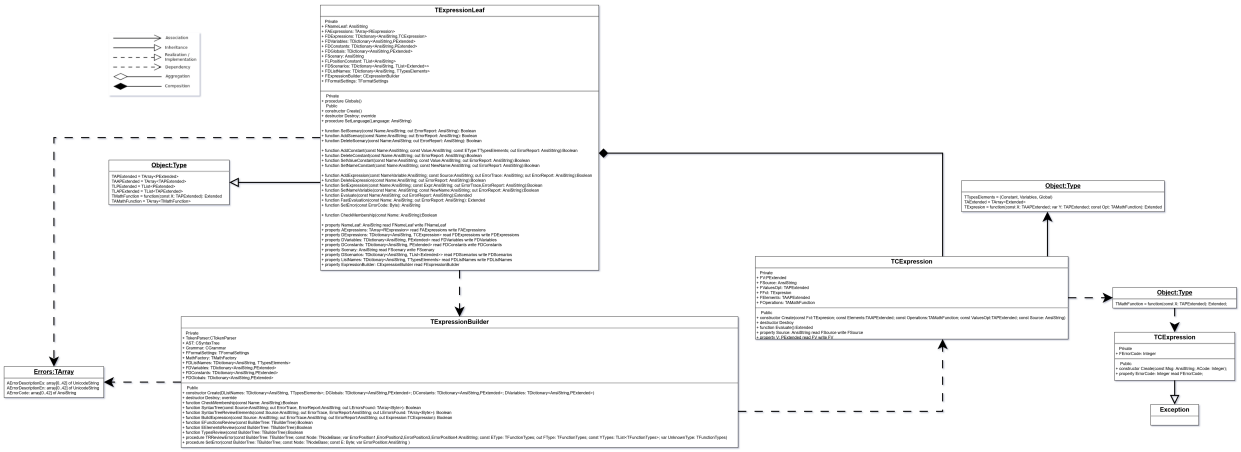
*Diagrama de la clase encargada de la gestión y uso de los elementos matemáticos*




Como se menciono anteriormente, esta clase se encarga de gestionar y usar los siguiente componentes, mostrando de esta manera que la arquitectura modular esta presente en el diseño general, bajo la cual se diseña el componente de código:

**Figura 20**

*Diagrama de clases del componente encargado de la gestión y uso de los elementos matemáticos*

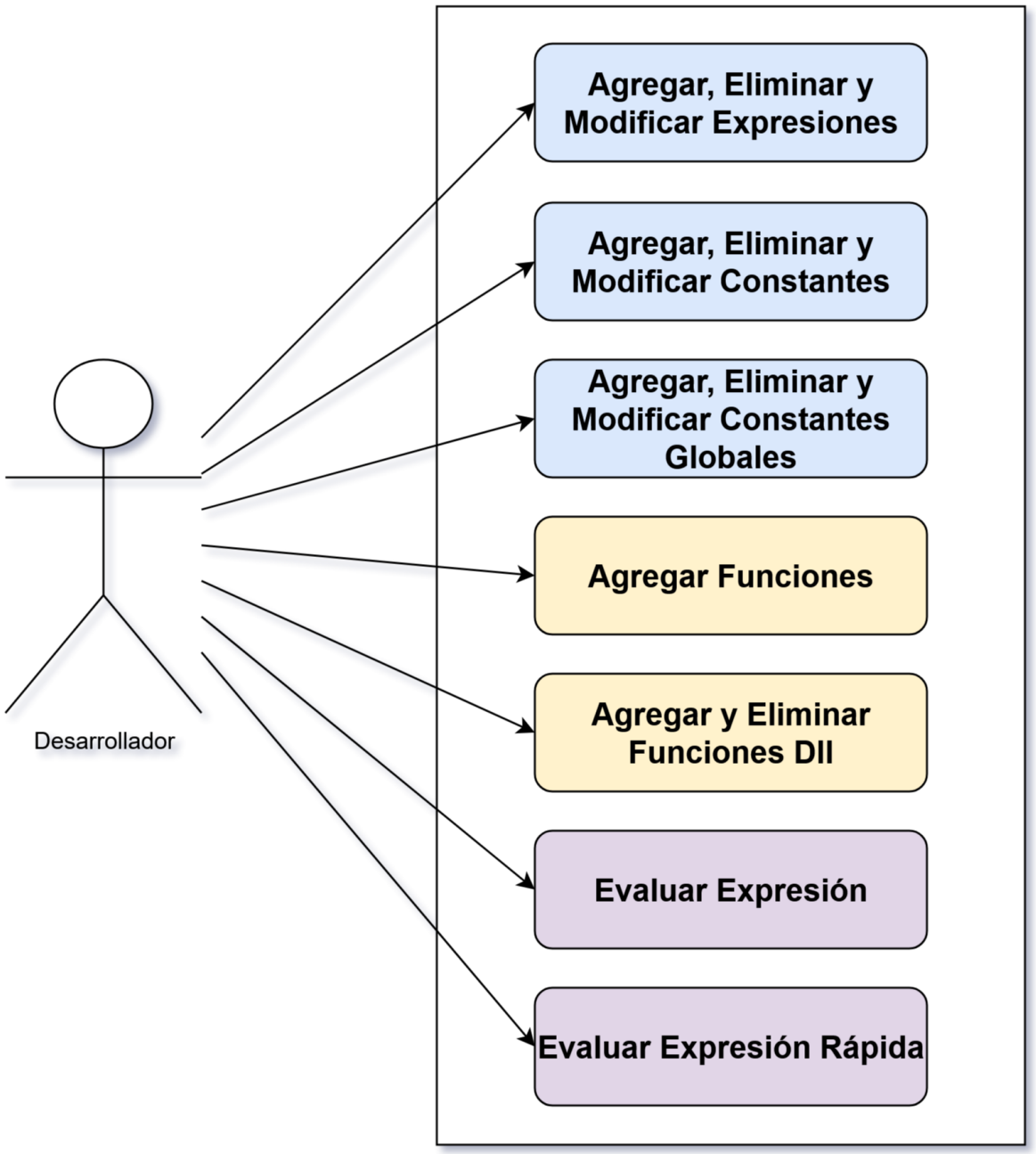



De esta manera se consigue satisfacer el diagrama de casos pensado para la interface de uso, la cual se presenta en la siguiente imagen (Figura 21). Esta satisface la metodología necesaria para construir hojas de expresiones que puedan representar diferentes tipos de sistemas, como los matemáticos (sistemas de ecuaciones) ó estadísticos (simulaciones por eventos discretos). Contando con la posibilidad de extender a diferentes áreas del conocimiento, las capacidades o funciones, empleando los métodos de agregación de funciones tanto puntero como DLL.

De esta manera se delega la responsabilidad de construir el orden de evaluación de los modelos basados en dinámica de sistemas (DS) o en estadística, junto a las funciones auxiliares que requiera cada simulación. El orden de evaluación de las expresiones matemáticas representa junto a las mismas expresiones, el modelo basado en DS, es de recalcar el reto y dificultad que este proceso plantea, alcanzándose con el software de DS Evolución.

**Figura 21**

*Diagrama de casos de uso*




Este componente se encarga de manejar lo siguientes errores relacionados con el funcionamiento de la clase controladora y del proceso de evaluación de las expresiones, ya que

se deben manejar los errores que pueden ocurrir al evaluar funciones en dominios vacíos o nulos, como es el caso de la división por cero, de esta manera se tienen el siguiente listado de errores manejados:

**Tabla 7**

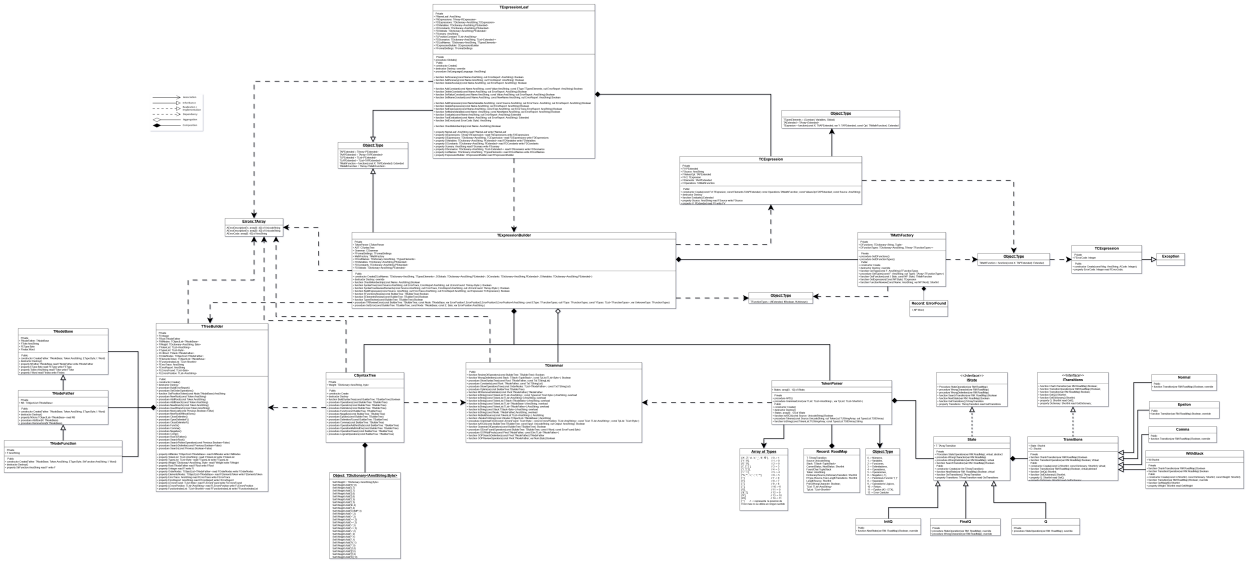
*Errores generados por la clase controladora y por el proceso de evaluación de expresiones*

| **Categoría** | **Descripción** |
|---|---|
| Operaciones matemáticas y funciones | División por cero.<br>`TAN`: puede dar error si el ángulo es $\pi/2 + k\pi$.<br>`COTAN`: no está definida para múltiplos de $\pi$.<br>`ASIN`: dominio inválido si el argumento no está entre -1 y 1.<br>`ACOS`: dominio inválido si el argumento no está entre -1 y 1.<br>`ACOSH`: requiere argumento $\geq 1$.<br>`ATANH`: requiere argumento entre -1 y 1.<br>Raíz cuadrada indefinida para valores menores a cero.<br>Logaritmo base 10 indefinido para valores menores a cero.<br>Módulo por cero.<br>Factorial indefinido para valores menores a cero.<br>Factorial solo definido para enteros.<br>`LN` indefinido para valores menores o iguales a 0.<br>`EXPRND`: la media debe ser un valor positivo. |
| Expresiones y elementos | El nombre del elemento ya existe.<br>Elemento no encontrado.<br>Expresión no encontrada.<br>Escenario no encontrado.<br>El escenario ya existe.<br>Valor inválido.<br>Error en evaluación rápida.<br>Elemento o nombre de variable no registrado.<br>Nombre inválido: debe comenzar con una letra (A–Z o a–z).<br>No se puede eliminar todos los escenarios.<br>Nombre del elemento o escenario vacío. |
| Fábrica de funciones (*Math Factory*) | El nombre de la función ya existe.<br>Error al cargar la `DLL`.<br>La función `DLL` a eliminar no existe.<br>Función no encontrada en la `DLL`. |

Por ultimo se presenta el componente de código en su totalidad, incluyendo las clases y estructuras de datos diseñadas:

**Figura 22**

*Diagrama de clases del interprete*



### *5.2.5. Gestión del Estado Durante el Proceso de Interpretación*

Hasta este punto se ha expuesto el proceso de interpretación y construcción de las expresiones matemáticas, dando una breve explicación del cómo se almacenan los datos e información a lo largo del proceso.

El diseño del proceso parte de un único tipo de entrada, correspondiente a un dato (expresión matemática almacenada en forma de una cadena o *String*) el cual deberá ser tratado, para posteriormente ser representado y evaluado. El proceso se puede resumir de la siguiente manera:

1. **Ingreso del dato:** La cadena o expresión ingresada representa un dato inicial para la computadora, ya que para nosotros la cadena es información representada de forma matemática.

2. **Segmentación de la cadena:** La cadena o expresión es segmentada según toda la sintaxis y gramáticas matemáticas necesarias, esto permite desglosar la cadena de forma coherente a la información representada en la expresión.

3. **Construcción del árbol de sintaxis:** Este proceso depende de la lista de tokens y tipos, lo cuales representan un conjunto de datos que pueden ser transformados en un árbol de sintaxis que representa la información contenida en la cadena o expresión.

4. **Revisión de la estructura de datos obtenida:** Una vez representada la información de la cadena, es necesario revisar que se cumplan igualmente algunas reglas gramaticales como lo son el correcto uso de los operadores comparativos, ademas de facilitar la optimización del árbol de sintaxis.

5. **Construcción evaluable de la expresión:** Por ultimo se construye una representación de cada expresión matemática en base al árbol de sintaxis construido.

Esta generalización del proceso como cualquier otro, requiere de un estado, el cual representa los datos e información necesarios en la construcción de la expresión interpretada, generalmente almacenadas dentro de las propias clases, algo que dificulta la modularidad del proyecto, ya que de esta forma las clases encargadas del proceso quedan sujetas al estado, requiriendo de múltiples instancias de diferentes clases para realizar el mismo proceso. Lo anterior plantea lo siguiente:

1. **Modularidad:** Cada componente realiza un proceso aislado e independiente, lo que implica que recibe un conjunto de datos que serán transformados en información o datos, necesarios para otro componente o proceso sin que afecte el estado interno del componente (conjunto de clases que realizan una tarea).

2. **Estado:** El estado representa la evolución del proceso en el tiempo, al que se ha sometido cada expresión para interpretarla. El estado se almacena en una clase aislada, con el objetivo de reutilizar la misma instancia de cada componente.

Bajo este diseño e implementación se logra construir un proceso que puede ser reutilizado y llevado a computación en paralelo de forma sencilla, ya que el estado del proceso

es aislado del propio proceso, el cual es modificado poco a poco por cada uno de los componentes, sin afectar a los componente encargados, permitiendo que cada hilo cuente con un estado propio e independiente, al tiempo que emplean la misma instancia del interprete sin afectar el proceso.

Para lograr este objetivo, se diseña la clase **BuilderTree** encargada de almacenar todos los datos e información obtenida en etapa de interpretación, junto a una registro (*Record* en Delphi) auxiliar el cual es empleado en el proceso encargado de construir la cadena de tokens junto a sus tipos, que posteriormente son transferidos a **BuilderTree**.

De esta forma el estado del proceso se aísla de la lógica del proceso, permitiendo optimizar tanto el uso de memoria, como facilitando el proceso de paralelo, el cual requiere que dos hilos o más, no accedan y modifiquen los mismos datos al tiempo, lo que se logra con este planteamiento; es de resaltar que Delphi provee una serie de elementos que permiten implementar computación en paralelo de forma fácil, contando una función *for* paralelizada, la cual requiere que los proceso llevados acabo no caigan en problemas de concurrencia al modificar áreas de memoria en común.

Es de esta manera que se presenta la arquitectura modular tanto del proceso como del estado mismo, logrando cumplir completamente con una arquitectura modular, uno de los requerimientos planteados para el proyecto.

## 5.3. Proceso de Construcción de las Expresiones

En este apartado se explorara el proceso de construcción de las expresión por medio de ejemplos, para esto se emplearán las pruebas diseñadas durante las etapas de pruebas, las cuales comprueban el componente de código; junto a esto se acompaña de ilustraciones que faciliten el proceso de presentación de resultados, el cual se aborda en gran parte a lo largo de esta sección.

### *5.3.1. Ejemplo 1:* $a + b - c + d$

El proceso de codificación de la expresión inicia con la tokenización y tipado de la cadena o expresión, para observar el resultado entregado por el componente se hace uso de la prueba *TestExpressionBuilderAPD*, el cual recibe como parámetro una cadena que contiene la representación de la expresión matemática, junto al número de veces que realizara el proceso de tokenización (prueba de carga), de esta forma se obtiene:

**Figura 23**

*Resultado de la prueba TestExpressionBuilderAPD al evaluar:* $a + b - c + d$

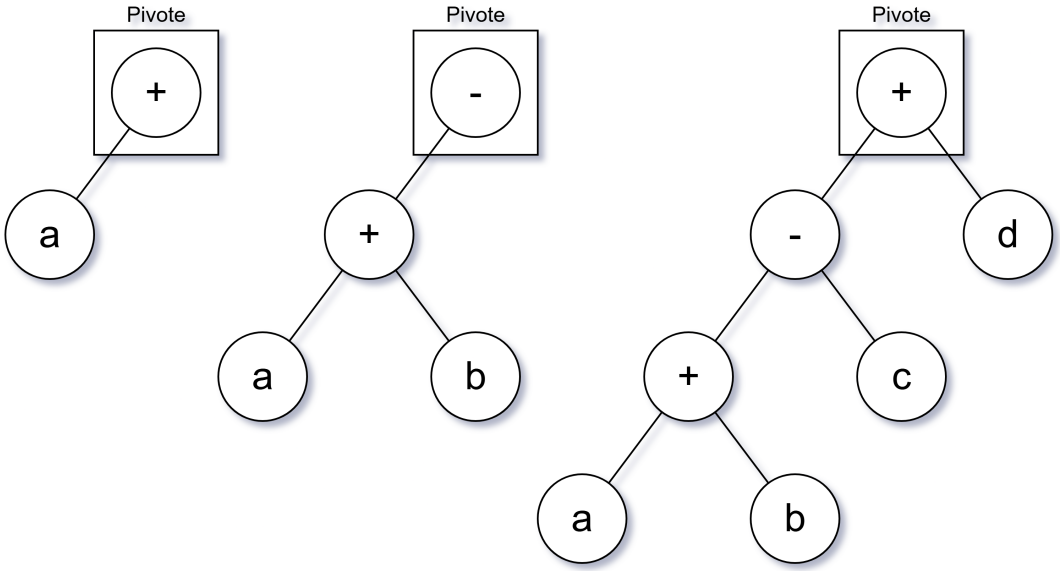
```
TestExpressionBuilderAPD --> Lexical Analysis - Execution Time
Iterations: 1000000
String: a+b-c+d
Tokens: [ "a"->1, "+"->5, "b"->1, "-"->5, "c"->1, "+"->5, "d"->1, "□"->11,  ]
Error Trace:
Error Report:
Total Time (ms): 2534
Average execution time (µs): 1,66
Total accumulated time (µs): 1663244,10
```

En el siguiente ejemplo se presenta el proceso de construcción del árbol de sintaxis, se realiza token a token, por lo que su complejidad es lineal, ya que realiza n iteraciones, donde n es equivalente a la cantidad de tokens encontrados en la expresión matemática. Iterando sobre la cadena el componente encargado de ordenar el árbol *SyntaxTree*, trabaja unicamente con los elementos cuyo tipo sea superior a 0 y 1, ya que estos tipos corresponden a los tipos de los números y variables, respectivamente.

La Figura 24 ilustra cómo el componente opera internamente, para esta expresión le toma 8 iteraciones construir el árbol, ya que el ultimo token en las cadenas siempre es un epsilon, es por esta razón que sera n elementos mas un tokens el numero de iteraciones que se requieren en la construcción del árbol de sintaxis, el proceso se presenta es la siguiente imagen:

**Figura 24**

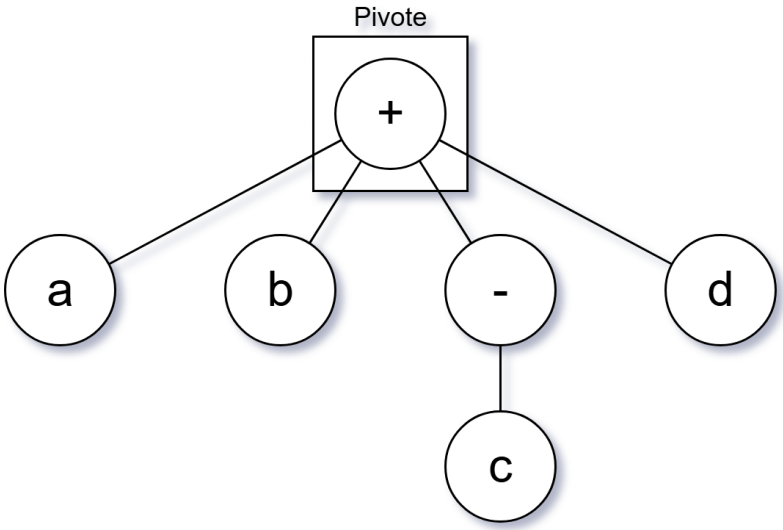
*Árbol de la expresión:* $a + b - c + d$




El proceso de optimización del árbol da como resultado la siguiente ilustración:

**Figura 25**

*Árbol optimizado de la expresión* $a + b - c + d$



Al utilizar la prueba *TestExpressionBuilderAST* se obtiene el siguiente resultado:

**Figura 26**

*Resultado de la prueba TestExpressionBuilderAST al evaluar:* $a + b - c + d$

```
TestExpressionBuilderAST --> Building Symtax Tree - Execution time
Iterations: 1000000
Hits: 1000000
String: a+b-c+d
Total Time (ms): 5972
Average execution time (micro s): 5,94
Total accumulated time (micro s): 5943932,00
Source: a+b-c+d
Error Trace:
Error Report:
Graph:
- ->  c
+ ->  a b - d

ReBuild Source:
Error Report:
Graph without parentheses:
- ->  c
+ ->  a b - d
```

En muchos casos se puede observar que el grafo obtenido y el grafo sin paréntesis son idénticos, debido a una optimización que se realizo en el componente encargado de ordenar el árbol. Se consigue esta mejora durante un proceso de mejora del algoritmo, donde se encuentra un forma simple y optima de ordenar el árbol (operadores de adición y sustracción) la cual se implementa y funciona, respondiendo de forma positiva durante todas las pruebas, esta aclaración se hace debido a que el grafo sin paréntesis muestra el grafo optimizado.

Por ultimo se puede evaluar esta expresión empleando la prueba de carga *TestMath-LeafsExpressionsAddExpression*, se debe tener en cuenta que esta prueba no contempla el uso de variables, debido a la complejidad que agregaría a la propia prueba y a que no se observaría ninguna diferencia, ya que números y variables tiene la misma representación, un conjunto de punteros de valores punto flotante. El resultado de esta prueba se presenta continuación:

**Figura 27**

*Resultado de la prueba TestMathLeafsExpressionsAddExpression al evaluar:* $1 + 2 - 3 + 4$

```
TestMathLeafsExpressionsAddExpression --> Expression evaluation tests
Evaluate: 4,000000000000000000000
# Error Report:
Iteraciones: 1000000
Aciertos: 0
String: 1+2-3+4
Total Time (ms): 101
Average execution time (micro s): 0,08
Total accumulated time (micro s): 78516,00
```

En el ejemplo nunca se presenta el orden de evaluación pero este siempre esta presente, ya que corresponde a la presentación lineal del propio grafo de una forma particular, ya que para obtenerse se requiere recorrer el grafo en profundidad y amplitud hacia la derecha, partiendo del nodo raíz, para este proceso se emplea una pila la cual almacena lo nodos en el árbol que cuenten con hijos, una vez se finaliza el proceso, la lista obtenida se invierte obteniendo el orden de evaluación. Este proceso es mas simple al usar una función recursiva pero menos eficiente debido al exceso de llamas a la función consecuente a la profundidad que puede llegar a tener el árbol y a los desbordamientos en memoria del procesador.

