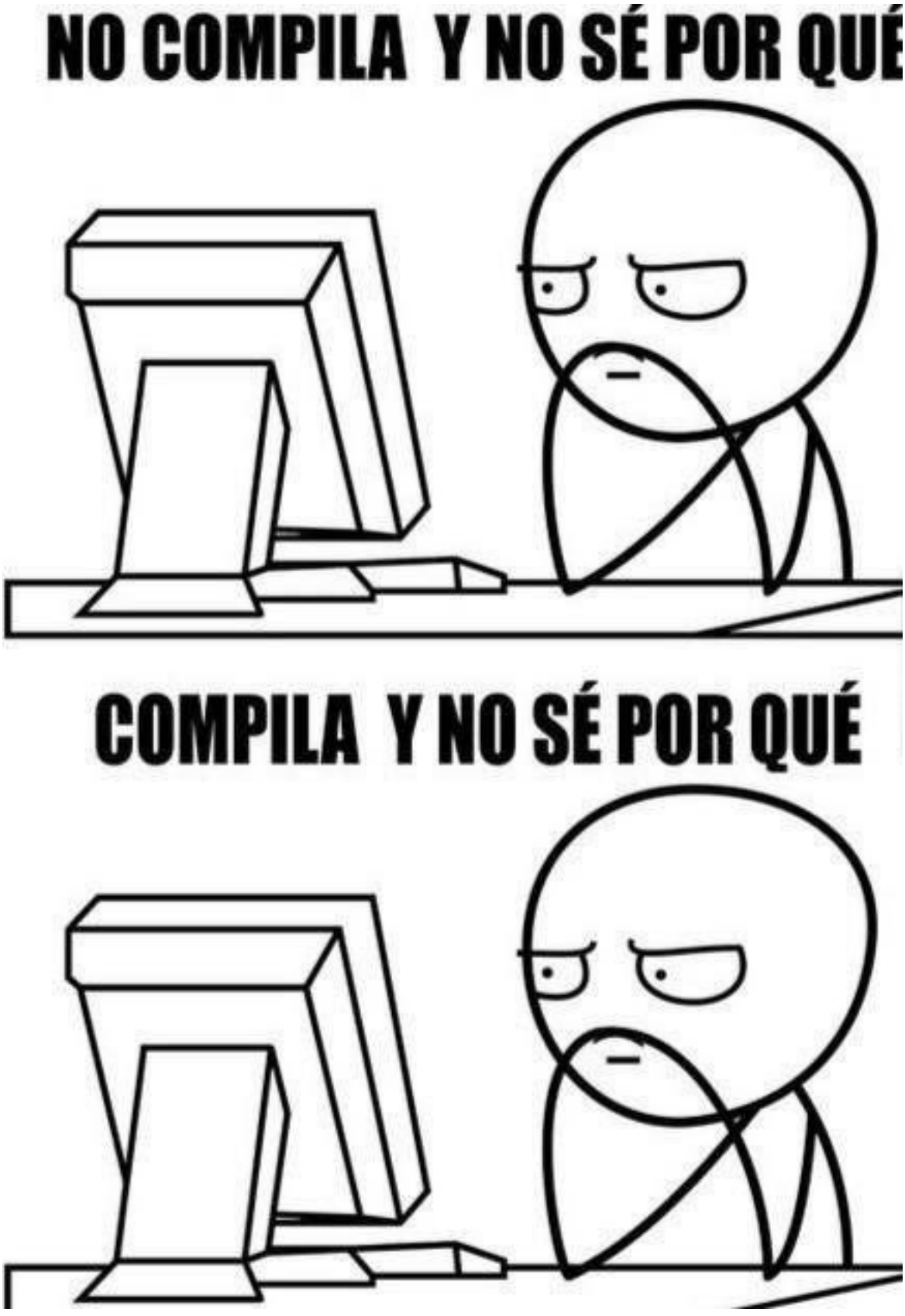
Informe del Proyecto: Intérprete del orientado Lenguaje de Propósito Específico (DSL)
Nombre: Ana Laura Hernández Cutiño – C121



Índice

Introducción

Objetivo del proyecto

En base a la realización de un previo videojuego de cartas desarrollado en la plataforma de Unity, se tomo como plan el implementar una expansión a la aplicación que permitiera a los futuros usuarios, relacionados con el ámbito de la programación o con los suficientes conocimientos al respecto, el poder incluir dentro de la misma cartas con su respectivo visual y efectos complementarios.

Contexto

Este proyecto se realiza bajo el marco del segundo semestre de la asignatura de Programación perteneciente a la carrera Ciencia de la Computación en La Universidad de La Habana, con el fin de aunar los conocimientos obtenidos durante el presente curso escolar y que conciernen el aprendizaje de C#, lenguaje de programación empleado en la implementación. Ademas de representar una importante parte de la nota final en dicha asignatura.

Descripción

Alcance

Su desarrollo y las funciones que presenta fueron creadas dentro de las exigencias del DSL impuesto y el alcance de su eficacia ostenta de unas bases solidas si bien no completo para su uso hasta el momento.

A continuación se presentan ejemplos de declaraciones de cartas y efectos válidos en cuanto al uso del lenguaje requerido durante la interpretación, si se considera lo suficientemente familiarizado como usuario, siéntase en libertad de omitirla y pasar directamente a la pagina 3.

```
card {
        Type: "Gold",
        Name: "Witch",
        Faction: "Northern Realms",
        Power: 10,
        Range: ["Melee","Ranged"],
        OnActivation: [
```

```
            {
                Effect: {
                    Name: "Damage",
                    Amount: 5
                },
                Selector: {
                    Source: "board",
                    Single: false, //for default is false
                    Predicate: (unit) => unit.Faction == "Northern Realms"
                },
                PostAction: {
                    Type: "ReturnToDeck",
                    Selector: {
                        Source: "parent",
                        Single: false,
                        Predicate: (unit) => unit.Power < 1
                    }
                }
            },
            {
                Effect: "Draw"
            }
        ]
    }
effect {
                Name: "Damage",
                Params: {
                    Amount: Number
                },
                Action: (targets, context) => {
                    for target in targets {
                        i = 0;
```

```
                while (i++ < Amount)
                    target.Power -= 1;
            };
        }
}
```

Análisis léxico

El **análisis léxico** es la primera fase del proceso de interpretación del código, donde el código fuente se convierte en una secuencia de *tokens*. Un *token* es una unidad mínima que tiene un significado en el contexto del lenguaje, como palabras clave, identificadores, operadores, números, etc. El código implementa un **analizador léxico** (también llamado lexer o scanner) en C#, utilizando expresiones regulares para identificar y clasificar diferentes elementos del código fuente.

Cómo funciona

El analizador léxico lee el código fuente línea por línea y lo divide en *tokens* utilizando varias expresiones regulares. Se encarga de reconocer palabras clave, identificadores, operadores, números y cadenas. A continuación, cómo funciona:

1. **Expresiones regulares**: Se utilizan para identificar diferentes tipos de tokens, como espacios en blanco, identificadores (palabras clave, nombres de variables), operadores, y números.
    - **whitespace**: Reconoce espacios en blanco.
    - **phrase**: Identifica los identificadores (que comienzan con una letra o guion bajo y pueden contener números).
    - **symbol**: Detecta operadores como `==`, `!=`, `+`, `-`, `&&`, `||`, entre otros.
    - **number**: Identifica números enteros y decimales.
2. **Procesamiento de líneas y columnas**: El lexer recorre el código línea por línea, procesando cada carácter para ver si coincide con alguna de las expresiones regulares. Si encuentra un símbolo o secuencia válida, lo agrega a una lista de tokens junto con su tipo y ubicación en el código (línea y columna).
3. **Identificación de tokens**: El método `AddMatch` se utiliza para agregar un token a la lista de tokens después de que se detecta una coincidencia con una expresión regular. Si el token es un signo o un operador, se verifica si ya existe en el diccionario de `Token.allTypes`. Si no está, se clasifica como un identificador.
4. **Errores léxicos**: El lexer también es capaz de detectar errores, como la presencia de un carácter inesperado, o cadenas sin comillas de cierre. Estos errores se agregan a una lista que es devuelta al final del análisis.
5. **Control de cadenas**: El lexer maneja de manera especial las cadenas entre comillas dobles (`"`). Si encuentra un carácter de comillas, activa un modo especial de captura para incluir todo lo que esté dentro de las comillas hasta encontrar el cierre.

6. **Comentarios**: Si detecta un comentario de línea (`//`), ignora el resto de la línea, permitiendo comentarios en el código sin interrumpir el análisis.
7. **Resultado final**: Al final del análisis, si no hay errores, se devuelve una lista de tokens. De lo contrario, devuelve la lista vacía y los errores encontrados.

Herramientas usadas

El principal recurso utilizado en el análisis léxico son las **expresiones regulares**, que permiten describir patrones de texto de manera flexible y eficiente. Aquí está cómo están implementadas las principales:

- **whitespace**: `\s+` — busca uno o más caracteres de espacio en blanco (incluyendo espacios, tabulaciones y saltos de línea).
- **phrase**: `[_a-zA-Z]+[_a-zA-Z0-9]*` — reconoce identificadores, que deben comenzar con una letra o guion bajo, seguidos de letras, guiones bajos o números.
- **symbol**: `==|!=|<=|>=|=|--|\+\+|&&|\|\||!|@|@@|=>|<|>|^|\+|\-|\*|\/|\.|\^` — identifica operadores comunes, como `==`, `!=`, `<=`, `>=`, `&&`, `||`, entre otros.
- **number**: `\d+(\.\d+)?` — identifica números enteros y decimales opcionales.

Ejemplo

card {

Name: "Warrior",

Power: 10,

Range: ["Melee", "Ranged"],

OnActivation: [{ Effect: "Damage", Amount: 5 }]

}

El proceso de análisis sería el siguiente:

1. La línea `card {` genera los tokens:
    - `card` → token tipo `Card` (palabra clave).
    - `{` → token tipo `OpenBrace` (llave abierta).
2. La línea `Name: "Warrior",` genera:
    - `Name` → token tipo `Name` (palabra clave).
    - `:` → token tipo `DoubleDot` (dos puntos).
    - `"Warrior"` → token tipo `String` (cadena de texto).
3. La línea `Power: 10,` genera:
    - `Power` → token tipo `Power`.
    - `10` → token tipo `Number`.
4. Se repite el proceso para el resto del código.

Al final, si no hay errores, tendrás una lista de tokens que representan cada componente del código y su tipo. Si hay algún error, como una cadena sin cerrar o un símbolo inesperado, se almacenará en la lista de errores para que el desarrollador pueda corregirlo.

Análisis Sintáctico

La estructura de las expresiones sigue el patrón de diseño basado en un árbol de sintaxis abstracta (AST) y utiliza el principio de doble despacho. Esto significa que el código organiza las expresiones como nodos de un árbol, donde cada nodo es evaluado de manera específica, dependiendo de su tipo y el operador que usa.

Aquí tienes una explicación detallada de cómo funciona esta estructura:

Expressions

1. Clase Abstracta `BinaryExpression<T>`

- Esta es una clase genérica que hereda de `Expression<T>` y representa una expresión binaria, es decir, una expresión que opera sobre dos operandos (izquierdo y derecho) con un operador en el medio.
- **Atributos**:
  - `leftValue`: el valor a la izquierda del operador.
  - `rightValue`: el valor a la derecha del operador.
  - `_operator`: el operador que define la operación entre los dos valores.
- **Métodos**:
  - `CheckError(out string error)`: realiza una verificación semántica de la expresión y devuelve cualquier error que encuentre.
  - `ToString()`: devuelve una representación textual de la expresión en forma de `<leftValue> <operator> <rightValue>`.
  - `CodeLocation`: obtiene la ubicación del código del operador (generalmente útil para depuración y manejo de errores).

1.1. Subclases Específicas para Diferentes Tipos de Expresiones

A partir de `BinaryExpression<T>`, se definen subclases especializadas para diferentes tipos de expresiones: matemáticas, booleanas, literales y de comparación. Cada una de estas subclases sobrescribe ciertos métodos de la clase base para adaptarse a su tipo de datos y operaciones específicas.

MathExpression

- Especializa `BinaryExpression<Number>`, donde el tipo de la expresión es numérico.
- **CheckError()**: verifica que el operador sea uno válido entre `+`, `-`, `*`, `/`, y `^` y que ambos operandos sean de tipo `Number`. Si no lo son, devuelve una lista de errores.

- **Interpret()**: evalúa la expresión matemática interpretando los operandos y aplicando la operación correspondiente. Si el operador es incorrecto o los valores no son números, lanza un error de ejecución.

BooleanExpression

- Especializa `BinaryExpression<bool>`, donde el tipo de la expresión es booleana.
- **CheckError()**: verifica que el operador sea uno válido entre && y `||` y que ambos operandos sean de tipo booleano. Si no lo son, agrega errores a la lista.
- **Interpret()**: evalúa la expresión booleana aplicando la operación lógica correspondiente entre los operandos. Si hay un error en el tipo, lanza una excepción.

