Informe del 2do Proyecto de Programación

Pixel Wall-E

Ciencia de la Computación

MATCOM 2024-2025

Meylí Jiménez Velázquez

C -121

Repositorio: MenwaLab/MosaicDroid

Introducción

Imagina esto: eres un artista en Venecia, Italia, y abrumado con clientes ansiosos por tener sus retratos pintados en un sereno paseo en góndola, programaste a un pequeño robot, Mosaic Droid, para que te ayude a dar vida a tus visiones de arte en píxeles.

Mosaic Droid es un programa que lee e interpreta comandos del lenguaje Pixel Wall-E, y pinta en píxeles sobre un canvas cuadrado.

Características de la Interfaz

Formato de Entrada

1. Editor de texto con numeración de líneas:
    - Al inicializar la ventana principal (**MainWindow**), se suscribe el evento **Editor.TextChanged**, que se dispara cada vez que el usuario escribe, pega o borra texto. En el manejador **Editor_TextChanged** se llama a **UpdateLineNumbers()**, función que:
    - Cuenta cuántas líneas existen en el texto actual (**Editor.LineCount**).
    - Genera una cadena con números de línea ascendentes, separados por saltos de línea.
    - La asigna a un control de texto de solo lectura ubicado a la izquierda.
    - De esta forma, cada vez que el usuario crea una nueva línea, la numeración se actualiza al instante. Cuando durante la compilación se detecta un error se construye un mensaje que incluye la ubicación exacta (**[línea,columna]**) y la descripción, y se muestra sobre la ventana con **MessageBox.Show(…)**. El usuario ve así claramente en qué fila y columna ocurrió la excepción, facilitando la depuración.

2. Importación de scripts con el botón **Cargar (Load)**

El comando “**Cargar (Load)**” abre un diálogo de archivos de Windows (**OpenFileDialog**) configurado con Filter = "**Mosaic scripts|*.pw**". Así, el usuario solo puede seleccionar ficheros con extensión **.pw**. Si confirma, leemos el contenido completo de la ruta elegida y lo volcamos en el editor:

De esta forma, el historial de código guardado se carga al instante para su edición o ejecución.

Redimensionar | Ejecutar | Guardar :

1. Canvas cuadrado con tamaño configurable

El lienzo de dibujo se basa en un **UniformGrid** llamado **PixelGrid**. Por defecto,

`CanvasSize = 40`, lo que crea una cuadrícula de 40×40 celdas blancas. Al pulsar el botón “**Redimensionar (Resize)**”, se abre un cuadro de texto que permite al usuario introducir un nuevo tamaño hasta un máximo de 300. Internamente, el método **ResizeCanvas()** ejecuta:

- Cálculo dinámico del tamaño de celda:

  ```
  cell = (CanvasSize > 0 && width > CanvasSize) ? Math.Floor(width / CanvasSize) : 15;
  ```

  De este modo, si el lienzo es muy pequeño (por ejemplo, **CanvasSize=10**), cada celda crece automáticamente (15 px mínimo) para que siga viéndose cuadriculada.

- Ajuste de filas y columnas:

  ```
  PixelGrid.Rows = PixelGrid.Columns = CanvasSize;
  ```

- Reconstrucción de las celdas: Borramos todos los hijos de **PixelGrid.Children**, luego se crea **CanvasSize²** bordes (Border) con fondo blanco y línea gris clara, y se añade en orden.

Este algoritmo garantiza que, sin importar cuán pequeña o grande sea la cuadrícula, el usuario siempre reciba un mosaico uniforme.

2. Ejecución del mosaico y manejo de errores

Al pulsar "**Ejecutar(Run)**":

- Se cancela cualquier ejecución previa y salva temporalmente el contenido del editor en "Code.pw".
- Fase de análisis estático:
    - Lexico: **Compiling.Lexical.GetTokens(...)** recoge tokens y errores léxicos.
    - Sintáctico: **Parser.ParseProgram()** construye el árbol de sintaxis y detecta errores de sintaxis.
    - Semántico: **prog.CheckSemantic(...)** valida tipos y contexto.
- Si existen errores en esas fases, se monta un mensaje con cada error (**[línea,columna] Mensaje**) y se muestra con **MessageBox.Show**, abortando la ejecución.
- Interpretación: Se crea un **InterpreterVisitor(CanvasSize, runErr)** y llama a **VisitProgram(prog)** dentro de un **Task.Run**.
    - Si salta una **PixelArtRuntimeException**, primero pinta en el canvas todo lo que se había procesado hasta el fallo (**PaintCanvas(interp)**), luego muestra el error.
    - Si no hay excepciones, va directamente a **PaintCanvas(interp)** y el mosaico completo se pintará.