### *5.3.2. Ejemplo 2:* $10 - if(5 > 3, 3, 5)$

Como resultado del análisis Léxico se obtiene la figura presentada a continuación, en esta se muestra el resultado de la prueba *TestExpressionBuilderAPD* donde la cadena que es evaluada contiene la función if, de esta manera se presenta el manejo de funciones, esta en particular cuenta con una complejidad mayor que otras funciones por lo que extiende la abstracción de la representación de funciones, pero ilustra perfectamente el proceso.

**Figura 28**

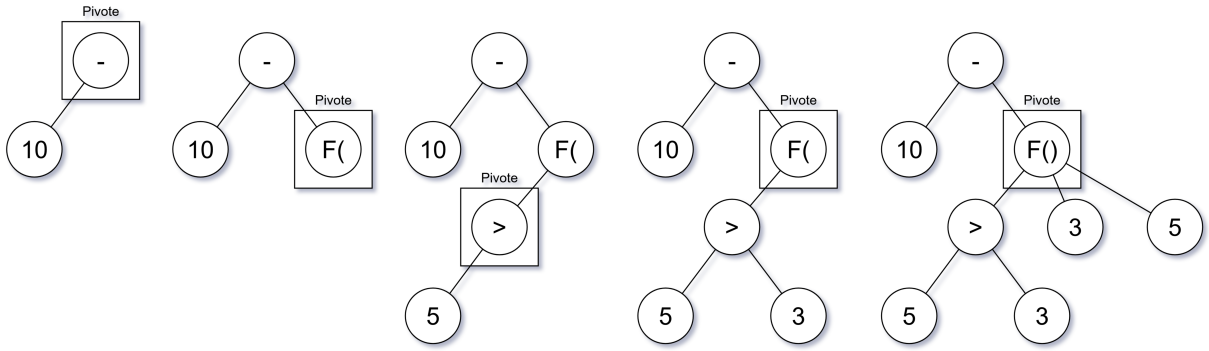
*Resultado de la prueba TestExpressionBuilderAPD al evaluar:* $10 - if(5 > 3, 3, 5)$

```
TestExpressionBuilderAPD --> Lexical Analysis - Execution Time
Iterations: 1000000
String: 10 - if(5>3, 3, 5)
Tokens: [ "10"->0, "-"->5, "if"->2, "("->7, "5"->0, ">"->4, "3"->0, ","->8,
"3"->0, ","->8, "5"->0, ")"->7, "□"->11,  ]
Error Trace:
Error Report:
Total Time (ms): 3892
Average execution time (µs): 2,93
Total accumulated time (µs): 2933545,00
```

El proceso de construcción del árbol de sintaxis se ilustra en la siguiente imagen:

**Figura 29**

*Árbol de la expresión:* $10 - if(5 > 3, 3, 5)$




El resultado de la prueba *TestExpressionBuilderAST* presenta como internamente el componente representa las funciones, mostrando que estas cuentan con mas información que un operador y operación. El resultado de la prueba se presenta en la figura siguiente:

**Figura 30**

*Resultado de la prueba TestExpressionBuilderAST al evaluar:* $10 - if(5 > 3, 3, 5)$

```
TestExpressionBuilderAST --> Building Symtax Tree - Execution time
Iterations: 1000000
Hits: 1000000
String: 10 - if(5>3, 3, 5)
Total Time (ms): 9893
Average execution time (micro s): 9,86
Total accumulated time (micro s): 9864351,70
Source: 10 - if(5>3, 3, 5)
Error Trace:
Error Report:
Graph:
> ->  5 3
if - F() ->  > 3 5
- ->  if - F()
+ ->  10 -

ReBuild Source:
Error Report:
Graph without parentheses:
> ->  5 3
if - F() ->  > 3 5
- ->  if - F()
+ ->  10 -
```

Por ultimo se realiza y se presenta en la imagen a continuación, el resultado arrojado por la prueba *TestMathLeafsExpressionsAddExpression* encargada unicamente de probar la evaluación de las expresiones interpretadas.

**Figura 31**

*Resultado de la prueba TestMathLeafsExpressionsAddExpression al evaluar:* $10 - if(5 > 3, 3, 5)$

```
TestMathLeafsExpressionsAddExpression --> Expression evaluation tests
Evaluate: 7,000000000000000000000
# Error Report:
Iteraciones: 1000000
Aciertos: 0
String: 10 - if(5>3, 3, 5)
Total Time (ms): 96
Average execution time (micro s): 0,08
Total accumulated time (micro s): 75995,80
```

### *5.3.3. Ejemplo 3:* 1&0&1&1

Como resultado del análisis Léxico se obtiene la figura presentada a continuación, en esta se muestra el resultado de la prueba *TestExpressionBuilderAPD* donde la cadena que es evaluada contiene operadores lógicos, de esta manera se presenta el manejo de la lógica booleana integrada en el interprete, esta en particular cuenta con una complejidad mayor que otros operadores ya que se incluye la capacidad de identificar elementos booleanos (0,1,false y true independiente del uso de minúsculas o mayúsculas). Es de esta manera que se obtiene:

**Figura 32**

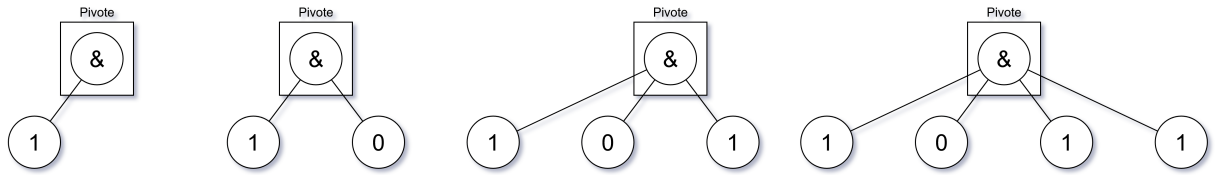
*Resultado de la prueba TestExpressionBuilderAPD al evaluar:* 1&0&1&1

```
TestExpressionBuilderAPD --> Lexical Analysis - Execution Time
Iterations: 1000000
String: 1 & 0 & 1 & 1
Tokens: [ "1"->0, "&"->9, "0"->0, "&"->9, "1"->0, "&"->9, "1"->0, "□"->11,  ]
Error Trace:
Error Report:
Total Time (ms): 2630
Average execution time (µs): 1,73
Total accumulated time (µs): 1732732,90
```

El proceso de construcción del árbol de sintaxis se ilustra en la siguiente imagen:

**Figura 33**

*Árbol de la expresión:* 1&0&1&1




El resultado de la prueba *TestExpressionBuilderAST* presenta como internamente el componente representa los operadores lógicos, aunque no se puede apreciar, el componente realiza un proceso de revision de tipos indicando si se están empleando los tipos correctos. El resultado de la prueba se presenta en la figura siguiente:

**Figura 34**

*Resultado de la prueba TestExpressionBuilderAST al evaluar:* 1&0&1&1

```
TestExpressionBuilderAST --> Building Symtax Tree - Execution time
Iterations: 1000000
Hits: 1000000
String: 1 & 0 & 1 & 1
Total Time (ms): 4924
Average execution time (micro s): 4,90
Total accumulated time (micro s): 4898128,50
Source: 1 & 0 & 1 & 1
Error Trace:
Error Report:
Graph:
& ->  1 0 1 1

ReBuild Source:
Error Report:
Graph without parentheses:
& ->  1 0 1 1
```

Por ultimo se realiza el proceso de evaluación empleando la prueba *TestMathLeafsExpressionsAddExpression* y se presentan los resultados en la imagen a continuación:

**Figura 35**

*Resultado de la prueba TestMathLeafsExpressionsAddExpression al evaluar:* 1&0&1&1

```
TestMathLeafsExpressionsAddExpression --> Expression evaluation tests
Evaluate: 0,000000000000000000000
# Error Report:
Iteraciones: 1000000
Aciertos: 0
String: 1 & 0 & 1 & 1
Total Time (ms): 82
Average execution time (micro s): 0,06
Total accumulated time (micro s): 60245,80
```

Cabe destacar que en caso de requerir emplear el resultado de evaluar una expresión matemática como elemento booleano, se cuenta con la función *Bool(x)* donde esta recibe un numero o variable y lo convierte en booleano compatible con el proceso.

### *5.3.4. Ejemplo 4:* $\{3 - 2 * [1 + 2 - (2^2 - 3)]\}$

Como resultado del análisis Léxico se obtiene la figura presentada a continuación, en esta se muestra el resultado de la prueba *TestExpressionBuilderAPD* donde la cadena que es evaluada contiene operadores aritméticos y muestra el uso de delimitadores, de esta manera se presenta el manejo de los mismos. A continuación se presentan el resultado de la prueba:

**Figura 36**

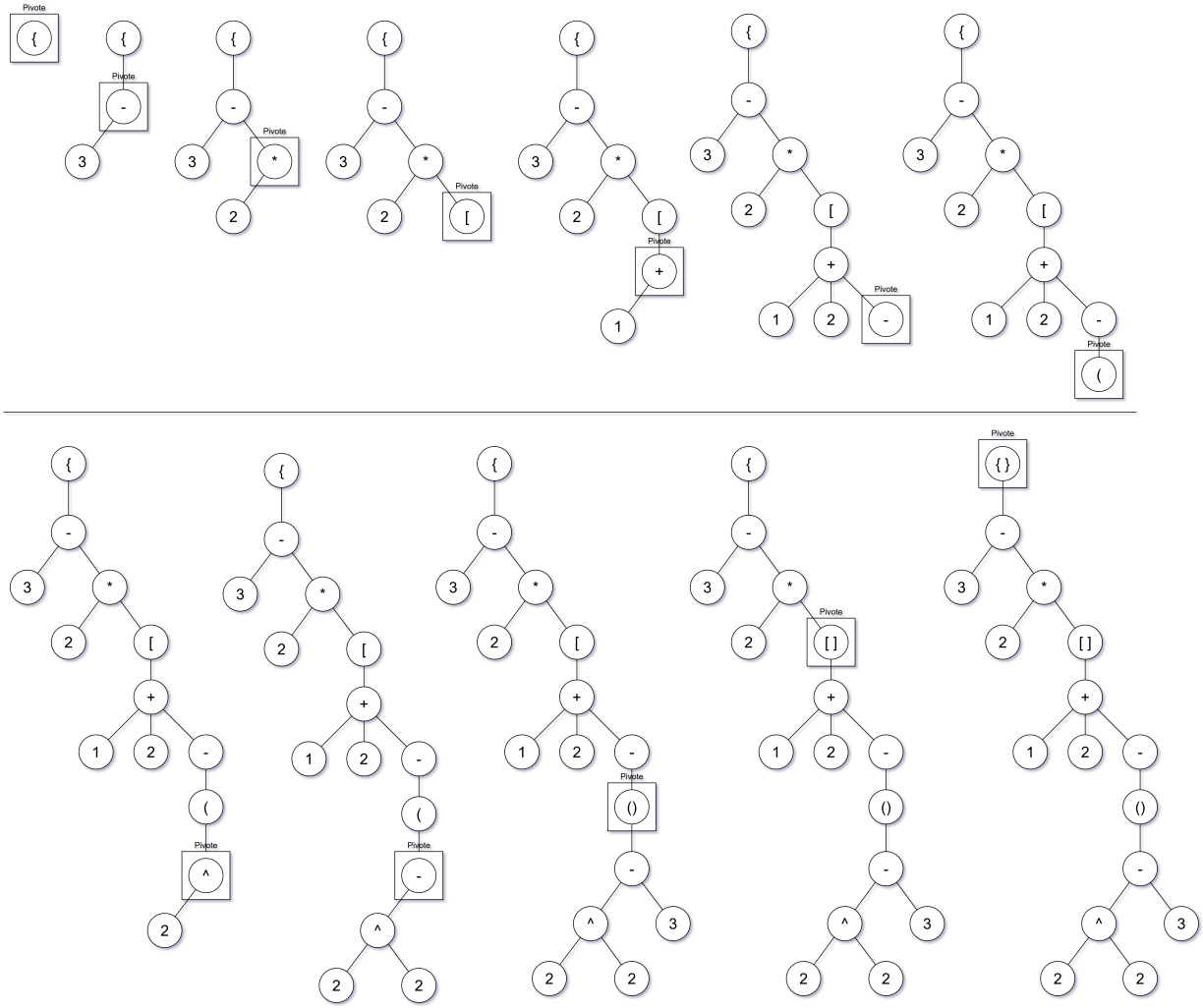
*Resultado de la prueba TestExpressionBuilderAPD al evaluar:* $\{3 - 2 * [1 + 2 - (2^2 - 3)]\}$

```
TestExpressionBuilderAPD --> Lexical Analysis - Execution Time
Iterations: 1000000
String: {3-2*[1+2 - ( 2^2 - 3)]}
Tokens: [ "{"->3, "3"->0, "-"->5, "2"->0, "*"->5, "["->3, "1"->0, "+"->5, "2"->0,
"-"->5, "("->3, "2"->0, "^"->5, "2"->0, "-"->5, "3"->0, ")"->3, "]"->3, "}"->3,
"□"->11,  ]
Error Trace:
Error Report:
Total Time (ms): 4094
Average execution time (µs): 3,12
Total accumulated time (µs): 3124117,00
```

El proceso de construcción del árbol de sintaxis se ilustra en la siguiente imagen:

**Figura 37**

*Árbol de la expresión:* $\{3 - 2 * [1 + 2 - (2^2 - 3)]\}$




El resultado de la prueba *TestExpressionBuilderAST* presenta en este caso en especifico el manejo interno de los delimitadores y como son empleados en la construcción del árbol de sintaxis, marcan la creación de un sub-árbol, cuya valor sera el resultado de su evaluación, concatenado al resto de la estructura. El componente respeta el orden y uso de los delimitadores representando adecuadamente su información. A continuación se presenta el resultado de la prueba:

**Figura 38**

*Resultado de la prueba TestExpressionBuilderAST al evaluar:* $\{3 - 2 * [1 + 2 - (2^2 - 3)]\}$

```
String: {3-2*[1+2 - ( 2^2 - 3)]}
Total Time (ms): 13071
Average execution time (micro s): 13,04
Total accumulated time (micro s): 13044069,30
Source: {3-2*[1+2 - ( 2^2 - 3)]}
Error Trace:
Error Report:
Graph:
^ ->  2 2
- ->  3
+ ->  ^ -
() ->  +
- ->  ()
+ ->  1 2 -
[] ->  +
* ->  2 []
- ->  *
+ ->  3 -
{} ->  +

ReBuild Source:
Error Report:
Graph without parentheses:
^ ->  2 2
- ->  3
+ ->  ^ -
- ->  +
+ ->  1 2 -
* ->  2 +
- ->  *
+ ->  3 -
```

Por ultimo se realiza el proceso de evaluación empleando la prueba *TestMathLeafsEx-*

*pressionsAddExpression* y se presentan los resultados en la imagen a continuación:

**Figura 39**

*Resultado de la prueba TestMathLeafsExpressionsAddExpression al evaluar:* $\{3 - 2 * [1 + 2 - (2^2 - 3)]\}$

```
TestMathLeafsExpressionsAddExpression --> Expression evaluation tests
Evaluate: -1,000000000000000000000
# Error Report:
Iteraciones: 1000000
Aciertos: 0
String: {3-2*[1+2 - ( 2^2 - 3)]}
Total Time (ms): 127
Average execution time (micro s): 0,11
Total accumulated time (micro s): 105738,70
```

## 5.4. Pruebas

En esta sección se abordaran las diferentes pruebas a las que se sometió el componente de interpretación, este proceso marco los puntos de evolución del proyecto, marcando el paso al siguiente ciclo, es por este motivo que se presentarán en orden de creación junto con su explicación. De esta forma se pretende dar soporte a los resultados que se definen como obtenidos.

Para el correcto desarrollo de las pruebas se definieron varios conjunto de datos, con el fin de comprobar si el componente de código desarrollaba el proceso de forma correcta, cada uno de estos conjunto esta compuesto de un conjunto de expresiones matemáticas escritas correctamente e incorrectamente, esto permitió contrastar los resultado y definirnos como positivos o negativos. En total se llegan a probar cerca de seiscientas expresiones matemáticas algunas de estas en un estado intermedio de su representación, como lo seria su estructura de árbol, contándose ademas con expresiones matemáticas escritas manualmente en forma de función con el objetivo de comprobar el proceso de evaluación.

Cada componente contó con un conjunto de pruebas, las cuales se presentaran por medio de la siguiente tabla:

**Tabla 8**

*Resumen de pruebas por componente*

| Prueba (nombre y descripción) | Tipo | Importancia |
|---|---|---|
| **Analizador léxico** | | |
| **Caracteres ASCII:** Verifica que todos los caracteres usados en las pruebas pertenezcan al conjunto ASCII permitido por el autómata. *(TestExpressionBuilderIsASCII)* | Unitaria | Media |
| **Sub-alfabetos de transición y estados:** Comprueba los caracteres admitidos por las transiciones de cada estado. *(TestTransitionSearchCharacter)* | Unitaria | Crítica |
| **Tokenización:** Evalúa el proceso de tokenización completo, incluyendo pruebas de carga. *(TestExpressionBuilderAPD)* | Integración y Carga | Alta |
| **Evaluación de expresiones:** Comprueba el proceso de tokenización y tipado de n expresiones. *(TestExpressionBuilderAPDNTokens)* | Integración | Alta |
| **Analizador sintáctico** | | |
| **Construcción del árbol de sintaxis:** Presenta visualmente el proceso de construcción del árbol. *(TestExpressionBuilderAST)* | Integración y Carga | Alta |
| **Evaluación del proceso de construcción:** Valida el proceso de construcción del árbol de sintaxis de n expresiones. *(TestExpressionBuilderASTNFunctions)* | Integración | Alta |
| **Pruebas de fugas de memoria:** Verifican la ausencia de fugas en el proceso de construcción hasta este punto. *(TestMemoryLeak, TestMemoryLeakNFunctions)* | Sistema | Crítica |
| **Interfaz de usuario** | | |
| **Orden de Evaluación:** Comprueba que el orden de evaluación creado sea el esperado. *(TestEBOrderNodes)* | Unitaria | Alta |
| **Creación de constantes normales y globales:** Verifica que el proceso de creación sea valido. *(TestMathLeafsExpressionsConstants, ConstantsGlobals)* | Unitaria | Media |
| **Eliminación de constantes normales y globales:** Verifica que el proceso de eliminación sea valido. *(TestMathLeafsExpressionsDeletesConstants, DeletesConstantsGlobals)* | Unitaria | Media |
| **Cambio de nombre de constantes normales y globales:** Evalúa el renombramiento. *(TestMathLeafsExpressionsSetNameConstants, SetNameConstantsGlobals)* | Unitaria | Media |
| **Cambio de valor de constantes normales y globales:** Evalúa la modificación de valores. *(TestMathLeafsExpressionsSetValueConstants, SetValueConstantsGlobals)* | Unitaria | Media |
| **Construcción de una expresión usando ExpressionBuilder:** Valida la construcción y evaluacion de la expresion. *(TestExpressionBuilderBuildExpression)* | Unitaria | Alta |
| **Revisión de elementos:** Verifica la existencia de variables antes de su uso. *(TestExpressionBuilderEElementsReview)* | Unitaria | Crítica |
| **Agregar expresión:** Evalúa que el almacenamiento y la evaluación sean correctas. *(TestMathLeafsExpressionsAddExpression)* | Integración y Carga | Alta |
| **Almacenamiento y evaluación de expresiones:** Verifica que la carga y evaluación de n expresiones corresponda a lo esperado. *(TestMathLeafsExpressionsBuildNExpression)* | Sistema | Crítica |
| **Agregar función puntero:** Comprueba la integración y uso en el intérprete. *(TestMathLeafsExpressionsAddFunction)* | Integración | Alta |
| **Agregar función DLL:** Verifica que se puedan cargar funciones desde DLL. *(TestMathLeafsExpressionsAddFunctionDll)* | Integración | Alta |

### *5.4.1. Caracteres ASCII*

Esta prueba se encarga de corroborar que los caracteres manejados dentro de Delphi para las pruebas como empleados por el propio autómata, corresponden al alfabeto definido. Para cumplir este propósito se define el siguiente proceso:

1. Se emplea un diccionario definido manualmente, el cual contiene unicamente el sub-alfabeto permitido por cada una de las diferentes transiciones.

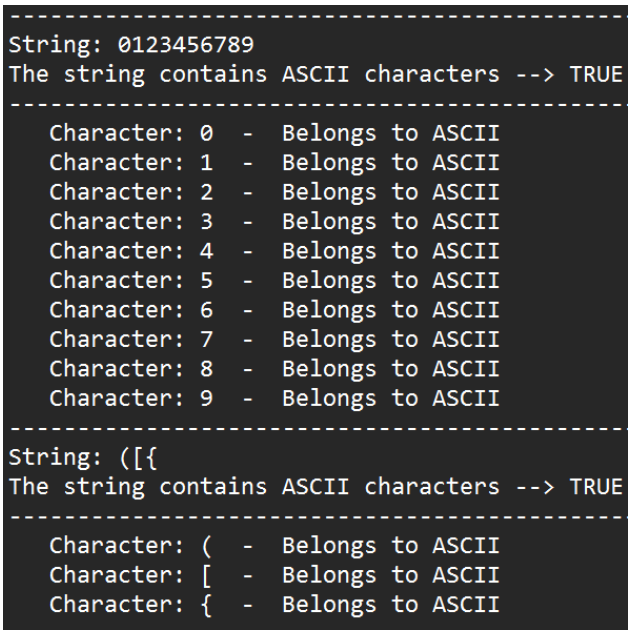
2. Se comprueba que los caracteres ASCII empleados para las pruebas son reconocidos como ASCII.

3. Se comprueba que al evaluar otros caracteres no pertenecientes a ASCII se reconocen como inválidos.

A continuación se presenta parte de los resultados. En caso de encontrar un error, se indicara, por ello su ausencia muestra que la revisión cumple con su objetivo.

**Figura 40**

*Resultados de las pruebas TestExpressionBuilderIsASCII*



### *5.4.2. Sub-alfabetos de Transición y Estados*

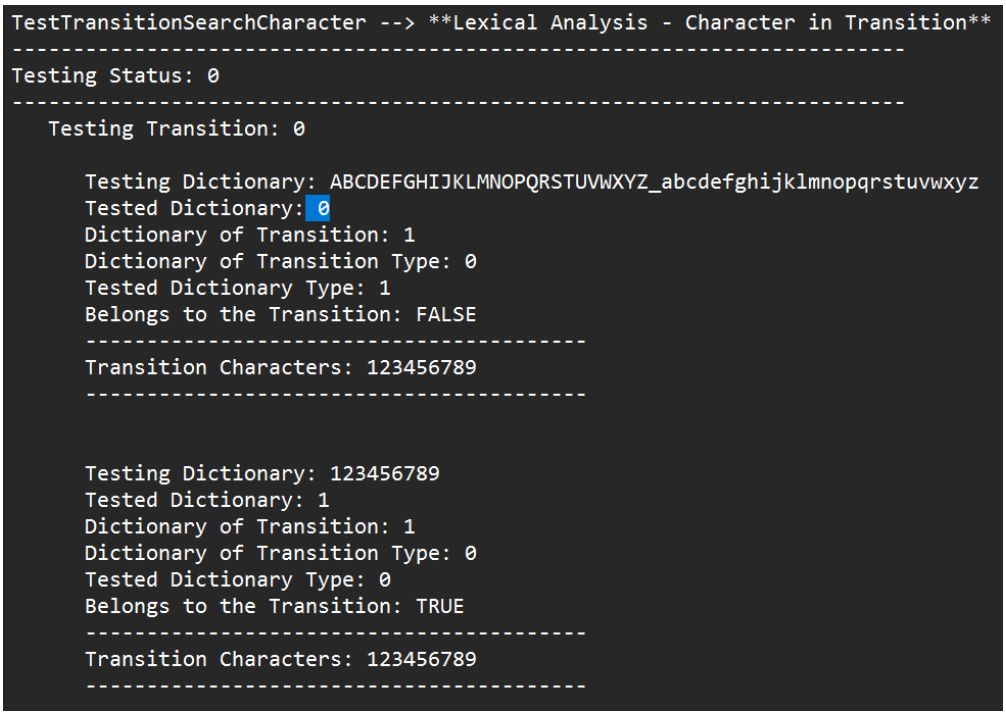
Esta prueba extiende los resultados y revisión de la anterior prueba, debido a que es necesario corroborar que los sub-conjuntos de alfabetos definidos para cada transición, reconozcan unicamente el sub-conjunto o sub-alfabeto definido como valido para transitar, comprobando de esta manera que el autómata diseñado trabaja conforme a la teoría de autómatas, reconociendo unicamente por transición un sub-conjunto de caracteres especifico, en base al conjunto universal definido para el autómata (alfabeto). Para cumplir este propósito se define el siguiente proceso:

1. Emplea eL autómata diseñado para el proceso de tokenización en la revisión de las transiciones.
2. Se emplea un diccionario definido manualmente, el cual contiene unicamente el sub-alfabeto permitido por cada una de las diferentes transiciones.
3. Se prueba cada uno de los caracteres en el diccionario definido.
4. Se prueban todos los sub-alfabetos definidos en cada una de las transiciones.
5. Se indica el resultado de cada evaluación en las revisiones 2 y 3 en un archivo de texto.

