1. Cards

Esta es la súper clase de la que heredan los dos tipos de carta protagonistas de nuestro juego (*MonsterCards* y *PowerCards*) las cartas de monstruos y las cartas de poder que no son más que habilidades que tendrán esos monstruos.

Cada carta tiene las siguientes propiedades:

-Nombre: No es más que el nombre de la carta
-Descripción pública: Es la descripción del efecto de la carta que se mostrara en el tablero (en el caso de las *PowerCards* debe ser lo más explícita y certera posible) o simplemente una descripción para argumentar la historia en el juego (en el caso de los *MonsterCards*).

Las cartas de monstruo además de los campos anteriores incluyen:

-Tipo: El tipo del monstruo sirve entre otras cosas para a la hora de atacar o recibir ataques, la cantidad de vida perdida es mayor o menor en dependencia de si el tipo del monstruo que recibe el ataque es débil o no contra el que no recibe.
-Estado: es el estado del monstruo (puede ser *Normal*, *Muerto* o *Envenenado*)
-Puntos de ataque:
-Puntos de vida:

	Planta	***Agua***	***Fuego***	***Aire***	***Tierra***
Planta	**0**	**1**	**-1**	**-1**	**1**
Agua	**-1**	**0**	**1**	**0**	**1**
Fuego	**1**	**-1**	**1**	**0**	**-1**
Aire	**1**	**0**	**0**	**0**	**1**
Tierra	**1**	**0**	**1**	**0**	**0**

El tipo que está en la fila es el que ataca y el que esta en la columna recibe el ataque 0 indica que el ataque es neutral ósea que no hay bonificación, 1 indica que es eficiente ósea pierde un 10% más de vida y -1 lo opuesto pierde un 10% menos.

Las cartas de poder contarán con las siguientes propiedades:

-Energía de activación: el usuario debe contar con una cantidad igual o superior de energía para poder usar dicha carta.
-Código: es el código que deberá escribir el creador de la carta. Para su correcta definición debe ser escrito de la siguiente manera:

```
Conditions{
   Condiciones que se deben cumplir para que la carta se pueda ejecutar
}
Actions{
   Acciones a realizar si se cumplen las condiciones
}
```

El funcionamiento del lenguaje esta mejor explicado en la sección 4. Lenguaje

La clase *Card* incluye además dos *enum* uno para los tipos y otro para los estados.
Los *MonsterCard* tienen dos métodos: *UpdateLife()* que modifica la vida del monstruo pero no permite que esta sea negativa y *WeaknessValue()* que dado el tipo del monstruo que la ataca devuelve si es débil, fuerte o neutral con respecto a este ataque.
Para el correcto funcionamiento del programa fue necesario implementar un método *Clone()* para estos objetos. Mas adelante explicaremos su uso.
Estado envenenado le quita el 3% de la vida al monstruo envenenado (mientras este porciento sea mayor que 1 sino le quita 1 un punto) al inicio del turno

2. CardsEngine

En este conjunto de clases es donde se desarrolla toda la lógica del juego:

Deck:

El objeto *Deck* no es más que la colección de *MonsterCards*(monstruos) y *PowerCards*(habilidades) que usted elije para jugar. Para este juego son 3 monstruos y 4 habilidades para cada uno (12 en total). Las habilidades están asociadas a los monstruos a través del diccionario *associations* que establece la relación de cada habilidad con el índice del monstruo en el array *monsters* del *Deck*.

Board:

El objeto *Board* se compone de dos propiedades: *hands*(Cartas en la mano) y *monsters*(Monstruos en el campo).

-*Monstruos en el campo*: por definición de nuestro juego cada jugador contara con tres monstruos a lo sumo en su campo.
-*Cartas en la mano*: cada jugador comenzara el juego con una cantidad de 5 cartas en la mano y en cada turno robara dos cartas más de su mazo de habilidades. Cada jugador podrá tener como máximo 7 cartas en la mano.

Game:

Esta clase es la que controla el flujo del juego, todo el contexto es almacenado en ella, por ello explicaremos detalladamente sus propiedades y funcionalidades:

Propiedades:

-***players***: Esta propiedad está definida por un array booleano de tamaño (cantidad de jugadores) que sindica que el jugador i es real, si es true, en cambio, si es false, es un NPC. Esto es necesario para llevar el control y saber si el juego necesita interactuar con un jugador real o si simplemente ejecutar el código del NPC.
-***decks***: los *decks* de todos los jugadores.