Esta secuencia permite aislar claramente las etapas de compilación, proporcionar mensajes de error precisos y, en caso de error en tiempo de ejecución, ofrecer feedback parcial sin perder el progreso.

3. Guardado de scripts con el botón **Salvar(Save)**

La función conecta un **SaveFileDialog** con **Filter = "Mosaic scripts|*.pw"**. Así, el usuario puede exportar su programa actual a disco en formato **.pw**, listo para compartir o **Cargar(Load)**.

4. Control de música de fondo: **Mute/Unmute**

La música se inicia en **StartMusic()**, donde comprueba que exista el archivo MP3 en **Assets/Osole_mio.mp3**, lo asigna a **BgMusic.Source** y llama a **BgMusic.Play()**. El evento **BgMusic_MediaEnded** reinicia la posición a cero y vuelve a reproducir. De este modo la canción se reproducirá indefinidamente. El botón **MuteBtn** alterna **BgMusic.IsMuted** y actualiza su etiqueta (**Content**) para mostrar "**Silenciar(Mute)**" o "**Reproducir(Unmute)**" según el estado.

5. Interfaz multilenguaje (**Español/Inglés**)

En el **ComboBox** de idiomas (**LangCombo**), al cambiar la selección se ejecuta **LangCombo_SelectionChanged**, que:

- Recupera el **Tag** del **ComboBoxItem** (“es” o “en”).
- Ajusta **Thread.CurrentThread.CurrentUICulture** a la cultura correspondiente.
- Llama a **ReloadAllTexts()**, que reasigna:
  1. Etiquetas y botones (**ResizeBtn, LoadBt**n, etc.) con sus cadenas de recursos.

  2. El panel derecho de documentación (**DocsBox**), leyendo el fichero **InstructionDocs_<lang>.txt**.

De esta forma, los textos de la UI, panel de ayuda y mensajes de error cambian al idioma elegido, garantizando accesibilidad y usabilidad para los usuarios.

Arquitetura y Lógica del Proyecto

Tokenizer

La etapa de *tokenización* convierte el texto fuente en unidades léxicas (tokens) básicas. En **MosaicDroid.Core.Token**:

- **TokenType** define categorías como **Instruction, GoTo, Operator, Bool_OP, Assign, Integer, Variable, Function, String, Color, Delimeter, Label, Jumpline**.
- **TokenValues** agrupa valores literales (por ejemplo, **"Spawn", "Color", "Move",** etc.).

1. Extensión del lenguaje:

- Se registran todas las instrucciones originales (**Spawn, Color, Size, DrawLine**, …) y ahora una nueva: **Move**, con su propio **TokenValues.Move**.
- Las funciones integradas (**GetActualX, GetColorCount, IsBrushColor**…) también se declaran como tokens de tipo **Function**.
- Al igual que todas las operaciones booleanas y aritméticas especificadas en los requisitos y las etiquetas y variables.

2. Secuencia de Tokens (**TokenStream**)

TokenStream encapsula la lista de tokens y un índice de posición:

- **Advance**() devuelve el token actual y avanza la posición.
- **MoveNext**(k) salta k posiciones.
- **Next**(type) comprueba y consume el siguiente token si coincide con type.
- **CanLookAhead**(k) y **LookAhead**(k) permiten inspeccionar tokens futuros sin avanzar.

Al refinar **LookAhead** y añadir **CanLookAhead**, se evita el error "**índice fuera de rango**" cuando se llama a **LookAhead** (por ejemplo, con una línea incompleta como **Spawn(0,0**.