LiteralExpression

- Especializa `BinaryExpression<string>`, donde el tipo de la expresión es una cadena de texto.
- **CheckError()**: verifica que el operador sea válido para operar con cadenas (@ o @@) y que ambos operandos sean de tipo `string`. Si no lo son, devuelve una lista de errores.
- **Interpret()**: combina las cadenas dependiendo del operador. El operador @ simplemente concatena las dos cadenas, mientras que @@ las concatena con un espacio intermedio.

ComparisonExpression

- Especializa `BinaryExpression<bool>`, donde el tipo de la expresión es una comparación que devuelve un valor booleano.
- **CheckError()**: verifica que el operador de comparación sea válido (por ejemplo, <, >, <=, >=, ==, `!=`) y que ambos operandos sean números.
- **Interpret()**: evalúa la expresión de comparación, devolviendo `true` o `false` dependiendo de la relación entre los operandos.

1.2. Método `CheckError`

El método `CheckError()` en cada subclase asegura que las operaciones sean semánticamente correctas antes de intentar interpretarlas. Esto es importante porque evita errores en tiempo de ejecución al validar que los operandos sean del tipo adecuado para la operación que se intenta realizar.

- Para **expresiones matemáticas**, asegura que los operandos sean números.
- Para **expresiones booleanas**, se asegura de que ambos operandos sean booleanos.
- Para **expresiones literales**, valida que ambos operandos sean cadenas.
- Para **expresiones de comparación**, se asegura de que ambos operandos sean números.

Si las verificaciones fallan, se genera un error con la ubicación precisa del problema, permitiendo un manejo claro y efectivo de los errores.

1.3. Método `Interpret`

Este método es el que realmente ejecuta la expresión. En cada subclase, sobrescribes este método para ejecutar la operación correspondiente, como sumar dos números o comparar dos valores. El

uso de un `switch` en base al valor del operador hace que el código sea fácil de seguir y de extender si decides agregar más operadores.

En caso de un error durante la interpretación (por ejemplo, si un valor no es del tipo esperado), se lanza una excepción, lo que permite manejar adecuadamente los errores en tiempo de ejecución.

1.4. Uso del Patrón AST

El código sigue el patrón de construcción de un árbol de sintaxis abstracta (AST), donde cada expresión es un nodo del árbol. Los nodos de las expresiones binarias tienen dos hijos: `leftValue` y `rightValue`, y un operador que los conecta. Este árbol puede ser evaluado recursivamente durante la interpretación.

Esta estructura implementa un sistema de expresiones que sigue el patrón de diseño **Visitor** y utiliza principios de **programación orientada a objetos** para representar y evaluar expresiones binarias en un lenguaje interpretado. A continuación, una explicación exhaustiva de cada componente:

1.5. Interfaces clave

1.5.1 `IExpression`

La interfaz `IExpression` representa una expresión que puede ser evaluada y verificada semánticamente. Los métodos importantes son:

- **`ExpressionType Type { get; }`**: Devuelve el tipo de la expresión (por ejemplo, número, booleano, cadena, etc.).
- **`(int, int) CodeLocation { get; }`**: Regresa la posición en el código donde la expresión fue definida, lo que es útil para la gestión de errores.
- **`object Interpret()`**: Evalúa la expresión y devuelve el resultado.
- **`bool CheckError(out List<string> errorsList)`**: Verifica los posibles errores semánticos en la expresión.
- **`bool CheckError(out string error)`**: Similar al anterior, pero devuelve una cadena con errores.

1.5.2 `IVisitable<T>` y `IVisitor<T>`

El patrón **Visitor** permite separar la lógica de procesamiento (evaluación de expresiones) de la estructura de datos (las propias expresiones).

- **`IVisitable<T>`**: Cualquier clase que implemente esta interfaz puede ser visitada por un visitante que ejecutará la lógica de procesamiento.
- **`IVisitor<T>`**: Representa al visitante, que aplica una operación sobre un elemento de tipo `IVisitable<T>`.

Este patrón permite que el procesamiento de las expresiones sea extensible sin modificar las clases de expresiones directamente.

1.6 Clase Abstracta `Expression<T>`

Esta es la clase base de la cual todas las expresiones derivan. Define el comportamiento genérico de las expresiones y permite la implementación del patrón Visitor. Aquí se declaran los métodos abstractos que las clases derivadas deben implementar:

- **`Accept(IVisitor<T> visitor)`**: Permite que un visitante procese la expresión.
- **`CheckError(out List<string> errorsList)`** y **`CheckError(out string error)`**: Métodos para verificar semánticamente la expresión.
- **`ExpressionType Type { get; }`**: Define el tipo de la expresión, implementado en clases derivadas.
- **`(int, int) CodeLocation { get; protected set; }`**: Posición de la expresión en el código.
- **`object Interpret()`**: Implementación de la evaluación de la expresión.

---

2. Clase `Callable`

Esta es una **clase abstracta** que hereda de la clase `Expression<object>` y representa la base para cualquier operación invocable, ya sea un **método** o una **propiedad** en el contexto del lenguaje interpretado.

Atributos:

- **`protected Token? caller`**: Este token representa la invocación de un método o propiedad. Contiene información sobre la ubicación del código y el nombre del método o propiedad que se está invocando.
- **`protected IExpression? callee`**: Esta es la expresión que se va a interpretar para obtener el objeto al que se le está llamando un método o propiedad.

Métodos:

- **`CheckError(out string error)`**: Este método revisa semánticamente la invocación, verificando si hay errores. Si se encuentran errores, los agrupa en una cadena.
- **`Type`**: Devuelve siempre `ExpressionType.Object`, ya que tanto los métodos como las propiedades devuelven objetos genéricos.

Este es el **esqueleto** de una invocación. Proporciona la estructura básica para manejar tanto propiedades como métodos, pero sin implementar el comportamiento específico, que será hecho por las subclases `Methods` y `Property`.

2.1 Clase `Methods`

Esta clase concreta extiende `Callable` y representa la invocación de un **método** en un objeto. Permite obtener un método por su nombre (almacenado en el `Token`) y ejecutarlo con los argumentos provistos.

Atributos:

- **`IExpression[] arguments`**: Este arreglo almacena los argumentos que serán pasados al método invocado.

Constructor:

- **`Methods(Token caller, IExpression callee, IExpression[] arguments = null)`**: Inicializa los atributos `caller`, `callee`, y `arguments`. `callee` es el objeto sobre el cual se invocará el método y `arguments` son los parámetros que se le pasarán al método.

Método Principal: `Interpret()`

Este es el corazón de la clase y se encarga de la **invocación dinámica de un método** en el objeto resultante de la evaluación de `callee`.

1. **Interpretar el callee**: El objeto sobre el cual se llamará el método se obtiene interpretando `this.callee`.
2. **Identificar el tipo de `callee`**:
    - Dependiendo de si el objeto es una instancia de `GameList`, `Number`, `Card`, o `string`, se determina su tipo en tiempo de ejecución usando `typeof`.
3. **Obtener el método mediante reflexión**:
    - **`method = type.GetMethod(caller.Value)`**: Se busca el método usando el nombre almacenado en el token `caller`. Si el método tiene sobrecargas, se maneja la excepción `AmbiguousMatchException`.
4. **Invocar el método**:
    - Si el método existe, se invoca usando `method.Invoke(callee, this.arguments)` pasando los argumentos correspondientes.
    - Si el método no existe, se lanza un error de ejecución con un mensaje detallado.
5. **Manejo de errores**:
    - Si el método no existe o los argumentos son incorrectos, se lanza un `RunningError` con la ubicación en el código (`caller.CodeLocation`).

Semántica y Errores:

El método `CheckError(out List<string> errorsList)` no está completamente implementado, pero sugiere que se debe verificar si el método está correctamente definido antes de la invocación.

2.2 Clase `Property`

Esta clase también extiende `Callable` y está diseñada para manejar la **accesibilidad a propiedades** de un objeto en lugar de métodos.

Constructor:

- **`Property(Token caller, IExpression callee)`**: Similar a `Methods`, inicializa los atributos `caller` y `callee`, pero no necesita argumentos adicionales.

Método Principal: `Interpret()`

Aquí se maneja el acceso a una propiedad de un objeto:

1. **Interpretar el callee**: Primero se evalúa el objeto `callee` sobre el que se accederá la propiedad.
2. **Determinar el tipo**: Se identifica el tipo de objeto (`GameList`, `Number`, `Card`, `string`, etc.) usando reflexión para obtener su tipo en tiempo de ejecución.
3. **Acceder a la propiedad**:
    - **`type.GetProperty(caller.Value)`**: Se busca la propiedad en el tipo de objeto usando el nombre almacenado en `caller`.
    - Si la propiedad existe, se obtiene su valor usando `GetValue(callee)`.
    - Si la propiedad no existe, se lanza un `RunningError`.

Semántica y Errores:

Similar a `Methods`, el método `CheckError(out List<string> errorsList)` lanza una excepción para indicar que se debe verificar que la propiedad esté correctamente definida antes de intentar acceder a ella.

2.3 Propósito del Diseño

El propósito de este código es permitir la **invocación dinámica de métodos y acceso a propiedades** en objetos dentro de un entorno interpretado. Esto se logra utilizando **reflexión**, que permite inspeccionar los tipos y miembros de un objeto en tiempo de ejecución sin necesidad de conocerlos en tiempo de compilación.

Reflexión

La **reflexión** es un mecanismo potente en C# que permite examinar y manipular el comportamiento de los objetos en tiempo de ejecución. En este caso, se usa para buscar métodos y propiedades por su nombre (`caller.Value`), lo cual permite que el intérprete ejecute métodos definidos dinámicamente en los objetos.

Patrón de Diseño Usado: Command

El uso de la clase `Callable` como clase base abstracta sugiere un **patrón de diseño Comando** (Command Pattern). Cada subclase (`Methods` y `Property`) encapsula una acción (invocar un método o acceder a una propiedad) que puede ejecutarse en un contexto específico.

3. Estructura `Number`

La estructura `Number` encapsula un número de tipo `double` y proporciona métodos para realizar operaciones aritméticas y de comparación.

Atributos:

- **`public double Value`**: Este es el único atributo de la estructura. Representa el valor numérico almacenado por la instancia de `Number`.

Constructor:

- **`public Number(double value)`**: Es un constructor que inicializa la estructura con un valor `double`.

Propiedades:

- **`public Number Opposite => new Number(-Value);`**: Esta propiedad de solo lectura devuelve una nueva instancia de `Number` con el valor opuesto del número almacenado en `Value`. Por ejemplo, si `Value = 5`, `Opposite` devolverá un `Number` con `Value = -5`.

Métodos Sobrescritos:

- **`public override string ToString()`**: Devuelve una representación en cadena del número almacenado en `Value`, es decir, su conversión a `string`.
- **`public override bool Equals(object? obj)`**: Este método sobrescribe `Equals` para comparar dos objetos de tipo `Number` en términos de igualdad. Devuelve `true` si ambos objetos tienen el mismo valor numérico.
- **`public override int GetHashCode()`**: Sobrescribe `GetHashCode` para devolver un código hash basado en el valor de `Value`, utilizando `HashCode.Combine`.

3.1 Comparación de Números

El código incluye varios métodos para realizar comparaciones entre instancias de `Number`.

- **`public bool Greater(object ob)`**: Devuelve `true` si el número actual es mayor que el número representado por `ob`.
- **`public bool Less(object ob)`**: Devuelve `true` si el número actual es menor que el número representado por `ob`.
- **`public bool GreaterEqual(object ob)`**: Devuelve `true` si el número actual es mayor o igual que el número representado por `ob`.
- **`public bool LessEqual(object ob)`**: Devuelve `true` si el número actual es menor o igual que el número representado por `ob`.

Estos métodos realizan la comparación entre dos instancias de `Number`. El objeto `ob` se convierte a `Number` usando la sentencia `ob is Number value` antes de realizar la comparación. Si `ob` no es un `Number`, la comparación no se realiza.

3.2 Operaciones Aritméticas

El código proporciona varios métodos para realizar operaciones aritméticas entre dos números:

- **`public Number Plus(Number value)`**: Suma el número actual con otro `Number` y devuelve un nuevo `Number` que contiene el resultado.
- **`public Number Minus(Number value)`**: Resta el `Number` proporcionado del número actual y devuelve un nuevo `Number` con el resultado.
- **`public Number Multiply(Number value)`**: Multiplica el número actual por otro `Number` y devuelve un nuevo `Number` con el resultado.
- **`public Number Divide(Number value)`**: Divide el número actual por otro `Number` y devuelve un nuevo `Number` con el resultado. (Se debería manejar el caso de división entre cero en una implementación completa.)
- **`public Number PowerTo(Number value)`**: Eleva el número actual a la potencia del número proporcionado (usa `Math.Pow` para realizar la operación) y devuelve un nuevo `Number` con el resultado.

