

# Segundo Proyecto de Programación

## Gwent++

Olivia Ortiz Arboláez

September 23, 2024

# Introducción

## ● Objetivo del Proyecto

En este proyecto contamos con un juego de cartas en Unity previamente implementado, en el cual es posible elegir un bando y jugar con las cartas pertenecientes a este. No obstante, se busca brindar una mejor experiencia en el juego a partir de la posibilidad de que el usuario pueda [crear sus propias cartas](#).

Fue proporcionado un **Lenguaje de Propósito Específico (DSL)** que es utilizado para ingeniar el código que definirá el comportamiento de las cartas que cree el usuario.

# Introducción



Figure: Nuevas opciones para la creación de cartas.

# Introducción

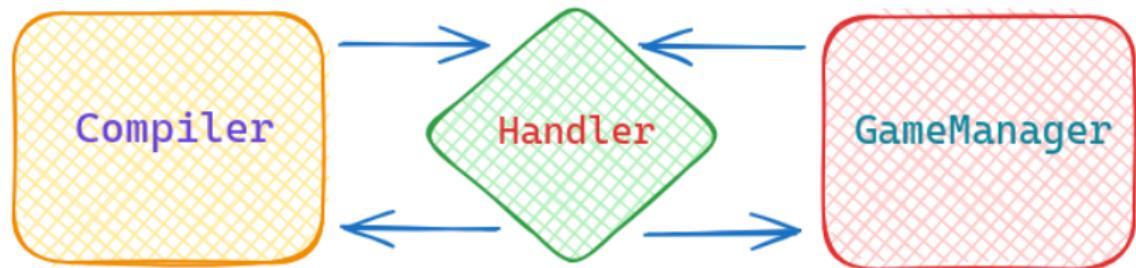
## • **GwentCompiler**

Para poder utilizar el **DSL** nos valdremos de un compilador, el que es (dicho de manera sencilla y directa) el encargado de interpretar el código que ingrese el usuario.

Buscando mantener lo más independientemente posible la parte visual de la parte lógica de este proyecto, se trabajó en el compilador para el lenguaje por separado y después se vinculó este con el juego en Unity.

# Introducción

Aunque luego se profundizará algo más respecto a la integración de ambos ámbitos (una vez abordemos el funcionamiento del compilador), lo principal en cuanto a la interacción entre los mismos es:



# Introducción

- **Compiler** : Interpretará el código y en caso de ser correcto lo convertirá en instrucciones para las cartas del juego.
- **Handler** : Punto de comunicación entre ambos ámbitos, enviará la entrada del usuario al compilador y de regreso transmitirá las instrucciones al GameManager.
- **GameManager** : Lleva a cabo dichas instrucciones.

# Introducción

El **GameManager** tendría semejanza con un administrador del juego, es un script (y por tanto un objeto en el propio juego) que sirve para gestionar cada fase del mismo. Teniendo en Handler un punto de partida para el intercambio de información sobre los eventos entre este y el Compilador, podremos obtener el comportamiento deseado.

Las cartas son creadas a partir de un **Json**, aprovechando la ventaja de poder crear **Prefabs**. Como breve recordatorio, el objeto fundamental de Unity para representar escenas y personajes son los **GameObjects**, y un **Prefab** no es más que un **GameObject** completamente configurado.