Un fragmento del archivo resultante de la prueba se presenta a continuación:

**Figura 41**

*Resultados de las pruebas TestTransitionSearchCharacter*




La prueba retorna la siguiente información:

1. Nombre de la prueba.
2. Estado que se prueba ”Testing Status: # q# → estado.
3. Transición evaluada.
4. Datos de la transición evaluada, donde se presenta, el diccionario de la transición, diccionario probado, identificador del diccionario que se prueba y su tipo, tipo del diccionario que se emplea para evaluar y si pertenece el diccionario a la transición junto a sus caracteres.
5. este proceso se realiza con todos los estados y todas sus transiciones, probando diferentes diccionarios ó alfabetos.

### *5.4.3. Tokenización*

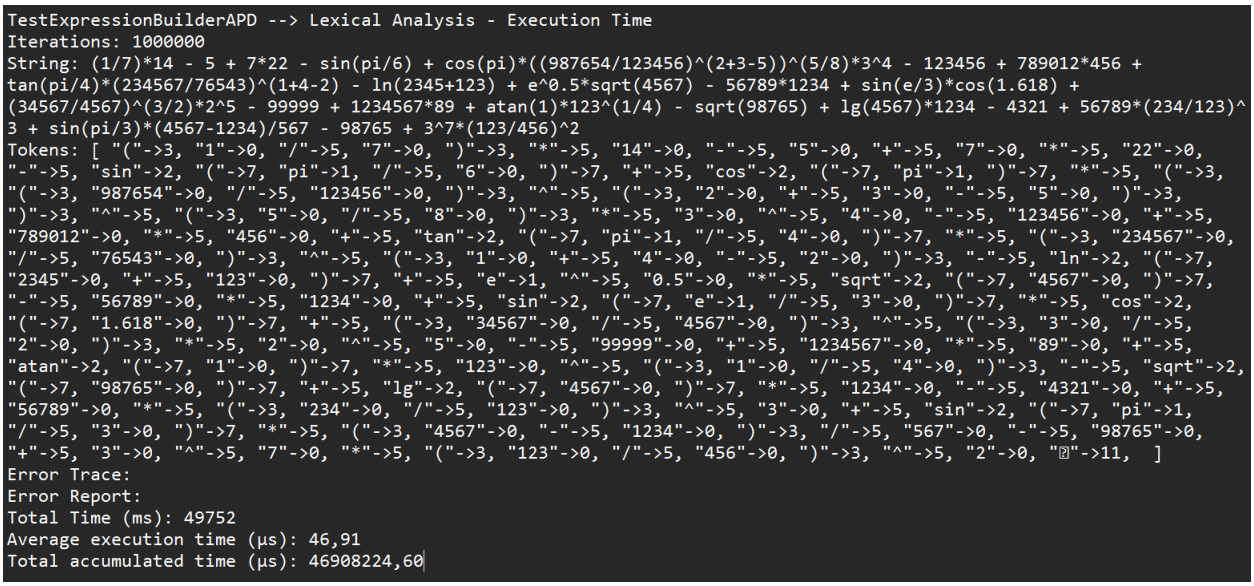
Esta prueba se diseña con dos objetivos, probar una cadena de forma aislada lo cual permite observar el resultado del proceso y depurar el mismo; por otro lado permite realizar pruebas de carga, al repetir el proceso n veces y medir el costo en tiempo que requiere en promedio tokenizar una cadena, siendo una prueba clave para continuar con las demás partes del proceso de interpretación. Para cumplir este propósito se define el siguiente proceso:

1. Crear una instancia del componente encargado del análisis léxico.
2. Recibe una sola expresión, permite evaluar capacidades por separado al probar casos específicos.
3. Se evalúa el proceso de tokenización, retornando un reporte y seguimiento del error según el proceso se acepte o no.
4. Se puede realizar una prueba de carga al definir la cantidad de iteraciones o evaluaciones que se desee probar. Retorna el tiempo promedio y total en $\mu$ y mili segundos.

Un fragmento del archivo resultante de la prueba se presenta a continuación:

**Figura 42**

*Resultados de las pruebas TestExpressionBuilderAPD*




La prueba retorna la siguiente información:

1. Nombre de la prueba.
2. Iteraciones realizadas.
3. Expresión procesada.
4. Lista de tokens y sus tipos.
5. Seguimiento del error.
6. Reporte o definición del error.
7. Tiempo total en mili segundos.
8. Tiempo promedio por iteración en $\mu$ segundos.
9. Tiempo total en $\mu$ segundos.

### *5.4.4. Evaluación de Expresiones*

Esta prueba permite evaluar de forma general el componente encargado de la tokenización, al pasar le una lista de expresiones, las cuales contiene un conjunto de expresiones correctas y otras incorrectas, revisadas manualmente. De esta forma se buscó errores en el proceso, al probar una gama amplia de expresiones, siendo importante resaltar el uso de chat-GPT en la creación de expresiones matemáticas aleatorias, lo que facilito el proceso de búsqueda de errores, esto permitió probar un mayor numero de casos diferentes, evaluando la capacidad de reconocer distintos aspectos contenidos en cada una de estas expresiones matemáticas. Para cumplir este propósito se define el siguiente proceso:

1. Crear una instancia del componente encargado del análisis léxico.
2. Recibe una lista de expresiones, permite evaluar las capacidades generales de tokenización.
3. Genera un reporte con información del proceso de construcción del listado de tokens y tipos de cada expresión.

Se presentan fragmentos del archivo resultante de la prueba:

**Figura 43**

*Resultados de las pruebas TestExpressionBuilderAPD conclusión del test*

```
#211
Tokens: [ "a"->1, ">"->4, "f"->1, "<="->4, "r"->1, "□"->10,  ]
Rebuild: a>f<=r.
Source:  a>f<=r.
Equals: TRUE

#212
Tokens: [ "asd"->1, "*"->5, "r"->1, "<="->4, "a"->1, ">="->4, "c"->1, "□"->10,  ]
Rebuild: asd*r<=a>=c.
Source:  asd*r<=a>=c.
Equals: TRUE

Total Hits: 113
Total Failures: 100
```

**Figura 44**

*Resultados de las pruebas TestExpressionBuilderAPD en el manejo de errores*

```
#184
Tokens: [ " -->(<-- "->11, "asd"->1, " -->,<-- "->11, "-"->6, "998"->0, "+"->5, " -->(<-- "->11, "1"->0, " -->,<--
"->11, "□"->10,  ]
Rebuild:  -->(<-- asd -->,<-- -998+ -->(<-- 1 -->,<-- .
Source:  (asd,-998+(1,.
Equals: FALSE

#185
Tokens: [ "c"->2, " -->(<-- "->11, " -->,<-- "->11, " -->,<-- "->11, " -->)<-- "->11, "□"->10,  ]
Rebuild: c -->(<--  -->,<--  -->,<--  -->)<-- .
Source:  c(,,).
Equals: FALSE

#186
Tokens: [ "f"->2, "("->7, "a"->1, ","->8, "p"->1, ")"->7, "+"->5, "("->3, "r"->1, " -->,<-- "->11, "s"->1, ")"->3,
"□"->10,  ]
Rebuild: f(a,p)+(r -->,<-- s).
Source:  f(a,p)+(r,s).
Equals: FALSE

#187
Tokens: [ " -->+<-- "->11, "1"->0, "*"->5, "2"->0, "□"->10,  ]
Rebuild:  -->+<-- 1*2.
Source:  +1*2.
Equals: FALSE
```

La prueba retorna la siguiente información:

1. Nombre de la prueba.
2. Evaluación de cada expresión, la cual cuenta con: lista de tokens y sus tipos, Reconstrucción de la cadena, expresión original sin espacios (se usa una función propia de Delphi) y por ultimo si la reconstrucción es igual a la cadena original.
3. Conteo total de aciertos.
4. Conteo total de fallos.

### *5.4.5. Construcción del Árbol de Sintaxis*

Esta prueba permite evaluar la construcción del árbol de sintaxis, presentándolo en sus diferentes etapas, a la vez que evalúa si contiene errores gramaticales y mostrándolos. Adicionalmente se puede realizar una prueba de carga al hacer el proceso requerido en la construcción el árbol de sintaxis, n veces. Para cumplir este propósito se define el siguiente proceso:

1. Crear una instancia del componente encargado de orquestar el proceso de construcción del árbol de sintaxis *SyntaxTree*, este realiza el proceso de tokenización para su funcionamiento.

2. Recibe una lista de expresión, permite evaluar capacidades generales en la construcción de árboles de sintaxis.

3. Genera un reporte de las pruebas llevadas acabo.

Se presentan fragmentos del archivo resultante de la prueba:

**Figura 45**

*Resultados de las pruebas TestExpressionBuilderAST*

```
TestExpressionBuilderAST --> Building Symtax Tree - Execution time
Iteraciones: 1000000
Aciertos: 1000000
String: (1/3)*3 - 1 + 3 * 15 - sin(3.1416/4) + cos(3.1416)
Total Time (ms): 16699
Average execution time (micro s): 16,67
Total accumulated time (micro s): 16669743,20
Source: (1/3)*3 - 1 + 3 * 15 - sin(3.1416/4) + cos(3.1416)
Error Trace:
Error Report:
Graph:
/ ->  1 3
() ->  /
* ->  () 3
- ->  1
* ->  3 15
/ ->  3.1416 4
sin - F() ->  /
- ->  sin - F()
cos - F() ->  3.1416
+ ->  * - * - cos - F()

ReBuild Source:
Error Report:
Graph without parentheses:
/ ->  1 3
* ->  / 3
/ ->  3.1416 4
sin - F() ->  /
- ->  1 sin - F()
* ->  3 15
cos - F() ->  3.1416
+ ->  * - * cos - F()
```

La prueba retorna la siguiente información:

1. Nombre de la prueba.
2. Iteraciones realizadas.
3. Total aciertos.
4. Expresión procesada.
5. Tiempo total de ejecución en mili segundo.
6. Tiempo promedio por expresión en mili segundo.
7. Tiempo total de ejecución en micro segundo.
8. Seguimiento del error y su reporte.
9. Grafo lineal.
10. Grafo lineal sin paréntesis.

### *5.4.6. Evaluación del Proceso de Construcción*

Esta prueba permite evaluar el comportamiento del componente encargado de la construcción del árbol de sintaxis, sobre un conjunto amplio de diferentes tipos de expresiones, para mejorar el alcance de esta prueba se emplean cadenas generadas con Chat-GPT en busca de casos no manejados y por ende, errores en el comportamiento esperado. Este proceso se complementa al realizar la construcción manual de cada una de los arboles de sintaxis, corroborando así la capacidad del componente. Para cumplir este propósito se define el siguiente proceso:

Se centro en realizar las siguientes actividades:

1. Crear una instancia del componente encargado de orquestar el proceso de construcción del árbol de sintaxis *SyntaxTree*.

2. Recibe una lista de expresiones diferentes a las empleadas para las pruebas de análisis léxico, permite evaluar una amplia gamas de casos.

3. Genera un reporte con información del proceso de construcción del cada árbol correspondiente a las diferentes expresiones en la lista de evaluación.

Se presentan fragmentos del archivo resultante de la prueba:

**Figura 46**

*Resultados de las pruebas TestExpressionBuilderASTNFunctions*

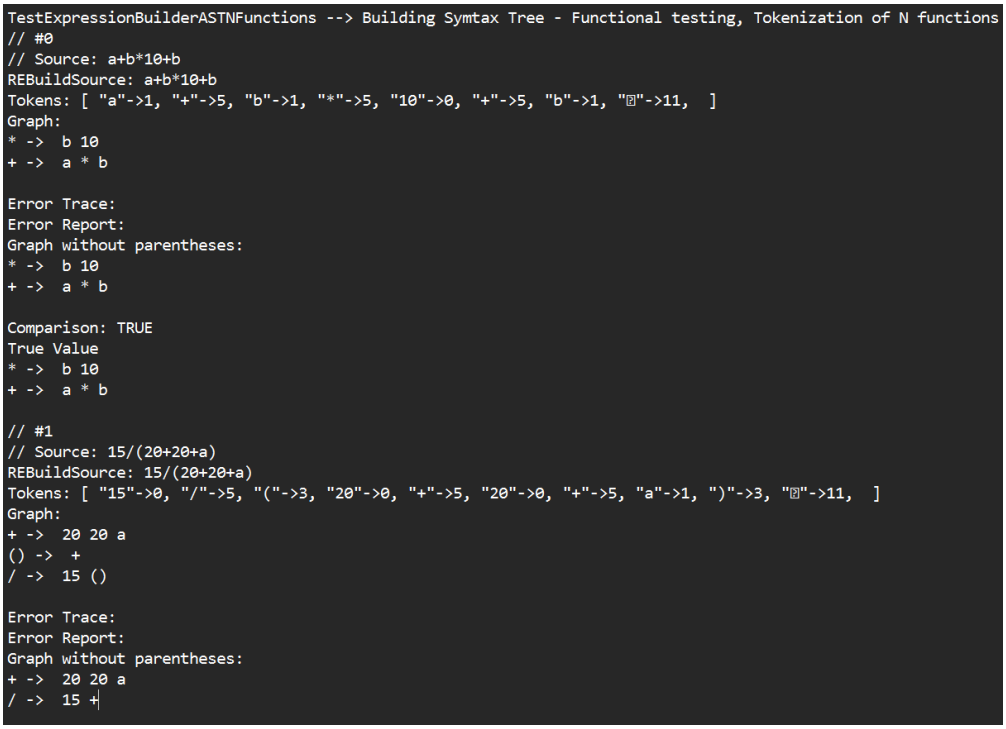


**Figura 47**

*Resultados de las pruebas TestExpressionBuilderASTNFunctionsconclusión del test*

```
// #150
// Source: if(TRUE, if("cadena", A, B), C)
Error Trace: if(TRUE,if( => " <= cadena => " <= ,A,B),C)
Error Report: { Errores:= [11101]: Caracteres invalidos detectados - Posicion del Error (13,20) }
Compare: FALSE
AST

// #151
// Source: if(123, 456, 789)
Error Trace: if( => 123 <= ,456,789)
Error Report: { Errores:= [21302]: Valor Numerico no permitido, se esperaba un valor Booleano - Posicion del Error (4) }
Compare: FALSE
AST

// #152
// Source: if(TRUE, if(FALSE, "A", B), "C")
Error Trace: if(TRUE,if(FALSE, => " <= A => " <= ,B), => " <= C => " <= )
Error Report: { Errores:= [11101]: Caracteres invalidos detectados - Posicion del Error (20,22,29,31) }
Compare: FALSE
AST

// #153
// Source: if(Bool(x), x > 5, 42)
Error Trace: if(Bool(x),x>5, => 42 <= )
Error Report: { Errores:= [21303]: La funcion "if" debe retornar el mismo tipo de valores - Posicion del Error (16) }
Compare: FALSE
AST

Total Tested: 393
Total Hits: 239
Total Failures: 154
Total True Counts: 239
Total False Counts: 154
```

La prueba retorna la siguiente información:

1. Nombre de la prueba.
2. Evaluación de cada expresión, la cual cuenta con: lista de tokens y sus tipos, Reconstrucción de la cadena, expresión original sin espacios (se usa una función propia de Delphi) y por ultimo si la reconstrucción es igual a la cadena original.
3. Conteo total de aciertos y fallos encontrados.
4. Conteo total de aciertos y fallos esperados.

### *5.4.7. Pruebas de Fugas de Memoria*

Esta prueba realiza unicamente el proceso de construcción del árbol, ya que en pruebas anteriores se llega a .ªbrir"la función con el objetivo de observar el proceso interno llevado acabo por el componente o componentes, lo que facilita el proceso de depuración pero dificulta

la búsqueda de fugas de memoria, ya que el proceso de abrir la función puede llegar a generar fugas de memoria falsas. Con el objetivo de evaluar si existen fugas de memoria se emplea un componente de terceros llamado FastMM4, el cual se centra en depurar las fugas y su origen, de esta manera se encuentran y se solucionan todas las fugas de memoria encontradas en el proceso, estas pruebas no cuentan con un registro que demuestre esta afirmación, ya que no arroja ningún resultado en caso de no encontrar fugas, por lo que el soporte o pruebas se basa en el testimonio del docente Emiliano Lince, el cual estuvo presente y observo las pruebas en funcionamiento, presentando adicionalmente una prueba la cual contaba con fugas de memoria que no se corrigieron, con el objetivo de mostrar que el componente encargado de las pruebas de fuga de memoria trabajaba de forma correcta y conforme a lo esperado, como soporte de esto se compartirá un vídeo a los evaluadores como pruebas adicionales de las afirmaciones hechas en este apartado.

1. Probar el componente que el proceso de construcción y evaluación de las expresiones se ejecuta sin fugas de memoria.

2. Se desarrollan dos funciones encargadas de esta tarea, una se centro en evaluar casos particulares, una expresión a la vez, la segunda función se encarga de evaluar la lista de expresiones definida en la prueba anterior.

3. Genera un reporte con información del proceso de construcción del cada árbol correspondiente a las diferentes expresiones en la lista de evaluación.

4. Se emplea un componente de código de terceros, especializado en depurar fugas de memoria, el componente se llama: *FastMM4* (le Riche, 2021).

Las pruebas de fuga de memoria se realizan empleando el componente FastMM4, el cual es un recubrimiento y a su vez reemplaza el componente encargado de manejar la memoria con el que cuenta delphi y el cual, se incluye en la compilación del programa durante las pruebas. Bajo estas condiciones se realiza la instalación del componente, se configura para

poder realizar pruebas de fuga de memoria, posteriormente se ejecuta el depurador y en caso de encontrar fugas de memoria aparecerá un mensaje de error y se incluirá en la carpeta de construcción del ejecutable, un reporte con el listado de incidencias de fugas de memoria y la función, procedimiento o método responsable.

**Figura 48**

*Resultados de las pruebas incluyendo FastMM4 sin eventos*

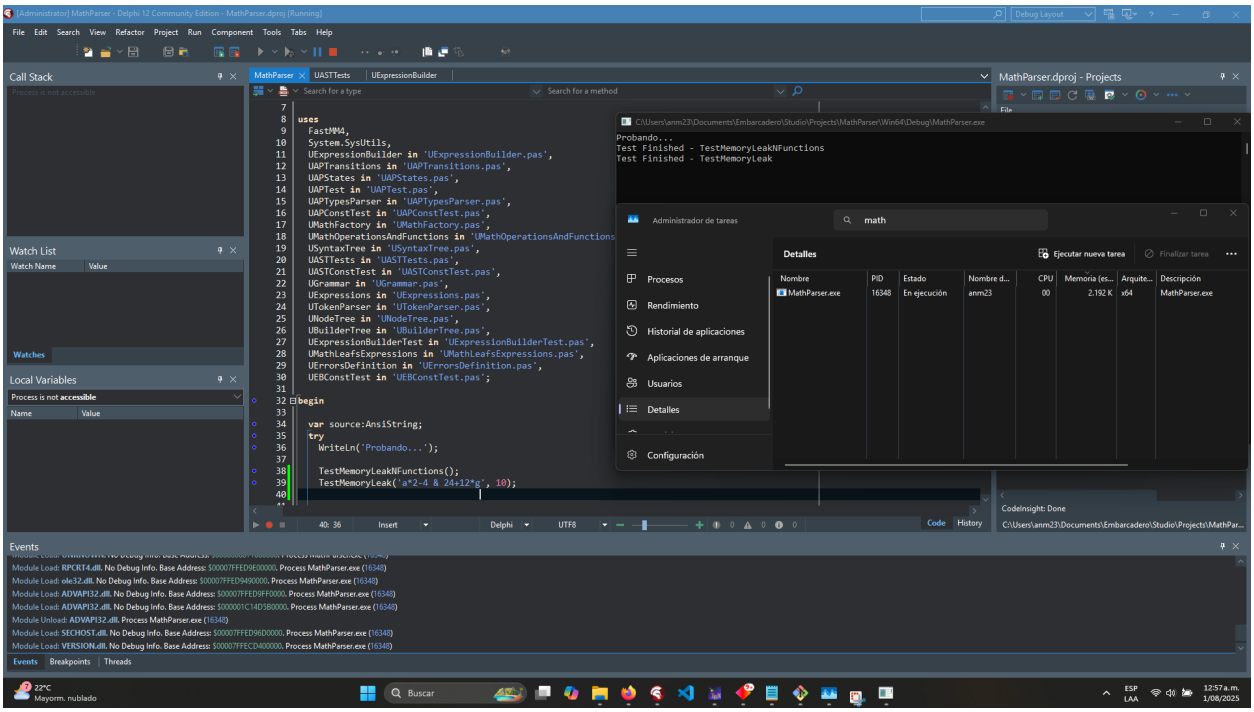
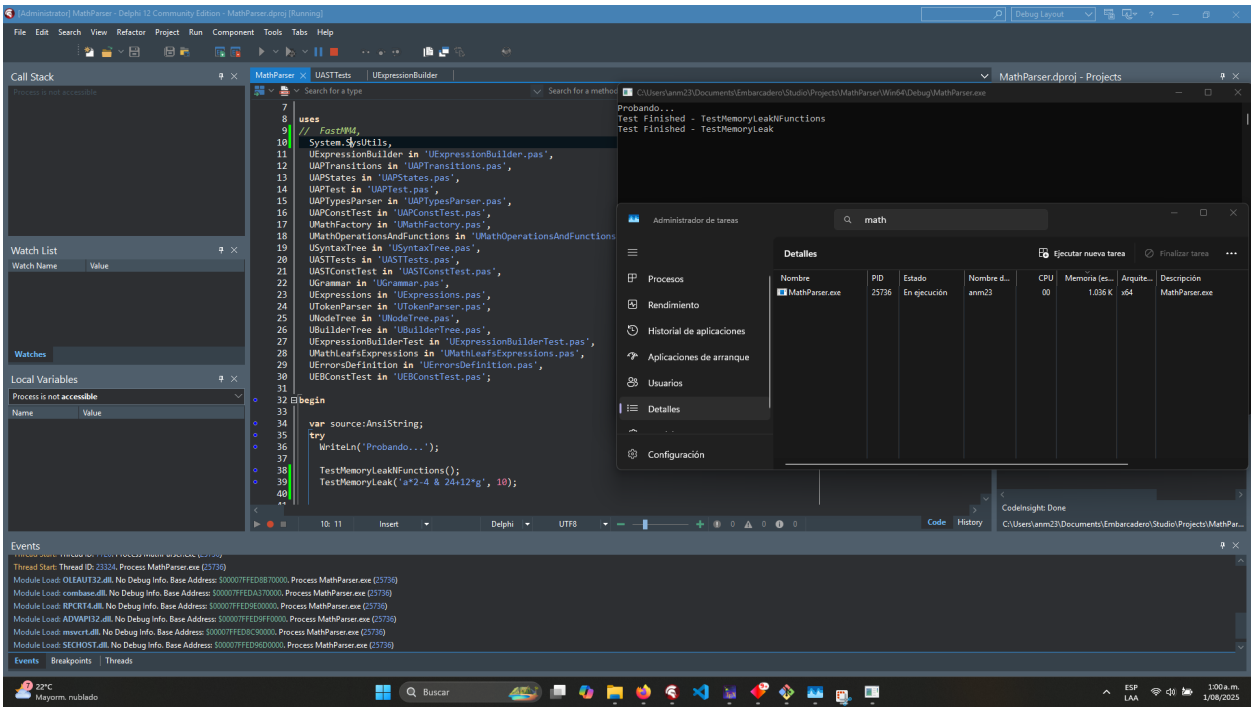


**Figura 49**

*Resultados de las pruebas sin incluir FastMM4*




se presentan el archivo resultante de cada prueba:

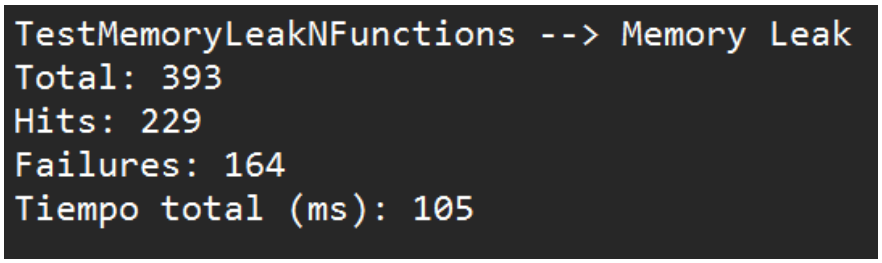
**Figura 50**

*Resultados de las pruebas TestMemoryLeak*

```
TestExpressionBuilderAPD --> Analisis Lexico - Tiempo de ejecuciin
Iteraciones: 1000000
Cadena: if(a>b,if(x=b,2,4),if(1<s>3,2,3))
Tiempo total (ms): 228074
Tiempo promedio por ejecuciin (micro s): 228,04
Tiempo total acumulado (micro s): 228039096,00
```

**Figura 51**

*Resultados de las pruebas TestMemoryLeakNFunctions*




La prueba retorna la siguiente información:

1. Nombre de la prueba.
2. Evaluación de cada expresión, la cual cuenta con: lista de tokens y sus tipos, Reconstrucción de la cadena, expresión original sin espacios (se usa una función propia de Delphi) y por ultimo si la reconstrucción es igual a la cadena original.
3. Conteo total de aciertos y fallos encontrados.
4. Conteo total de aciertos y fallos esperados.