-***turn***: lleva el número de turno en el que se está.
-***currentPlayer***: es el jugador que está jugando.
-***energyPonits***: lleva la cantidad de puntos de energía que tiene cada jugar actualmente, necesarios para poder activar habilidades.
-***losers***: esta propiedad es un array booleano que indica que le jugador *i* perdió si está en true. Cuando quede solo uno en *false* se acabó el juego y este gana el juego.
-***board***: Aquí se almacena el objeto tablero relacionado a este juego
-***npcs***: en una lista guardamos las instancias de todos los *npcs* del juego.

Para controlar el flujo del juego desde fuera de la clase *Game* tenemos siguientes métodos:

-***UpdateTurn()***: Pasa al siguiente turno y reinicia el contador del jugador actual.
-***NextPlayer()***: Pasa al siguiente jugador.
-***SetPlayer()***: Pone el contador de jugador actual en un jugador en específico.
-***UpdateEnergy()***: Actualiza la energía, es usado tanto cuando un jugador juega una carta como para actualizar la energía al comienzo del turno.
-***UpdateLosers():*** Establece que el jugador dado por parámetros perdió.

Otros métodos de la clase son:

-***CanPlay()***: este método compara la energía necesaria para jugar una carta con la energía del jugador y si la primera es menor o igual que la segunda entonces se puede jugar la carta.
-***PlayCard()***: Este método se ejecuta cada vez que un jugador decide que carta usar y contra quien. El monstruo objetivo se decide según la formación del jugador objetivo, el primer monstruo vivo será seleccionado. Posteriormente se intenta ejecutar el efecto de la carta, si se cumplieron las condiciones para ejecutar las acciones de la carta entonces se ejecutan (este proceso está más detallado en la sección 4. Lenguaje), se verifica si el monstruo objetivo murió en caso de que, si se cambia el estado del mismo y se eliminan las cartas asociadas a ese monstruo de la mano del jugador objetivo, se actualiza la energía del jugador quitando el costo energético de la carta jugada y se remueve la carta usada de la mano.

```
47      public Game Clone()
48      {
49          Game newGame = new Game(this.players, 200, this.decks);
50
51          Deck[] newDecks = new Deck[this.decks.Length];
52
53          for (int i = 0; i < this.decks.Length; i++)
54          {
55              newDecks[i] = this.decks[i].Clone();
56          }
57
58          newGame.energyPoints = Engine.Clone<int>(this.energyPoints);
59          newGame.losers = Engine.Clone<bool>(this.losers);
60          newGame.npcs = Engine.Clone<NPC>(this.npcs.ToArray()).ToList<NPC>();
61
62          newGame.turn = this.turn;
63          newGame.SetPlayer(this.currentPlayer);
64          newGame.board = this.board.Clone(decks);
65          newGame.decks = newDecks;
66          return newGame;
67      }
```

NPC:

Los objetos de tipo NPC solo contarán con un índice para identificarlos.
PlayTurn(): es el método que se invoca cuando le toca jugar al NPC.

La jugada del NPC se procesa con los métodos:

-***Potencia()***: este método recibe la mano del NPC como parámetro y crea el conjunto potencia de las cartas, podando por el requerimiento de energía. Posteriormente es llamado el método…
-***Permutation()***: este método permuta cada una de los conjuntos potencia y con cada una de las permutaciones llama al método... (Ambos métodos *Potencia()* y *Permutation()* fueron implementados con BackTracking)
-***EvaluateBetterPlay():*** este método clona la instancia de *Game* y ejecuta las jugadas de la permutación que se recibió. Se evalúa el resultado final en el siguiente método.
-***EvaluateStatistics():*** que recibe los siguientes datos:
1. Vida del oponente.
2. Vida del jugador actual.
3. Energía del jugador actual.
4. Cartas en la mano del jugador actual.

A partir de esta información se evalúa la mejor jugada que es inversamente proporcional a la vida del oponente y directamente proporcional a la vida del jugador actual, a su energía y a la cantidad de cartas en su mano. Importante señalar que cada dato es tratado de manera particular, asignándole una relevancia distinta a consideración nuestra. Por ejemplo, el indicador (Vida del oponente) puede aportar a la calidad de la jugada de 0 a 5 puntos (0 si no hubo daño, 5 si todos los monstruos murieron). Por otra parte, la vida y la energía del jugador pueden aportar 2 puntos cada una y la cantidad de cartas en la mano como máximo 1 punto. La puntuación perfecta para una jugada sería 10.