3.3 Propósito General

La estructura es muy útil en un intérprete porque permite definir claramente las operaciones que se pueden realizar sobre números, como comparación, aritmética, y manipulación de su representación interna. Además, al ser una estructura (`struct`), tiene beneficios de rendimiento porque se almacena en la **pila** en lugar del **montón**, lo que puede ser importante en ciertas aplicaciones.

3.4 Integración en un Intérprete

Por ejemplo, al interpretar una operación aritmética como `5 + 3`, cada uno de esos números sería una instancia de `Number`, y el intérprete llamaría al método `Plus` para sumar ambos valores.

---

4. Clase `UnaryOperation`

Esta clase hereda de `Expression<object>` y representa una operación unaria que involucra un operador y un valor (el operando). Está diseñada para manejar operadores unarios, como la negación de un número o la negación lógica.

Atributos:

- **`protected Token _operator`**: Representa el operador unario (por ejemplo, `-` o `!`). Los tokens suelen contener información sobre el tipo de símbolo que se está procesando y su ubicación en el código.
- **`protected IExpression value`**: Es la expresión a la que se aplica la operación unaria. Puede ser un número o una expresión booleana.

Constructor:

- **`public UnaryOperation(Token _operator, IExpression value)`**: Inicializa la operación unaria con un operador y una expresión.

Métodos principales:

- **`CheckError(out List<string> errorsList)`**: Realiza una verificación semántica sobre la operación unaria. Verifica si el tipo de `value` es un número o un valor booleano. Si no lo es, agrega un mensaje de error y retorna `false`.
    - Si el tipo es `Object`, lanza una excepción con un mensaje de error.
    - Si el tipo es `Number` o `Boolean`, la operación es válida y retorna `true`.
- **`Interpret()`**: Interpreta la operación unaria en tiempo de ejecución. Realiza una de dos operaciones:
    - Si el operador es `-`, devuelve el valor opuesto de un número (negación).
    - Si el operador es `!`, devuelve el valor lógico inverso de un booleano.

  Si el operador no es válido, lanza una excepción con la ubicación del error.
- **`Type`**: Devuelve el tipo de la expresión basada en el tipo del operando `value`.

---

5. Clase `UnaryExpression<T>`

Esta es una clase abstracta que hereda de `Expression<T>`. Define el comportamiento general de una **expresión unaria**. Esta clase es la base para otras clases específicas que representan expresiones unarias de diferentes tipos (por ejemplo, `UnaryCallable`, `UnaryDeclaration`, etc.).

Atributos:

- **`protected T? value`**: Es el valor (u objeto) sobre el que se aplicará la operación unaria.

Métodos principales:

- **`CheckError(out string error)`** y **`CheckError(out List<string> errorsList)`**: Estos métodos son abstractos en la clase `Expression<T>` y aquí se implementan como siempre válidos, devolviendo `true`.
- **`ToString()`**: Devuelve una representación en cadena del valor de la expresión.

Propósito:

Esta clase proporciona una base para definir expresiones unarias genéricas, permitiendo reutilizar código para expresiones más específicas que hereden de ella.

---

6. Clase `UnaryCallable`

Esta clase hereda de `UnaryExpression<Callable>` y representa una **expresión unaria que opera sobre un `Callable`** (una función o método).

Métodos principales:

- **`Interpret()`**: Devuelve la interpretación del valor (es decir, invoca el método o función representado por el `Callable`).
- **`Type`**: Devuelve el tipo de la expresión, basado en el tipo del `Callable`.

Propósito:

Permite que las operaciones unarias trabajen con funciones o métodos que son invocables.

---

7. Clase `UnaryDeclaration`

Hereda de `UnaryExpression<Declaration>` y representa una **expresión unaria que opera sobre una declaración** (una variable o valor asignado).

Métodos principales:

- **`Interpret()`**: Devuelve la interpretación de una declaración. En este caso, llama al método `ValueGiver()` de la declaración para obtener el valor de la variable o constante.
- **`Type`**: Devuelve el tipo de la declaración.

Propósito:

Permite realizar operaciones unarias sobre declaraciones de variables o constantes.

---

8. Clase `UnaryObject`

Hereda de `UnaryExpression<object>` y representa una **expresión unaria que opera sobre un objeto genérico**.

Métodos principales:

- **`Type`**: Determina el tipo de la expresión en función del valor del objeto. Utiliza un patrón de coincidencia (`switch`) para verificar si el objeto es un número, cadena, lista, carta, o simplemente un objeto genérico.
- **`Interpret()`**: Devuelve el objeto tal cual sin realizar ninguna operación sobre él.

Propósito:

Permite trabajar con operaciones unarias sobre cualquier tipo de objeto, interpretando correctamente su tipo.

---

9. Clase `UnaryValue`

Hereda de `UnaryExpression<Token>` y representa una **expresión unaria que opera sobre un token**. Un token puede ser un valor booleano, numérico o una cadena de texto.

Métodos principales:

- **`Interpret()`**: Interpreta el token dependiendo de su tipo. Convierte el token a su valor correspondiente:
    - Si el token es `True`, devuelve `true`.
    - Si el token es `False`, devuelve `false`.
    - Si es un número, convierte el valor a `Number`.
    - Si es una cadena, elimina las comillas y devuelve el valor de la cadena.
- **`Type`**: Devuelve el tipo de la expresión en función del tipo del token (booleano, número, o cadena).

Propósito:

Esta clase es útil para interpretar valores literales (como números o booleanos) en el contexto de una operación unaria dentro de un intérprete.

Propósito de las clases unarias

El propósito general de estas clases es proporcionar una estructura que permita manejar **operaciones unarias** dentro de un **intérprete**. Cada clase específica se encarga de un tipo particular de expresión (objetos, funciones, declaraciones, etc.), lo que permite a un intérprete realizar operaciones como la negación de números o valores booleanos, y aplicar estas operaciones en diferentes contextos.

Declarations

Se definió una clase `Declaration` que implementa la interfaz `IStatement`, la cual representa una **declaración** o **asignación** en un lenguaje interpretado. Esta declaración permite asociar variables con valores en un entorno (por ejemplo, un ámbito o un contexto). Cómo funciona:

1. Interfaz `IStatement`

Esta interfaz declara las siguientes propiedades y métodos que deben implementar las clases que la hereden:

- **`CheckError(out List<string> errorsList)`**: Método que verifica si la declaración o el statement tiene sentido semántico en el contexto en que se usa. Devuelve una lista de errores si los hay.
- **`(int, int) CodeLocation`**: Una propiedad que indica la ubicación del código (línea y columna) en el que ocurre el statement.
- **`RunIt()`**: Método que ejecuta la declaración, es decir, interpreta la asignación o modificación de una variable.

---

2. Clase `Declaration`

La clase `Declaration` representa una declaración o asignación de una variable. Puede trabajar con diferentes operadores (como asignación, incremento, decremento, etc.).

Atributos:

- **`Environment environment`**: El entorno en el que se declara o se asigna la variable. Un entorno es donde se almacenan los valores de las variables. Si no se proporciona, por defecto usa el entorno global.
- **`IExpression? value`**: El valor que se va a asignar a la variable. Puede ser `null` si la declaración es solo de una variable sin asignar un valor.
- **`Token token`**: El token que representa la variable que se está declarando o asignando.
- **`Token operation`**: El token que representa el operador que se está utilizando en la declaración. Por ejemplo, puede ser `=`, `++`, `--`, `+=`, o `-=`.

Constructor:

- **`Declaration(Token token, Environment environment = null, Token operation = null, IExpression value = null)`**: Este constructor inicializa la declaración. Si no se proporciona un entorno, usa el global. También ejecuta inmediatamente la declaración mediante el método `RunIt()`.

Propiedades:

- **`CodeLocation`**: Devuelve la ubicación del código donde ocurrió la operación. Si no hay una operación, se utiliza la ubicación del token.
- **`Type`**: Devuelve el tipo de la expresión (por ejemplo, número, booleano, cadena, etc.) que está asociada con la variable. Si el valor no está definido, lo obtiene del entorno.

Métodos principales:

- **`CheckSemantic(out List<string> errorsList)`**: Verifica la validez semántica de la declaración. Verifica si el valor asignado a la variable es semánticamente correcto:
    - Si el valor no es `null`, se verifica su validez.
    - Si el valor ya existe en el entorno y no es válido, agrega un error a la lista.
    - Retorna `true` si todo es correcto, y `false` si hay errores.
- **`ValueGiver()`**: Devuelve el valor de la variable desde el entorno, ejecutando la declaración si es necesario.
- **`RunIt()`**: Este es el método clave, que **ejecuta** la declaración y realiza la operación correspondiente:
    - Si la operación es `IncreaseOne` (`++`), incrementa el valor de la variable en 1.
    - Si la operación es `DecreaseOne` (`--`), decrementa el valor de la variable en 1.
    - Si la operación es `Assign` (`=`), asigna el valor proporcionado a la variable.
    - Si la operación es `Increase` (`+=`), incrementa el valor de la variable por la cantidad especificada en `value`.
    - Si la operación es `Decrease` (`-=`), decrementa el valor de la variable por la cantidad especificada en `value`.

  Si la operación no es válida, lanza un error de análisis sintáctico.

Excepciones manejadas:

- **`ParsingError`**: Si la operación no es válida o el tipo de operación no es reconocido, lanza esta excepción con un mensaje de error, indicando la ubicación de la declaración problemática.
- **`NullReferenceException`**: Se maneja para evitar errores cuando alguna referencia es `null`.

Resumen

La clase `Declaration` se utiliza para declarar variables y asignarles valores, gestionando operaciones comunes como `=`, `++`, `--`, `+=`, y `-=`. La clase trabaja en conjunto con un entorno para almacenar y recuperar los valores de las variables. Además, incluye validaciones semánticas para asegurarse de que las operaciones sean correctas y lanza excepciones si se intenta realizar una operación inválida.

---

3. Clase `Block`

La clase `Block` representa un bloque de instrucciones en el contexto de un lenguaje interpretado. Un bloque de instrucciones es una colección de declaraciones o sentencias que se ejecutan secuencialmente. Esta clase implementa la interfaz `IStatement`, lo que le permite ser parte del sistema de ejecución de instrucciones del intérprete.

1. Atributos

- **`IEnumerable<IStatement> statements`**: Una colección de objetos que implementan la interfaz `IStatement`. Representa el conjunto de sentencias que forman el bloque. Estas sentencias se ejecutarán en el orden en que aparecen en la colección.

2. Constructor

- **`Block(IEnumerable<IStatement> statements)`**: Constructor que inicializa el bloque con una colección de sentencias. El bloque ejecutará estas sentencias en el orden dado.

3. Métodos

- **`public void RunIt()`**: Ejecuta todas las sentencias en el bloque secuencialmente. Itera sobre cada sentencia en la colección `statements` y llama a su método `RunIt()`. Este es el método principal para ejecutar el bloque de instrucciones.
- **`public bool CheckSemantic(out List<string> errorsList)`**: Verifica la validez semántica de todas las sentencias en el bloque.
    - **`string attention = "";`**: Variable para acumular mensajes de atención que se lanzarán como excepciones más tarde.
    - **`errorsList = new List<string>();`**: Lista para acumular errores semánticos detectados durante la verificación.

- **`foreach (var Statement in statements)`**: Itera sobre cada sentencia en el bloque.
    - **`if (!Statement.CheckSemantic(out List<string> temperrorsList))`**: Llama al método `CheckSemantic` de cada sentencia. Si hay errores, los agrega a `errorsList`.
    - **`catch (Attention a)`**: Captura excepciones de tipo `Attention` lanzadas por las sentencias. Los mensajes de estas excepciones se acumulan en `attention`.
- **`if (attention != "") throw new Attention(attention);`**: Si se acumuló algún mensaje de atención, lanza una excepción `Attention` con todos los mensajes concatenados.
- **`return errorsList.Count == 0;`**: Devuelve `true` si no se encontraron errores, de lo contrario, `false`.

Resumen

La clase `Block` actúa como un contenedor para una secuencia de sentencias. Permite ejecutar todas las sentencias en un bloque de manera secuencial y verificar su validez semántica. Si alguna sentencia genera un error o una advertencia, el bloque puede manejarlo adecuadamente, acumulando errores y lanzando excepciones si es necesario.

4.1 Clase `If`

Propósito: Ejecuta un bloque de código si una condición se evalúa como `true`. Si la condición es `false`, puede ejecutar un bloque `else` opcional.