# Introducción

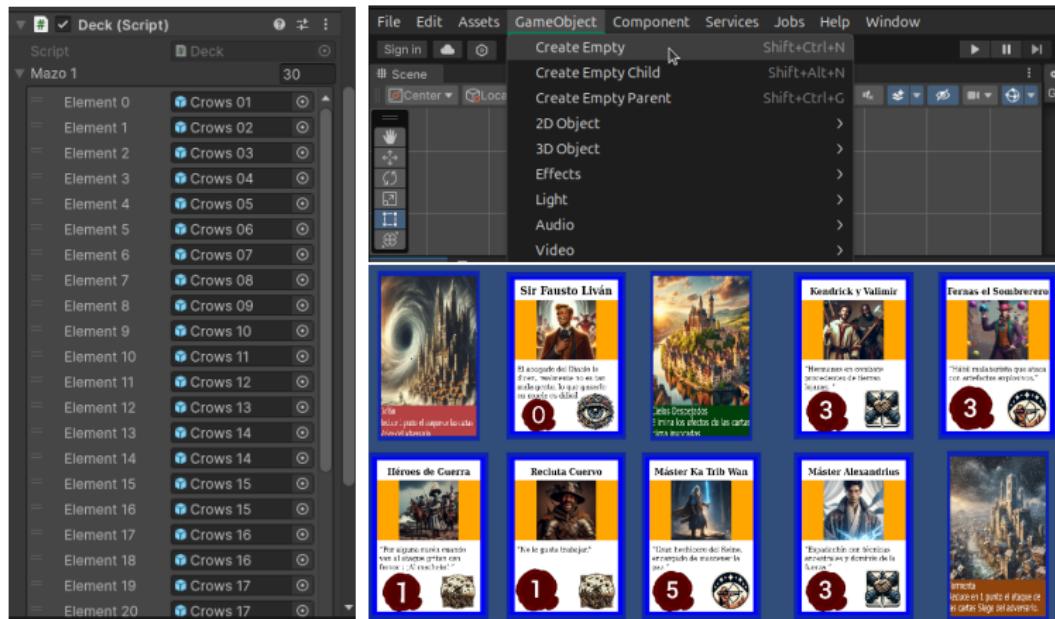
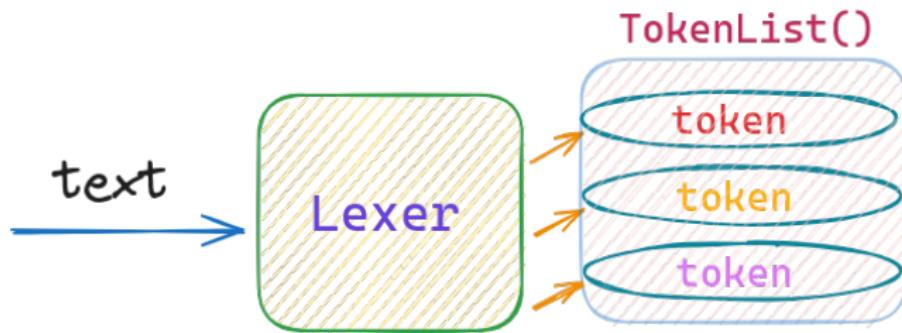


Figure: Haremos exactamente lo mismo pero creando los Prefabs a través de código.

# Lexer

- **Fase 1: Obtener lista de tokens.**

La entrada del usuario, no es más que simple texto en un inicio. El Lexer se encargará convertir este texto en una secuencia de elementos que tengan significado para nuestro compilador. La idea es ir reconociendo los caracteres recibidos en el texto, para ir identificando si pertenecen a nuestro lenguaje y dividiéndolos en tokens.



# Lexer

## ¿Qué es un token?

Es un objeto que representa un valor, un concepto o una entidad, y en el contexto de nuestro compilador, los usaremos para guardar la información y su tipo, que obtendremos del texto.

De C# conocemos que un string es una secuencia de caracteres, por lo que mediante un puntero y algunos métodos de ayuda que lo modifican, se inicia así el proceso de tokenización.

Keywords	Symbols
< "Action" , Tokentype.ActionKeyword >	< "@" , Tokentype.DoubleAtSymb >
< "card" , Tokentype.CardKeyword >	< "!" , Tokentype.ExclamationSymb>

Figure: Algunos ejemplos de tokens.

# Lexer

Se denomina **Keyword** o palabra clave a una palabra reservada que tiene un significado en un lenguaje de Programación. Como ejemplo en este lenguaje tenemos **effect**, **Name**, **while**. Es prioridad del Lexer reconocerlas así como cualquier otro carácter que consista en un token válido para el mismo. No guardamos ni los espacios en blanco, ni el carácter # que añadimos para permitir escribir comentarios en el código.