Engine:

Esta clase es estática y fue necesaria para implementar métodos auxiliares.

-***LoadCards()***: esta función carga todas las cartas desde archivos tipo JSON y las almacena en dos listas, definidas también en esta clase.
-***SaveMonsterCard()*** y ***SavePowerCard()*** son utilizados para guardar permanentemente las cartas creadas por cualquier jugador.
-***GetInitialHand()***: Solo es utilizada para repartir la mano inicial de 5 cartas a cada jugador, completamente en aleatorio.
-***Draw()***: Simplemente toma una carta *random* del *Deck* cuando es el inicio del juego, en los siguientes turnos solo toma una carta del *Deck*, si el monstruo asociado a esta, no ha muerto aun.
-***PlayerWins()***: Verifica si un jugador gano, esto ocurre cuando solo queda un jugador sin perder. Se ejecuta al final de cada turno.
-***PlayerLose()***: Determina si un jugador perdió, esto sucede cuando todos sus monstruos

han muerto. Se ejecuta después de cada ataque.
-***MonsterDied()***: determina si la vida de un monstruo llegó a cero. Se ejecuta después de cada ataque.
-***DeleteDeadMonsterPowerCards()***: Elimina de la mano las cartas asociadas a un monstruo que murió.
-***UpdateMonsterState()***: Este método al inicio del turno realiza las acciones referentes a los estados de cada monstruo (hasta el momento solo se realizan acciones para los monstruos envenenados).
-***ActionDraw()***: este es el método utilizado para robar cartas del *Deck*, tanto por la acción *Draw* del lenguaje, como al inicio del turno de cada jugador.
-***Clone()***: es un método auxiliar utilizado para clonar array de cualquier tipo.

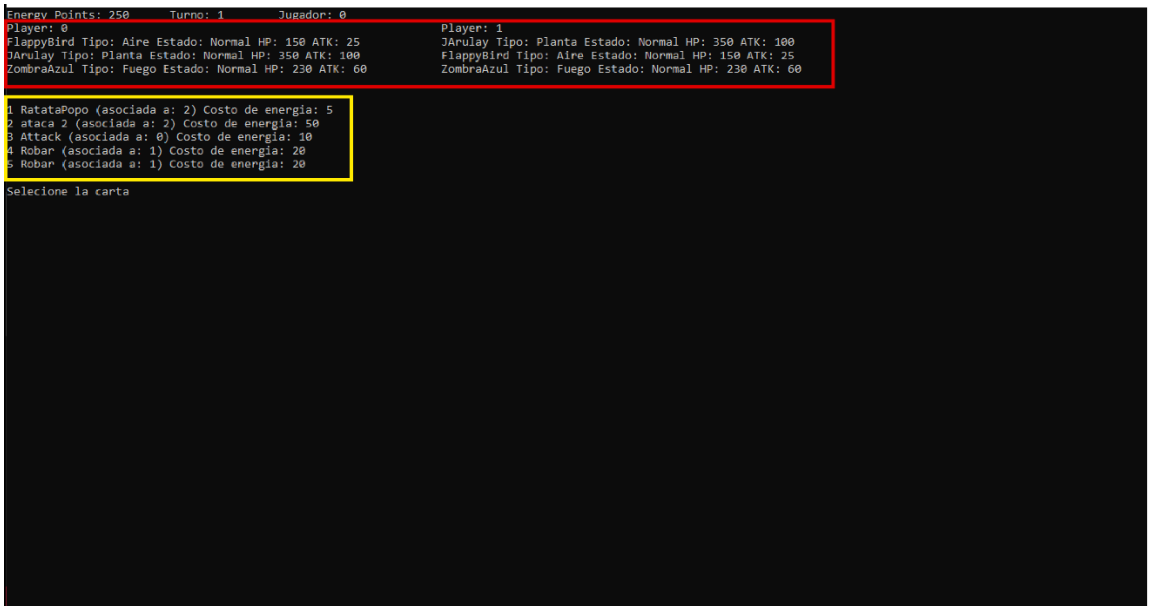
3. ConsoleApp

Bueno esta es la aplicación de consola que permite visualizar e interactuar al usuario con nuestro juego.