3. Inicialización de Palabras Clave y Operadores (**Analyzer**)

La clase **Compiling.Lexical** configura el analizador léxico registrando:

- Operadores aritméticos con **RegisterOperator("+", TokenValues.Add)** . Análogamente con ***, -, /, %,** ****.
- Operadores booleanos (**&&, ||, ==, >=, <=, >, <**).
- Asignación con **<-**.
- Delimitadores (**(,), ,, [,]**).
- Instrucciones y funciones con **RegisterKeyword("Spawn", TokenValues.Spawn), RegisterKeyword("GetColorCount", TokenValues.GetColorCount)**, etc. Análogamente con las funciones.
- Textos literales ("…") con RegisterText("\"", "\"").

  Nota: La clase **TokenValues** actúa como fuente única de valores constantes; el **analizador** y el **tokenizador** usan estas mismas constantes para garantizar consistencia.

4. Análisis Léxico (**LexicalAnalyzer**)

El **lexer** integra **TokenReader**, **operators, keywords y texts** para producir tokens:

1. Lectura de líneas en blanco y saltos: cuando **Peek**() es '**\n'**, genera **Jumpline**.
2. Ignorar espacios en blanco: **ReadWhiteSpace()**.
3. Etiquetas: detecta [...], emitiendo **Delimeter / Label / Delimeter**.
4. Identificadores: **ReadID**(out id), luego:
   - Si keywords.**Contains**(id), emite **Instruction, GoTo o Function** según el valor.
   - Si va tras **paréntesis/coma/asignación**, se trata como **Variable**.
   - Si va al final de línea, se considera **Label**.
5. Números: **ReadNumber**(out number) → Integer.
6. Textos y colores: **MatchText**(...) genera String o Color.
7. Símbolos: **MatchSymbol**(...) itera sobre operators, emite **Operator, Bool_OP, Assign o Delimeter.**

8. Error de carácter: **UnrecognizedChar** si no empata nada.

Finalmente, devuelve **IEnumerable<Token>** que alimenta al **TokenStream**.

Parser

Es la fase que toma la secuencia de tokens generada por el **Lexer** y construye el Árbol de Sintaxis Abstracta (**AST**). Su objetivo es validar la estructura gramatical y traducir cada instrucción, etiqueta, asignación y expresión en nodos.

Entrada

- Recibe un **TokenStream** ya tokenizado y una lista en la que irá acumulando errores de parseo o semánticos tempranos.

- **ParseProgram**()
  Punto de entrada. Crea un **ProgramExpression** y un listado temporal de **ASTNode**.

Desecha líneas en blanco iniciales y verifica que el primer token sea **Spawn**(...); en caso contrario, llama a **ErrorHelpers** para guardar este error y termina el parseo.

Bucle principal y switch

Dentro de while **(_stream.CanLookAhead())**, el Parser mira siempre el token actual con

`var la = _stream.LookAhead();` y usa un **switch** sobre `la.Type`:

- **TokenType.Jumpline**: Simplemente salta líneas en blanco:
- **TokenType.Label**: Cuando el lexer emite un token de tipo **Label**, el Parser:

    1. Comprueba con un **Regex** que no coincida con nombres de instrucción (**Spawn**...) mal interpretados.
    2. Consume el token con **_stream.Advance()** y registra la etiqueta en **program.LabelIndices**.
    3. Detecta duplicados

```
if (program.LabelIndices.ContainsKey(lbelTok.Value))

       ErrorHelpers.DuplicateLabel(_errors, lbelTok.Location,lbelTok.Value);
else
    program.LabelIndices[lbelTok.Value] = lbelTok.Location;
```

4. Exige un salto de línea después y avanza.

- **TokenType.Instruction**
  Llama a **ParseInstruction**(), obtiene un nodo concreto (ej. **DrawLineCommand**), lo agrega a statements y después invoca **EnsureNewlineAfter**(node) para verificar nueva línea.

- **TokenType.GoTo**
  Similar: **ParseGoTo**() devuelve un **GotoCommand**, que también comprueba más tarde en la semántica si la etiqueta existe.