### *5.4.8. Orden de Evaluación*

Esta prueba surge del problema que plantea emplear una función recursiva en arboles que pueden ser muy profundo, ya que puede generar un proceso de sobrecarga de llamadas, y con ello puede traer desbordamientos en la memoria del procesador, por ello y debido a la profundidad que pueden tomar algunas estructuras de las expresiones, se planteo como obligatorio cambiar el método de construcción del orden de evaluación, para esto se diseña una función que implementa una pila por la utilidades que ofrece esta estructura de datos en Delphi, esta metodología no presenta ningún problema relacionado al tamaño, pero si puede ingresar errores significativos en el funcionamiento del sistema. Lo anterior presenta la importancia de contar con una prueba que verifique el correcto funcionamiento de la

funcion encargada de construir el orden de evaluación. Para cumplir este propósito se define el siguiente proceso:

1. Crear una instancia del componente encargado de orquestar el proceso de construcción del árbol de sintaxis *ExpressionBuilder* y construcción del orden de evaluación.
2. Recibe una lista de expresión, permite evaluar capacidades generales en la construcción del orden de evaluación, verifica cada elemento con la estructura del árbol real o esperada.
3. Genera un reporte de las pruebas llevadas acabo.

Se presentan fragmentos del archivo resultante de la prueba:

**Figura 52**

*Resultados de las pruebas TestEBOrderNodes*

```
TestEBOrderNodes --> building the order of the expression
#0
Source: a+b*10+b
Error in the Source:
Comparison: TRUE
Order Operations:
* ->  b 10
+ ->  a * b

True Order:
* ->  b 10
+ ->  a * b

#1
Source: 15/(20+20+a)
Error in the Source:
Comparison: FALSE
Order Operations:
+ ->  20 20 a
/ ->  15 +

True Order:
+ ->  20 20 a
() ->  +
/ ->  15 ()
```

La prueba retorna la siguiente información:

1. Nombre de la prueba.
2. Error en la expresión procesada.
3. Comparación del resultado obtenido, con el esperado.
4. Orden de evaluación.
5. Orden real o esperado.

### *5.4.9. Creación y Eliminación de Constantes Normales y Globales*

En este apartado se trataran cuatro pruebas que revisan un mismo aspecto del componente de código diseñado, pero que cuenta con cuatro funcionalidades asociadas, siendo los elementos matemáticos definidos como constantes globales y normales, donde las primeras tiene un valor fijo que no cambia con el escenario, mientras las segundas, su valor depende del escenario, ya que facilita y mejora el proceso de evaluación (gracias a la implementación de punteros). Para cumplir este propósito se define el siguiente proceso:

1. Crear una instancia del componente definido como interface de uso *TExpressionLeaf* para probar las funcionalidades relacionadas con agregar y eliminar lo elementos de las expresiones (tipos de constantes).
2. Crea nombres de forma aleatoria para crear uno de los dos tipos de constante con un valor asociado, generado de forma aleatoria, estos datos son agregados en una lista de elementos para corroborar la veracidad de los datos almacenados por el componente de código.
3. Elige elementos al azar de un tamaño definido como el 20 % de los elementos generados, de esta forma se realizara la prueba de eliminación.
4. Revuelve los elementos en su totalidad.

5. Comprueba la lista de elementos existan y sean coherentes a los valores definidos.

6. Genera un reporte de las pruebas llevadas acabo.

Se presentan fragmentos de las pruebas realizadas:

**Figura 53**

*Resultados de las pruebas agregación de constantes: TestMathLeafsExpressionsConstants*

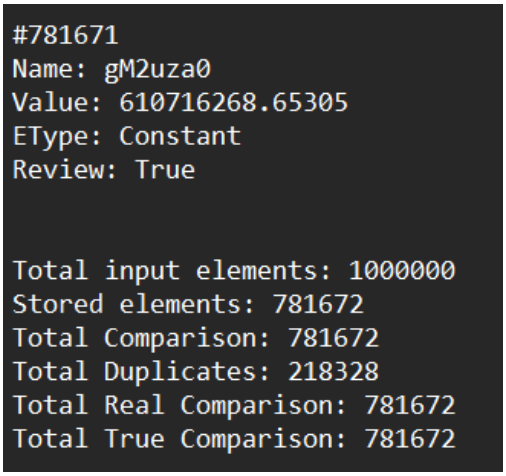



**Figura 54**

*Resultados de las pruebas eliminación de constantes: TestMathLeafsExpressionsDeletesConstants*

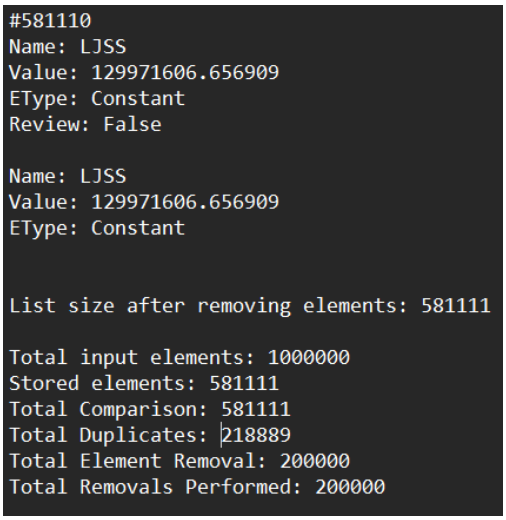


**Figura 55**

*Resultados de las pruebas agregación de constantes globales: TestMathLeafsExpressionsConstantsGlobals*

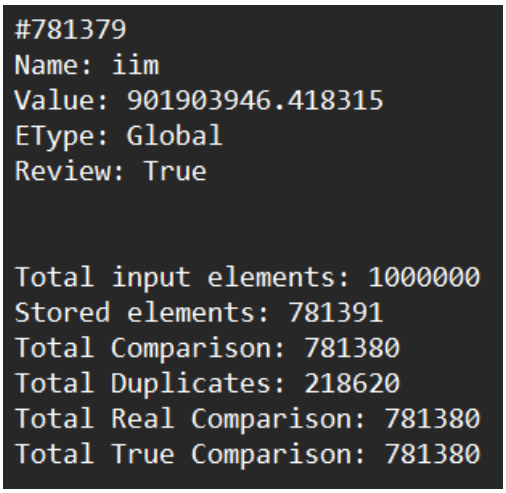
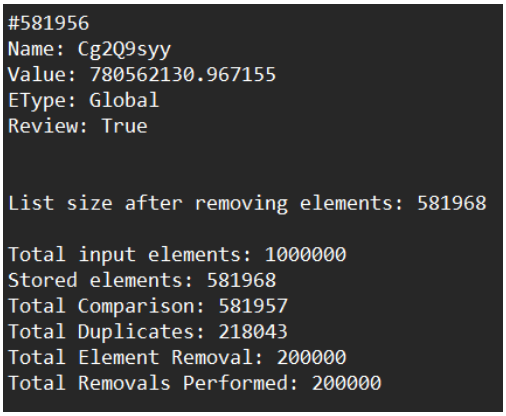



**Figura 56**

*Resultados de las pruebas eliminación de constantes globales: TestMathLeafsExpressionsDeletesConstantsGlobals*




La prueba retorna la siguiente información:

1. Nombre de la prueba.

2. Listado de elementos creados con nombre, valor y tipo.

3. Resultados del proceso de comparación con los elementos esperados.

4. Cantidad elementos definidos para la creación.

5. Elementos almacenados.

6. Elementos totales comparados.

7. Elementos duplicados.

### *5.4.10. Cambio de Nombre y Valor en Constantes Normales y Globales*

En este apartado se continua con la revisión de los elementos matemáticos, pero en este punto se centra en los cambios que pueden sufrir los elementos y no en los procesos de construcción y eliminación de elementos. Para cumplir este propósito se define el siguiente proceso:

1. Crear una instancia del componente definido como interface de uso *TExpressionLeaf* para probar las funcionalidades relacionadas con agregar y eliminar lo elementos de las expresiones (tipos de constantes).

2. Crea nombres de forma aleatoria para crear uno de los dos tipos de constante con un valor asociado, generado de forma aleatoria, estos datos son agregados en una lista de elementos para corroborar la veracidad de los datos almacenados por el componente de código.

3. Elige elementos al azar de un tamaño definido como el 20 % de los elementos generados, de esta forma se realizara la prueba de eliminación.

4. Revuelve los elementos en su totalidad.

5. Comprueba la lista de elementos existan y sean coherentes a los valores definidos con sus cambios respectivos.

6. Genera un reporte de las pruebas llevadas acabo.

Se presentan fragmentos de las pruebas realizadas:

**Figura 57**

*Resultados de las pruebas de cambio de nombres en constantes: TestMathLeafsExpressionsSetNameConstants*

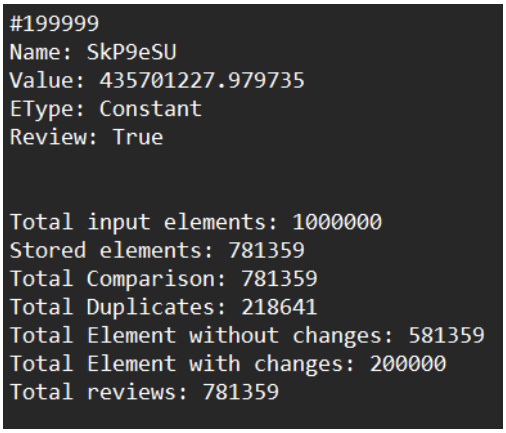



**Figura 58**

*Resultados de las pruebas de cambio de valores en constantes: TestMathLeafsExpressionsSetValueConstants*

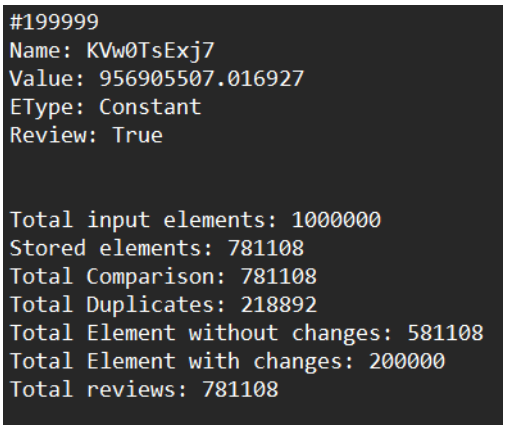


**Figura 59**

*Resultados de las pruebas de cambio de nombres en constantes globales: TestMathLeafsExpressionsSetNameConstantsGlobals*

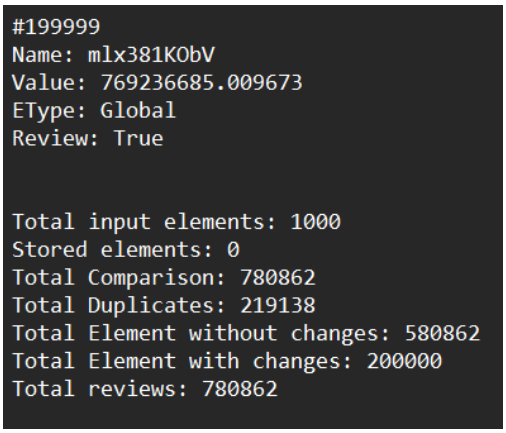
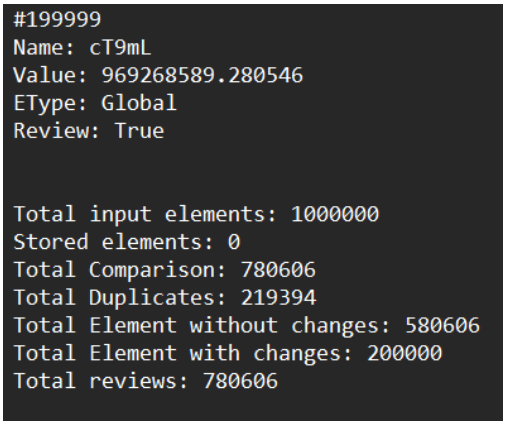



**Figura 60**

*Resultados de las pruebas de cambio de valores en constantes globales: TestMathLeafsExpressionsSetValueConstantsGlobals*




La prueba retorna la siguiente información:

1. Nombre de la prueba.
2. Listado de elementos creados con nombre, valor y tipo.
3. Resultados del proceso de comparación con los elementos esperados.
4. Cantidad elementos definidos para la creación.

5. Elementos almacenados.

6. Elementos totales comparados.

7. Elementos duplicados.

### *5.4.11. Construcción de Una Expresión usando ExpressionBuilder*

Esta prueba se centra en comprobar que la clase encargada de orquestar el proceso de construcción de una expresión funcione adecuadamente, adicional a esto, sirve como una prueba de carga del proceso de evaluar una expresión construida. Para cumplir este propósito se define el siguiente proceso:

1. Crear una instancia del componente encargado de orquestar el proceso de construcción de la expresión empleando *ExpressionBuilder*.

2. Recibe una expresión, permite evaluar y depurar el proceso general encargado de la construcción de una expresión.

3. Mide el costo en tiempo de evaluar n veces una expresión.

4. Genera un reporte de las pruebas llevadas acabo.

Se presentan fragmentos del archivo resultante de la prueba:

**Figura 61**

*Resultados de las pruebas TestExpressionBuilderBuildExpression*

```
TestExpressionBuilderBuildExpression --> Building Symtax Tree - Execution time
Evaluate: 28768401,9030190340000000000000
Iteraciones: 1000000
Aciertos: 0
String: (1/3)*3 - 1 + 3 * 15 - sin(3.1416/4) + cos(3.1416) * ((324234 / 31432)
^ (1 + 2 - 4)) ^ (3/4) * power(2,3) - 42423 + 234234 * 123
Total Time (ms): 219
Average execution time (micro s): 0,19
Total accumulated time (micro s): 191998,60
```

La prueba retorna la siguiente información:

1. Nombre de la prueba.
2. Resultado de evaluar la expresión.
3. Cantidad de iteraciones.
4. Expresión interpretada.
5. Tiempo total en mili segundos.
6. Tiempo promedio de ejecución por evaluación de expresión interpretada en $\mu$ segundos.
7. Tiempo total acumulado por el proceso en $\mu$ segundos.

#### *5.4.12. Revisión de Elementos*

Esta prueba permite comprobar el funcionamiento del método encargado de revisar que las variables dentro de una expresión, se hayan creado previamente, de esta manera se previenen errores al usar elementos no declarados y graniza que la metodología de manejo y evaluación de los elementos planteada, se cumpla, retirando responsabilidades en la usabilidad del componente de código. Para cumplir este propósito se define el siguiente proceso:

1. Crear una instancia de la clase definida como interface de uso del componente de código: *TExpressionLeaf*.
2. Se define una lista de funciones con elementos marcados por medio de una lista, para comprobar que el proceso se realiza correctamente. La lista contiene elementos verdaderos y falso, marcados intencionalmente para probar el funcionamiento.
3. Una vez se agrega una expresión se comprueba que el resultado, ya sea falso o verdadero, corresponde con el resultado de revisar los elementos creados o no dentro de la hoja de expresiones.
4. Genera un reporte de las pruebas llevadas acabo.

Es importante resaltar que se contó con expresiones escritas correctamente, cuyo único problema recae en contener elementos no contenidos, de esta manera se comprueba esta funcionalidad del componente. Se presentan fragmentos del archivo resultante de la prueba:

**Figura 62**

*Resultados de las pruebas TestExpressionBuilderEElementsReview*

```
TestBuilderExpressionEElementsReview --> Check for the existence of elements in the expression

Correct expressions:


// #0
// Source: (massDensity * accelerationRate) - frictionCoefficient * 0.5
ErrorTrace:
ErrorReport:

// #1
// Source: potentialEnergy + kineticEnergy - (forceMagnitude * displacementVector)
ErrorTrace:
ErrorReport:

// #2
// Source: if(bool(isReady), TotalCount / RetryCount, MaxLimit * errorCode)
ErrorTrace:
ErrorReport:
```

**Figura 63**

*Conclusión de los resultados de las pruebas TestExpressionBuilderEElementsReview*

```
// #72
// Source: PRED(5) + FACT(6) + ABS(-8) + COSH(1) * LOG(2) - PROD(2, 3)
ErrorTrace: PRED(5)+FACT(6)+ABS(-8)+COSH(1)* => LOG <= (2)-PROD(2,3)
ErrorReport: { Errores:= [21102]: Funcion desconocida:  - Posicion del Error (33) }

// #73
// Source: IF(X > 5, COS(DEGTORAD(X)) + RANDOM(1, 10), LOG(10)) * 3
ErrorTrace: IF(X>5,COS(DEGTORAD(X))+RANDOM(1,10), => LOG <= (10))*3
ErrorReport: { Errores:= [21102]: Funcion desconocida:  - Posicion del Error (38) }

Total Tested: 422
Total Hits: 346
Total Failures: 74
Total True Counts: 348
Total False Counts: 74
```

La prueba retorna la siguiente información:

1. Nombre de la prueba.
2. Resultado de revisar las expresiones.
3. Total de expresiones probadas.

4. Total de aciertos encontrados.

5. Total de fallos encontrados.

6. Total de casos verdaderos.

7. Total de casos falsos.

### *5.4.13. Agregar Expresión*

Esta prueba permite comprobar el funcionamiento de la interface de uso en el manejo de expresiones, lo que incluye crear y evaluar. Para cumplir este propósito se define el siguiente proceso:

1. Crear una instancia de la clase definida como interface de uso del componente de código: *TExpressionLeaf*.

2. Se pasa una expresion que se quiera interpretar.

3. Se evalúa la expresión interpretada.

4. Puede realizarse una prueba de carga al definir n pruebas de evaluación.

5. Genera un reporte de las pruebas llevadas acabo.

Se presentan fragmentos del archivo resultante de la prueba:

**Figura 64**

*Resultados de las pruebas TestMathLeafsExpressionsAddExpression*

```
TestMathLeafsExpressionsAddExpression --> Expression evaluation tests
Evaluate: 10,000000000000000000000
# Error Report:
Iteraciones: 1000000
Aciertos: 0
String: 1+2+3+4
Total Time (ms): 89
Average execution time (micro s): 0,07
Total accumulated time (micro s): 66191,40
```

La prueba retorna la siguiente información:

1. Nombre de la prueba.
2. Resultado de evaluar la expresión.
3. Reporte de errores.
4. Iteraciones.
5. Expresión interpretada.
6. Tiempo total en mili segundos.
7. Tiempo promedio en $\mu$ segundos.
8. Tiempo total acumulado en $\mu$ segundos.

### *5.4.14. Almacenamiento y Evaluación de Expresiones*

Esta prueba permite comprobar el correcto manejo y evaluación de n expresiones, ya que en la practica se espera que una hoja de expresiones gestione n expresiones interpretadas, adicional a esto se agregan variables a las expresiones y se les asigna valores aleatoriamente, con el objetivo de evaluar estas expresiones y comparar sus resultados, para alcanzar este objetivo fue necesario escribir manualmente expresión a expresión verificando su veracidad, de esta forma se puede comprobar que el componente realmente funciona. Para cumplir este propósito se define el siguiente proceso:

1. Crear una instancia de la clase definida como interface de uso del componente de código: *TExpressionLeaf*.
2. Se define una lista de funciones con elementos marcados por medio de una lista, para comprobar que el proceso se realiza correctamente. La lista contiene elementos verdaderos y falso, marcados intencionalmente para probar el funcionamiento.
3. Se evalúa la expresión escrita manualmente y se compara con el valor obtenido del interprete.