Atributos:

- **`(int, int) codeLocation`**: Guarda la ubicación en el código donde se encuentra la instrucción `if`, para manejar posibles errores o advertencias.
- **`IExpression conditional`**: La expresión condicional que se evaluará. Debe ser de tipo booleano.
- **`IStatement content`**: El bloque de código que se ejecuta si la condición es `true`.
- **`IStatement? elseContent`**: Un bloque opcional que se ejecuta si la condición es `false`.

Métodos:

- **`CheckSemantic`**:
    - Verifica que la condición sea de tipo booleano.
    - Comprueba semánticamente el bloque principal (`content`) y el bloque `else` (si existe).
    - Si la condición no es booleana, devuelve un error.
- **`RunIt`**:

- Interpreta la condición.
- Si es `true`, ejecuta el bloque `content`; si es `false`, ejecuta el bloque `elseContent` (si existe).

4.2 Clase `For`

Propósito: Itera sobre una colección (como una lista) y ejecuta un bloque de código para cada elemento.

Atributos:

- **`Environment environment`**: El entorno en el que se ejecutan las variables del bucle.
- **`Token token`**: El identificador de la variable de iteración.
- **`IExpression collection`**: La colección sobre la que se iterará. Debe ser de tipo `List`.
- **`IStatement content`**: El bloque de código que se ejecuta en cada iteración.

Métodos:

- **`CheckSemantic`**:
  - Verifica que el `content` sea semánticamente válido.
  - Verifica que la colección sea de tipo `List`. Si no lo es, lanza una excepción de tipo `Attention`.
- **`RunIt`**:
  - Itera sobre cada elemento de la colección. Si no es una colección válida, lanza un error de ejecución.
  - Para cada elemento, asigna el valor a la variable identificada por el `token` y ejecuta el bloque `content`.

4.3 Clase `While`

Propósito: Ejecuta un bloque de código mientras una condición sea `true`.

Atributos:

- **`IExpression conditional`**: La expresión condicional que se evaluará en cada iteración. Debe ser de tipo booleano.
- **`IStatement content`**: El bloque de código que se ejecuta mientras la condición sea `true`.
- **`(int, int) codeLocation`**: La ubicación de la instrucción en el código, útil para depuración y manejo de errores.

Métodos:

- **`CheckSemantic`**:

    - Verifica que el `content` sea semánticamente válido.
    - Verifica que la condición sea de tipo `Boolean`. Si no lo es, devuelve un error.
- **`RunIt`**:
    - Mientras la condición sea `true`, ejecuta el bloque de código. Si la condición no es booleana, lanza un error de ejecución.

4.4 Clase `Log`

Propósito: Muestra el valor de una expresión en la consola.

Atributos:

- **`IExpression Value`**: La expresión cuyo valor se mostrará.

Métodos:

- **`CheckSemantic`**:
    - Verifica que la expresión `Value` sea semánticamente válida. Si no lo es, devuelve un error.
- **`RunIt`**:
    - Interpreta la expresión y muestra su valor en la consola mediante `Console.WriteLine`.

Resumen

Estas clases implementan instrucciones fundamentales en lenguajes de programación:

- **Condicionales (`If`)**: Ejecutan bloques de código dependiendo del resultado de una condición booleana.
- **Ciclos (`For`, `While`)**: Ejecutan bloques de código varias veces, iterando sobre una colección o mientras una condición sea verdadera.
- **Salida (`Log`)**: Permite mostrar información en la consola para depuración o interacción con el usuario.

Cada una de estas clases utiliza un enfoque semántico para verificar que las operaciones sean válidas antes de ejecutarse, asegurando que no haya errores en tiempo de ejecución.

---

5. Clase `Environment`

Este código define una clase llamada `Environment`, que tiene un papel crucial en la gestión de las variables y su contexto en el entorno de un lenguaje interpretado. La clase se usa para almacenar y acceder a expresiones (`IExpression`) en un entorno jerárquico, lo que permite tener múltiples "niveles" de variables y acceder a ellas de manera apropiada según su alcance. A cada entorno se le puede asignar un entorno "padre" (`parent`), creando una jerarquía. Esto es importante para permitir la búsqueda de variables en entornos anidados. El concepto es similar a los entornos de ejecución o espacios de nombres que ves en lenguajes como Python o JavaScript.

Atributos

- **`static Environment? global`**: Representa el entorno global. Solo existe una instancia global a lo largo de la ejecución del programa, y todas las demás instancias pueden referenciarlo si es necesario.
- **`Environment? parent`**: Un entorno puede tener un entorno padre. Si el entorno actual no contiene una variable, buscará en su entorno padre (de manera recursiva si es necesario).
- **`Dictionary<string, IExpression> expr`**: Almacena las variables y expresiones asociadas dentro del entorno actual. Las claves son los nombres de las variables (de tipo `string`), y los valores son las expresiones que representan las variables (`IExpression`).

Métodos

1. **Constructor `Environment(Environment? parent = null)`**:
    - Se puede crear un entorno con o sin un entorno padre.
    - Si no se pasa ningún padre, significa que este entorno es el nivel más alto (a menos que sea el entorno global).
2. **Propiedad `static Environment Global`**:
    - Devuelve la instancia del entorno global. Si aún no se ha creado, se crea la primera vez que se solicita.
    - El entorno global tiene una variable especial `context`, que es una referencia a `GameContext.Context`, lo que parece estar relacionado con el contexto general del juego.
3. **Método `IExpression Get(string key)`**:
    - Busca una variable por su nombre (`key`) en el entorno actual o en sus padres.
    - Si la variable no se encuentra en el entorno actual, se busca recursivamente en los entornos padres hasta encontrarla.
4. **Método `void Set(IExpression value, Token token)`**:
    - Asigna un valor (`IExpression`) a una variable (identificada por `token`).
    - Si la variable ya existe en el entorno actual, la sobrescribe.
    - Si la variable existe en algún entorno padre, la cambia usando el método `Change`.
    - Si no existe, la agrega al entorno actual.
    - Hay una restricción especial: no se permite redefinir la palabra `context`, ya que está reservada para un uso específico dentro del entorno global.
5. **Método `void Change(string key, IExpression value)`**:
    - Cambia el valor de una variable existente en el entorno padre.
    - Si la variable no está en el entorno padre inmediato, se busca en otros padres más arriba en la jerarquía.
6. **Método `bool Contains(string key)`**:
    - Verifica si una variable existe en el entorno actual o en cualquiera de sus padres.
7. **Método `bool Search(string key)`**:
    - Busca una variable solo en los entornos padres, no en el entorno actual.

8. **Método `void Reset()`**:
    - Reinicia el entorno global, creando una nueva instancia.
9. **Propiedad indexadora `IExpression this[string token]`**:
    - Es una indexación que permite obtener y establecer valores en el entorno mediante un nombre de variable (`token`).
    - En el caso de obtener (`get`):
        - Si la variable existe en el entorno actual, se devuelve su valor.
        - Si no existe en el entorno actual, se busca en los padres.
        - Si no se encuentra, lanza un error de análisis (`ParsingError`), indicando que la variable no ha sido declarada.
    - En el caso de establecer (`set`):
        - Si la variable existe en el entorno actual, se cambia su valor.
        - Si existe en un entorno padre, se cambia usando el método `Change`.
        - Si no se encuentra en ningún entorno, lanza un error de análisis.

Ejemplo de Uso

Tomando por hecho que existe una estructura de código donde declaras una variable `x` en un entorno padre, y luego en un entorno hijo intentas acceder a `x`. Si `x` no está en el entorno hijo, el programa buscará en el padre usando la lógica de herencia de entornos.

---

6. Clase `EffectState`

La clase `EffectState` define el estado de un efecto en el sistema de interpretación del juego. Esta clase maneja la definición de los efectos, su verificación semántica y su ejecución dentro de un entorno que incluye parámetros, el contexto de juego, y los objetivos de los efectos.

Atributos

1. **`IExpression name`**:
    - El nombre del efecto, almacenado como una expresión, que debe ser evaluado como una cadena para identificar el efecto en el sistema.
2. **`IStatement action`**:
    - La acción que se ejecutará cuando el efecto sea activado. Se espera que sea otro bloque de código o una serie de instrucciones encapsuladas en una declaración.
3. **`static Dictionary<string, EffectState> effects`**:
    - Un diccionario estático que almacena todos los efectos definidos en el sistema, donde la clave es el nombre del efecto y el valor es el objeto `EffectState` que lo representa.
4. **`static Dictionary<string, string> declaredEffects`**:

    - Un diccionario estático que almacena los efectos declarados, pero con un enfoque en el código para realizar un seguimiento de los efectos escritos en el código fuente.
5. **`Dictionary<string, ExpressionType> _parameters`**:
    - Un diccionario que almacena los parámetros que debe tomar el efecto. La clave es el nombre del parámetro y el valor es el tipo de la expresión esperada.
6. **`(int, int) codeLocation`**:
    - Al igual que en la clase anterior, este atributo almacena la ubicación del efecto en el código para facilitar la depuración y los mensajes de error.
7. **`Token context`** y **`Token targets`**:
    - Estos tokens representan el contexto en el que se ejecutará el efecto (por ejemplo, una carta específica, una situación de juego) y los objetivos a los que se aplicará el efecto.
8. **`Environment environment`**:
    - Representa el entorno donde se ejecutan las expresiones y declaraciones. Este objeto gestiona las variables y su estado durante la ejecución del efecto.
9. **`bool taken`**:
    - Esta bandera indica si los parámetros del efecto fueron correctamente validados antes de ejecutar la acción.

Constructor

El constructor inicializa los atributos del estado del efecto, estableciendo el nombre, la acción a ejecutar, los parámetros, y el entorno donde este efecto será ejecutado. También asigna el contexto y los objetivos del efecto, permitiendo que los efectos sean declarados dinámicamente.

1. Se recorren los parámetros que se le pasan al constructor, y se verifican y asignan sus tipos en el diccionario `_parameters`.
2. Se añade cada parámetro al entorno del juego con `environment.Set()`, preparando el entorno para la ejecución del efecto.

Métodos

`StartOver()`

- **Propósito**: Reinicia los diccionarios estáticos de efectos y efectos declarados. Esto es útil cuando se quiere "reiniciar" el sistema de efectos, como en el inicio de una nueva partida.

`CheckSemantic(out List<string> errors)`

- **Propósito**: Verifica si el efecto está bien definido y es semánticamente válido.
- **Funcionamiento**:
    - Primero, verifica si el nombre del efecto es una cadena (`ExpressionType.String`), y si ya ha sido declarado previamente.
    - Luego, realiza una verificación semántica de la acción (`action`) del efecto. Si hay errores, estos se agregan a la lista `errors`.

- Si no hay errores, el efecto se añade al diccionario estático `effects` y `declaredEffects`.
- Si hay advertencias (atenciones), se lanza una excepción de tipo `Attention`.

`RunIt()`

- **Propósito**: Ejecuta la acción asociada al efecto si los parámetros fueron correctos. Si los parámetros no han sido tomados (validados), lanza un error.

`Take(List<(Token, IExpression)> _parameters, IExpression targets)`

- **Propósito**: Valida los parámetros que se le pasan al efecto y los asigna al entorno.
- **Funcionamiento**:
  - Primero, se asegura de que el contexto y los objetivos están correctamente establecidos en el entorno.
  - Luego, recorre los parámetros recibidos y verifica si coinciden con los tipos esperados que fueron declarados cuando se creó el efecto.
  - Si un parámetro no es necesario, lanza una advertencia. Si el tipo del parámetro es incorrecto o no ha sido declarado, lanza un error.
  - Si todo es correcto, marca la bandera `taken` como `true`, indicando que los parámetros han sido validados correctamente.

Flujo General

1. **Declaración del efecto**:
   - Se crea una instancia de `EffectState` con su nombre, acción, y parámetros. Se realiza una verificación semántica con el método `CheckSemantic()` para asegurarse de que todo es válido.
2. **Validación de los parámetros**:
   - Antes de ejecutar el efecto, se utiliza el método `Take()` para validar que los parámetros pasados coinciden con los tipos esperados.
3. **Ejecución del efecto**:
   - Si todo está correcto (los parámetros han sido validados), el método `RunIt()` ejecuta la acción del efecto.

---

7. Clase `Activation`

La clase `Activation` es una parte del sistema de interpretación para la activación de efectos en un juego. Está diseñada para manejar la verificación semántica y la ejecución de efectos basados en ciertas condiciones, como expresiones, selectores y parámetros que deben cumplir con una semántica particular antes de ser ejecutados.

Atributos

1. **`IExpression effect`**:

- Representa el efecto que se va a activar. Es una expresión que debe ser evaluada para identificar qué efecto específico se debe aplicar.