- **TokenType.Variable**
  Discrimina entre asignaciones (**x <- expr**) y etiquetas independientes(variable seguida de salto de línea), o bien emite

```
ErrorHelpers.UnexpectedToken(_errors, la.Location, la.Value);
```

- Si no encaja en ninguna forma válida:
  `default`
  Cualquier otro token también se reporta con **ErrorHelpers.UnexpectedToken(...)** y luego se sincroniza hasta la siguiente línea en blanco (**Synchronize**()).
- Este método Synchronize garantiza que cuando haya un error de sintaxis, permita al programa "recuperarse" y seguir analizando sin que todo se rompa. **`Synchronize`**( ) avanza *rápido* hasta el próximo salto de línea, para retomar el análisis desde un punto coherente.
- Tras el **`while`**, si todavía hay tokens y el siguiente es `TokenType.Jumpline`, entonces hace un único **`_stream.MoveNext().`**
- Con esto "consume" esa línea en blanco, dejando el cursor ya en la primera posición útil de la siguiente línea de código.

Construcción de cada nodo

ParseInstruction()

Mediante **switch(instr.Value)** crea el nodo correcto:

Cada comando lee sus argumentos con **ParseArgumentList**(), que consume paréntesis, comas y devuelve una lista de expresiones.

- **ParseAssignment**()
  Consume el '**<-**' y construye sub-árbol de expresión

- **ParseGoTo()**
  Maneja la sintaxis exacta **GoTo [label] (condition)** extrayendo los delimitadores [] y () en orden.

Prioridad de operadores y AST

Para garantizar precedencia correcta (por ejemplo, || sobre &&, y multiplicación sobre suma), el Parser implementa métodos en cascada:

```
ParseExpression() → ParseLogicalOr()
```

- **ParseLogicalOr()**
  Mientras el token sea **Bool_OP** y **Value == TokenValues.Or:**

```
var right = ParseLogicalAnd();
left = new LogicalOrExpression(left, right, left.Location);
```

1. **ParseLogicalAnd()**
   Análogo para **&&,** llamando a **ParseEquality**().

2. **ParseEquality()**
   Para **==**, construye **LogicalEqualExpression**.

3. **ParseRelational()**
   Menor, mayor, etc., cada uno como nodo distinto.

4. **ParseAdditive()**
   Detecta + y -, anidando Add o Sub, llamando internamente a **ParseTerm**().

5. **ParseTerm()**
   Detecta *, /, %, creando Mul, Div, **ModulusExpression**, y llama a **ParsePower**().

6. **ParsePower()**
   Soporta exponenciación recursiva (^).

7. **ParseFactor()**
   Extrae literales (**Number, StringExpression, ColorLiteralExpression**), variables,

llamadas a función (**ParseFunctionCall**()), o expresiones entre paréntesis.

Este diseño “en etapas” garantiza que, al construir el AST, los nodos de mayor precedencia queden más profundos en el árbol. Por ejemplo, en a + b * c, **ParseTerm**() primero creará un **Mul**(b,c), y luego **ParseAdditive**() generará **Add**(a, **Mul**(b,c)).

Registro de errores

Siempre que algo no encaja, el Parser invoca métodos de **ErrorHelpers**:

Cada llamada añade un **ParseError** a la lista **_errors**, pero el Parser intenta continuar (“**error recovery**”) moviéndose hasta el siguiente **Jumpline** con **Synchronize**(), de modo que se puedan reportar múltiples errores en una sola pasada.

Con todo ello, al terminar **ParseProgram**() el Parser devuelve un **ProgramExpression** con:

- Un diccionario **LabelIndices** lleno de etiquetas y posiciones.
- Una lista **Statements** con cada nodo AST en orden secuencial.
- Un listado de **Errors** que recoge todos los fallos de sintaxis detectados.

Este AST es la base para la siguiente fase de análisis semántico y, finalmente, la ejecución.