# Lexer

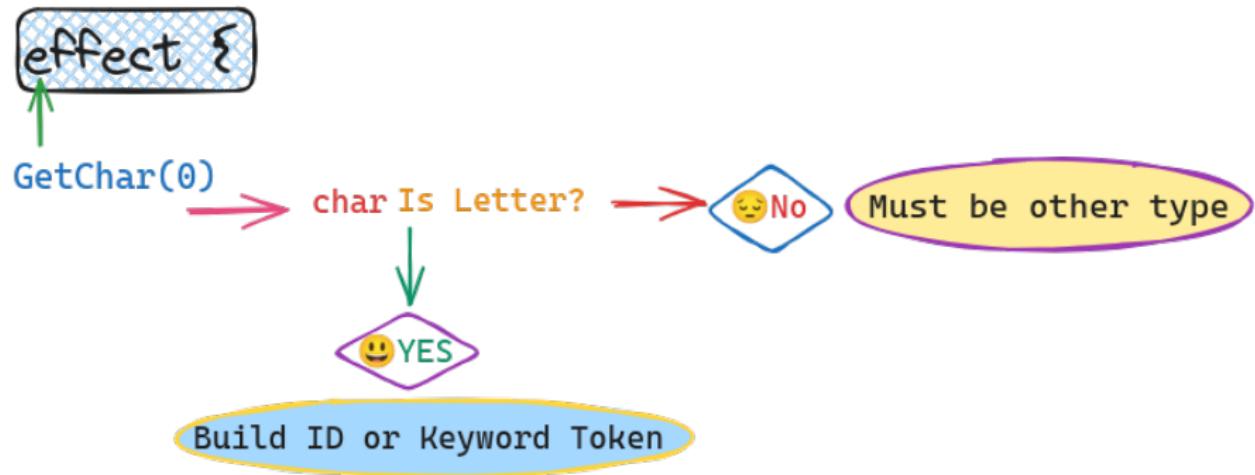


Figure: Ejemplo de reconocimiento.

# Método Tokenize

```
public List<Token> Tokenize()
{// Devuelve una lista con los tokens, cada token tiene su tipo, y su valor.
    Token token;
    List<Token> tokenlist = new List<Token>();
    do
    {
        token = GetToken(); // Sigue construyendo los tokens hasta que
                           // llegues al final del texto.
        if (token.Type != TokenType.WhiteSpaceToken && token.Type
            != TokenType.CommentToken && token.Type !=
            TokenType.SymbolNewLineToken)
        {
            tokenlist.Add(token);
        }
    }
    while (token.Type != TokenType.EOFToken);

    return tokenlist; }
```

# Parser

- Fase 2: Obtener expresiones.

Ahora bien, al igual que en nuestro idioma, el uso correcto del vocabulario no garantiza la comprensión si la estructura sintáctica es incorrecta, debemos verificar si nuestros tokens están organizados de manera sintácticamente correcta para el DSL.



# Parser

**Expresiones:** Una expresión es un conjunto de tokens que podemos evaluar para obtener un resultado. Estas expresiones se componen de:

- Tokens: Son los símbolos básicos que se utilizan para construir la expresión. Tal como vimos pueden ser números, variables, operadores, etc.
- Operadores: Son los símbolos que se utilizan para realizar operaciones entre los tokens. Funciones: Son bloques de código que se pueden llamar para realizar una operación específica (ejemplo aritméticas, lógicas).
- Variables: Son los tokens TokenType.ID , se utilizan para almacenar valores.
- Funciones: Son bloques de código que se pueden llamar para realizar una operación específica.

# Parser

## Expresiones principales

(Para mayor detalle véase el PDF de la orientación)

- CardExpression: Si esta expresión se compila correctamente, podremos crear una nueva carta en el juego. Cuenta con 2 importantes detalles, la declaración (Nombre, Parámetros, Tipo) y la Llamada al Efecto de la carta (OnActivation).
- EffectExpression: Si esta expresión se compila correctamente, podremos crear un nuevo efecto que será ejecutado cuando se juegue una carta que lo tenga asociado.

# Parser

Para el parseo de expresiones usaremos una lógica similar a la de manejo del puntero en el Lexer. Aunque encontraremos algunas diferencias, pues en este caso usamos un Parser Recursivo Descendente. Antes teníamos sólo caracteres, pero ahora con los tokens se pueden formar distintas expresiones. Un ejemplo de esto:

- PowerExpression: 2 ,
- Pero podríamos tener también: PowerExpression: 2 + 3,
- Sin embargo no es válido si recibimos:PowerExpression: "Aguacate",

# Parser

Para no perdernos con esto, en nuestro Parser se empieza parseando las expresiones más generales que a su vez parsean expresiones más pequeñas que están contenidas en ellas. Entonces hemos de tener en cuenta en qué momentos se nos pueden presentar las expresiones, obtener la seguridad de que están sintácticamente correctas y claro, evaluarlas.