2. **`EffectState statement`**:
    - Contiene el estado del efecto que se ha interpretado. Este es un objeto que será ejecutado después de haber sido identificado a partir de `effect`.
3. **`IExpression selector`**:
    - Una expresión que representa un selector opcional, que define a qué unidades o elementos del juego se va a aplicar el efecto.
4. **`List<(Token, IExpression)> _params`**:
    - Una lista de parámetros que se pasan al efecto. Cada parámetro tiene un token y una expresión asociada que define su valor.
5. **`(int, int) codeLocation`**:
    - Guarda la ubicación en el código donde fue declarada la activación del efecto. Esto se utiliza para la depuración y para generar mensajes de error más detallados.

Constructor

- El constructor inicializa la activación del efecto, el selector, los parámetros, y la ubicación en el código. Esto permite que se cree una instancia de `Activation` con todas las propiedades necesarias para realizar la verificación semántica y la ejecución posterior.

Métodos

ErrorsCheck(IExpression expression, List<string> errors)

- **Propósito**: Este método verifica que una expresión sea válida semánticamente. Si no lo es, añade un error a la lista de errores.
- **Funcionamiento**:
    - Llama al método `CheckSemantic()` de la expresión, que verifica si la expresión es válida.
    - Si `CheckSemantic()` devuelve falso, agrega el error generado a la lista de errores.
    - Si ocurre una excepción de tipo `Attention`, captura el mensaje y lo devuelve.
- **Uso**: Este método es utilizado para verificar múltiples expresiones dentro de la clase `Activation` y evitar duplicación de código.

CheckSemantic(out List<string> errorsList)

- **Propósito**: Verifica si la activación del efecto es semánticamente válida antes de que pueda ser ejecutada.
- **Funcionamiento**:
    - Crea una lista de errores que será llenada si se encuentran problemas.
    - Usa `ErrorsCheck` para verificar la expresión `effect` y el `selector`.
    - Verifica que `effect` sea de tipo `String`. Si no lo es, añade un error.
    - Verifica que `selector`, si no es nulo, sea de tipo `List`.
    - Recorre la lista de parámetros (`_params`) y verifica que cada uno de ellos sea semánticamente válido.

- Intenta recuperar el efecto correspondiente a `effect` desde `EffectState.DeclaredEffects`. Si el efecto no está definido, lanza un error de clave no encontrada.
- Si todo es correcto, asigna el efecto a `statement` y llama a su método `Take()` para aplicar los parámetros y el selector.
- Si se acumulan mensajes de atención, lanza una excepción de tipo `Attention`.
- Devuelve `true` si no hubo errores; de lo contrario, devuelve `false`.

RunIt()

- **Propósito**: Ejecuta el efecto una vez que ha pasado la verificación semántica.
- **Funcionamiento**: Llama al método `RunIt()` del objeto `statement`, que ejecuta la lógica del efecto almacenado.

Flujo General

1. **Verificación Semántica**:
    - Antes de que se active un efecto, se verifican sus componentes (el efecto en sí, el selector, y los parámetros) para asegurarse de que son correctos. Si hay errores, se añaden a la lista y se detiene la activación.
2. **Ejecución del Efecto**:
    - Si todo es semánticamente correcto, se recupera el efecto correspondiente a través de un diccionario llamado `EffectState.DeclaredEffects`. Luego, se ejecuta el efecto utilizando el método `RunIt()`.

---

8. Clase `CardState`

Esta clase administra el estado de una carta en el juego, y se encarga de activar efectos cuando se cumplen ciertas condiciones.

1. **`List<Activation> activations`**:
    - Se almacena una lista de activaciones (`Activation`) que representan las distintas activaciones posibles de la carta. Estas activaciones contienen efectos que la carta puede ejecutar.
2. **`EffectState effect`**:
    - La referencia a `EffectState` no se maneja directamente en `CardState`, sino que está encapsulada dentro de una instancia de `Activation`, que controla tanto el efecto como el selector (objetivos) y parámetros.

3. **`void ActivateEffect(IExpression selector, List<(Token, IExpression)> parameters)`**:
    - Este método maneja la activación de los efectos en la carta, ahora utilizando una instancia de la clase `Activation`.

- Cada activación está asociada a un efecto en particular y puede tener un selector (objetivos) y parámetros que serán pasados a la activación.
- Se crea una instancia de `Activation` con los parámetros y el selector adecuado, y se llama a `CheckSemantic()` para verificar que todo está correcto antes de activar.

Flujo de activación

El flujo de activación en `CardState` sería el siguiente:

1. **Crear la activación**: Cuando ocurre un evento en el juego que debería activar un efecto, se instancia una nueva activación (de tipo `Activation`) con el efecto específico, el selector y los parámetros adecuados.
2. **Chequeo semántico**: Antes de ejecutar el efecto, se realiza una verificación semántica con `CheckSemantic()` en la clase `Activation` para asegurarse de que el efecto y sus parámetros sean válidos.
3. **Ejecución**: Si el chequeo semántico es exitoso, el método `RunIt()` de la activación se ejecuta, activando el efecto en el contexto de la carta.

---

9. Clase `OnActivation`

La clase `OnActivation` tiene la función de definir qué ocurre cuando un efecto específico es activado en una carta o un componente del juego. Con base en la clase `Activation`, la lógica de activación ahora debe integrarse de la siguiente manera:

1. **`List<Activation> activations`**:
   - Similar a `CardState`, esta clase manejará una lista de activaciones, donde cada activación contiene un efecto, un selector (objetivos) y parámetros específicos.
2. **`EffectState statement`**:
   - Aunque antes se manejaba directamente dentro de la clase `OnActivation`, ahora `statement` es parte de la clase `Activation`, y los efectos se activan a través de instancias de esa clase.

1. **`void TriggerActivation(IExpression effect, IExpression selector, List<(Token, IExpression)> parameters)`**:
   - Este método ahora debe crear una instancia de `Activation` cuando se activa un efecto en respuesta a un evento del juego.
   - Los efectos, el selector y los parámetros pasados al método son utilizados para crear una activación. Luego, se llama a `CheckSemantic()` en la clase `Activation` para verificar la validez de los parámetros antes de ejecutar el efecto.

Flujo de activación

El proceso para manejar una activación sería:

1. **Recibir un efecto**: Cuando un evento de activación ocurre en el juego, se pasan el efecto, selector y parámetros al método `TriggerActivation()`.
2. **Instanciar `Activation`**: Se crea una nueva instancia de `Activation` con el efecto y los detalles proporcionados.
3. **Validar semánticamente**: `CheckSemantic()` en la activación se ejecuta para verificar que todos los parámetros son correctos y coinciden con las expectativas del efecto.
4. **Ejecutar el efecto**: Si la validación es exitosa, `RunIt()` es llamado para ejecutar el efecto en el contexto adecuado.

---

10. Clase `Localizer`

La clase `Localizer` es una expresión que maneja el acceso a un elemento dentro de una lista en el contexto del juego.

Propósito de la clase `Localizer`

Esta clase permite acceder a un elemento dentro de una lista (`GameList`) utilizando un índice que puede ser una expresión evaluada dinámicamente.

Atributos

- **`(int, int) codeLocation`**:
    - Guarda la ubicación del código en términos de línea y columna para facilitar la depuración o lanzar errores semánticos con la ubicación exacta donde ocurrieron.
- **`IExpression localizer`**:
    - Representa la expresión que se supone que devuelve una lista o una estructura de datos indexable, como un `GameList`.
- **`IExpression index`**:
    - Es la expresión que debe evaluarse para obtener el índice dentro de la lista.

Métodos

1. **`Interpret()`**:
    - Ejecuta la expresión y devuelve el valor correspondiente en la lista.
    - Interpreta la expresión `localizer`, que debería devolver un objeto de tipo `GameList`, y luego usa el valor de la expresión `index` para acceder a un elemento dentro de esa lista.
1. **`CheckSemantic(out string error)`**:

- Verifica que la expresión `localizer` sea de tipo `List`. Si no lo es, genera un error indicando que no se pueden realizar operaciones de indexación sobre algo que no sea una lista.
- Luego, verifica que la expresión `index` sea de tipo `Number`. Si no lo es, genera un error porque los índices deben ser números.

Manejo de errores:

- Si `localizer` o `index` son de tipo `Object` (no tipado), se lanza una excepción de tipo `Attention` con un mensaje de advertencia, informando que las operaciones no están permitidas en ese objeto.
- Si el tipo de `localizer` no es una lista, o si `index` no es un número, se agregan errores al string `error`.

2. **`CheckSemantic(out List<string> errorsList)`**:

- Similar al método anterior, pero aquí los errores se recopilan en una lista en lugar de una cadena. Esto permite un manejo más estructurado de los errores semánticos cuando se realiza la verificación de tipo.
- Agrega los errores específicos relacionados con el tipo de la expresión `localizer` y `index`.

Flujo general

1. **Interpretación**: Se espera que `localizer.Interpret()` devuelva una lista (`GameList`), y luego se evalúa `index.Interpret()` para obtener el índice que seleccionará un elemento de esa lista.
2. **Verificación semántica**: Antes de ejecutar la operación, se verifica que `localizer` sea una lista y que `index` sea un número. Si no lo son, se lanzan errores detallados que incluyen la ubicación del error en el código, facilitando la depuración.

---

11. Clase `Predicate`

La clase `Predicate` es una expresión que encapsula una condición o filtro que se aplica sobre un objeto de tipo `Card` en el entorno del juego. Es un tipo de expresión que representa un predicado, es decir, una función que devuelve un valor booleano (verdadero o falso) sobre un objeto.

Atributos

- **`Environment environment`**:
  - El entorno donde se almacenan y gestionan las variables y sus valores durante la ejecución. Se utiliza para asociar la carta (`Card`) con el token correspondiente dentro del entorno.
- **`Token token`**:
  - Un token que representa un identificador o referencia en el código. Este token se utiliza para asociar la carta con la expresión.

- **`IExpression expression`**:
    - La expresión que se evaluará para devolver un valor booleano. Esta expresión es la que realmente define la condición que debe cumplirse para que el predicado sea verdadero o falso.

Métodos

1. **`Interpret(Card card)`**:
    - Este es un método interno que recibe una carta (`Card`) y la asigna al token dentro del entorno. Luego, evalúa la expresión en ese contexto.
    - El resultado de la evaluación de la expresión debe ser de tipo booleano, ya que el predicado tiene que devolver un valor verdadero o falso.
    - La carta se asocia al token dentro del entorno mediante la llamada `environment.Set(new UnaryObject((-1, -1), card), token);`, y luego se ejecuta la evaluación de la expresión.
2. **`Interpret()`**:
    - Sobrescribe el método `Interpret()` de la clase base `Expression<object>`. Este método devuelve una instancia de `Predicate<Card>`, donde el predicado será la función `Interpret(Card card)` previamente definida.
    - Es importante notar que este método no devuelve directamente un valor booleano, sino un predicado que puede aplicarse a diferentes cartas.
3. **`Type`**:
    - El tipo de la expresión es `ExpressionType.Predicate`, lo que indica que esta expresión representa un predicado que se evaluará sobre un objeto.
4. **`CodeLocation`**:
    - Devuelve la ubicación en el código donde se encuentra el token. Calcula esta ubicación sumando la longitud del valor del token a la posición original. Este valor es útil para generar mensajes de error que indiquen la ubicación precisa del código donde ocurrió el problema.

Verificación Semántica

La verificación semántica garantiza que la expresión que se está evaluando en el predicado sea válida y coherente con el tipo de datos esperado (en este caso, un valor booleano).

1. **`CheckSemantic(out string error)`**:
    - Verifica que la expresión sea de tipo booleano. Si no lo es, se agrega un mensaje de error especificando que la expresión a la derecha del token no es un booleano.
    - También lanza una excepción de tipo `Attention` si la expresión es de tipo `Object`, ya que no se puede operar con objetos sin tipo definido en este contexto.
2. **`CheckSemantic(out List<string> errorsList)`**:

- Similar al método anterior, pero recopila los errores en una lista de strings. Esto permite un manejo más estructurado de los errores cuando hay múltiples problemas de tipo en el código.

Flujo General

1. **Interpretación**:
    - El método `Interpret()` devuelve un predicado que puede aplicarse a una carta. Cuando el predicado se aplica a una carta, el entorno se actualiza con la carta actual, y luego se evalúa la expresión asociada al predicado para determinar si es verdadera o falsa.
2. **Verificación semántica**:
    - Antes de ejecutar el predicado, se verifica que la expresión sea de tipo booleano. Si no es así, se generan errores detallados para facilitar la depuración del código.

---

12. Clase `Selector`

La clase `Selector` es una expresión que realiza una selección de objetos, de diferentes fuentes (como el mazo del jugador, el campo de batalla, la mano, etc.), usando un predicado que define una condición para seleccionar ciertos elementos. También permite especificar si se debe devolver un solo elemento o una lista de elementos que coincidan con el predicado.