Analisis Técnico Detallado del AST

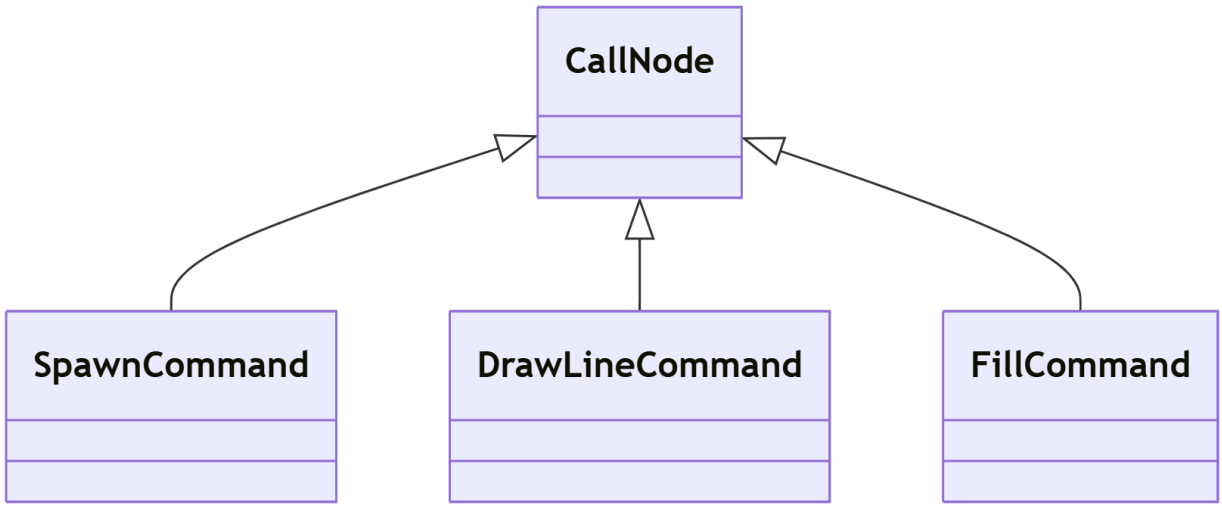
Expresiones Atómicas (Atoms)

- Propósito: Representan los bloques fundamentales del AST (hojas del árbol sintáctico).
- Comportamiento:
    1. **Number:**
        - Almacena valores numéricos (`double`).
        - `IsInt` verifica si es entero sin conversión explícita.
        - Semántica siempre válida (no requiere contexto).

2. **StringExpression:**
    - Elimina comillas automáticamente en el constructor.
    - Tipo fijo `ExpressionType.Text`.
3. **VariableExpression:**
    - Consulta el contexto para verificar existencia y tipo.
    - Reporta error si la variable no está declarada.
4. **JumpLineExpression:**
    - Marcador sintáctico (no afecta semántica).
5. **NoOpExpression:**
    - "Placeholder" para operaciones inválidas.
    - Tipo `ErrorType` para propagar errores.

Operadores Aritméticos (Add y similares)

- Flujo:
    1. CheckSemantic:
        - Verifica que ambos operandos sean `ExpressionType.Number`.
        - Propaga errores de subexpresiones.
    2. Evaluate:
        - Ejecuta recursivamente los operandos.
        - Calcula el valor resultante.
- Extensibilidad: Operadores como `Subtract`, `Multiply` siguen la misma estructura en `BinaryExpression`.



- **SpawnCommand**:
    - Garantiza un único `Spawn` por programa (`ctx.SpawnSeen`).
- **DrawLineCommand**:
    - **`ExtraArgumentChecks`** valida rangos:

```
 ArgumentSpec.EnsureDirectionInRange(...); // dirX/dirY en [-1,0,1]

ArgumentSpec.EnsurePositive(...); // distancia > 0
```

Análogamente para **DrawCircle** y **DrawRectangle.**

- **FillCommand**: Sin lógica adicional (comando sin parámetros).

Manejo de Colores (WPF)

Componentes:

- **`ColorCommand`**: Instrucción para establecer color.
- **`ColorLiteralExpression`**: Representa literales de color.

```
// En ColorCommand:

ColorValidationHelper.ValidateColorArgument(Args, 0, Location, errors);


// En ColorLiteralExpression:

public ColorLiteralExpression(string color, ...)

{

    Value = color; // Almacena string crudo ("#RRGGBB", "Red", etc.)

}
```

- Compatibilidad WPF:
    - Los colores se manejan como strings.