Las clases: *CardCreator*, *DeckCreator* e *IDE*, tienen una implantación trivial que guían al usuario a crear una carta, un *Deck* y a programar el efecto de una carta de poder respectivamente.

En la clase *Program* están definidos todos los métodos estáticos q controlan el

El método ***ShowBoard()*** imprime en pantalla el contexto del juego de la siguiente forma.



En el rectángulo rojo se pueden apreciar los monstruos del jugador 0 y del jugador 1 en este caso, donde se puede apreciar el nombre, el tipo, el estado, la vida y el ataque del monstruo, información necesaria para decidir una buena jugada.

Por otra parte, en el rectángulo amarillo se aprecia la mano del jugador actual, de cada carta se específica: nombre, a que monstruo está asociada y su costo de energía. Al

seleccionar una, se podrá leer su descripción y decidir si jugarla o no, en caso positivo se debe elegir un jugador objetivo.

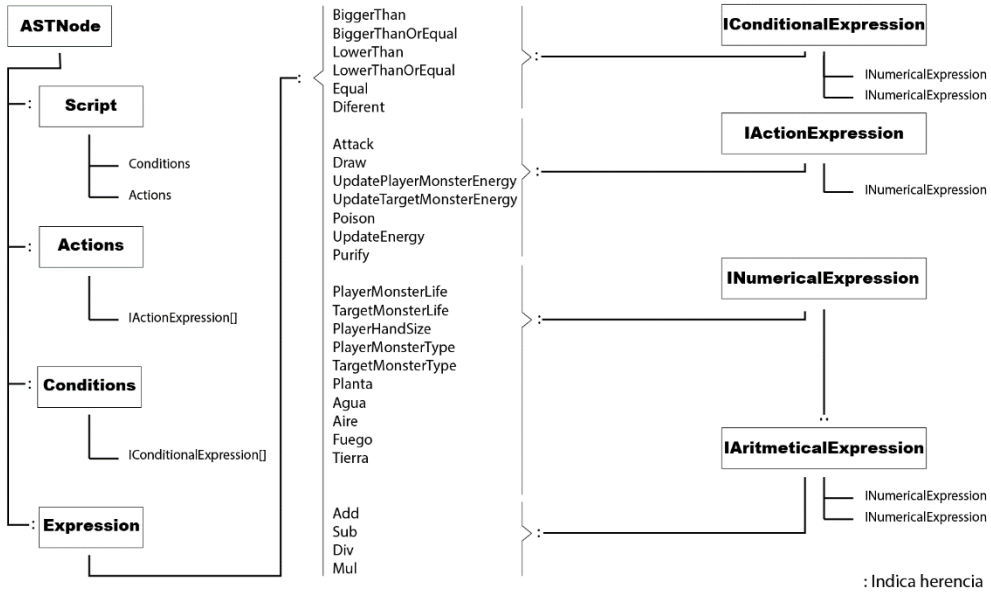
-Algunas aclaraciones:

Para jugar una carta se debe tener energía mayor o igual a la de activación de la misma. En caso de que tenga condiciones de activación, también se deben cumplir.
Cuando un monstruo muere sus cartas asociadas desaparecerán del *Deck* y de la mano.
Un jugador pierde cuando todos sus monstruos han muerto.
Un jugador gana cuando todos sus enemigos son eliminados.

4. Lenguaje

Durante el proceso de creado de un *PowerCard* es necesario dotarla de un script que es el efecto de la carta para esto fue necesario implementar el lenguaje del cual hablaremos a continuación.

Los elementos del lenguaje tienen la siguiente estructura:



Una vez escrito el código que no es mas que una cadena de texto este pasa por un proceso de tokenizacion y parseo para convertir dicha cadena en información relevante y procesable por c#. Lo primero en este proceso es tokenizar la cadena para esto usamos un objeto de tipo *Tokenizer* que a su vez depende de otro de tipo Reader que son clases que cuentan con las herramientas necesarias para realizar este proceso. Lo segundo es el proceso de parseo llevado a cabo por un objeto de tipo *Parser*. Pero antes de explicar el funcionamiento hablemos de un par de conceptos necesarios para entender dichos procesos.