Para la evaluación de expresiones fue creado un nuevo tipo, el **GwentObject**, que contará con el valor del token, pero tendremos un GwentType (GwentNumber, GwentString, GwentBool), que servirá de ventaja a la hora de comprobar qué tipo de objeto devuelve una expresión. Realmente su utilidad recae en que a la hora de evaluar, tenemos que contar con un objeto que represente y abarque todos los resultados sin perder la información.

# Parser

Para las expresiones, nos creamos una `IExpression`, esta interfaz nos permitirá trabajar con ellas de forma cómoda. Por tanto todas las expresiones que hagamos tendrán que proporcionar su propia implementación de estos métodos.

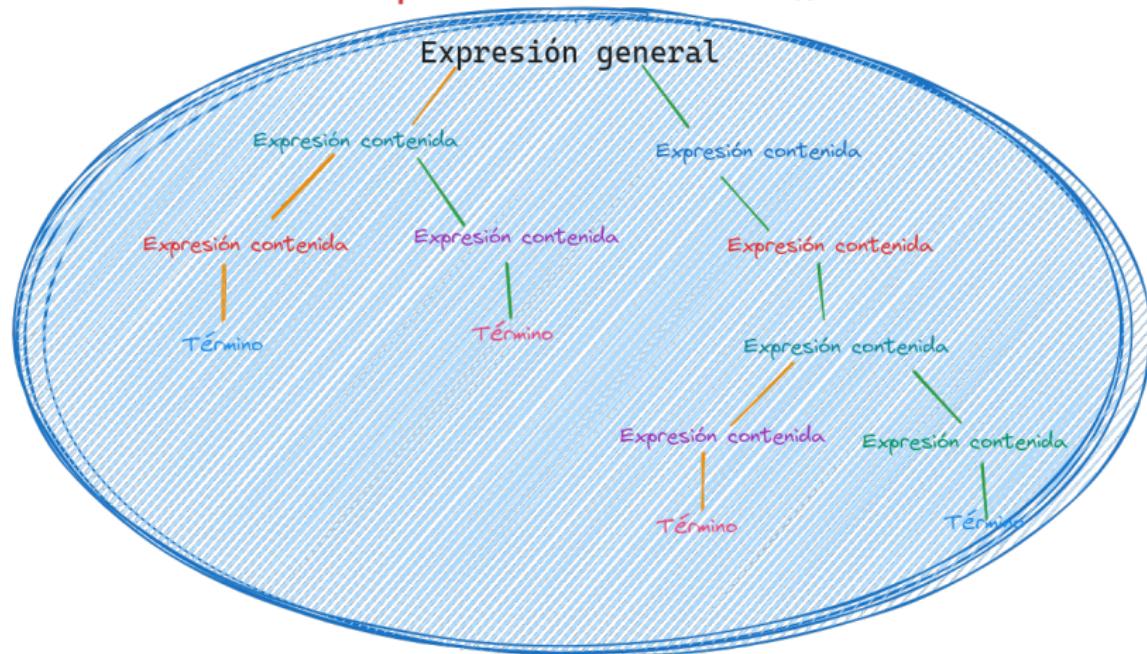
```
public interface IExpression {  
  
    GwentObject Evaluate();  
  
    bool CheckSemantic();  
  
    GwentType ReturnType{get;}  
}
```

# Parser

- Evaluate: Evalúa la expresión y devuelve un GwentObject (este puede ser GwentNull), pero también hace viable que en caso de recibir una expresión como parámetro puede pedir que esta se evalúe a sí misma.
- CheckSemantic: Verifica que no hayan errores de tipo en la expresión y pide comprobar también la semántica de las expresiones que tenga contenidas.
- ReturnType: Devuelve el tipo de retorno esperado de la expresión.