- `ColorValidationHelper` verifica formatos válidos usando `System.Windows.Media.Colors`.

Etiquetas (Labels)

- Registros:
  - Las etiquetas se almacenan en **`ProgramExpression.LabelIndices`**.
  - El contexto (**`ctx`**) centraliza las declaraciones para resolución de `GOTO`.

Instrucción GOTO

Flujo Semántico:

1. Verifica sintaxis de la etiqueta (**`IsValidLabel`**).
2. Comprueba existencia en el contexto (**`ctx.IsLabelDeclared`**).
3. Valida que la condición sea booleana:

Funciones (**FunctionCallExpression**)

Mecanismo:

```
public override bool CheckSemantic(...)
{
    // Reutiliza validación de CallNode via CommandNode
    var tempNode = new CommandNode(Name, Args, Location);
    return tempNode.CheckSemantic(...);
}
```

- Herencia:
  - Funciones como `GetColorCount` heredan de `FunctionCallExpression`.
  - Añaden validaciones específicas (ej: `ValidateColorArgument`).
  - Así para el resto de las funciones

Patrón de Diseño:

- Reutilización: La validación se delega a `CallNode` evitando duplicación.
- Extensibilidad: Nuevas funciones implementan `Evaluate` y `CheckSemantic` específicos.
- Nuevos comandos (ej. `MoveCommand`) se añaden extendiendo `CallNode` sin modificar código existente.
- Los nuevos comandos/funciones no requieren modificar el núcleo del intérprete.
- El parser genera comandos mediante `TokenValues`, sin lógica incrustada.
- Patrón Visitor:
    - Permite añadir nuevas operaciones (ej. depuración, optimización) creando nuevos visitors sin tocar los nodos AST.

Operadores Booleanos

```
public override void Evaluate()

{

    // Soporta números y textos

    if (Left.Type == ExpressionType.Number)

        Value = (double)Left.Value > (double)Right.Value;

    else

        Value = string.Compare((string)Left.Value, (string)Right.Value) > 0;

}
```

- Genericidad:
    - `CheckSemantic` fuerza operandos del mismo tipo (numérico o textual).
    - Otros operadores (<, ==, etc.) siguen idéntica estructura.

Sistema de Ejecución y Validación

1. Semantic Checker con Herencia

Componentes Clave:

- **ArgumentRegistry**: Catálogo centralizado de especificaciones para funciones/comandos
- **ArgumentSpec**: Define aridad y tipos esperados
- **ColorValidator**: Valida colores usando el sistema WPF

Flujo de Validación Semántica:

1. CallNode.CheckSemantic():
    - Consulta **`ArgumentRegistry`** para obtener especificaciones
    - Verifica aridad (número de argumentos)
    - Valida tipos de argumentos
2. Validación Específica (en comandos como `DrawLineCommand`):
    - Usa **`ArgumentSpec.EnsureDirectionInRange()`** para valores [-1,0,1]
    - Usa **`ColorValidationHelper`** para validar formatos de color

2. Sistema de Ejecución con Patrón Visitor

Componentes:

- **IStmtVisitor**: Interfaz para comandos
- **MatrixInterpreterVisitor**: Implementa la ejecución
- **ExpressionEvaluatorVisitor**: Evalúa expresiones

Flujo de Ejecución:

1. `VisitProgram()` inicia la ejecución
2. Para cada statement:
    - Si es GOTO con condición verdadera: salta a etiqueta
    - Si es comando: llama al método `Visit` correspondiente
    - Incrementa contador de pasos (para luego detectar posibles bucles infinitos)

3. Sistema de Errores Multilenguaje

Arquitectura:

Implementación Multilenguaje:

Para proporcionar funcionalidad de múltiples idiomas, se utilizó la biblioteca **System.Resources de .NET**, que permite gestionar las cadenas de los diferentes idiomas a través de archivos **.resx**. Así el programa pueda cambiar entre inglés y español según la preferencia del usuario.

¿Cómo funciona el cambio de idioma según la entrada del usuario?:

Cuando el usuario selecciona la opción **"Idioma(Language)"** en el menú principal, el programa le pide que elija entre español o inglés.Según la elección, se establece una cultura específica utilizando la clase **CultureInfo de .NET**:

Esta cultura activa determina qué archivo se utilizará para mostrar los mensajes de error, botones y panel de información.