Token: un token es un objeto que representa cada unida de código que tiene importancia para nuestro lenguaje dígase palabras claves o simplemente símbolos y

números. En nuestro caso almacenan la posición en la que aparecen en el *script* (valiéndose de un objeto de tipo Position que guarda las fila y la columna en la que aparece dicho token), el pedazo de código que representa y el tipo de ese pedazo de código que puede ser: *symbol*, *number*, *keyword*, *eof* (end of file) o *unknown* (para cuando hay texto que no es válido en nuestro lenguaje).

Error: un error es un objeto que como su nombre indica representa la aparición de un problema tanto sintáctico como semántico. En nuestro lenguaje este almacena la posición en donde se encuentra el *error* en el *script* (con un objeto Position) y el mensaje de descripción del error.

Position: No es más que una estructura de datos que almacena la línea y la columna donde aparece tanto un *token* como un *error*.

Ahora si empecemos.
El proceso de tokenizacion que no es mas que tomar la cadena de caracteres e ir creando tokens a partir de lo que vamos leyendo se realiza de conjunto en las clases *Tokenizer* y *Reader*.

El objeto Tokenizer tiene tres propiedades:

-***errors***: una lista de errores.
-***tokens***: una lista de tokens.
-***reader***: un objeto de tipo Reader.

El objeto Reader tiene las siguientes propiedades:

-***symbols***: todos los posibles símbolos que pueden ser un token
-***code***: la cadena de caracteres que representa el script
-***codeByLines***: el código separado por líneas
-***line***: la línea actual
-***column***: la columna actual
-***eof***: la posición del final del archivo
-***errors***: una lista de errores

Una vez llamado el método ***Tokenize()*** son cargados todos los keywords y símbolos validos en nuestro lenguaje estos estaban previamente almacenados en la clase estática *TokenCodes* (solo sirve de contenedor para los keywords y los simbolos). Y mientras el *reader* no lea el final la cadena con el método ***CheckEOF()*** intentamos leer un token. Primero desechamos los espacios en blanco para lo cual no valemos del metodo ***IsWhiteSpace()*** de *reader*. Luego tratamos de leer los tokens uno a uno primero los keywords para esto por cada posible keyword llamamos al método ***TryReadToken()*** de reader que nos dice si estamos leyendo o no el código de token que le estamos pasando por parámetro en caso positivo agregamos a la lista de tokens un nuevo token en la posicion actual del código en la que estamos y con el pedazo de código que le corresponde. Si no logramos leer ningún keyword pasamos a los símbolos en caso de no poder leer ningún símbolo, intentamos con un numero usando el método

TryReadNumber() de *reader* que nos devolverá un token de tipo numérico si logro leer un numero correctamente. En ultima instancia de no poder leer nada de lo anterior llamamos al metod ***Read()*** que devuelve un token de tipo *unknown* con el pedazo de código que no es válido para nuestro lenguaje y agrega un error de expresión desconocida a la lista. De esta manera se repite el proceso hasta el final de la cadena.

Del objeto tipo Reader solo nos faltó por mencionar un método el ***CheckEOL()*** que verifica si llegamos al final de una línea pasa ala siguiente y resetea el contador de las columnas.

El proceso de parseo es llevado a cabo por la clase *Parser* valiéndose de la lista de tokens generado en el proceso anterior. Para desplazarse por la lista de tokens y obtener los tokens, el parser tiene tres métodos:

-***UpdateIndex()***: avanza al siguiente token de la lista
-***SetIndexAt()*** se desplaza el índice a un token específico
-***GetIndex()***: devuelve el índice del token actual

Estos tres métodos se encargan de que nunca el índice se salga de la lista de tokens. Durante el proceso de parseo va quedando constituido el AST (Árbol de sintaxis abstracta por sus siglas en inglés) del código de la carta. Lo primero que tenemos que tratar de parsear es un objeto de tipo *Script* compuesto por dos objetos: Conditions y *Actions*, (método ***TryParseScript()***)este proceso no es más que tratar de parsear un objeto *Conditions* y luego un objeto *Actions* en caso de que no hallan errores durante el parseo estos son almacenados en el objeto Script

Para parsear un objeto *Conditions,* compuesto por una lista de *IConditionalExpression* (método ***TryParseConditions()***), verificamos que el *token* actual tenga valor "*Conditions*" (para esto nos valemos del método ***CheckToken()***)en caso negativo se agrega un error a la lista, luego verificamos el token para "*{*" una vez llegado a este punto comenzaremos a tomar todos los tokens, hasta que encontremos alguno de estos tokens: "*}*", "*Actions*" o "*EOF*", y trataremos de parsear expresiones que implementen *IConditionalExpression* con el método ***TryParseCondition()*** y las iremos agregando a la lista de condiciones. Una vez encontremos alguno de los tokens mencionados anteriormente verificamos si es "*}*" y así finaliza el parseo de *Conditions*.