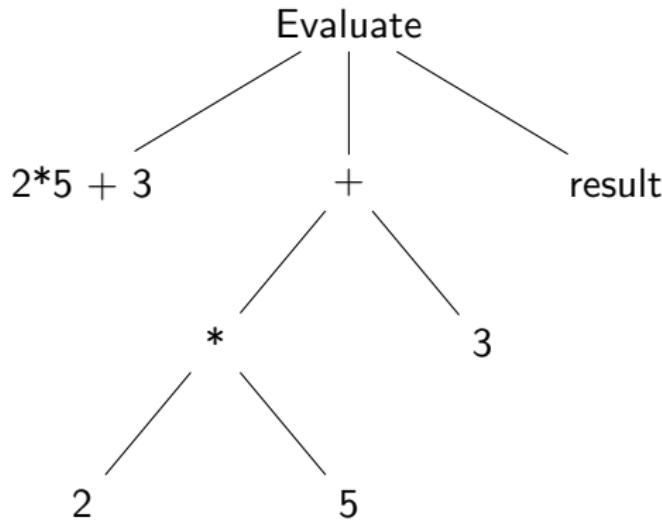
# Parser

## Expression.Evaluate()



# Parser

Ejemplo de evaluación de una expresión sencilla.



# Parser

El Scope es el ámbito o alcance en el que definimos o utilizamos las variables y expresiones que se encuentran en una sección específica del código. En el Parser hay expresiones que definen su propio Scope o simplemente se lo pasan a otras expresiones.

**CheckSemantic** también inicializa las variables en el Scope para que posteriormente en el Evaluate estén seteadas con sus valores y no se produzcan errores. Claramente, si se utiliza una variable no declarada anteriormente obtendremos un error.

# Parser

## Scope

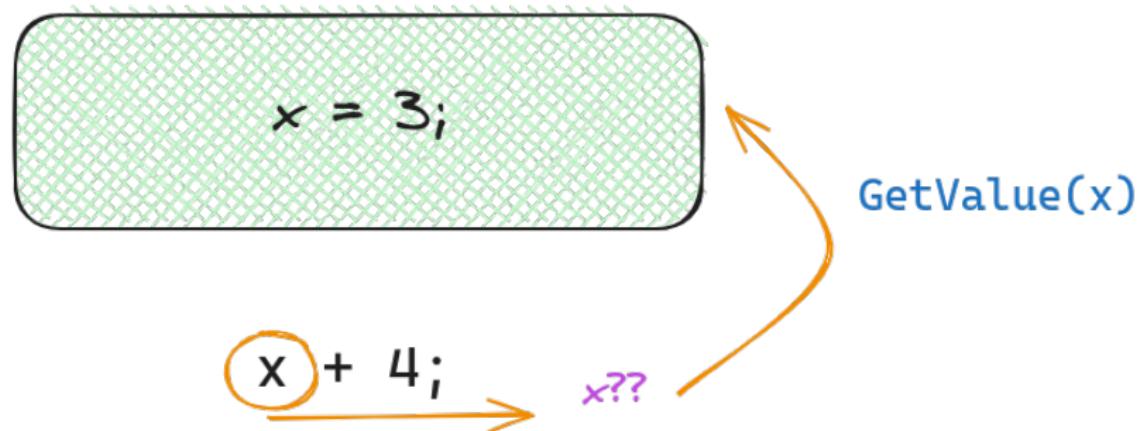


Figure: Ejemplo de Scope, también podemos setear y en caso de que no se encuentre la variable en el Scope actual, pedirá buscarla en el Scope padre de este hasta que la encuentre o suelte una excepción.

# Parser

En cuanto a la creación de las expresiones, basta con implementar la `IExpression` adecuándose a las características de cada una. Aquí trataremos algunas implementaciones que pueden resultar interesantes y sirven de ejemplo.

- **Binary Expression:** Para esta expresión fue creada una clase abstracta que será la base de todas las expresiones binarias. Recibimos 2 expresiones y un `Func` genérico para poder realizar la operación que queramos. El `Func` representa funciones que devuelven un valor (`GwentObject`).

# Parser

```
public abstract class BinaryExpression : IExpression {  
    public IExpression left;  
    public IExpression right;  
    public Func<GwentObject, GwentObject, GwentObject>  
        operation;  
}
```

En los útiles del proyecto contamos con un diccionario para que resulte un tanto más sencillo reconocer un tipo de token y enviar la función deseada como parámetro.

```
public static Dictionary<TokenType, (Func<GwentObject, GwentObject,  
GwentObject>, string)> PredicatesDict = new Dictionary<  
TokenType, (Func<GwentObject, GwentObject, GwentObject>,  
string)>  
{  
    { TokenType.PlusOperatortoken, (GwentPredicates.Sum, "+") },  
    { TokenType_MINUS_MINUS_MinusOperatortoken, (GwentPredicates.Sub, "-") },  
    { TokenType.MultiplicationOptoken, (GwentPredicates.Mul, "*") }  
}
```



# Parser

- Context Expression: Las expresiones tipo context están más conectadas con el tablero. En estas podemos acceder a listas de cartas y al evaluarlas podemos provocar un comportamiento determinado en la partida.