4. Genera un reporte de las pruebas llevadas acabo.

Se presentan fragmentos del archivo resultante de la prueba:

**Figura 65**

*Resultados de las pruebas TestMathLeafsExpressionsBuildNExpression*

```
TestMathLeafsExpressionsBuildNExpression --> Evaluation tests of N expressions

Correct expressions:


// #0
// Source: (massDensity * accelerationRate) - frictionCoefficient * 0.5
  massDensity = 728,2849159091709400000000
  accelerationRate = 660,6222789268939500000000
  frictionCoefficient = 284,4977898057550100000000
Comparison passed -> Real Value: 480978,9919610950400000000000 = Obtain Value: 480978,9919610950400000000000
ErrorTrace:
ErrorReport:
Comparison: TRUE

// #1
// Source: potentialEnergy + kineticEnergy - (forceMagnitude * displacementVector)
  potentialEnergy = 899,2016569245610200000000
  kineticEnergy = 362,9547436721619800000000
  forceMagnitude = 71,6276704333723010000000
  displacementVector = 66,7649700772016900000000
Comparison passed -> Real Value: -3520,0628725870438000000000 = Obtain Value: -3520,0628725870438000000000
ErrorTrace:
ErrorReport:
Comparison: TRUE
```

**Figura 66**

*Conclusión de los resultados de las pruebas TestMathLeafsExpressionsBuildNExpression*

```
// #72
// Source: PRED(5) + FACT(6) + ABS(-8) + COSH(1) * LOG(2) - PROD(2, 3)
ErrorTrace: PRED(5)+FACT(6)+ABS(-8)+COSH(1)* => LOG <= (2)-PROD(2,3)
ErrorReport: { Errores:= [21102]: Funcion desconocida:  - Posicion del Error (33) }

// #73
// Source: IF(X > 5, COS(DEGTORAD(X)) + RANDOM(1, 10), LOG(10)) * 3
ErrorTrace: IF(X>5,COS(DEGTORAD(X))+RANDOM(1,10), => LOG <= (10))*3
ErrorReport: { Errores:= [21102]: Funcion desconocida:  - Posicion del Error (38) }

Total Tested: 422
Total Evaluations: 345
Total Failures: 74
Total True Counts: 348
Total False Counts: 74
Total Evaluation passed: 301
```

La prueba retorna la siguiente información:

1. Nombre de la prueba.

2. Resultado de comparar los resultados obtenidos en la evaluación interpretada y escrita manualmente, registra si se encuentra una expresión con errores.

3. Total de expresiones probadas.

4. Total de evaluaciones realizadas.

5. Total de fallos encontrados.

6. Total de casos verdaderos.

7. Total de casos falsos.

8. Total de evaluaciones encontradas como verdaderas.

#### *5.4.15. Agregar Función Puntero*

Esta prueba permite comprobar el funcionamiento de la funcionalidad integrada que permite agregar funciones puntero bajo un formato especifico. Para cumplir este propósito se define el siguiente proceso:

1. Crear una instancia de la clase definida como interface de uso del componente de código: *TExpressionLeaf*.

2. Se definen funciones bajo el formato definido.

3. Se cargan las funciones.

4. Se realiza un proceso de interpretación y evaluación de una expresión que contenga una o varias funciones agregadas.

5. Genera un reporte de las pruebas llevadas acabo.

Se presenta un ejemplo donde se agrega una función que recibe 4 parámetros y los suma, junto al resultado de su evaluación en la siguiente imagen:

**Figura 67**

*Resultados de las pruebas TestMAthLeafsExpressionsAddFunction*

```
TestMAthLeafsExpressionsAddFunction --> Add Function - Execution time
Evaluate: 30,000000000000000000000
Iteraciones: 1000000
Aciertos: 0
String: suma(1,2,3,4) * 3
Total Time (ms): 168
Average execution time (micro s): 0,15
Total accumulated time (micro s): 145597,30
```

La prueba retorna la siguiente información:

1. Nombre de la prueba.
2. Resultado de evaluar la expresión.
3. Iteraciones.
4. Expresión interpretada.
5. Tiempo total en mili segundos.
6. Tiempo promedio en $\mu$ segundos.
7. Tiempo total acumulado en $\mu$ segundos.

### *5.4.16. Agregar Función DLL*

Esta prueba permite comprobar el funcionamiento de la funcionalidad integrada que permite agregar funciones desde un DLL bajo un formato especifico. Para cumplir este propósito se define el siguiente proceso:

1. Crear una instancia de la clase definida como interface de uso del componente de código: *TExpressionLeaf*.
2. Se definen funciones bajo el formato definido y se compilan en una DLL.

3. Se cargan las funciones.

4. Se realiza un proceso de interpretación y evaluación de una expresión que contenga una o varias funciones agregadas.

5. Genera un reporte de las pruebas llevadas acabo.

Se presenta un ejemplo donde se agregan tres funciones que reciben un numero especifico de parámetros, junto al resultado de su evaluación en la siguiente imagen:

**Figura 68**

*Resultados de las pruebas TestMAthLeafsExpressionsAddFunctionDll*

```
TestMAthLeafsExpressionsAddFunctionDll --> Add Function DLL - Execution time
ErrorReport:
Evaluate: 32,181818181818120000000
ErrorReport:
Iteraciones: 100000
Aciertos: 0
String: Prueba1(1,2,3,4) * 3 + Prueba2(1,2,3,4) / Prueba3(1,2,3,4)
Total Time (ms): 38
Average execution time (micro s): 0,37
Total accumulated time (micro s): 36726,80
```

La prueba retorna la siguiente información:

1. Nombre de la prueba.

2. Resultado de evaluar la expresión.

3. Iteraciones.

4. Expresión interpretada.

5. Tiempo total en mili segundos.

6. Tiempo promedio en $\mu$ segundos.

7. Tiempo total acumulado en $\mu$ segundos.

## 5.5. Presentación de Resultados

En esta sección se abordaran los resultado obtenidos, para esto se abordara la arquitectura modular y otros aspectos que se tuvieron en cuenta durante el desarrollo, así como principios de diseño SOLID. Por ultimo se trataran los requerimientos alcanzados y se presentara una implementación completa del interprete de expresiones.

### *5.5.1. Precisión*

El componente de código al centrarse en simulaciones requiere de un nivel de precisión lo suficientemente alto, para poder tenerse en cuenta en la simulación e diversos fenómenos que requieran de precisión. Por esto mismo y para probar el tamaño máximo de una expresión se realiza la siguiente prueba.

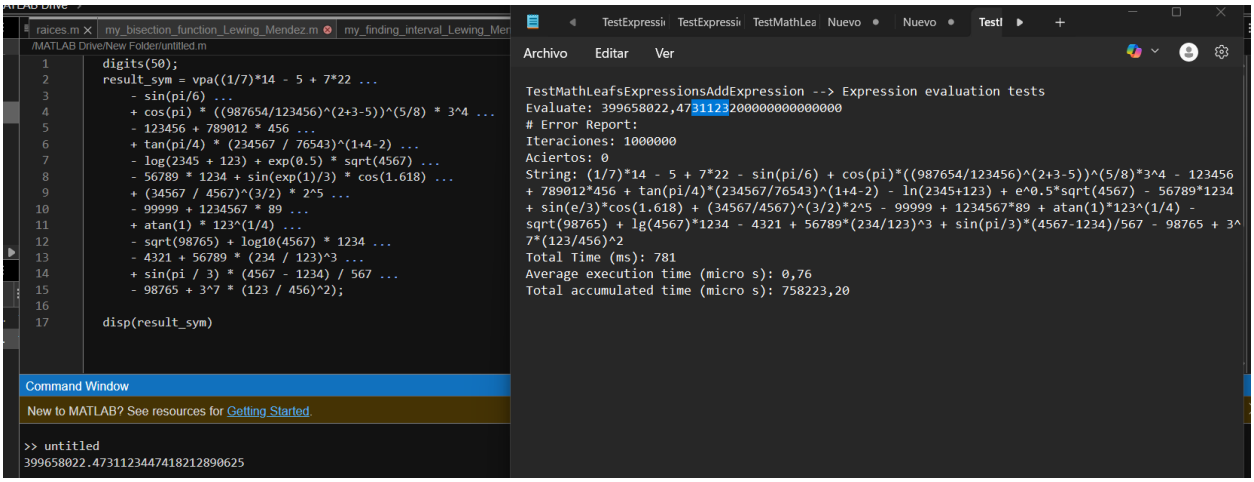
1. **Entorno Matlab:** Para poder tener una idea de la precisión del componente de código se escribirá la misma expresión matemática en el lenguaje Matlab, para contrastarse los resultados.

2. **Evaluar la expresión empleando MEL:** Se utiliza la prueba diseñada para probar el proceso de agregar y evaluar una expresión, retornando el valor de la evaluación, este se compara con el obtenido de Matlab.

   El resultado de esta prueba se presenta en la siguiente imagen:

**Figura 69**

*Comparación de resultados*




Calculando el error se tiene:

1. **Error absoluto:** $E_a = 2{,}47418212890625 \times 10^{-8}$
2. **Error relativo:** $E_r = 6{,}190748064046943 \times 10^{-17}$
3. **Error porcentual:** $E_{\%} \approx 6{,}19 \times 10^{-15}\,\%$

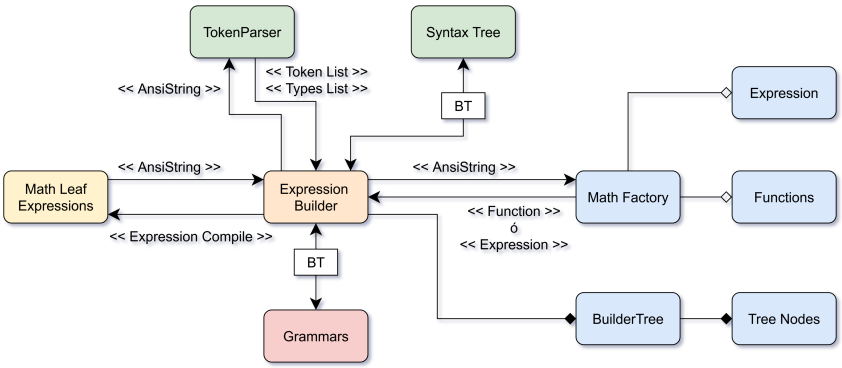
Lo que corresponde a valores a errores de truncamiento, aun así, se contempla y se prueba que aun en expresiones muy grandes y con operaciones que pueden generar propagaciones de error, Delphi cuenta con un motor matemático robusto y permite manejar una precisión excelente. A pesar de esto es de reconocer que para cálculos que requieran de un mayor numero de decimales, puede que el componente no sea adecuado, debido a que el tamaño máximo de los números en Delphi es de 8 bits.

En esta prueba se puede ademas observar el tamaño que pueden llegar a tomar las expresiones, llegando a tener al menos 1000 caracteres.

### *5.5.2. Arquitectura Modular*

Como se presento en la Figura 9 el componenete se divide en sub componentes con el objetivo de facilitar la mantenibilidad del software, debido a que al dividir en partes el proceso se consigue centralizar las fuentes de errores, de esta manera un error estará asociado a un único proceso y sera mas sencillo ubicar su fuente, así como también se mejora la capacidad para extender el software, modificando procesos específicos en lugar de partes.

Lo descrito coincide con los diagramas de clases UML, así como con las descripciones correspondientes que se han hecho a lo largo del documento.




Nota. Imagen repetida de la Figura 9.

La Modularidad del componente se centra en la clase *BuilderTree* ya que esta almacena el estado del proceso de construcción de una expresión, esto implica una instancia única (u objeto), que se mueve entre métodos de clases, que se encargan de recibir unos datos almacenado por la instancia, realizar el proceso correspondiente y por ultimo, entregarle el resultado de su proceso a *BuilderTree*, esta ausencia de estado en las clases facilita llevar el componente a computación en paralelo, claro que esto como una mejor en futuros proyectos.

El que se plantee como posible y viable este tipo de modificaciones demuestra un diseño modular.

### *5.5.3. Principios SOLID, Extensibilidad y Mantenibilidad*

Los principios de diseño SOLID se centran en facilitar la mantenibilidad y extensibilidad del software, por este motivo se tuvieron en cuenta en el diseño las clases, ya que esto daría como soporta la requerimientos no funcionales de Modularidad y extensibilidad. A continuación se describe la presencia de cada principio en el proyecto:

1. **Principio de responsabilidad única:** Es el principio predominante en el diseño de clases, cada clase se encarga de una parte del proceso en especifico.

2. **Principio de abierto/cerrado:** El diseño de las clase se planteo para extenderse, llegando a requerir agregar elementos a una lista pero eliminando la necesidad de modificar código, cumpliendo abierto para extensión, cerrado para modificación.

3. **Principio de sustitución de Liskov:** Este principio se aplica en la construcción del autómata, ya que en otras partes del código no se observaría gran impacto o beneficio, al ser tareas o roles muy específicos.

4. **Principio de segregación de clases:** Las interfaces implementadas son implementadas en su totalidad y utilizadas, por lo que este principio se respeta aunque no se abordo directamente.

5. **Principio de inversión de dependencias:** Este principio es la base del funcionamiento del autómata, ya que las transiciones presentan la necesidad de hacer lo mismo pero de formas diferentes, es el caso, de la necesidad de disponer de transiciones que hagan un proceso especifico cuando reconozca uno o alguno entre varios caracteres, cuyo caso es el de la coma, o los paréntesis o delimitadores, esto facilita agregar elementos reconocibles y por ende la extensibilidad del autómata, permitiendo ampliar las capacidades de interpretación.

El componente en el que se empleo estos principios exhaustivamente es el *TokenParser* ya que desde el inicio se considero que seria la fuente de muchos cambios, modificaciones y

ajustes, por lo que el proceso se subdividió en tareas muy especificas, empleando el principio de inversión de dependencias, para esto se centro el diseño en que actividades realiza cada actor, donde:

1. **Estados:** Existen tres tipos de estados, cada uno realiza la misma actividad, permite pasar al siguiente estado o no, ademas de realizar el proceso de construcción de los tokens tipados, ademas téngase en cuenta que los estados finales definen el token y su tipo, mientas que los estados normales unicamente construyen el token.

2. **Transiciones:** Podría decirse que existen muchos tipos de transiciones, ya que el objetivo que se pretendía alcanzar con el diseño, era que se pudieran agregar transiciones que realizaran una tarea única o especifica según se requiriera, como es el caso de la coma, o la transición con pila, cuyas actividades son muy diferentes a las demás, este tipo de implementación se basa en el uso de interfaces y herencia. Lo anterior permite tener una sola lista, con objetos del mismo tipo, sobre los cuales se itera llamando se el mismo método y cada método, realiza un proceso diferente empleando un registro de datos, que representa el estado del proceso.

3. **Gramáticas:** Agregar un símbolo o carácter especial, como el caso de & y | depende de poder agregar un nuevo conjunto de estados y sus transiciones, posterior a esto se debe ingresar la lógica correspondiente al proceso de construcción de árbol de sintaxis, donde unicamente se deberá agregar la lógica necesaria para manejar el nuevo tipo, el nuevo tipo y el peso del tipo; el proceso de agregar estos elementos se centra en agregar mas no en modificar. Emplear interfaces daba como resultado una clase que tenia un arreglo de pesos, y clases hijas que cambian de método, pero esto agregaría lógica innecesaria sin una mejora visible.

Es de esta manera que se implementan promueve la mantenibilidad y extensibilidad del componente de código, este proceso probablemente requiera de comprender el proceso de interpretación de una cadena o expresión.

### *5.5.4. Cumplimiento de los Requerimientos*

En el presente proyecto se cumplen la totalidad de los requerimientos planteados, tanto los principales como aquellos definidos opcionales, los cuales no son obligatorios, ya que el compilador de formulas integrado en Evolución carece de dichas capacidades.

Es de aclarar que muchos de los requerimientos se alcanzan progresivamente a lo largo del desarrollo del componente de código, debido a que los requerimientos dependen de múltiples factores o procesos complementarios, corroborando se el logro mediante las diferentes pruebas de software, es por este motivo que no se menciona donde o cuando son alcanzados cada uno de los requerimientos a lo largo del documento.

Bajo esta idea se presenta a continuación dos subsecciones donde se presentaran la totalidad de los requerimientos alcanzados y cuales componentes son los encargados y responsables de cumplir con dicho requerimiento.

**Tabla 9**

*Implementación de requerimientos funcionales*

| ID | Nombre | Componentes Encargados |
|---|---|---|
| RF1 | Caracteres permitidos | Analisis Léxico |
| RF2 | Reconocimiento de números | Analisis Léxico |
| RF3 | Manejo de variables | Analisis Léxico, Analisis Gramatical, Interface de Uso |
| RF4 | Manejo de funciones | Analisis Léxico, Analisis Gramatical, Interface de Uso |
| RF5 | Reconocimiento de delimitadores | Analisis Léxico, Analisis Gramatical |
| RF6 | Operaciones matemáticas | Analisis Léxico, Analisis Gramatical |
| RF7 | Operadores de comparación | Analisis Léxico, Analisis Gramatical |
| RF9 | Retorno de valores | Analisis Léxico, Analisis Gramatical |
| RF10 | Definición de elementos | Interface de Uso |
| RF11 | Detección de errores léxicos y sintácticos | Analisis Léxico, Analisis Gramatical |
| RF12 | Mensajes de error en expresiones | Analisis Gramatical, Interface de Uso |
| RF13 | Mensajes de error generales | Interface de Uso |
| RF14 | Carga de funciones en tiempo de ejecución | Analisis Gramatical, Interface de Uso |
| RF15 | Carga de funciones desde DLL | Analisis Gramatical, Interface de Uso |

#### 5.5.4.1 Cumplimiento de los Requerimientos Funcionales.

**Tabla 10**

*Implementación de requerimientos no funcionales*

| ID | Nombre | Cómo se alcanzan |
|---|---|---|
| RNF1 | Lenguaje de implementación | El componente se desarrollo en su totalidad en la versión de Rad Studio 12, empleando el lenguaje Delphi, usando la licencia adquirida por el grupo SIMON. |
| RNF2 | Capacidad de almacenamiento | El componente cuenta con pruebas que instancia un millón de elementos, finalizando sin errores. |
| RNF3 | Desempeño | Las pruebas presentan mejores tiempos de evaluación que el compilador de formulas con el que cuenta Evolución. |
| RNF4 | Modularidad | Como se mostró a lo largo del desarrollo del proyecto, el proceso se reparte en componentes aislados que reciben algo y entregan algo, de esta manera se asegura la arquitectura modular. |
| RNF5 | Interfaz desacoplada | Según lo presentado, se cuenta con una interface de uso que facilita la integración y uso del componente. |
| RNF6 | Extensibilidad | El componente se desarrolla pensando en la extensibilidad de sus capacidades, como se presenta en su sección respectiva. |

#### 5.5.4.2 Cumplimiento de los Requerimientos No Funcionales.

### *5.5.5. Implementación del Componente de Código: MELView*

Para facilitar la presentación de resultados del componente se preparo una interface gráfica, capaz de emplear el componente en su totalidad, lo que permite a los usuario intermedios familiarizarse con las capacidades del componente de código, ayudando a entender el como se deberá integrar, bien sea, en Evolución o en cualquier otro proyecto que pueda aprovecharlo.

La funcionalidad central del componente de código se centra en la evaluación de expresiones matemáticas, el usuario deberá seguir una idea básica bajo la cual podrá interpretar cada una de las expresiones que desee, siempre que la expresión cumpla con una serie de reglas: léxico, gramaticales y semánticas. El proceso requerido para construir una expresión es el siguiente:

1. **Declaración de elementos:** Se deben declara previamente las constantes normales y las globales, que vayan a ser empleadas en cualquier expresión, cada elemento cuenta

con su propio apartado en el cual se pueden gestionar.

2. **Declaración de expresiones:** Cada expresión interpretada debe contener elementos previamente declarados, así como los nombres de variables (nombre del elemento que almacena el resultado de evaluar la expresión) que pueda contener.

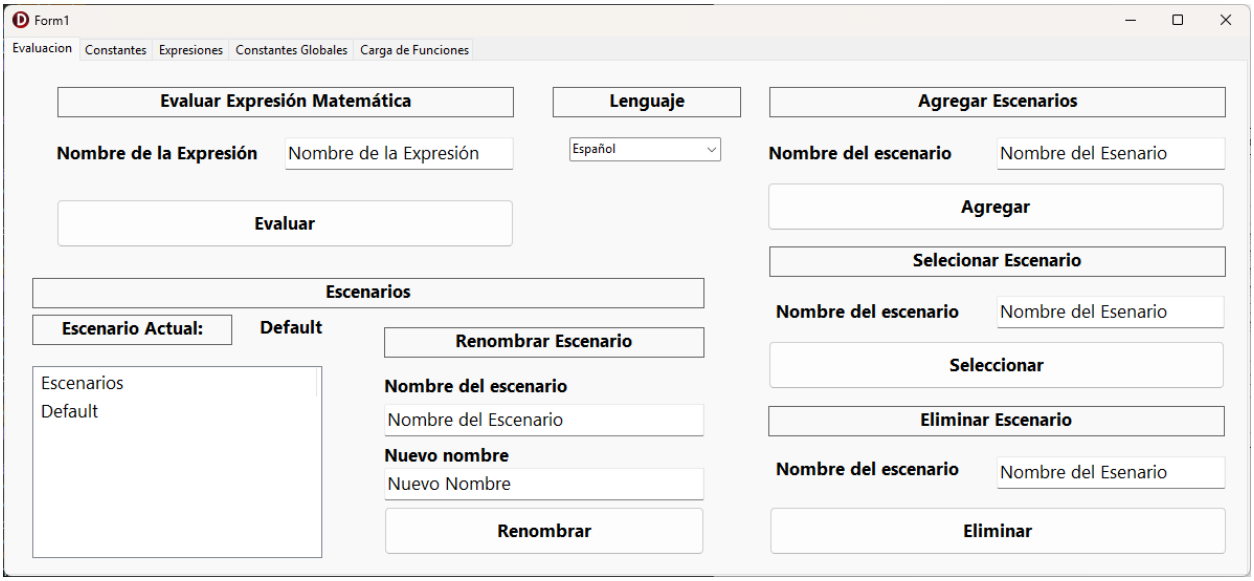
Debe tener en cuenta que cada expresión puede contar con cuatro tipos de elementos, siendo números, constantes normales, constantes globales y variables donde cada uno cuenta con un valor numérico asociado, pero cuya naturaleza es diferente en el sentido practico, ya que cada uno se define bajo métodos diferentes.

Teniendo en cuenta lo anterior, se presentan en las subsecciones siguiente, los apartados que el desarrollador y evaluadores podrán emplear para observar el funcionamiento del componente de código de forma visual.

**5.5.5.1 Ventana de Evaluación.** Este apartado presenta las funcionalidades necesarias para visualizar y evaluar el correcto funcionamientos del proceso de evaluación de expresiones, basado en alguno de los diferentes escenarios definidos.

**Figura 70**

*Aplicación: ventana de evaluación*



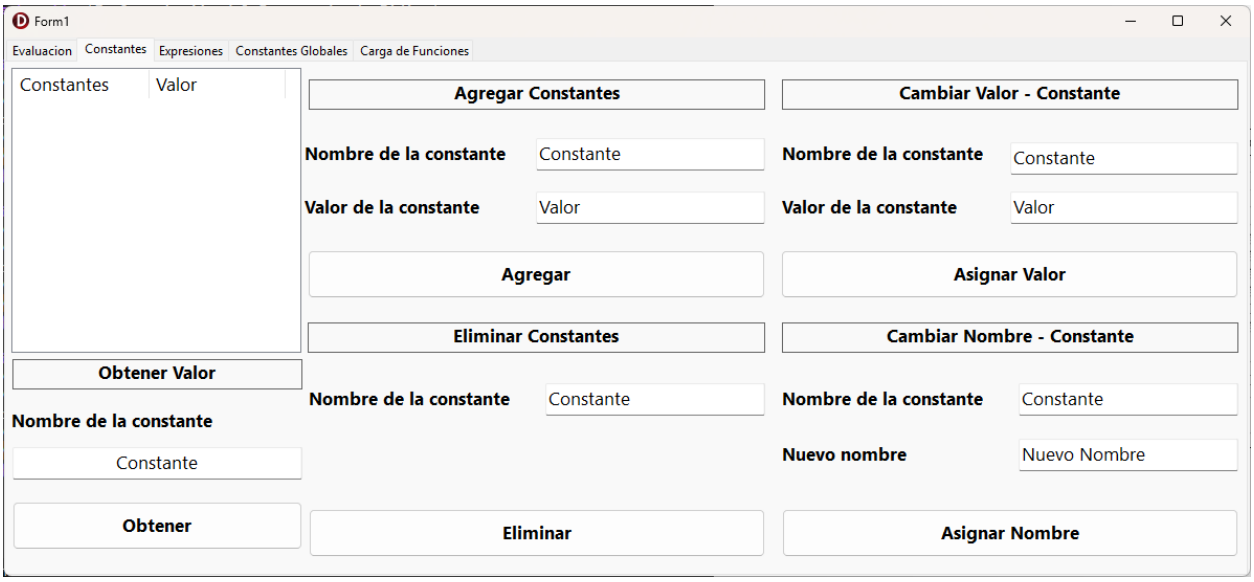
La pestaña *Evaluación* cuenta con las siguientes funcionalidades acorde a las capacidades del componente de código, continuación son descritas:

1. **Evaluar expresión matemática:** permite evaluar la expresión en base a los valores definidos.
2. **Lenguaje:** Permite cambiar de lenguajes entre ingles y español.
3. **Escenarios:** Permite visualizar los escenarios existente, así como el actual, adicionalmente permite cambiar el nombre de los escenarios existentes.
4. **Agregar Escenarios:** Permite crear escenarios.
5. **Seleccionar Escenario:** Permite definir el escenarios bajo el cual se evaluaran las expresiones.
6. **Eliminar Escenario:** Permite borrar algún escenario creado, téngase en cuenta que esto elimina la información almacenada en el mismo.