Atributos

1. **`IExpression predicate`**:
    - Expresión que representa el predicado, es decir, la condición lógica que determina qué elementos deben ser seleccionados.
2. **`IExpression source`**:
    - Expresión que representa la fuente desde donde se van a seleccionar los elementos. La fuente puede ser, por ejemplo, el mazo del jugador, el campo de batalla, la mano, etc.
3. **`IExpression parent`**:
    - Esta expresión parece representar una fuente de elementos padre, que puede ser útil si la selección depende de algún tipo de jerarquía en el juego.
4. **`IExpression single`**:
    - Una expresión booleana que indica si se debe devolver un único elemento o una lista de elementos que cumplan con el predicado.
5. **`(int, int) codeLocation`**:
    - La ubicación en el código donde se encuentra esta expresión. Se usa principalmente para la gestión de errores y advertencias, proporcionando información sobre dónde ocurrió un problema en el código.

Métodos

1. **`Interpret()`**:
    - Este método se encarga de interpretar la selección. Primero, determina la fuente de los elementos a través de la expresión `source`. Algunas fuentes posibles incluyen:
        - `parent`: una fuente padre.
        - `playerDeck`: el mazo del jugador.
        - `opponentDeck`: el mazo del oponente.
        - `playerField`: el campo de batalla del jugador.
        - `opponentField`: el campo de batalla del oponente.
        - `playerHand`: la mano del jugador.
        - `opponentHand`: la mano del oponente.
        - `board`: el tablero.
    - Después de obtener la fuente, aplica el predicado a esa lista de elementos para filtrar aquellos que cumplen la condición.
    - Dependiendo del valor de `single`, el método devuelve ya sea un solo elemento (el primero que cumple el predicado) o una lista de todos los elementos que lo cumplen.
2. **`CheckSemantic(out List<string> errorsList)`**:
    - Verifica que la expresión tenga sentido en el contexto semántico. Asegura que los tipos sean correctos y que las expresiones involucradas en la selección sean válidas.
    - Verifica que la fuente sea una cadena o un objeto adecuado.
    - Comprueba que la expresión `single` sea booleana.
    - Asegura que el predicado tenga el tipo adecuado (`Predicate`).
    - Si hay algún problema, se agrega a `errorsList` para su posterior manejo.
3. **`CheckSemantic(out string error)`**:
    - Similar al anterior, pero devuelve un solo mensaje de error en lugar de una lista.

Flujo de Trabajo

1. **Interpretación de la Fuente**:
    - El método `Interpret()` primero interpreta la fuente especificada (por ejemplo, el mazo del jugador o el campo de batalla) para obtener una lista de cartas.
2. **Aplicación del Predicado**:
    - Luego, se aplica el predicado a esa lista, devolviendo solo los elementos que cumplen la condición especificada.
3. **Selección Única o Múltiple**:
    - Dependiendo de la expresión `single`, se devuelve ya sea un solo elemento o una lista de elementos. Si `single` es verdadero, se devuelve solo el primer elemento que cumple con el predicado; de lo contrario, se devuelve toda la lista filtrada.

Verificación Semántica

- **Fuente**:

    - La fuente debe ser una cadena o un objeto, como se verifica en los métodos de comprobación semántica.
    - Si la fuente es `"parent"` y no se proporciona un `parent`, se genera un error.
- **Predicado**:
    - El predicado debe ser de tipo `Predicate`, y se verifica que su semántica sea correcta antes de usarlo.
- **Single**:
    - El campo `single` debe ser de tipo booleano, indicando si se debe seleccionar un solo elemento o no. Si no se proporciona, por defecto se usa un valor `false`.

Posibles Fuentes

- `parent`: Se refiere a una fuente "padre" asociada.
- `playerDeck`: Mazo del jugador.
- `opponentDeck`: Mazo del oponente.
- `playerField`: Campo de batalla del jugador.
- `opponentField`: Campo de batalla del oponente.
- `playerHand`: Mano del jugador.
- `opponentHand`: Mano del oponente.
- `board`: El tablero del juego.

Ejemplo de Uso

Si, por ejemplo, el `source` es `"playerDeck"` y el `predicate` indica que solo se deben seleccionar cartas con un ataque superior a 5, la clase `Selector` devolverá todas las cartas del mazo del jugador que cumplan esa condición. Si `single` es verdadero, solo devolverá la primera carta que cumpla el predicado.

---

13. Clase `Record`

La clase `Record` representa una declaración de cartas en el contexto de un juego, donde se definen efectos y cartas que serán ejecutados en un determinado momento.

Atributos

1. **`bool runned`**:
    - Este campo indica si la ejecución del `Record` ya ha ocurrido. Se utiliza para asegurar que las cartas no se vuelvan a ejecutar accidentalmente si el método `RunIt()` se llama más de una vez.
2. **`List<Card> declaredCards`**:
    - Una lista que contiene las cartas que se han declarado y que se han registrado en la ejecución del `Record`.
3. **`List<IStatement> effects`**:

- Lista de efectos asociados al `Record`. Estos efectos son ejecutables y también son declaraciones (`IStatement`).

4. **`List<IStatement> cards`**:
    - Lista de cartas, representadas como declaraciones, que serán procesadas por el `Record`.
5. **`public (int, int) CodeLocation`**:
    - Representa la ubicación en el código donde se encuentra la declaración. Esta clase devuelve siempre `(0, 0)` para este valor porque no se tiene en cuenta para el control de errores.

Métodos

1. **`SetCards()`**:
    - Este método retorna la lista de cartas que han sido declaradas y ejecutadas. Si aún no se ha ejecutado el `Record` (es decir, si `runned` es falso), se llama al método `RunIt()` para ejecutarlo.
2. **`CheckSemantic(out List<string> errorsList)`**:
    - Verifica si las declaraciones de cartas y efectos tienen sentido semánticamente. Utiliza el método `Errors()` para procesar cada lista de cartas y efectos y verificar si hay errores.
    - Si ocurre una excepción `Attention`, se recopilan los errores y se lanzan.
3. **`Errors(List<IStatement> list, ref List<string> errorsList)`**:
    - Método auxiliar que recorre una lista de declaraciones (`cards` o `effects`) y llama a su método `CheckSemantic()`. Si se detectan errores, se agregan a la lista de errores `errorsList`.
    - Si alguna declaración lanza una excepción `Attention`, se captura y se agrega a la variable `attention`.
4. **`RunIt()`**:
    - Ejecuta las declaraciones de cartas si el `Record` no ha sido ejecutado previamente. El flujo de trabajo del método es el siguiente:
        1. Se almacena el número actual de cartas en `CardState.Cards`.
        2. Se recorren las declaraciones de cartas y se ejecutan.
        3. Después de ejecutar las declaraciones, se obtienen las nuevas cartas creadas (desde `CardState.Cards`), a partir del número inicial.
        4. Las cartas declaradas se almacenan en `declaredCards` y se marca `runned` como verdadero para evitar ejecuciones posteriores.

Flujos Importantes

1. **Declaración y Ejecución de Cartas**:

- Las cartas son procesadas mediante una lista de declaraciones (`cards`), que se ejecutan secuencialmente usando el método `RunIt()`. Las nuevas cartas se agregan a `CardState.Cards`, y se extraen las cartas declaradas después de la ejecución.

2. **Chequeo Semántico**:
    - Tanto las cartas como los efectos asociados al `Record` son verificados semánticamente antes de su ejecución. Si algún elemento en la lista no pasa el chequeo, se recopilan los errores y se lanzan las excepciones pertinentes.
3. **Control de Errores**:
    - El método `Errors()` maneja el control de errores tanto para las cartas como para los efectos, agregando los errores a una lista y lanzando excepciones si es necesario.

Resumen

- La clase `Record` permite definir y ejecutar un conjunto de cartas y efectos en el juego.
- Usa listas de declaraciones (`IStatement`) para procesar tanto las cartas como los efectos.
- El método principal, `RunIt()`, ejecuta las declaraciones y registra las cartas creadas.
- Los métodos de verificación semántica aseguran que las declaraciones sean válidas antes de su ejecución.

---

14. Clases de Excepciones Personalizadas herederas de Exception

1. **Clase `Attention`**:
    - Se usa para lanzar errores relacionados con la lógica que necesita una atención especial. El mensaje de error se almacena y se sobrescribe la propiedad `Message` de la clase base.
    - **Constructor**:
        - Acepta un string `message` que especifica el error o situación que requiere atención.
    - **Uso esperado**:
        - Es ideal para manejar situaciones donde un error lógico es detectado en una operación específica, como cuando un valor inesperado o fuera de lugar se encuentra en la semántica del programa.
2. **Clase `RunningError`**

- Se usa para capturar errores relacionados con la ejecución del programa o del juego, posiblemente cuando algo inesperado ocurre durante el procesamiento de las cartas o la ejecución de acciones.
- **Constructor**:
    - Almacena un mensaje de error que puede ser mostrado o registrado para depuración.

throw new RunningError("An unexpected error occurred during execution.");

Clase `ParsingError`:

- Hereda de `Exception`.

- Maneja errores relacionados con el análisis o parsing, por ejemplo, cuando el compilador o intérprete encuentra un error durante el análisis de la entrada del usuario.
- **Constructor**:
    - Acepta un mensaje que indica la causa del error durante el análisis sintáctico o semántico.

Resumen

Estas tres excepciones permiten categorizar y manejar errores de manera específica dentro del flujo del programa:

- **`Attention`**: Para situaciones que requieren una acción o verificación especial.
- **`RunningError`**: Para manejar problemas en la ejecución, como fallas lógicas durante la operación.
- **`ParsingError`**: Para indicar problemas en la fase de análisis o parsing del código.

---

15. Clase `GameContext`

La clase `GameContext` es una implementación importante dentro del sistema del juego, ya que actúa como un **contexto global** que gestiona el estado del juego, específicamente la interacción entre jugadores y los objetos del tablero. Vamos a desglosarla punto por punto sin mostrar el código directamente.

Patrón Singleton

La clase sigue el **patrón singleton**, lo que significa que solo puede haber una única instancia de `GameContext` en todo el programa. Esto es clave para manejar el estado global del juego, asegurando que todos los jugadores y elementos interactúen dentro de un único entorno compartido. La propiedad `Context` es la que implementa este patrón, verificando si ya existe una instancia del contexto, y si no, crea una nueva.

Diccionario de Jugadores

En `GameContext`, hay un **diccionario** que mapea facciones (por ejemplo, `Clouds` y `Reign`) a jugadores específicos. Esto permite que el contexto gestione múltiples jugadores y sus respectivas facciones, asignando roles claros a cada uno.

Tablero de Juego

El tablero del juego es otra parte central del contexto. La clase tiene una referencia al objeto `Board`, que se inicializa usando el contexto de uno de los jugadores (en este caso, el jugador `Clouds`). Este tablero contiene información clave, como el jugador que tiene el turno actual (`TriggerPlayer`) y otros datos que permiten gestionar el estado del juego.

Métodos de Acceso a Elementos del Juego

La clase tiene varios métodos que permiten **acceder a diferentes aspectos del juego**, como las manos de los jugadores, su campo de batalla, el cementerio y el mazo. Para cada uno de estos,

existen métodos para obtener tanto los elementos del jugador que tiene el turno actual como del jugador contrario. Cada uno de estos métodos devuelve un objeto de tipo `GameList`, que parece ser una lista de cartas asociadas al jugador correspondiente.

Comprobación Semántica y Ejecución

Como parte de la implementación de la interfaz `IExpression`, `GameContext` tiene métodos para **verificar la validez semántica** del contexto y ejecutar acciones sobre él. Aunque en esta clase específica la verificación semántica no realiza operaciones complejas (simplemente retorna `true`), está presente para garantizar la consistencia y validez del contexto cuando sea necesario.

La **ejecución** del contexto simplemente devuelve el contexto mismo. Esto tiene sentido, ya que el contexto es el entorno global del juego, y no necesita "ejecutarse" en el mismo sentido que otras expresiones o acciones del juego.

Análisis Sintáctico

Propósito General del Parser

Este parser tiene el objetivo de analizar la lista de tokens generados por el lexer y organizar la estructura del código del jugador en un formato que el intérprete pueda ejecutar. Se trata de una parte clave del análisis sintáctico, donde se revisa que las combinaciones de tokens sigan las reglas gramaticales de tu lenguaje.

Componentes Clave:

1. **Stack<Environment> environments**: Los *environments* en este contexto gestionan el ámbito y el contexto de variables o entidades como cartas y efectos, utilizando una pila para manejar entornos anidados.
2. **input y pos**:
    - `input` se usa como un buffer para almacenar datos intermedios mientras se parsea y se puede restablecer con el método `Reset`.
    - La propiedad `pos` se utiliza para generar mensajes de error específicos sobre la ubicación del problema en el código del usuario. Si ocurre un error, puede dar feedback sobre qué token causó el error y dónde está ubicado.
3. **Parse()**: Este es el método principal que dirige el análisis sintáctico. Sigue un ciclo mientras no consuma un token final. Dentro del ciclo, el parser va evaluando tokens con las condiciones `LookAhead` para determinar si un bloque es un "efecto" o una "carta".
    - `LookAhead` parece verificar el tipo de los tokens sin avanzar el iterador. Esto te permite anticipar qué viene y decidir cómo procesarlo.
    - Si se encuentra un error, como una declaración incorrecta, se lanza una excepción `ParsingError`, que puede interrumpir el análisis o manejarse para continuar en un modo de pánico (es decir, saltando tokens hasta encontrar uno que sea seguro procesar).