Tenemos que parsear expresiones de Acceso a Zonas del juego y expresiones de métodos que trabajan sobre una lista de cartas.

- Acceso a zonas del juego: podemos pedir una zona del juego y al evaluar esta expresión nos devolverá una lista de las cartas en dicha zona.
- Métodos sobre lista de cartas: podemos realizar una acción a las cartas de una lista. Estos métodos los parseamos según reciban o no parámetros.

Si tenemos una carta, podemos acceder a su datos. Esto es mediante las expresiones Set y Get de CardProperties.

# Parser

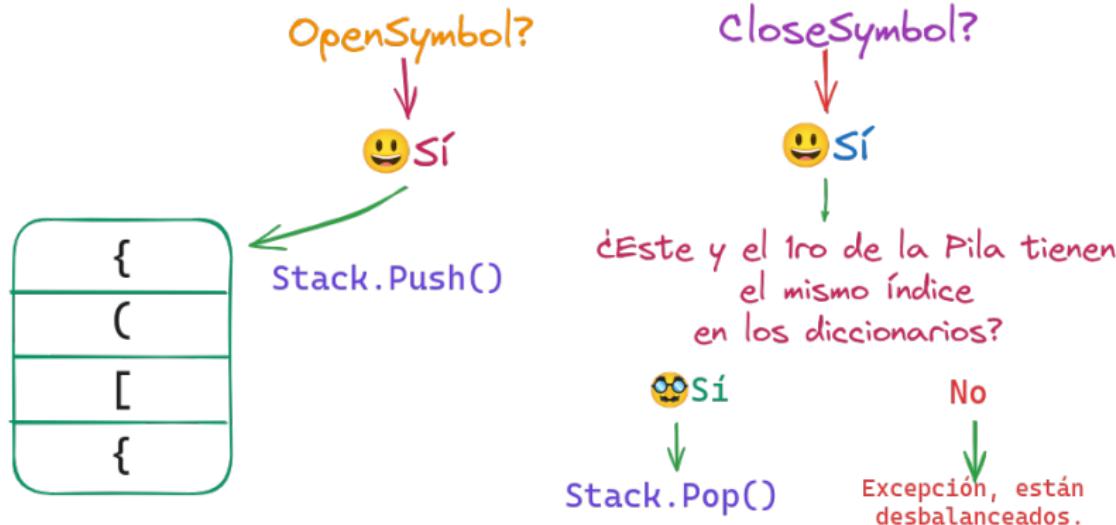
Con una idea hecha de lo que son las expresiones, podemos entrar de lleno en el Parser. Existen métodos auxiliares para mover el puntero<sup>1</sup> por cada token de la lista recibida del Lexer.

Simplemente vamos comprobando si el tipo de los tokens recibidos se ajusta a la estructura del lenguaje. Comprobamos si no están desbalanceados los tokens de las llaves, corchetes y paréntesis y empezamos a construir las expresiones.

---

<sup>1</sup>Es recomendable tener en cuenta siempre dónde queda el puntero a la hora del parseo de expresiones y utilizar el mismo estándar para todas, la mayor parte de los errores enfrentados en el Parser fueron errores de puntero

# Parser



Si al terminar el proceso quedó algún elemento en la pila, están desbalanceados también.

Figure: Representación del método CheckSymbolBalance.

# Parser

Una vez empezamos a parsear las expresiones principales, usaremos recursivamente el método ParseExpression para obtener las expresiones contenidas en las generales.

```
private IExpression ParseExpression(Scope scope)
{
    if (Current.Type == TokenType.KeywordWhiletoken)
        return ParseWhileExpression(scope);

    else if (Current.Type == TokenType.KeywordFortoken)
        return ParseForExpression(scope);

    else if (Current.Type == TokenType.KeywordIftoken)
    {
        return ParseIfExpression(scope);
    }

    return ParseAssignmentExpression(scope);}
```