**5.5.5.2 Ventana de Constantes.** Este apartado presenta las funcionalidades necesarias para visualizar y evaluar el correcto funcionamientos en la creación y manejo de constantes normales (su valor siempre es el mismo, independiente del escenario).

**Figura 71**

*Aplicación: ventana de constantes*



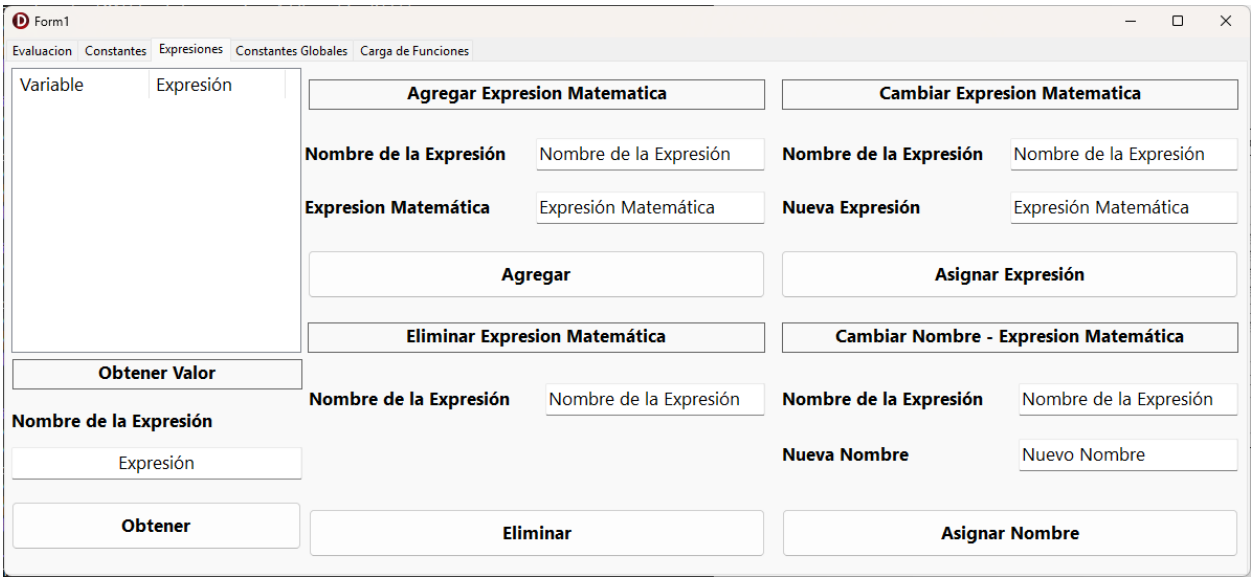

La pestaña *Constantes* cuenta con las siguientes funcionalidades acorde a las capacidades del componente de código, continuación son descritas:

1. Tabla de Elementos: Presenta los elementos creados por el usuario y su respectivo valor.
2. Obtener Valor: Retorna el valor que contiene un elemento.
3. Agregar constantes: Permite crear una nueva constante en la hoja de expresiones.
4. Eliminar constantes: Elimina una constante existente.
5. Cambiar valor - constante: Cambiar el valor que almacena la constante.
6. Cambiar nombre - constante: Cambia el nombre de una constante, manteniendo el mismo valor.

**5.5.5.3 Ventana de Expresiones.** Este apartado presenta las funcionalidades necesarias para visualizar y evaluar el correcto funcionamientos de la creación y manejo de expresiones.

**Figura 72**

*Aplicación: ventana de expresiones*




La pestaña *Expresiones* cuenta con las siguientes funcionalidades acorde a las capacidades del componente de código, continuación son descritas:

1. Tabla de expresiones: Presenta las expresiones interpretadas por el usuario y el nombre de variable asociado.

2. Obtener Valor: Retorna el valor o resultado contenido en el nombre de la variable asociada.

3. Agregar expresión matemática: Permite interpretar o agregar una expresión, haciendo la disponible, para su evaluación. El nombre de la expresión equivale al nombre de variable.

4. Eliminar expresión matemática: Eliminar una expresión empleando el nombre de variable asociado.

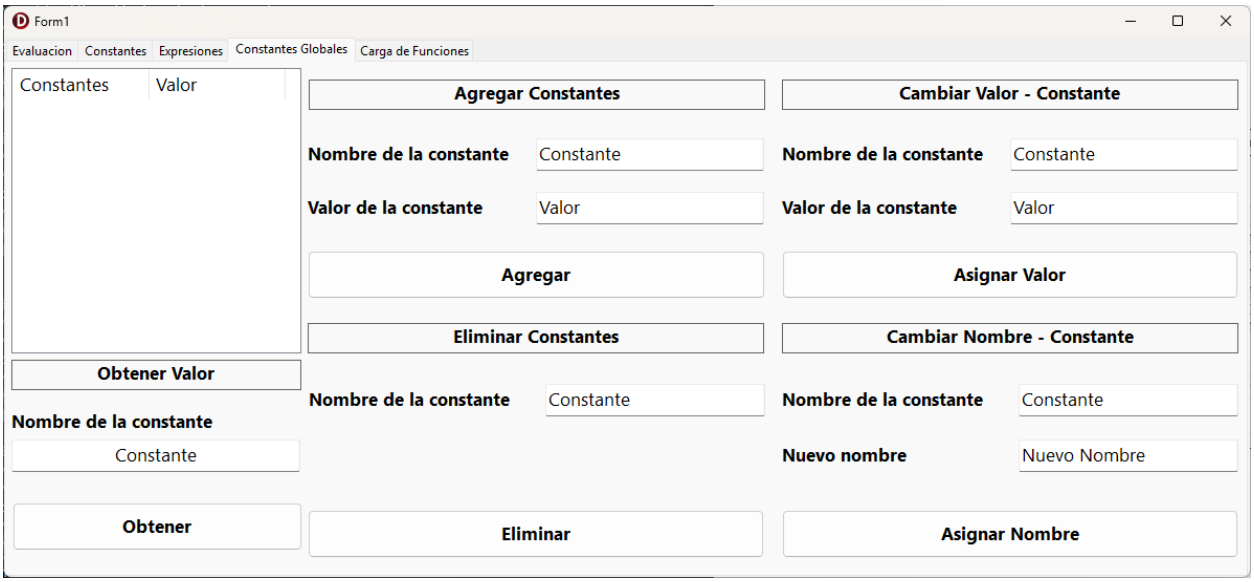
5. Cambiar expresión matemática: Emplea el nombre de variable asociado, para cambiar la expresión que contiene, permitiendo corregir la definición de la expresión declarada.

6. Cambiar nombre de la expresión: Permite cambiar el nombre de variable asociado.

**5.5.5.4 Ventana de Constantes Globales.** Este apartado presenta las siguiente funcionalidades, necesarias para visualizar y evaluar el correcto funcionamientos de la creación y manejo de las constantes globales.

**Figura 73**

*Aplicación: ventana de constantes globales*



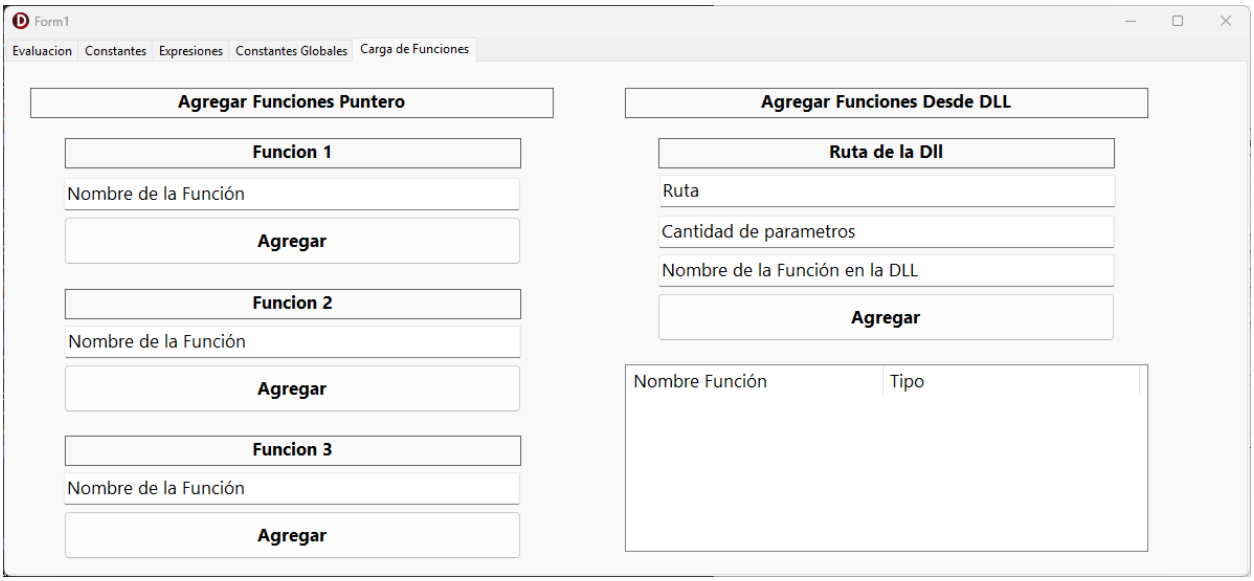

La pestaña *Constantes Globales* cuenta con las siguientes funcionalidades acorde a las capacidades del componente de código, continuación son descritas:

1. Tabla de Elementos: Presenta los elementos creados por el usuario y su respectivo valor.
2. Obtener Valor: Retorna el valor que contiene un elemento.
3. Agregar constantes: Permite crear una nueva constante en la hoja de expresiones.
4. Eliminar constantes: Elimina una constante existente.
5. Cambiar valor - constante: Cambiar el valor que almacena la constante.
6. Cambiar nombre - constante: Cambia el nombre de una constante, manteniendo el mismo valor.

**5.5.5.5 Ventana de Carga de Funciones.** Este apartado presenta las siguiente funcionalidades, necesarias para visualizar y evaluar el correcto funcionamientos de la agregación de funciones puntero y DLL.

**Figura 74**

*Aplicación: ventana de funciones*




La pestaña *Carga de Funciones* cuenta con las siguientes funcionalidades acorde a las capacidades del componente de código, continuación son descritas:

1. Agregar funciones puntero: Este apartado permite agregar tres funciones que han sido escritas en el código de la aplicación MELView, pasadas como puntero y agregadas por medio de la interface.

2. Agregar funciones desde DLL: Permite cargar funciones desde una DLL externa, para esto emplea una ruta absoluta de la DLL, cantidad de parámetros que admite la función y por ultimo el nombre de la función dentro de la DLL.

## 6. Conclusiones

El desarrollo del proyecto se llevo acabo de forma satisfactoria, esto gracias a la metodología empleada, lo que facilito adaptarse a los cambios y agregación de nuevas funcionalidades. La metodología en espiral permitió actuar adecuadamente ante diferentes inconvenientes que se encontraron durante el desarrollo del proyecto, como replantear el diseño general, mejoras o ajustes grandes durante los diferentes ciclos del proyecto, fue clave en la construcción del componente de código MathExpressionLeaf (MEL).

El soporte del proceso de desarrollo bajo la metodología en espiral, se almacena mediante el uso de un repositorio en git-hub, el cual cuenta con mas de 54 cambios guardados en el historial del repositorio, bajo tres ramas, las cuales representan o soportan el desarrollo incremental y cíclico que se realizo bajo las tres etapas iniciales planteadas. De esta manera se registra el proceso de desarrollo a lo largo del tiempo, es de destacar un inconveniente durante el desarrollo, el cual motiva a la creación del repositorio, ya que por un fallo en la computadora, los datos del proyecto se borran de forma permanente y es necesario reconstruir el código en base a los diagramas y una primera versión en código monolítico; por lo que el registro presenta como fecha inicial el mes de enero del año 2025, teniendo como fecha real de inicio el mes de agosto del año 2024.

Junto a esto se presenta una revisión de las diferentes pruebas realizadas, cuyos resultados sirven como soporte de los resultados alcanzados, asi como de la calidad del software descrita, es importante resaltar que se ponen a disposición de los evaluadores los documentos que contiene los resultados completos de las pruebas, terminando de dar soporte del trabajo realizado. Para la etapa de pruebas se emplearon alrededor de 600 expresiones diferentes para probar el componente de código, las cuales requirieron de un proceso de construcción manual, con el propósito de tener certeza de los resultados de las pruebas. Logrando con esto probar el cumplimiento de todos los requerimientos planteados y descritos en el documento.

Dentro de los resultados alcanzados se destaca las capacidades de interpretar cual-

quier tipo de expresiones bajo la definición soportada por Evolución (No soporta ecuaciones puesto que el símbolo igual corresponde a un operador de comparación), las cuales pueden contener operadores aritméticos, lógicos, comparativos, jerarquías de paréntesis y funciones; reconociendo y manejando al rededor de 50 tipos diferentes de errores contenidos en las expresiones, los cuales se relacionan con el léxico, la gramática, los tipos (Booleanos y Flotantes), la evaluación de las expresiones (división por cero) y relacionados con el uso del interprete. Esto junto a grandes capacidades para evaluar y manejar los diferentes elementos que puede contener, donde presenta tiempos de evaluación muy cercanos a funciones compiladas lo que pretende mejorar los tiempos de simulación en modelos de gran tamaño y complejidad, por ultimo, la capacidad de manejo de un volumen de hasta un millón de elementos, con lo que se espera satisfacer las necesidades que pueden presentar los modelos desarrollados por los usuarios de Evolución.

Por ultimo se destaca la extensibilidad del componente de software, lo que permitirá la continuación del proyecto, junto a su arquitectura modular e interface de uso que plantea una fácil integración y uso del componente en el software Evolución, así como en otras herramientas de software; es importante resaltar la aplicación demo desarrollada, la cual esta pensada como presentación del componente de código, ademas de ser una herramienta de soporte al proceso evaluativo del proyecto.

Se concluye de esta manera, que el interprete de expresiones desarrollado y diseñado en el presente proyecto de grado, cumplió con todos los requerimientos planteados, lográndose ademas, otros objetivos complementarios definidos como opcionales, que mejoran el proceso de interpretación de los diferentes tipos de expresiones matemáticas.

## 7. Recomendaciones

Para la continuación del proyecto se propone partir de la agregación de las siguiente funcionalidades, con el objetivo de familiarizare con la arquitectura del proyecto:

- **Arreglos:** Implica poder contar con funciones u operaciones que puedan trabajar con

conjuntos de elementos definidos por el usuario de la forma $[a, b, c, d]$ equivalente a una lista, suponiendo que los elementos estén definidos. Este proceso de integrar esta funcionalidad es relativamente sencillo dentro del interprete, mientras el manejo que deberá darle la lógica en dinámica de sistemas puede variar en su complejidad. Plantea una dificultad Alta - Critica.

- **Negación:** Es una operación que actualmente se realiza empleando la función *Not()* pero en lenguajes modernos esta operación se puede realizar mediante un símbolo como "!"(admiración) u otras notaciones. Esta característica implica cambios unicamente en el interprete por lo que es un ejercicio perfecto para entender la arquitectura modular integrada. Plantea una dificultad Baja.

- **Optimización de Operaciones:** Esta mejora involucra específicamente el árbol de sintaxis, el cual deberá ser recorrido en busca de operaciones constantes, lo que recae en $(a+3+4-c) \rightarrow (a+7-c)$. Este proceso aunque parece sencillo es complejo debido a que es necesario entender el papel que cumple el árbol de sintaxis, es de resaltar que esta actividad mejora el tiempo de evaluación y requiere modificaciones unicamente en el interprete, por lo que es perfecto para entender el funcionamiento del interprete. Plantea una dificultad Media.

- **Manejo de Espacios en Blanco:** Este problema es fundamental para la construcción de un interprete de lenguaje funcional, ya que implica reconocer el espacio en blanco como un delimitador, no tiene impacto alguno en el interprete actual, pero es un ejercicio fundamental que ayudara a entender en profundidad el funcionamiento del interprete. Plantea una dificulta Media.

- **Vectores:** Evolución cuenta en la actualidad con soporte para operaciones con vectores, pero el interprete no soporta esta funcionalidad, este problema, mas allá de mejorar o reemplazar la funcionalidad con la que ya se cuenta, plantea un ejercicio practico para entender el manejo de capacidades de calculo por parte del interprete; integrar esta

funcionalidad implica entender todo el proceso para agregar los bloques de código necesarios en las diferentes partes del interprete, ya que esta característica involucra todo el proceso de forma simple pero especifica, requiriendo de un conocimiento profundo de como funciona el interprete; es de recalcar que esta mejora se plantea en un sentido pedagógico por ello no se plantean cambios en el código fuente de Evolución. Plantea una dificultad Media - Alta.

- **Caracteres Especiales:** Jugar a agregar caracteres especiales, como aquellos que están definidos en la computadora pero de los cuales no podemos prescindir específicamente, es muy útil para comprender la extensibilidad del interprete, a la vez que se pueden ir agregando funcionalidades que permitan por ejemplo reconocer saltos de linea, lo que facilitara la implementación de un lenguaje de programación funcional a largo plazo. La dificultad es variable y depende de la característica con las que se requiera contar, muchas veces podría ser una característica invisible para el usuario final, la cual cumpliría un papel de captar información o datos importantes de la cadena.

Una vez se contemplan algunas funcionalidades integrables, se puede continuar con la construcción de un lenguaje funcional, para este propósito se recomienda como ejemplo el lenguaje LISP, el cual cuenta con una estructura y metodología ampliamente usadas en el mundo y conocidas, facilitando el aprendizaje y adopción del mismo.

Por ultimo y no menos importante, se puede contemplar la integración del interprete en Evolución, este proceso puede llegar a ser complejo, debido a que el software cuenta con una alta complejidad y con múltiples funcionalidades integradas, requiriendo una variedad de ajustes. Para este proceso se recomiendo estudiar el código fuente de Evolución junto a los documentos de soporte disponibles.

## Referencias Bibliográficas

Abelson, H., Sussman, G. J., & Sussman, J. (1996). *Structure and Interpretation of Computer Programs* (2nd). MIT Press / McGraw-Hill.

Ball, S. (2020, septiembre). *[InfoGraphic] DevOps, desarrollo y RAD Studio* [Blog, Embarcadero]. Consultado el 16 de septiembre de 2025, desde https://blogs.embarcadero.com/es/infographic-devops-development-and-rad-studio/?utm_source=chatgpt.com

Cantù, M. (2008, diciembre). *Delphi and Unicode* [Documento técnico publicado por Embarcadero Technologies. Consultado el 21 de julio de 2025]. https://d2ohlsp9gwqc7h.cloudfront.net/images/old/pdf/Delphi-Unicode181213.pdf

Casales, L. (2019, mayo). *Quién fue Edsger Dijkstra* [Accedido: 2 de abril de 2025]. https://es.linkedin.com/pulse/qui%C3%A9n-fue-edsger-dijkstra-leonardo-casales

Embarcadero Technologies. (2025). *Migration & Upgrade Center: Unicode* [Accedido el 21 de julio de 2025]. https://www.embarcadero.com/es/rad-in-action/migration-upgrade-center#unicode

Embarcadero Technologies. (s.f.). *Delphi* [Consultado el 19 de julio de 2025]. https://www.embarcadero.com/es/products/delphi

Hannon, B., & Ruth, M. (s.f.). *Modeling Dynamic Biological Systems* (Segunda Edición).

Hitchins, D. K. (s.f.). *System Engineering: A 21st Century Systems Methodology.*

Isee Systems, Inc. (s.f.). *About Isee Systems* [Consultado el 18 de julio de 2025]. https://www.iseesystems.com/about.aspx

le Riche, P. (2021). *FastMM4: Fast memory manager for Delphi and C++ Builder* [Versión 4.993, liberada el 10 de agosto de 2021. Consultado el 1 de agosto de 2025]. https://github.com/pleriche/FastMM4

Microsoft Support. (s.f.). *Control de la aplicación y explorador en la aplicación Seguridad de Windows* [Accedido el 21 de julio de 2025]. https://support.microsoft.com/es-

es/windows/control-de-la-aplicaci%C3%B3n-explorador-en-la-aplicaci%C3%B3n-seguridad-de-windows-8f68fb65-ebb4-3cfb-4bd7-ef0f376f3dc3

Moral, S. (s.f.). *Teoría de Autómatas y Lenguajes Formales* [Departamento de Ciencias de la Computación e I.A., ETSI Informática]. https://www.cubawiki.com.ar/images/4/47/Apunte_tleng_completo_2.pdf

Morales, A. G. (2011). *Manual desarrollo de software basado en tecnologías orientadas a componentes: formación para el empleo*. Editorial CEP.

Nystrom, R. (2021). *Crafting Interpreters*. Genever Benning.

Pressman, R. S. (2010). *Ingeniería del software: Un enfoque práctico* (Séptima edición). McGraw-Hill Interamericana Editores, S.A. de C.V.

Rastogi, R. (2015). An Exhaustive Review for Infix to Postfix Conversion with Applications and Benefits [Accedido: 2 de abril de 2025]. https://www.researchgate.net/publication/292137900_An_Exhaustive_Review_for_Infix_to_Postfix_Conversion_with_Applications_and_Benefits

Sosa, H. H. A., Dyner, I., Espinosa, A., Garay, H. L., & Sotaquirá, R. (2001). *Pensamiento sistémico: Diversidad en búsqueda de Unidad*.

Technologies, E. (2015). *Working with Packages and Components - Overview* [DocWiki, RAD Studio Athens. Consultado el 16 de septiembre de 2025]. Consultado el 16 de septiembre de 2025, desde https://docwiki.embarcadero.com/RADStudio/Athens/en/Working_with_Packages_and_Components_-_Overview

Technologies, E. (2025). *Partner Programs — Embarcadero* [Página web, consultado el 16 de septiembre de 2025]. Consultado el 16 de septiembre de 2025, desde https://www.embarcadero.com/es/resources/partner-programs?utm_source=chatgpt.com

Wallarm Learning Center. (2025, abril). *¿Qué es ASLR (aleatorización del diseño del espacio de direcciones)?* [Consultado el 21 de julio de 2025]. https://lab.wallarm.com/what/que-es-aslr-aleatorizacion-del-diseno-del-espacio-de-direcciones/?lang=es

Wolf, C. E. (s.f.). *Algoritmo de Shunting Yard, Dijkstra* [Texto original publicado por Carol E. Wolf de la Universidad Pace; adaptado por P. Oser. Accedido: 2 de abril de 2025]. https://mathcenter.oxford.emory.edu/site/cs171/shuntingYardAlgorithm/

# Apéndices

## Apéndice A

## ¿Qué es MathExpressionLeaf?

MathExpressionLeaf (MEL) es un componente de software que interpreta y almacenas expresiones matemáticas y constantes en tiempo de ejecución. Se diseña con el objetivo de ser integrable en software de simulación, los cuales requieran cargar y evaluar expresiones matemáticas de forma dinámica, es por esto mismo que cuenta con la capacidad de tener uno o más escenarios de evaluación.