4. **Panic Mode**: Este es un mecanismo que ayuda a manejar errores de sintaxis sin interrumpir todo el proceso de parsing. La idea es que, si algo falla, el parser "entra en pánico" y consume tokens hasta encontrar un lugar seguro para reanudar el análisis.

Esta parte inicial del parser se centra en organizar la infraestructura para procesar las declaraciones de cartas y efectos.

Private Helpers

1. EnumeratorOfTokens (Clase Interna)

Esta clase es fundamental en el parser, ya que actúa como un iterador especializado sobre la lista de tokens que recibes del lexer. Aquí se encuentran varios métodos que permiten moverse y mirar tokens de manera controlada:

- **`Current` y `Previous`**: Te permiten acceder al token actual y al anterior sin necesidad de mover el iterador. Esto es útil en varias fases del análisis sintáctico cuando necesitas referenciar tokens cercanos.
- **`TryLookAhead`**: Este método verifica el siguiente token en la lista sin mover el iterador. Es un mecanismo clave para tomar decisiones sobre el tipo de estructura sintáctica que sigue en el código sin modificar el estado del parser. Si ya has alcanzado el final de los tokens, retorna `null`.
- **`MoveNext`**: Avanza al siguiente token en la lista. Si has llegado al final de los tokens, marca una condición de finalización (poniendo `current` en -2), lo que es útil para detener el proceso de parsing.
- **`Reset`**: Este método reinicia el iterador, situando el índice en la posición inicial, preparándolo para un nuevo análisis.

2. LookAhead y Consume

Estos son métodos esenciales que permiten verificar y consumir tokens:

- **`LookAhead`**: Verifica si el token actual es de uno de los tipos indicados en los parámetros (usando el array de tipos `TokenType[]`). Si lo es, el parser avanza (`MoveNext()`). Este es un patrón clásico en parsers para adelantar la verificación de tokens sin avanzar hasta estar seguro de que el token es correcto para lo que necesitas analizar.
- **`Consume`**: Similar a `LookAhead`, pero sin avanzar el iterador. Se usa cuando solo necesitas verificar si el token actual pertenece a alguno de los tipos especificados.

3. Comma y AllocateExpr

- **`Comma`**: Verifica si el siguiente token es una coma (utilizada para separar elementos en listas, como en argumentos o declaraciones múltiples). Si no es una coma, retorna `true` si el token actual es el delimitador `CloseBrace` (indicando el final de un bloque).
- **`AllocateExpr`**: Método destinado a gestionar la asignación de expresiones en tu lenguaje. Usa una combinación de `LookAhead` para verificar la presencia de un token

(`DoubleDot`) importante en el lenguaje para comparaciones o declaraciones. Si no encuentra la secuencia esperada, lanza un `ParsingError` con un mensaje específico sobre la estructura esperada.

Helpful Methods

1. `Declaration()`

Este método maneja las declaraciones de variables dentro del código fuente.

- **Propósito**: Detecta la declaración de una variable, verificando si está seguida por un punto y coma (`;`) para definirla sin inicialización, o por un operador de asignación (`Assign`), incremento (`Increase`), o decremento (`Decrease`) para inicializarla o modificarla en el mismo paso.
    - **Proceso**: Si encuentra un `;`, genera una instancia de `Declaration` para indicar que la variable ha sido declarada. Si encuentra un operador de asignación o incremento, usa el método `Comparing()` para procesar la expresión que se asignará a la variable.
    - **Errores**: Si el formato no es correcto, lanza una excepción `ParsingError`, que incluye la ubicación en el código donde ocurrió el problema.

2. `SimpleStatement()`

Este método se encarga de procesar sentencias simples, como logs o declaraciones de variables.

- **Propósito**: La sentencia más común manejada aquí es un log (`Log`) o una declaración. También asegura que las sentencias terminen con un punto y coma (`;`), lo cual es típico en lenguajes de programación que utilizan esta convención.
    - **Proceso**: Si el token siguiente es un identificador, el método invoca `Declaration()` para procesar una declaración de variable. Si el token es un `Log`, invoca `Comparing()` para evaluar la expresión a registrar.
    - **Errores**: Si no encuentra un punto y coma o la sentencia está incompleta, lanza un `ParsingError`.

Este método es útil para manejar las declaraciones simples y mantener una estructura clara en el parser.

3. `If()`

Este método procesa las sentencias condicionales (`if`).

- **Propósito**: Verifica la condición de un `if` y luego procesa el cuerpo de la sentencia condicional, que puede ser una sentencia simple o un bloque de código delimitado por llaves (`{}`). También maneja las sentencias `else`.
    - **Proceso**: Comienza buscando un paréntesis abierto (`(`) para encerrar la condición del `if`. Luego, usa `Comparing()` para evaluar la condición. Si encuentra un bloque

`{}`, invoca `ActionBody()` para manejar el cuerpo del `if`. Si no, utiliza `SimpleStatement()` para evaluar una única sentencia.
- **Else**: Si detecta un `else`, puede procesar un nuevo bloque de código o una sentencia simple.
- **Errores**: Si el formato es incorrecto (por ejemplo, si faltan paréntesis o llaves), lanza un `ParsingError`.

Este método sigue la estructura estándar de los parsers descendentes recursivos para sentencias condicionales.

4. `While()`

Este método maneja las sentencias `while`, que crean bucles basados en una condición.

- **Propósito**: Evalúa una condición booleana y ejecuta un bloque de código mientras la condición sea verdadera.
    - **Proceso**: Al igual que el método `If()`, comienza buscando un paréntesis abierto para encerrar la condición, luego usa `Comparing()` para evaluarla. Si encuentra un bloque `{}`, invoca `ActionBody()`. Si no, ejecuta una sola sentencia con `SimpleStatement()`.
    - **Errores**: Si la estructura del `while` no es correcta (falta de paréntesis o llaves), lanza un `ParsingError`.

Este método refleja la estructura común de los bucles `while` en lenguajes como C, Java, y similares, y maneja tanto bucles con un solo cuerpo como bucles más complejos con múltiples sentencias.

5. `For()`

Este método maneja la sentencia de bucle `for`.

- **Propósito**: Procesa un bucle `for` que itera sobre una colección. El parser espera encontrar un identificador (que será el "item" o variable de iteración) y luego una colección sobre la que iterar, seguida por un cuerpo de sentencias que define lo que se ejecutará en cada iteración.
    - **Proceso**:
        1. Busca un identificador (`TokenType.Identifier`) que representa el elemento actual de la iteración. Si no encuentra el identificador, lanza un error.
        2. Luego, usa `Comparing()` para evaluar la expresión de la colección que será iterada.
        3. El cuerpo del bucle puede ser una única sentencia o un bloque encerrado entre llaves (`{}`). Si encuentra un bloque, invoca `ActionBody()`, y si no, usa `SimpleStatement()` para manejar una sola sentencia.
    - **Errores**: Si la sintaxis no es correcta (por ejemplo, no encuentra un identificador o colección), lanza una excepción `ParsingError`.

El `For()` sigue un enfoque típico de bucles de iteración. El elemento clave aquí es que permite tanto un cuerpo de bucle complejo con múltiples sentencias (bloque `{}`) como una única sentencia.

6. `ActionBody()`

Este método maneja bloques de código, como los que aparecen dentro de `if`, `while`, `for` y otros bloques de control de flujo.

- **Propósito**: Procesa un conjunto de sentencias dentro de un bloque de código encerrado entre llaves (`{}`). Puede contener múltiples sentencias, incluyendo condicionales (`if`), bucles (`while`, `for`), y declaraciones simples.
    - **Proceso**:
        1. **Manejo de Entornos**: Antes de comenzar, si hay más de un entorno en la pila (`environments`), se empuja un nuevo entorno sobre el actual. Esto permite manejar correctamente las variables locales en bloques de código anidados.
        2. Luego, entra en un bucle mientras no encuentre una llave de cierre (`TokenType.CloseBrace`), procesando cada sentencia según su tipo:
            - Si encuentra un `if`, invoca el método `If()`.
            - Si encuentra un `while`, invoca el método `While()`.
            - Si encuentra un `for`, invoca el método `For()`.
            - Si no encuentra ninguno de estos, asume que la sentencia es simple y la procesa con `SimpleStatement()`.
        3. Si se encuentra un error de sintaxis, el parser entra en "modo de pánico" (`PanicMode()`), ignorando tokens hasta que encuentra un punto y coma (`;`) o una llave de cierre, para seguir procesando.
        4. **Manejo de Entornos al Final**: Al final, si se había añadido un entorno nuevo al inicio, lo elimina para volver al entorno anterior.
    - **Errores**: Si la sintaxis de alguna sentencia es incorrecta (por ejemplo, falta un punto y coma o una llave de cierre), se lanza una excepción `ParsingError`.

El manejo de entornos dentro de bloques de código es esencial para asegurarse de que las variables locales a un bloque no se "filtren" a otros entornos. Esta es una estrategia común en lenguajes de programación para implementar ámbitos léxicos.

7. `EffectAllocation()`

Esta función se encarga de analizar las asignaciones de efectos y post-acciones, que posiblemente tengan parámetros y selectores asociados.

- **Propósito**: La función espera procesar una estructura que defina un efecto y opcionalmente una acción posterior (`PostAction`), ambos dentro de llaves (`{}`). También puede incluir parámetros (por ejemplo, identificadores o valores) y un selector que actúa como filtro o condición para aplicar el efecto.
    - **Proceso**:
        1. Verifica que el código comience con una llave de apertura (`TokenType.OpenBrace`).

2. Recorre el contenido del bloque que contiene la definición del efecto:
    - **`EffectParam`**: Define los parámetros asociados al efecto. El cuerpo del efecto puede definirse dentro de un bloque (`{}`) o puede ser una expresión simple.
    - **`Selector`**: Si se define un selector, este debe estar dentro de un bloque y procesarse con el método `Selector()`.
    - **`PostAction`**: Similar al efecto, puede incluir un bloque de parámetros y un selector específico para la post-acción.
3. Al final, verifica que todos los elementos esenciales estén definidos correctamente, de lo contrario, lanza errores de sintaxis.

- **Errores**: Si se omite alguna parte esencial, como el cierre de llaves o la asignación de un efecto o selector, se lanza una excepción `ParsingError`.

Este método devuelve un par de activaciones (`Activation`), una para el efecto principal y otra para la post-acción, si existe.

8. `EffectAllocationBody()`

Este método procesa el cuerpo de un efecto o post-acción, que puede incluir parámetros y selectores.

- **Propósito**: Dentro de un bloque de efectos o post-acciones, se encargará de asociar expresiones a parámetros y selectores. Si un selector está presente, lo delega al método `Selector()`.
    - **Proceso**:
        1. Recorre las llaves (`{}`) que contienen la declaración del efecto o post-acción.
        2. Procesa los parámetros asociados al efecto. Estos parámetros se añaden a la lista `_params` como pares `(Token, IExpression)`.
        3. Si hay un selector dentro del bloque, lo maneja con el método `Selector()`, que permite definir filtros o condiciones adicionales.
        4. Se asegura de que cada parámetro o selector esté correctamente definido. Si no lo está, lanza un error.
    - **Errores**: Si hay errores de sintaxis en la declaración de parámetros o selectores, como la falta de llaves o nombres, lanza un error.

Este método devuelve el selector si se define dentro del cuerpo del efecto.

9. `Selector()`

Este método procesa la creación de un selector, que actúa como un filtro o condición dentro de la declaración de un efecto.

- **Propósito**: Un selector puede tener múltiples componentes, como el origen de los datos (`Source`), una condición específica (`Single`) y un predicado (`Predicate`). Todos estos son expresiones que determinan cuándo y cómo aplicar el efecto.
    - **Proceso**:
        1. Recorre el bloque encerrado en llaves que contiene la definición del selector.

2. Procesa los diferentes componentes del selector:
    - **`Source`**: Define de dónde provienen los datos o sobre qué entidad se aplica el efecto.
    - **`Single`**: Define una condición específica.
    - **`Predicate`**: Define un filtro o condición adicional, como una expresión lógica.
3. Asegura que tanto el `Source` como el `Predicate` estén definidos correctamente.

- **Errores**: Si faltan partes esenciales del selector, como la fuente de datos o el predicado, lanza una excepción `ParsingError`.

El método `Selector()` devuelve una expresión `Selector` que incluye todas las condiciones y filtros definidos.