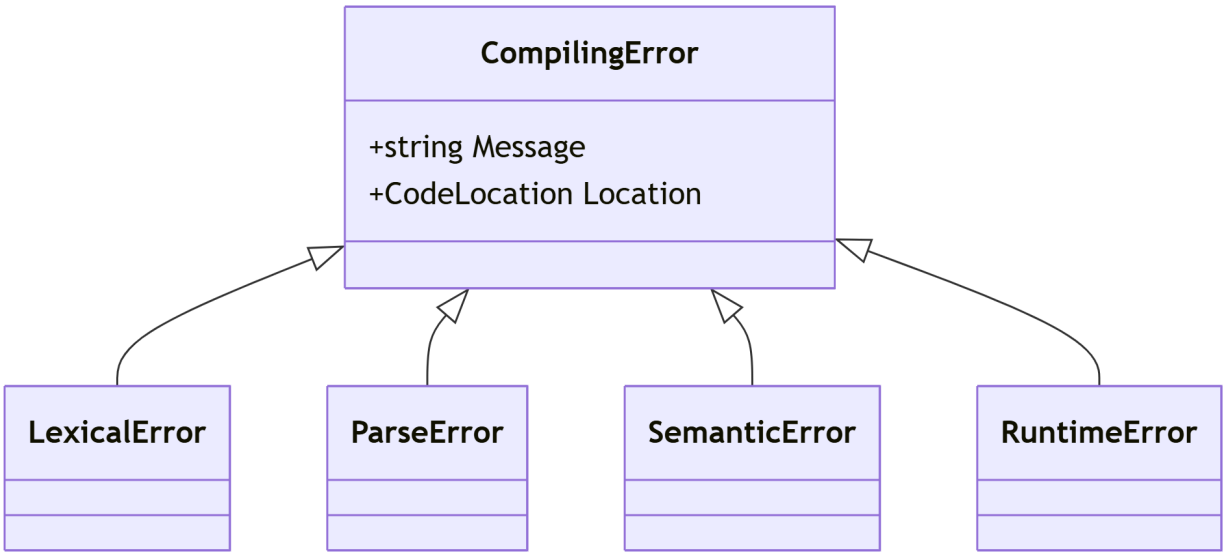
Se crearon archivos .resx: uno para español (**Strings.es.resx**) y otro para inglés **(Strings.resx)**. Cada archivo contiene las cadenas localizadas para los mensajes que aparecen en el programa. De esta manera, se facilita la adición de nuevos idiomas en el futuro implemente creando un archivo .**resx** para el nuevo idioma.

Cada vez que se necesita mostrar un mensaje en el programa, el texto se obtiene dinámicamente del archivo de recursos correspondiente utilizando la clase **ResourceManager**. Esta clase proporciona acceso a las cadenas localizadas basadas en la cultura actual.

Para mantener una implementación clara y evitar errores, cada mensaje que debe ser mostrado al usuario en el código está vinculado a una entrada única en los archivos **.resx.** Esto asegura que todos los textos del programa puedan ser traducidos y mantenidos de forma centralizada. Además, este enfoque permite cambiar el idioma sin modificar el código base, ya que todos los textos están separados en archivos .resx.

Esta implementación asegura que los usuarios disfruten de una experiencia personalizada, adaptada al idioma que elija, mejorando así la accesibilidad y usabilidad del programa

Jerarquía de Errores:



- Léxicos: Caracteres no reconocidos, strings sin cerrar
- Sintácticos: Paréntesis faltantes, tokens inesperados
- Semánticos: Variables no declaradas, tipos incorrectos

- Runtime: División por cero, movimiento fuera de límites

Resumen del Flujo:

Código fuente → TokenReader → LexicalAnalyzer (+ keywords/operators/texts) → TokenStream → Parser → AST → Semántica → Interprete → Canvas

Este pipeline modular hace que cada capa (tokenización, léxica, parsing, semántica, ejecución) permanezca independiente y fácil de mantener