MEL soporta la creación de una hoja de simulación con un conjunto de expresiones matemáticas (cada expresión cuenta con un nombre de variable asignado) y dos conjuntos diferentes de constantes:

- **Constantes globales:** Se emplean para almacenar valores que no cambian como pi y el número de Euler.
- **Constantes:** Se emplean para almacenar valores que cambian según el escenario definido para la evaluación.

Cada expresión puede emplear constantes generales o globales, conforme a como se defina y se requiera, contándose con la posibilidad de usar el valor resultante de evaluar otras expresiones, empleando para esto, su nombre de variable definido.

## Apéndice B

## Requisitos

El componente cuenta con los siguientes requisitos:

- Debe emplearse Rad studio 12 (Se desconoce si funciona en otras versiones del IDE).
- Debe compilarse sobre Windows 10 u 11.

Cabe aclarar que el componente de software no emplea elementos modernos de programación, empleando únicamente código puro en pascal para su desarrollo, por lo que debería ser compatible con al menos la versión 10 de Rad Studio.

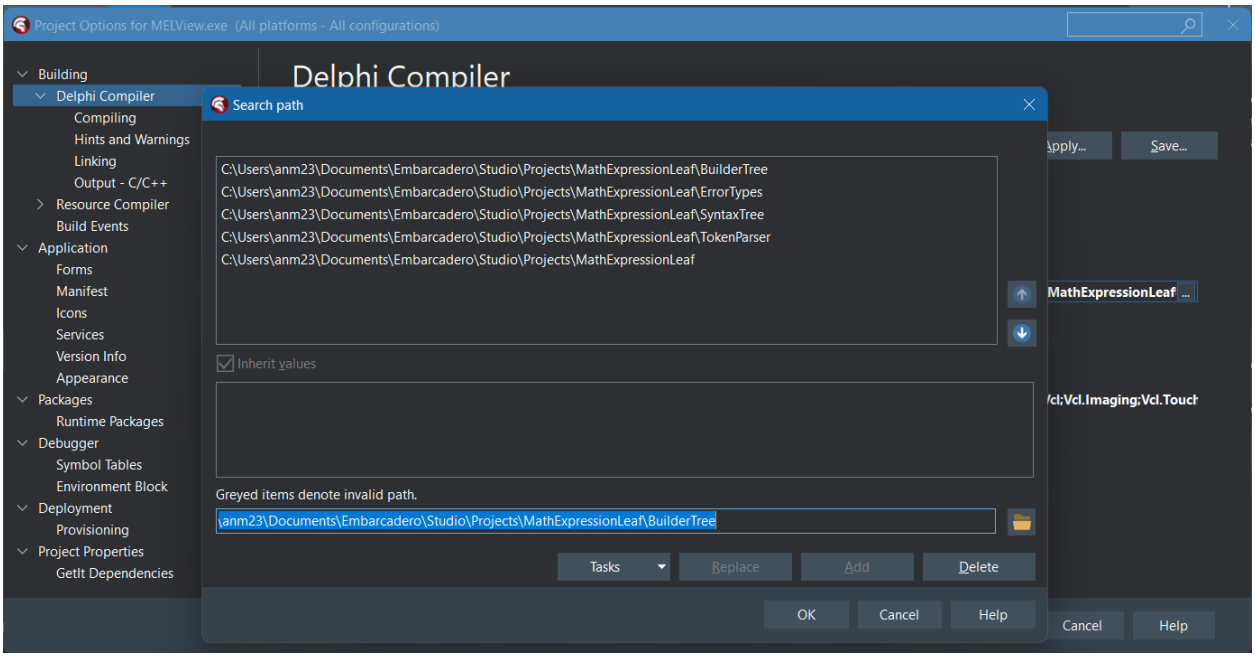
## Apéndice C

## Instalación

Para instalar el componente de código basta con agregar al Path de búsqueda de Rad Studio, cabe aclarar que se deberán incluir las subcarpetas del componente, esto con el objetivo de contar con un componente más organizado, o en su defecto emplear el archivo (.dcp). Como ejemplo de lo anterior se presenta la siguiente imagen:

**Figura 75**

*Agregación de rutas para la integración del interprete*




En la anterior imagen se agregan al path de búsqueda la carpeta general del proyecto llamada MathExpressionLeaf junto a 4 subcarpetas, las cuales corresponden a los nombres BuilderTree, ErrorTypes, SyntaxTree y TokeParser.

## Apéndice D

## Integración

La integración del componente está pensada para ser lo más simple y sencilla posible, para este propósito NO se delega responsabilidad sobre el cómo funciona el componente, el desarrollador que desee integrar el interprete en alguno de sus proyectos deberá usar únicamente la unidad UMathExpressionLeaf usando la clase MathExpressionLeaf, como ejemplo se presenta lo siguiente:

**Código 2.** *Integración del interprete*

```
Uses UMathExpressionLeaf;

Var MEL: TExpressionLeaf;

Implementation

MEL := TExpressionLeaf.Create;
```

La clase TExpressionLeaf contiene todos los métodos necesarios para administrar y evaluar los elementos matemáticos de la simulación. Los métodos se explicaran en los siguientes apéndices.

## Apéndice E

## Configuración del Idioma

Define en que idioma se devolverán los reportes de errores, recibe un AnsiString el cual debe ser equivalente a 'Español' o 'English', en caso de recibir algo diferente asignara por defecto el idioma español para el sistema del reporte de errores, este procedimiento se define como:

**Código 3.** *Método para la definición del lenguaje de presentación de los reportes de errores*

```
procedure SetLanguage(Language: AnsiString);
```

# Apéndice F

# Métodos de la Hoja de Expresiones

En este apéndice se detallan los métodos que permiten emplear y extender el comportamiento de la hoja de expresiones. Se incluyen operaciones con funciones puntero, DLLs, escenarios, constantes, expresiones y propiedades de acceso.

## Agregar Funciones Puntero

Agrega una función puntero definida como:

**Código 4.** *Definición requerida por la función puntero*

```
function NombreFuncion(X:TArray<PExtended>):Extended;
```

Los parámetros que recibe la función corresponden a el nombre que se utilizará para llamar a la función dentro de las expresiones, el número de parámetros que recibe la función (este valor debe ser el correcto de lo contrario arrojará error al compilar la función), un puntero a la función que se desea pasar y por último una cadena AnsiString que contendrá el reporte de errores de haberse.

**Código 5.** *Método para agregar una función puntero*

```
function AddFunction(const Name: AnsiString;
  const Parameters: Byte;
  const Fct:TMathFunction;
  out ErrorReport: AnsiString):Boolean;
```

## Agregar Funciones DLL

Agrega una función de forma dinámica mediante DLLs de la siguiente forma (No se emplea Sharemem):

**Figura 76**

*Formato de las funciones en la DLL requerido para la agregación*

```
function Prueba1(const X: array of PExtended): Extended; stdcall;
begin
  Result := X[0]^ + X[1]^ + X[2]^ + X[3]^;
end;
//
function Prueba2(const X: array of PExtended): Extended; stdcall;
begin
  Result := X[0]^ * X[1]^ * X[2]^ * X[3]^;
end;

function Prueba3(const X: array of PExtended): Extended; stdcall;
begin
  Result := X[0]^ - X[1]^ + X[2]^ * X[3]^;
end;

exports
  Prueba1 name 'Prueba1',
  Prueba2 name 'Prueba2',
  Prueba3 name 'Prueba3';


begin
end.
```

Los parámetros que recibe la función corresponden a el nombre que se utilizará para llamar a la función dentro de las expresiones este nombre también corresponde al nombre dentro de la DLL, el número de parámetros que recibe la función (este valor debe ser el correcto de lo contrario arrojará error al compilar la función), la ruto global donde se encuentra la DLL y por último una cadena AnsiString que contendrá el reporte de errores de haberse.

**Código 6.** *Método para agregar una función DLL*

```
function AddFunctionDll(const Name: AnsiString;
  const Parameters: Byte;
  const Dir:String;
  out ErrorReport: AnsiString):Boolean;
```

### Eliminar Funciones DLL

Elimina una función DLL previamente cargada, sino se encuentra retorna un reporte de error.

Los parámetros que recibe la función corresponden al nombre con el que se agregó la función DLL previamente y por último una cadena AnsiString que contendrá el reporte de errores de haberse.

**Código 7.** *Método para eliminar una función DLL*

```
function DeleteFunctionDll(const Name:AnsiString;
  out ErrorReport: AnsiString):Boolean;
```

### Obtener Valor

Permite obtener el valor de cualquier elemento matemático dentro de la hoja de expresiones, retorna un reporte de error sino encuentra el nombre del elemento. Las variables retornan cero si nunca se ha evaluado la expresión correspondiente.

Los parámetros que recibe la función corresponden al nombre con el que se agregó la función DLL previamente y por último una cadena AnsiString que contendrá el reporte de errores de haberse.

**Código 8.** *Método para obtener un valor asociado a un elemento*

```
function GeValue(const Name:AnsiString;
  out ErrorReport: AnsiString):Extended;
```

### Gestión de Escenarios

**Definir Escenario:** Permite definir el escenario bajo el cual se evaluarán las expresiones matemáticas, retorna False y un reporte de error en caso de no encontrar el reporte.

Los parámetros que recibe la función corresponden al nombre con el que se agregó el escenario previamente y por último una cadena AnsiString que contendrá el reporte de errores de haberse.

**Código 9.** *Método para elegir un escenario de evaluación*

```
function SetScenary(const Name:AnsiString;
  out ErrorReport: AnsiString): Boolean;
```

**Agregar Escenario:** Permite agregar un escenario bajo el cual se evaluarán las expresiones matemáticas, retorna False y un reporte de error en caso de no encontrar el reporte.

Los parámetros que recibe la función corresponden al nombre con el que se identificará el escenario previamente y por último una cadena AnsiString que contendrá el reporte de errores de haberse.

**Código 10.** *Método para agregar un escenario*

```
function AddScenary(const Name:AnsiString;
  out ErrorReport: AnsiString): Boolean;
```

**Eliminar Escenario:** Permite eliminar un escenario existente, retorna False y un reporte de error en caso de no encontrar el reporte.

Los parámetros que recibe la función corresponden al nombre con el que se agregó el escenario previamente y por último una cadena AnsiString que contendrá el reporte de errores de haberse.

**Código 11.** *Método para eliminar un escenario*

```
function DeleteScenary(const Name:AnsiString;
  out ErrorReport: AnsiString): Boolean;
```

**Cambiar Nombre de Escenario:** Permite cambiar el nombre de un escenario existente, retorna False y un reporte de error en caso de no encontrar el reporte.

Los parámetros que recibe la función corresponden al nombre con el que se agregó el escenario previamente, el nuevo nombre del escenario y por último una cadena AnsiString que contendrá el reporte de errores de haberse.

**Código 12.** *Método para cambiar el nombre de un escenario*

```
function SetNameScenary(const Name:AnsiString;
  const NewName:AnsiString;
  out ErrorReport: AnsiString): Boolean;
```

### Gestión de Constantes

**Agregar Constante:** Permite agregar constantes ya sean globales o generales, para esto emplea una variable de tipo, tenga en cuenta que el valor que se le asigne se les asignara a todos los escenarios, retorna False y un reporte de error en caso de no encontrar el reporte. Los parámetros que recibe la función corresponden al nombre de la constan, el valor que almacenará dicha constante, el tipo de constante y por último una cadena AnsiString que contendrá el reporte de errores de haberse.

**Código 13.** *Método para agregar una constante*

```
function AddConstant(const Name:AnsiString;
  const Value:AnsiString;
  const EType:TTypesElements;
  out ErrorReport: AnsiString): Boolean;
```

**Eliminar Constante:** Permite eliminar una constante previamente creada, retorna False y un reporte de error en caso de no encontrar el reporte.

Los parámetros que recibe la función corresponden al nombre con el que se agregó la constante previamente y por último una cadena AnsiString que contendrá el reporte de errores de haberse.

**Código 14.** *Método para eliminar una constante*

```
function DeleteConstant(const Name:AnsiString;
  out ErrorReport: AnsiString):Boolean;
```

**Cambiar Valor de Constante:** Permite cambiar el valor de una constante previamente creada, retorna False y un reporte de error en caso de no encontrar el reporte.

Los parámetros que recibe la función corresponden al nombre con el que se agregó la constante previamente, el nuevo valor que tendrá la constante y por último una cadena AnsiString que contendrá el reporte de errores de haberse.

**Código 15.** *Método para cambiar el valor asociado a una constante*

```
function SetValueConstant(const Name:AnsiString;
  const Value:AnsiString;
  out ErrorReport: AnsiString):Boolean;
```

**Cambiar Nombre de Constante:** Permite cambiar el nombre de una constante previamente creada, retorna False y un reporte de error en caso de no encontrar el reporte.

Los parámetros que recibe la función corresponden al nombre con el que se agregó la constante previamente, el nuevo nombre de la constante y por último una cadena AnsiString que contendrá el reporte de errores de haberse.

**Código 16.** *Método para cambiar el nombre de una constante*

```
function SetNameConstant(const Name:AnsiString;
  const NewName:AnsiString;
  out ErrorReport: AnsiString):Boolean;
```

### Gestión de Expresiones

**Agregar Expresión:** Permite agregar una expresión matemática y almacena su resultado en una variable que puede ser llama desde otras expresiones, tenga en cuenta que el nombre de la variable identifica a la expresión, retorna False y un reporte de error en caso de no encontrar el reporte.

Los parámetros que recibe la función corresponden al nombre de variable que tendrá la expresión, la expresión matemática, una cadena AnsiString que contendrá el seguimiento del error y por último una cadena AnsiString que contendrá el reporte de errores de haberse.

**Código 17.** *Método para agregar una expresión*

```
function AddExpression(const NameVariable:AnsiString;
  const Source:AnsiString;
  out ErrorTrace: AnsiString;
  out ErrorReport: AnsiString):Boolean;
```

**Eliminar Expresión:** Permite eliminar una expresión matemática, retorna False y un reporte de error en caso de no encontrar el reporte.

Los parámetros que recibe la función corresponden al nombre de variable de la expresión y por último una cadena AnsiString que contendrá el reporte de errores de haberse.

**Código 18.** *Método para la eliminación de una expresión*

```
function DeleteExpression(const Name:AnsiString;
  out ErrorReport: AnsiString):Boolean;
```

**Cambiar Expresión:** Permite cambiar la expresión asignada a una variable, retorna False y un reporte de error en caso de no encontrar el reporte.

Los parámetros que recibe la función corresponden al nombre de variable de la expresión, la nueva expresión matemática, una cadena AnsiString que contendrá el seguimiento del error y por último una cadena AnsiString que contendrá el reporte de errores de haberse.

**Código 19.** *Método para cambiar la expresión asociada a una variable - expresión*

```
function SetExpression(
  const Name:AnsiString;
  const Expr:AnsiString;
  out ErrorTrace,ErrorReport: AnsiString):Boolean;
```

**Cambiar Nombre de Variable:** Permite cambiar el nombre de variable con el que se definió la Expresión, tenga en cuenta que esto puede generar errores al evaluar expresiones previamente definidas y que usen un nombre de variable que no existe, retorna False y un reporte de error en caso de no encontrar el reporte.

Los parámetros que recibe la función corresponden al nombre de variable de la expresión, nuevo nombre de variable para la expresión y por último una cadena AnsiString que

contendrá el reporte de errores de haberse.

**Código 20.** *Método para el cambio de nombre de una variable - expresión*

```
function SetNameVariable(const Name: AnsiString;
  const NewName:AnsiString;
  out ErrorReport: AnsiString):Boolean;
```

**Evaluar Expresión:** Permite evaluar una expresión al pasar su nombre de variable, retorna False y un reporte de error en caso de no encontrar el reporte.

Tenga en cuenta que esta función contiene dentro un try catch que atrapara errores que puedan ocurrir al evaluar la expresión, este tipo de error corresponde a errores de dominio matemático, un ejemplo sencillo de estos es la división por cero, de esta forma cada función cuenta con el manejo adecuado de su error si es que cuenta con uno.

Los parámetros que recibe la función corresponden al nombre de variable de la expresión y por último una cadena AnsiString que contendrá el reporte de errores de haberse.

**Código 21.** *Método para la evaluación de expresiones*

```
function Evaluate(const Name:AnsiString;
  out ErrorReport: AnsiString):Extended;
```

**Evaluación Rápida de Expresión:** Permite evaluar una expresión al pasar su nombre de variable, retorna False y un reporte de error en caso de no encontrar el reporte.

Tenga en cuenta que esta función no cuenta con un try catch, por lo que no atrapara el error y podría ocasionar que su programa dejara de funcionar. Los errores conocidos están manejados y no causaran ningún tipo de error, pero al emplear funciones puntero o dinámicas mediante dlls, generaran error.

Los parámetros que recibe la función corresponden al nombre de variable de la expresión y por último una cadena AnsiString que contendrá el reporte de errores de haberse.

**Código 22.** *Método para la evaluación rápida de expresiones*

```
function FastEvaluation(const Name: AnsiString;
  out ErrorReport: AnsiString): Extended;
```

### Verificación de Declaraciones

Permite saber si un elemento ha sido declarado bajo un nombre especifico, retorna un Booleano y un reporte de error en caso de no encontrar el reporte.

Los parámetros que recibe la función corresponden al nombre de variable de la expresión.

**Código 23.** *Método para la verificación de declaración de elementos*

```
function CheckMembership(const Name: AnsiString):Boolean;
```

## Apéndice G

## Propiedades de la Hoja de Expresiones

Las propiedades se definieron con la finalidad de facilitar el acceso al estado de la hoja, donde este corresponde a un conjunto de elementos. Úselos bajo su responsabilidad, en la práctica no son necesarios, pero puede existir algún caso en el que resulten útiles. Modificar los fuera del contexto de la hoja de expresiones puede ocasionar errores.

**Código 24.** *Propiedad de la hoja de expresiones*

```
property NameLeaf: AnsiString read FNameLeaf write FNameLeaf;

property AExpressions: TArray<RExpression> read FAExpressions
  write FAExpressions;

property DExpressions: TDictionary<AnsiString, TCExpression>
  read FDExpressions write FDExpressions;

property DVariables: TDictionary<AnsiString, PExtended>
  read FDVariables write FDVariables;

property DConstants: TDictionary<AnsiString, PExtended>
  read FDConstants write FDConstants;

property Scenary: AnsiString read FScenary write FScenary;

property DScenarios: TDictionary<AnsiString, TList<Extended>>
  read FDScenarios write FDScenarios;

property ListNames: TDictionary<AnsiString, TTypesElements>
  read FDListNames write FDListNames;

property ExpressionBuilder: CExpressionBuilder
  read FExpressionBuilder;
```

## Apéndice H

## Repositorio GitHub

Para el desarrollo del proyecto fue necesario contar con un repositorio en GitHub que permitiera gestionar las versiones del código, ya que la metodología en espiral, al no establecer un camino fijado, favoreció la refinación progresiva de la estructura, la cual se ajustó a la solución esperada en función de los requerimientos obligatorios y de aquellos definidos como opcionales.

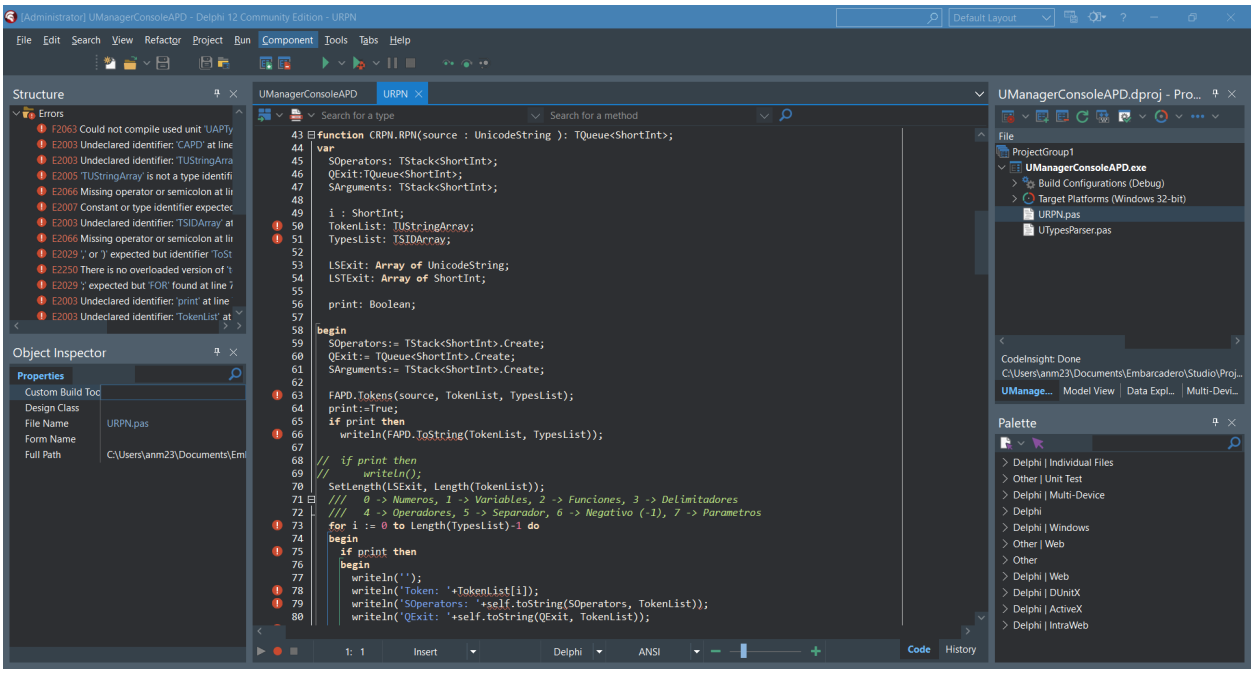
La creación del repositorio también se ve motivada debido a la perdida de un primer prototipo en el cual, se creaba un interprete muy sencillo basado en un autómata monolítico, junto a un proceso simple de evaluación basado en el algoritmo de Shutinyard de Edsger Dijkstra, esta primera versión caía en malas practicas o a lo que se le conoce como código espagueti, donde cualquier modificación o intento por ampliar las capacidades de interpretación

recaía en errores de lógica, que cada vez eran mas complejos de solucionar. Es importante resaltar que también se contaba con una primera versión del autómata lista para la etapa de pruebas exhaustivas, de la cual, no se logro recuperar nada y se debió reconstruir todo partiendo de una versión con errores y prematura del código monolítico del tokenizador.