AST Building

En esta sección del parser, estás trabajando con la creación y manipulación de expresiones. El objetivo aquí es construir un árbol de expresiones (AST) que representarán las operaciones que el intérprete deberá evaluar en tiempo de ejecución.

1. UnaryExpr()

El propósito de este método es manejar las expresiones unarias, que son aquellas donde un operador afecta a un único operando, como `!` (negación lógica) o `-` (negación aritmética):

- **Proceso**: El método usa `LookAhead` para detectar los operadores unarios (`Not` y `Minus`). Si los encuentra, llama al método `Simple()` (que no has mostrado aún, pero probablemente maneja expresiones simples como literales o variables). Luego, devuelve una instancia de `UnaryOperation`, que representa la operación unaria en el árbol de sintaxis abstracta (AST).
- Si no hay un operador unario, delega el procesamiento a `StringConcatenation()`, que probablemente se encargue de concatenar cadenas o manejar expresiones más simples.

2. BooleanExpr()

Este método gestiona las expresiones booleanas que involucran los operadores lógicos `&&` (AND) y `||` (OR):

- **Proceso**: Primero llama a `UnaryExpr()` para asegurarse de que cualquier expresión unaria se procese primero. Luego, entra en un ciclo `while` que busca operadores lógicos. Si los encuentra, construye una `BooleanExpression` que combina la expresión actual y la siguiente expresión unaria.
- Este patrón sigue la regla de precedencia de operadores, donde los operadores unarios tienen mayor precedencia que los operadores lógicos.

3. Part()

Aquí se manejan las expresiones que involucran la potencia matemática (`^`):

- **Proceso**: Llama primero a `BooleanExpr()` para procesar cualquier expresión booleana que pueda preceder a la operación de potencia. Luego, si encuentra un operador de potencia (`PowerTo`), construye una `MathExpression` con la expresión actual y la siguiente expresión booleana.
- Este enfoque modular permite que las operaciones de potencia tengan mayor precedencia que las sumas o restas, pero menor que las operaciones unarias o lógicas.

4. Term()

Este método es para manejar las sumas y restas (`+` y `-`):

- **Proceso**: Primero llama a `Part()`, para asegurarse de que las operaciones de potencia y booleanas se manejen antes de considerar sumas o restas. Si encuentra un `+` o `-`, construye una `MathExpression`, combinando la expresión actual con la siguiente expresión.
- Esto sigue el patrón clásico de parsers que respetan la precedencia de operadores, donde la adición y sustracción se procesan después de las potencias.

5. Comparing()

Este método gestiona las comparaciones relacionales, como `>`, `<`, `>=`, `<=`, `==`, y `!=`:

- **Proceso**: Llama primero a `Term()` para manejar las sumas, restas y cualquier otra operación aritmética. Luego, si encuentra uno de los operadores relacionales, construye una `ComparisonExpression` con la expresión actual y la siguiente.
- Las comparaciones tienen una precedencia menor que las operaciones aritméticas, pero mayor que las operaciones booleanas, lo que asegura que se evaluarán en el orden correcto.

Resumen General:

- **Descomposición Modular**: Se descomponen las expresiones en sus componentes más simples (unarias, booleanas, matemáticas) y aplicando operadores según su precedencia. Cada método se encarga de una parte específica de la gramática de expresiones, lo que hace que el código sea fácil de mantener y extender.
- **Construcción de AST**: Cada vez que se encuentra un operador, se crea una nueva expresión en el AST (`UnaryOperation`, `BooleanExpression`, `MathExpression`, etc.), lo que permite que el intérprete evalúe estas expresiones más tarde.
- **Precedencia de Operadores**: Sigue estrictamente las reglas de precedencia, lo que garantiza que las operaciones más prioritarias (como unarias o potencias) se procesen antes que las menos prioritarias (sumas, restas o comparaciones).

Cards and effects parsing

1. `Card()`

En este fra función `Card()` se encarga de analizar y crear una estructura que represente una carta en el contexto de tu juego. Aquí se procesan diversos atributos de la carta, como su nombre, tipo, rango, facción, poder, y los efectos que ocurren en su activación.

1. Inicialización de Variables

- **Propósito: Estas variables almacenan los diferentes atributos de la carta. Se inicializan como `null` (o listas vacías en el caso de `range` y `onActivation`), y posteriormente serán asignadas cuando se encuentren los tokens correspondientes en el código de entrada.**
- **Atributos**:
    - `name`: El nombre de la carta.
    - `type`: El tipo de la carta.
    - `range`: El rango de la carta (puede ser una lista de valores).
    - `faction`: La facción a la que pertenece la carta.
    - `power`: El poder de la carta.
    - `onActivation`: Los efectos que se activan cuando se juega la carta.

2. Estructura de Bucle

- **Propósito**: El bucle principal analiza todos los tokens hasta que encuentre una llave de cierre (`TokenType.CloseBrace`), que indica el fin de la declaración de la carta.
- **Manejo de errores**: Si algún token no sigue la sintaxis esperada, el parser lanza un error de tipo `ParsingError`, detalla el problema y maneja el error usando el método `PanicMode()` para recuperarse y seguir procesando.

3. Análisis de Atributos

Dentro del bucle se identifican varios tipos de atributos posibles para la carta:

- **`Name`, `Type`, `Faction`, `Power`**:
    - Cada uno de estos atributos se analiza y asigna a la variable correspondiente solo si aún no ha sido asignado (comprobación con `null`). Se utiliza el método `AllocateExpr()` para crear la expresión que representa cada atributo.
- **`Range`**:

  Este atributo puede ser una lista de valores. Se verifica la presencia de los delimitadores (`DoubleDot`, `OpenBracket`, `CloseBracket`) y se agregan los elementos de rango a la lista `range` usando el método `Comparing()`.
- **`OnActivation`**:

- Este atributo es más complejo, ya que puede contener múltiples efectos. Se usa el método `EffectAllocation()` para analizar cada efecto dentro de `OnActivation` y se almacenan en una lista. Este proceso puede ser recursivo si se utilizan corchetes (`[ ]`).

4. Validación Final

Una vez que el bucle termina, se realizan validaciones para asegurarse de que los atributos esenciales (como `name`, `type`, `range`, y `faction`) se hayan declarado correctamente. Si falta alguno, se lanza un error.

5. Retorno

Finalmente, si todos los atributos esenciales se han procesado correctamente, se devuelve un objeto `CardState` con toda la información de la carta.

Resumen

- **Funcionalidad**: Esta función se asegura de que la declaración de una carta en tu juego esté bien estructurada, comprobando cada uno de los atributos esperados y asignando sus valores correspondientes. Además, maneja posibles errores de sintaxis mediante `ParsingError`.
- **Estructura**: El código está bien modularizado, lo que facilita el mantenimiento y la comprensión del flujo del análisis.
- **Puntos Críticos**: Los atributos obligatorios (como `name`, `type`, `range` y `faction`) deben estar presentes, de lo contrario, se genera un error. Además, el manejo de efectos complejos en `OnActivation` agrega una capa de complejidad que está bien resuelta con el uso de listas y el método `EffectAllocation()`.

Este método `Effect()` es parte de un parser que analiza y construye una estructura que representa un "efecto" en tu juego de cartas. El objetivo principal es asegurarse de que los efectos estén definidos correctamente, con parámetros, acciones y un cuerpo asociado.

Explicación de las partes principales:

1. Effect()

1. Inicialización de Variables

- **Propósito**: Se inicializan varias variables que contendrán los diferentes aspectos del efecto:
    - **`environments`**: Apila un nuevo entorno basado en el entorno anterior para manejar el scope de los efectos.
    - **`name`**: Contendrá el nombre del efecto, el cual se asignará al encontrar el token correspondiente.
    - **`paramsAndType`**: Lista de pares que almacena los parámetros y sus tipos.
    - **`body`**: Representa el cuerpo de la acción del efecto.
    - **`codeLocation`**: Almacena la posición del código actual, útil para manejar errores y reportar en qué parte ocurre un problema.
    - **`targets`** y **`context`**: Almacenan los identificadores que representan los "objetivos" y el "contexto" de la acción.

2. Estructura de Análisis

- **Propósito**: El bloque `do-while` se utiliza para analizar los tokens que representan el efecto. El ciclo se detiene cuando se encuentra una llave de cierre (`TokenType.CloseBrace`).
- **Manejo de errores**: Si el parser detecta un error en la sintaxis, se lanza un `ParsingError` y se intenta recuperar con el método `PanicMode`.

3. Análisis de Atributos

Durante el bucle se analiza cada parte del efecto:

- **Nombre del Efecto (`name`)**:
- **Propósito**: Si se encuentra un token que representa el nombre del efecto, se asigna usando `AllocateExpr()`. Solo se permite un nombre por efecto.
- **Parámetros (`paramsAndType`)**:
- **Propósito**: Se analizan los parámetros del efecto. Si hay varios parámetros, se esperan corchetes `{}` que los delimiten. Se agregan a la lista `paramsAndType` usando el método auxiliar `Param()`.
- **Errores**: Se verifican varios errores posibles como no cerrar los corchetes, falta de tipos para los parámetros, etc.
- **Cuerpo del Efecto (`body`)**:
- **Propósito**: Aquí se define la acción del efecto, su cuerpo (`body`). Si ya se ha definido un cuerpo previamente, se lanza un error. Los tokens que definen el objetivo y el contexto también son analizados. Se espera que el cuerpo sea una expresión lambda o un simple statement.
    - **Errores**: Como en las demás secciones, hay múltiples verificaciones de errores posibles, como la falta de corchetes, lambdas o paréntesis.

4. Manejo de Errores

El manejo de errores es central en el parser. Cada vez que se lanza un `ParsingError`, el parser intenta recuperarse usando el método `PanicMode()`.

5. Validación Final

Al final del análisis, se verifica que el efecto tenga un nombre y un cuerpo válidos. Si alguno de estos elementos falta, se lanza un error.

6. Retorno

Si todo está en orden, se devuelve un objeto `EffectState` que representa el efecto completamente analizado.

7. Método Auxiliar `Param()`

El método auxiliar `Param()` es responsable de analizar cada parámetro y su tipo:

- **Propósito**: Este método verifica que cada parámetro tenga un identificador y un tipo asociado. Si alguno de estos falta, se lanza un error.

Resumen

- **Función principal (`Effect()`)**: Analiza y construye un "efecto" en el juego, asegurándose de que se declaren correctamente su nombre, parámetros, cuerpo, objetivos y contexto.
- **Manejo de errores**: Se implementa una robusta detección y recuperación de errores para manejar problemas de sintaxis.
- **Retorno**: Si todo está correcto, se devuelve un objeto `EffectState` con toda la información relevante del efecto.

Este método es clave para el análisis semántico de los efectos en tu juego de cartas, asegurando que cada efecto tenga una estructura válida antes de ser procesado o interpretado.

de intérprete en el juego de cartas:

Conclusiones

En el desarrollo de este intérprete para el juego de cartas, se implementaron con éxito las funcionalidades clave para la correcta interpretación de las definiciones de cartas y efectos, asegurando que el sistema pueda procesar estructuras complejas de manera eficiente y robusta. El uso de técnicas de análisis léxico y sintáctico permitió definir con precisión las reglas de sintaxis del lenguaje específico del dominio (DSL), garantizando que los programadores y jugadores puedan crear cartas y efectos personalizados de forma sencilla y sin errores.

La implementación del **parser** y el manejo de errores fueron aspectos centrales del proyecto. Se logró diseñar un sistema capaz de detectar y corregir errores sintácticos de forma eficaz, utilizando el modo de pánico para recuperarse de fallos sin interrumpir el flujo de ejecución. Esto proporciona una experiencia más sólida y menos propensa a fallos durante la creación de nuevas cartas y reglas.

Además, el uso de estructuras como **árboles de sintaxis abstracta (AST)** y el manejo de entornos para parámetros y acciones brindó una arquitectura flexible que soporta la extensión futura del intérprete. La implementación de expresiones y declaraciones siguiendo patrones de diseño claramente definidos facilita el mantenimiento del código y su escalabilidad.

En resumen, este proyecto de intérprete no solo cumplió con los objetivos funcionales propuestos, sino que también establece una base sólida para futuras mejoras, tales como la optimización del rendimiento o la adición de nuevas características al lenguaje de cartas. La estructura modular y bien organizada del código permitirá incorporar cambios sin comprometer la estabilidad del sistema actual.

Finalmente, el enfoque tomado a lo largo de este proyecto refuerza el aprendizaje de principios fundamentales de programación como el análisis léxico, sintáctico, manejo de errores y generación de árboles de sintaxis abstracta, conocimientos que son aplicables no solo en el ámbito de los juegos, sino también en otros dominios que involucren la interpretación y ejecución de lenguajes específicos.

Referencias

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

Katrib, M. (2020). *Empezar a programar. Un enfoque multiparadigma con C#*. Editorial UH.