# Parser

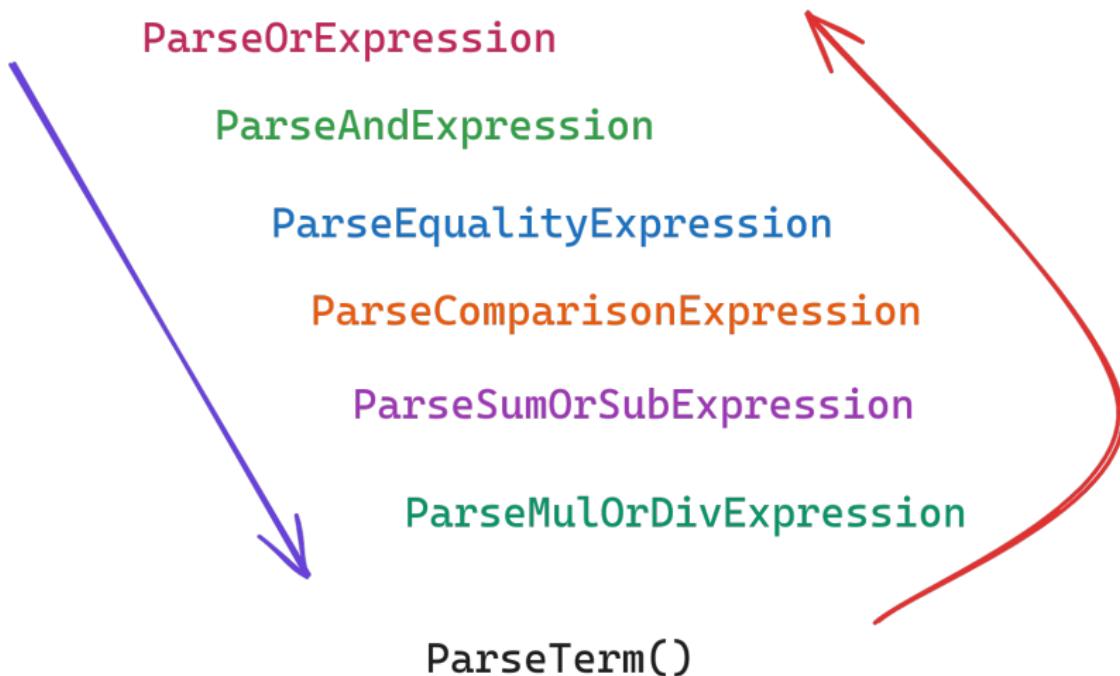


Figura: Representación de la forma de parsear operaciones según su

# Parser

Para este punto, hemos creado nuestra lista de expresiones, que guardaremos como un nuevo objeto AST<sup>2</sup>. En este AST podemos llamar a `Run()` y se ejecutará el chequeo semántico por cada una de las expresiones de la lista y posteriormente (si todo resulta bien) se evaluarán estas expresiones y tendremos un `Json` que podrá ser procesado por el juego para crear las cartas.

A modo de que las cartas creadas sean reconocibles de forma visual en el juego, fue añadido el campo `Image` en la declaración de la expresión tipo carta, si no es especificado ningún valor, la imagen se establecerá a la imagen por defecto.

---

<sup>2</sup>Árbol de Sintaxis Abstracta (AST, por sus siglas en inglés) es una representación abstracta de código fuente como un árbol de nodos sin preocuparse por los detalles de la sintaxis del lenguaje.

## Lanzar Efecto

Una vez tengamos las cartas en el tablero, si una carta con un efecto asignado es jugada, llamará al Compilador mediante el handler y pedirá que ejecute el efecto. Hay expresiones que evaluamos sólo de esta forma, ya que en el punto en que creamos la carta el juego todavía no ha empezado entonces obtendríamos **Reference not set to an instance of the object** en Unity. Los efectos son identificados por el nombre que tienen, y si estos no han sido declarados previamente dará error a la hora de crear la carta, por lo cual en este punto el efecto de la carta puede ser encontrado. Mediante el selector podemos seleccionar el objetivo al cuál será lanzado el efecto.

# Lanzar Efecto



# Conclusiones

- El juego ha obtenido una nueva funcionalidad que resulta en interesantes partidas si es usada correctamente. Es posible editar los mazos predefinidos y restablecerlos.
- Este proyecto ha servido como acercamiento hacia el funcionamiento y propósito de los compiladores así como del empleo de la plataforma Unity.