En enero de 2025 el equipo de cómputo donde se desarrollaba el intérprete sufrió una falla que provocó la pérdida total de la información almacenada en las carpetas Documentos, Descargas y en parte de otras ubicaciones. Aunque se intentó recuperar el código con diferentes programas de recuperación de archivos, solo se obtuvieron fragmentos incompletos que sirvieron como base para rehacer el tokenizador, un proceso que requirió cerca de un mes, a continuación se presenta parte de ese código:

**Figura 77**

*Código recuperado del Tokenizador con múltiples errores*




Posterior a esto se desarrolla el proyecto empleando el repositorio GitHub con el cual se logra solventar problemas similares, entre los cuales se presentan:

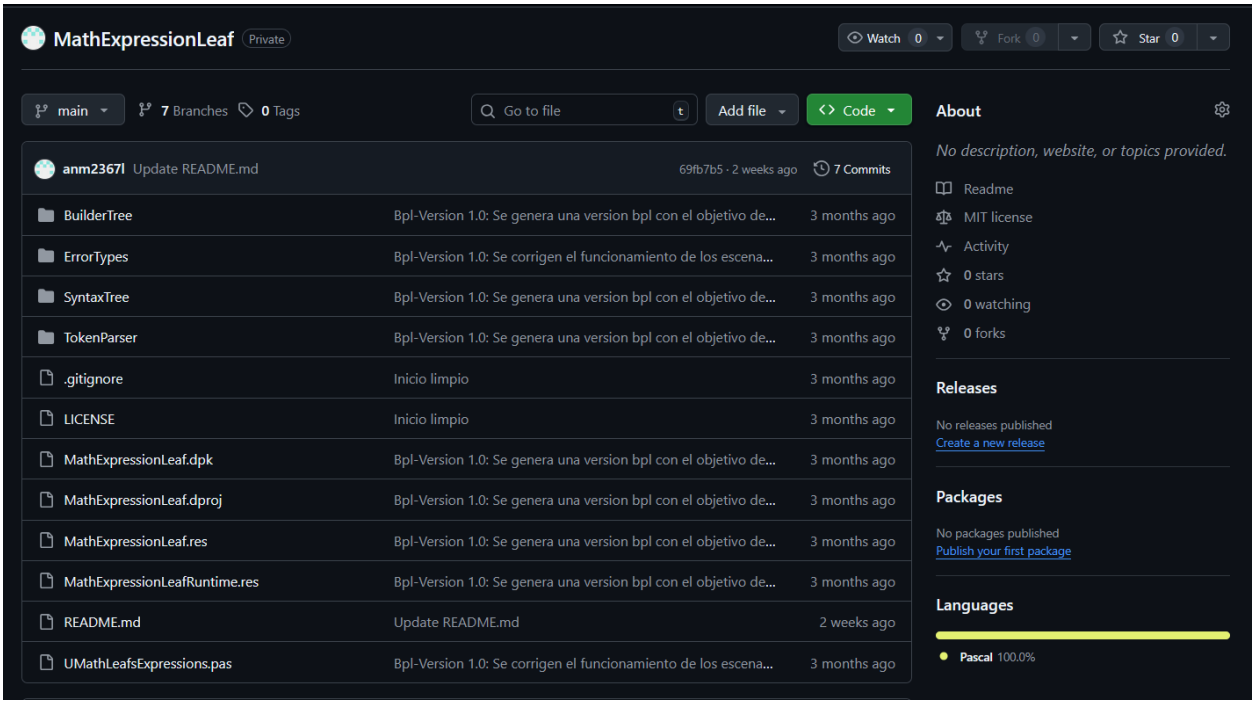
- Cambios en el código que producían errores.

- Introducción de errores por equivocación o desconocimiento.
- Formateo del equipo de computo.

Gracias a este enfoque se consolidó una rama principal que contiene la última versión estable del proyecto, donde el repositorio recibe el nombre de *MathExpressionLeaf* y se presenta en la siguiente imagen:

**Figura 78**

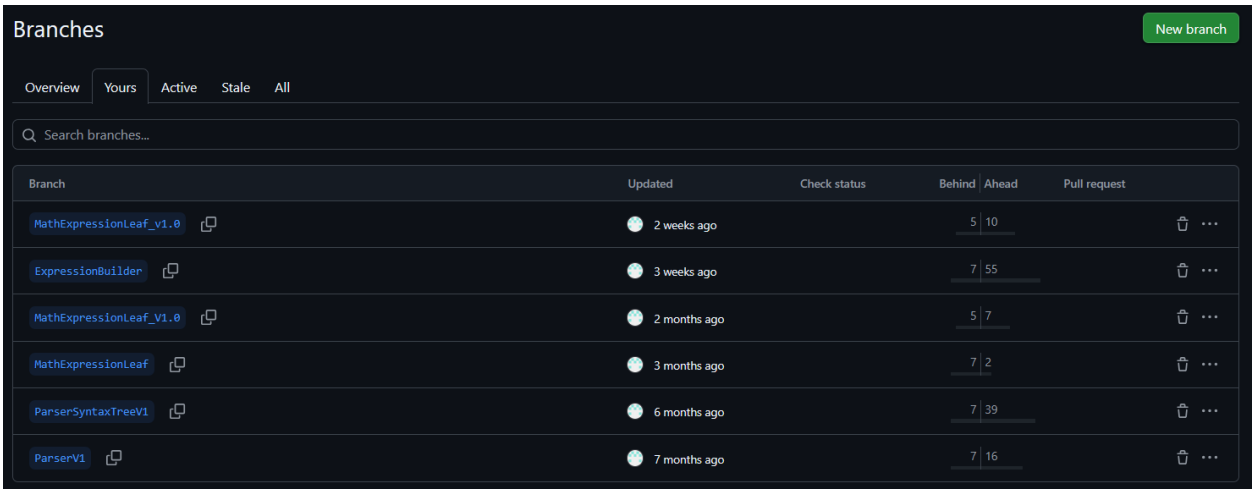
*Versión estable disponible en la rama principal (Main)*




De igual manera, el historial de ramas refleja la reconstrucción completa del componente de código, proceso que se llevó a cabo entre enero y febrero de 2025. La siguiente figura presenta el registro de ramas utilizadas:

**Figura 79**

*Ramas empleadas en el repositorio GitHub*




## Apéndice I

## Historial de Commits en el Repositorio GitHub

En este apéndice se abordara la presentación del historial de cambios en el repositorio y su progreso, para esto se presentan a continuación las cuatro ramas principales que se emplearon para la construcción del interprete de expresiones matemáticas, es de recalcar que estas ramas representa los ciclos principales de desarrollo bajo la metodología en espiral. Las ramas abordaron el desarrollo como se presenta en la siguiente lista:

1. **ParserV1:** Esta rama engloba todo el desarrollo y ciclos en el proceso de construcción del componente encargado de tokenizar la cadena.

2. **ParserSyntaxTreeV1:** Esta rama engloba todo el desarrollo y ciclos en el proceso de construcción del componente encargado de la representar expresión matemática mediante un árbol de sintaxis abstracto.

3. **ExpressionBuilder:** Esta rama engloba todo el desarrollo y ciclos en el proceso de construcción del componente encargado de transformar el árbol de sintaxis en un

objeto evaluable que representa exactamente la información almacenada en cada cadena o expresión matemática.

4. **MathExpressionLeaf_v1.0:** Esta rama engloba la etapa final, en la cual se ordeno la estructura de componentes en carpetas, junto a cambios y correcciones pequeñas en el funcionamiento del interprete. Representa la versión estable del componente antes de ser llevada a la rama principal (Main).

## Tabla 11

*Historial de commits de la rama ParserV1*

| ID | Fecha | Descripción |
|---|---|---|
| 28ccdab | 2025-01-24 11:27:42 | Initial commit. |
| 6eaddb8 | 2025-01-24 11:54:31 | Definición de la estructura (Inicial): Se establecieron las clases y su herencia como marco general de desarrollo. |
| 45006d5 | 2025-01-24 12:57:28 | Definición de la estructura (Inicial): Se estructuraron clases en base al diagrama, implementando herencia, métodos abstractos y virtuales. |
| e3bdff9 | 2025-01-24 19:19:20 | Definición de la estructura (Inicial): Se agregó el código inicial sin probar de cada función, basado en una versión funcional previa. Incluye código parcial en UStates, UTransition y UTypesParser. |
| 40f2170 | 2025-01-26 00:48:31 | Definición de la estructura (Inicial): Se completaron las clases y elementos del algoritmo. Aunque el código está completo, todavía no funciona correctamente y no se han realizado pruebas. |
| 29b5f77 | 2025-01-26 02:13:56 | Definición de la estructura (Inicial): El código compila y funciona con cadenas de prueba, aunque aún no se ha probado con el autómata completo. Se espera que los errores provengan de detalles de lógica o imprevistos menores. |
| dc57b7d | 2025-01-26 02:50:37 | Definición de la estructura (Inicial): Se solucionaron errores simples en el funcionamiento de clases y métodos. Además, se realizaron pruebas de tiempo promedio para generar tokens de una cadena. |
| 1702864 | 2025-01-28 00:18:27 | Definición de la estructura (Inicial): Se crearon funciones y procedimientos en UTest para probar el autómata. Incluye pruebas de rendimiento al tokenizar funciones, pruebas de reconstrucción de cadenas desde tokens y validación de transiciones del autómata con distintos caracteres. Los resultados se guardan en la carpeta Test_Records. |
| 529aa79 | 2025-01-29 01:38:31 | Definición de la estructura (Inicial): Se agregó un procedimiento de prueba para validar que las cadenas ingresadas contengan caracteres válidos de ASCII limitado y extendido. Esta funcionalidad también se añadió al Analyst, permitiendo tokenizar cadenas limpias y habilitadas. |
| 962925e | 2025-01-29 19:17:57 | Definición de la estructura (Inicial): Se agregó a cada clase capacidad de modularidad mediante interfaces como contrato, lo que facilita la extensibilidad del código y lo hace más compacto. No se requieren constantes cambios y las pruebas mostraron estabilidad. Aunque el componente parece más lento al tokenizar, esto puede deberse a los numerosos writeln de depuración que serán eliminados. |
| 8e0ebd7 | 2025-01-29 20:09:44 | Definición de la estructura (Inicial): Se hicieron ajustes pequeños en UTest para mejorar el almacenamiento de las pruebas. |
| 3d2cab1 | 2025-01-30 00:39:20 | Definición de la estructura (Inicial): Se realizaron cambios ligeros, aunque se requiere reformular el manejo de las funciones para solucionar un error con la lectura del símbolo. |
| f4f715b | 2025-01-31 01:00:34 | Definición de la estructura (Inicial): Se logró implementar los cambios pensados. Ahora el autómata tokeniza adecuadamente la cadena e identifica elementos que no corresponden con la lógica, lo que permite una comunicación más clara sobre los errores en las funciones ingresadas. Además, se modularizó más el proceso, independizando partes del autómata y facilitando la extensibilidad del algoritmo. |
| 46b28fd | 2025-01-31 01:00:34 | Definición de la estructura (Inicial): Se completaron los ajustes y se solucionó el error al mostrar caracteres erróneos en la cadena. Con esta actualización se da por completada la fase inicial. Se realizaron pruebas en cada punto de interés del componente, permitiendo concluir que el código es estable y apto para su uso. |
| 3078e20 | 2025-02-02 01:31:22 | Definición de la estructura (Inicial): Se completaron los ajustes y se solucionó el error al mostrar caracteres erróneos en la cadena. Con esta actualización se da por completada la fase inicial. Se realizaron pruebas en cada punto de interés del componente, permitiendo concluir que el código es estable y apto para su uso. |
| 8f15eb0 | 2025-02-03 22:14:59 | Definición de la estructura (Inicial): Se realizaron cambios que mejoran la eficiencia del algoritmo, el costo computacional se redujo en gran medida al dejar de emplear una función de Delphi que borraba espacios en blanco; en su lugar se agregó el manejo de espacios en blanco al autómata, basado en el conteo de espacios, lo que mejoró la eficiencia en más de un 60 %. |

## Tabla 12

*Historial de commits de la rama ParserSyntaxTreeV1*

| ID | Fecha | Descripción |
|---|---|---|
| 341ec33 | 2025-02-05 19:51:08 | Definición de la estructura (Intermedia): Se inicia la construcción del árbol de sintaxis buscando estructuras más compactas. En teoría, se pasa de $n$ a $n/2$ iteraciones, mejorando la velocidad de respuesta. |
| 980bf4a | 2025-02-05 23:24:21 | Definición de la estructura (Intermedia): Se agregan transiciones para reconocer operadores lógicos de comparación. |
| a00702e | 2025-02-10 00:36:51 | Definición de la estructura (Intermedia): Se crea una función para construir un árbol de sintaxis óptimo para operaciones matemáticas. Aún falta probar otras operaciones. |
| 5fd1bf5 | 2025-02-11 17:14:54 | Definición de la estructura (Intermedia): Se añade la lógica para manejar números negativos como suma/resta agrupada, factorizando el signo negativo. También se ajusta el manejo de paréntesis con pesos de delimitadores mayores, encapsulando operaciones y cerrando ramas automáticamente. |
| e1ceab1 | 2025-02-11 18:37:29 | Definición de la estructura (Intermedia): Se agregan funciones para la abstracción de operaciones de agregar, insertar y crear ramas. |
| c2110dc | 2025-02-13 15:10:10 | Definición de la estructura (Intermedia): Se corrigen errores relacionados con paréntesis y la construcción del árbol de sintaxis. |
| c9d1a0c | 2025-02-13 15:55:38 | Definición de la estructura (Intermedia): Se corrige un error en el manejo de paréntesis: el peso de `]` fue asignado como 6 en lugar de `[` con peso 5. Se añaden pruebas para las reglas de sintaxis. |
| 9468b98 | 2025-02-15 20:01:25 | Definición de la estructura (Intermedia): Se corrigen errores en la lógica de manejo de paréntesis para operadores de multiplicación/división. Se añaden cadenas de prueba para validar el árbol de sintaxis. |
| faf90fd | 2025-02-16 02:30:16 | Definición de la estructura (Intermedia): Se corrige un error de conversión entre `UnicodeString` y `AnsiString`, que no reconocía ñ/Ñ. No afecta si en el futuro se emplea `UTF16`. |
| a5303d1 | 2025-02-16 16:17:10 | Definición de la estructura (Intermedia): Se corrige un error al leer espacios iniciales en la función `isAnsii`. Además, se define la lógica de manejo de funciones (`funct`, `comma`, `delimiterf`) vinculadas a la tokenización. |
| fc72cd2 | 2025-02-17 21:47:56 | Definición de la estructura (Intermedia): Se redefine la forma de destruir nodos del árbol de sintaxis, usando una lista global `LNodes` para liberar memoria al final del proceso. Esto permite ejecuciones paralelas sin interferencias. Se completan operaciones, incluyendo operadores de comparación. Falta probarlos exhaustivamente. |
| ddc00cb | 2025-02-18 03:12:41 | Definición de la estructura (Intermedia): Se agrega un estado para reconocer el valor 0 en el tokenizador. El código se muestra resistente al agregar nuevos estados y transiciones. Queda pendiente validar la función 96 y la construcción correcta del árbol. |
| f068ac1 | 2025-02-19 20:01:25 | Definición de la estructura (Intermedia): Cambios pequeños. Se desarrolla la gramática de operadores lógicos. Backup de cambios. |
| 0ffafd9 | 2025-02-23 19:16:45 | Definición de la estructura (Intermedia): Se agrega función para analizar la gramática de operadores lógicos (`>`, `<`, etc.). |
| 38d8120 | 2025-02-24 02:12:01 | Definición de la estructura (Intermedia): Se finaliza la función gramatical encargada de operadores lógicos. Falta definir casos particulares de combinaciones como `>=`, `<=`, `<>`. |
| 48b88c3 | 2025-02-26 18:52:42 | Definición de la estructura (Intermedia): Se completa el análisis de gramáticas, quedando pendiente la fase de pruebas. El intérprete tokeniza, crea el árbol de sintaxis y analiza reglas gramaticales satisfactoriamente. |
| 3a08a99 | 2025-02-28 01:01:45 | Definición de la estructura (Intermedia): Se inician pruebas de gramáticas y del árbol de sintaxis. Se corrige una transición omitida y se añaden más cadenas de prueba, incluidas las erróneas anteriores. Se cambia el uso de destructor directo por `free`. |
| fa1bd25 | 2025-03-03 02:49:35 | Definición de la estructura (Intermedia): Se modifican constructores de `AST`, `FactoryFunctions` y `TokenParser` (añadida para modularizar). Se implementa el patrón Singleton y se ajustan llamadas a constructores/destructores. El código pasa todas las pruebas. |
| 3b6d803 | 2025-03-07 22:21:58 | Definición de la estructura (Intermedia): Se completa la construcción intermedia con avances en `MathExpression` y sus unidades relacionadas. El proceso de creación y optimización del árbol finaliza. |
| 0fd3a48 | 2025-03-11 22:13:21 | Definición de la estructura (Intermedia): Se resuelven problemas de fuga de memoria en el tokenizador y sus pruebas. Este apartado queda resuelto y sin fugas. Se emplea `FastMM4` para seguimiento de memoria. |
| 86187b7 | 2025-03-11 23:16:00 | Definición de la estructura (Intermedia): Se resuelven problemas de fuga de memoria trabajando en el syntaxtree. Se corrigen fugas al emplear `buildSyntaxTree`. |
| 72b8e27 | 2025-03-18 23:50:46 | Definición de la estructura (Intermedia): Se resuelven problemas de fugas de memoria en la construcción y revisión del árbol de sintaxis. El proceso cerrado no presenta fugas, pero en pruebas individuales aún aparecen errores por referencias no liberadas. Se soluciona encapsulando la lógica de construcción del árbol, permitiendo al compilador liberar correctamente los punteros. |
| a1825fa | 2025-03-25 22:21:34 | Definición de la estructura (Intermedia): Se resuelven todos los problemas de fugas de memoria, con esto se actualizan todos los test. Aunque en pruebas individuales aún persisten fugas menores, al ejecutar el componente como caja negra estas son nulas. Se finaliza esta etapa. |

## Tabla 13

*Historial de commits de la rama ExpressionBuilder*

| ID | Fecha | Descripción |
|---|---|---|
| 2dc0ece | 2025-04-03 19:16:20 | Definicion de la estructura (Final): Se añade capacidad de diagnosticar errores, indicando tipo, descripción y posición inicial en la expresión. |
| 5d5607b | 2025-04-05 01:26:57 | Definicion de la estructura (Final): Evolución de componentes para reconocer operadores lógicos. Se añade la función return, se optimiza almacenamiento de elementos eliminando hash y usando nombres definidos por el usuario. |
| 37c4791 | 2025-04-05 22:47:51 | Definicion de la estructura (Final): Se realizan pequeñas correcciones en los mensajes de error. Se añade la opción de cambiar idioma (inglés y español). |
| 3f1f4b9 | 2025-04-25 04:59:02 | Definicion de la estructura (Final): Se finaliza la construccion del codigo que permite contruir la expresion, incluyendo evaluacion, declaracion de variables y revision de tipado. El interprete admite extended y boolean, lo que facilita la depuracion. |
| b357495 | 2025-04-30 18:53:12 | Definicion de la estructura (Final): Se inicia la depuracion del codigo basado en pruebas unitarias y funcionales. Se desarrollan test para manejo de constantes y constantes globales. Todo funciona sin fugas de memoria. |
| 0502cd2 | 2025-05-05 03:06:48 | Definicion de la estructura (Final): Pruebas enfocadas en expresiones booleanas y tipado. Se corrigen errores y se realizan ajustes. Se han probado 20 tipos de errores. El componente soporta hasta 2 millones de caracteres. |
| 9f88dce | 2025-05-08 17:08:05 | Definicion de la estructura (Final): Se corrigen errores en el proceso de revisar la existencia de funciones y constantes. Se agregan mas expresiones en pruebas para medir la capacidad del componente. Se han testeado 393 cadenas (154 con errores, 239 sin errores). Hasta el momento el componente se muestra solido y soporta cadenas extensas. |
| 8aebba3 | 2025-05-15 01:59:35 | Definicion de la estructura (Final): El proceso de evaluacion se realiza de forma natural y sin errores. Se solucionaron otros problemas, se redujo el tamaño del componente final y se optimizó su eficiencia. Se solucionaron errores relacionados con operaciones booleanas y manejo de elementos. |
| bf9b391 | 2025-05-16 04:57:27 | Definicion de la estructura (Final): Se mejora el proceso de probar las funciones, al crear un conjunto de funciones como expresion en cadena y como una funcion compilada. Se realizan otros ajustes. |
| 2b09ee5 | 2025-05-20 00:25:34 | Definicion de la estructura (Final): Se corrigen todas las funciones, se revisa la precision de los calculos y estos presentan una baja propagacion de error. Faltan mas pruebas para este apartado, pero la revision no encontro mayor solucion, ya que se basa en el manejo nativo de delphi. Se da por completada la etapa final, quedando pendiente unicamente el proceso de depuracion de funciones individuales y agregar metodos numericos. |
| 566ea95 | 2025-06-13 01:44:16 | Definicion de la estructura (Final): Se completa la depuracion de mathleafexpression, este proceso incluye la revision de fugas de memoria, en todo el proceso no se presenta ninguna fuga de memoria, se da por finalizada esta etapa. Queda pendiente probar las funcionalidades de agregar funciones tanto en codigo delphi como en dll. |
| 6419c63 | 2025-06-13 23:39:44 | Definicion de la estructura (Final): Se completa la definicion de las funcionalidades de agregacion de funciones tanto como extencion dentro del lenguaje delphi, como por agregacion por medio de funciones contenidas desde una dll. Se realizan las pruebas correspondientes del manejo de memoria, se concluyen las pruebas sin fugas de memoria y con completa funcionalidad. |
| be7e1f6 | 2025-06-14 03:11:00 | Definicion de la estructura (Final): Se organiza la estructura del componente de software. Se comprueba que el componente continua funcionando. No presenta errores. |
| 9ae68f5 | 2025-08-09 17:25:34 | Actualizacion de rutas |
| d0bff86 | 2025-08-10 02:44:29 | Actualizacion de rutas |
| 218a39e | 2025-08-23 18:08:12 | cambios sin guardar |

**Tabla 14**

*Historial de commits de la rama MathExpressionLeaf_v1.0*

| ID | Fecha | Descripción |
|---|---|---|
| 9b108e8 | 2025-06-14 03:42:12 | Inicio limpio |
| d4823ec | 2025-06-14 03:45:00 | Bpl-Version 1.0: Se genera una versión bpl con el objetivo de facilitar la integración. |
| 66a7939 | 2025-06-16 03:03:50 | Bpl-Version 1.0: Se corrige readme. |
| ce9aff1 | 2025-06-16 03:06:26 | Bpl-Version 1.0: Se corrige readme. |
| 8ce9514 | 2025-06-16 03:10:59 | Bpl-Version 1.0: Corrigiendo cambios entre versiones, falso error, esta es la última versión del código. |
| 7ac2a96 | 2025-06-17 18:12:15 | Bpl-Version 1.0: Corrigiendo cambios entre versiones, falso error, esta es la última versión del código. |
| 840703c | 2025-06-17 18:18:30 | Bpl-Version 1.0: Agregando ejecutable de la interfaz de prueba, la cual busca presentar al usuario las funcionalidades del componente. |
| ae13417 | 2025-06-21 19:48:43 | Bpl-Version 1.0: Correcciones según la implementación de la interfaz gráfica, con esto se termina de revisar y probar el componente de código. Faltarían pruebas de caja negra por parte de un usuario externo. |
| e2b7773 | 2025-07-10 16:15:44 | Se agrega la carpeta `examples` con un ejemplo de aplicación del componente. |
| cf847ac | 2025-08-25 00:50:46 | Corrigiendo errores: se corrige el manejo de errores en funciones con más o menos parámetros de los esperados, al igual que el almacenamiento por defecto del valor obtenido tras evaluar la expresión. |
| cce38fd | 2025-08-25 15:14:24 | Se agrega documentación en las funciones, indicando qué tarea realiza cada función o conjunto de funciones. |
| 1dfe8eb | 2025-08-25 15:40:38 | Corrección de errores introducidos en la última corrección: el error aparecía al devolver el número de parámetros de la función en lugar del índice de la función encontrada. |