Para parsear un objeto *Actions,* compuesto por una lista de *IActionExpression* (método ***TryParseActions()***) el proceso es muy similar al anterior solo que primero verificamos el token "*Actions*", en caso de haber código entre el "}" de Conditions y el token "*Actions*" se agrega un error y en vez de tratar de parsear expresiones que implementen *IConditionalExpression* tratamos de parsear expresiones que implementen *IActionExpression* con el método ***TryParseAction()***.

Podemos parsear dos tipos de expresiones que implementan *ICondtionalExpression* estos dos tipos de expresiones no existen de manera explícita en el código, pero sí de manera implícita. Estas son las que comparan meramente dos expresiones que implementen *IAritmeticalExpression* o las que comparan tipos de monstruo (planta, agua, etc, …) que en última instancia implementan *INumericalExpression* pues los elementos de un *enum* pueden ser tratados como valores numéricos. Lo primero que tratamos de parsear es un expresión de la relacionada con los tipos para esto usamos

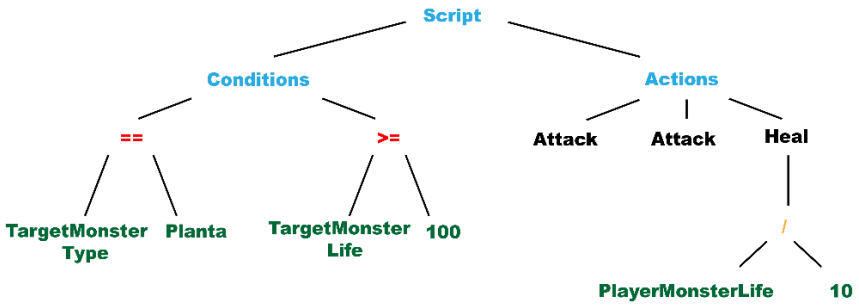
GetTypeVariable() y le damos el token actual si no obtenemos null inmediatamente tratamos de parsear un "==" y luego otro expresión de la misma especie solo de esta forma obtenemos una *IConditionalExpression* valida. En caso de no obtener la primera expresión de variable de tipo tratamos de parsear una expresión numérica con ***ParseExpression()***, luego tratamos de obtener una condición comprobando si el token actual es un operador de comparación valido y luego tratamos de parsear otra expresión numérica. Si todo este proceso ocurrió sin errores al objeto que implementa *IConditionalExpression* se le agrega las expresiones parseadas para luego evaluarlas.

Para parsear una acción es más sencillo (método ***TryParseAction()***) primero tratamos de parsear un token de acción con el método ***GetAction()*** que nos devuelve un objeto que implementa *IActionExpression* si el token es válido. Luego verificamos que venga "("(si la expresión necesita parámetros, tratamos de parsear una expresión numérica) y termine con un ")".

El parseo de una expresión que implementa *INumericalExpression* pero solo con números, variables numéricas y operadores aritméticos es un proceso bastante mas engorroso, las expresiones están divididas en niveles digamos de prioridad para poder lograr el orden operacional de la aritmética. En nuestro caso tenemos solo las operaciones: suma, resta, multiplicación y división. El nivel 1_ es el de las expresiones con menos prioridad, dígase suma y resta el 2_ es para la multiplicación y la división y el 3 es para los números y las variables numéricas (pues sin números no tenemos con que operar). Además, tenemos otros dos niveles el 1 y 2. ¿Por qué dos niveles 1 y dos niveles 2? Pues tenemos dos posibles maneras de parsear una expresión aritmética: una es cuando ya tenemos la expresión que va a su izquierda y otra cuando no la tenemos. El nivel 1 es para cuando no tenemos parte izquierda y el nivel 1_ es para cuando ya lo tenemos, homólogamente pasa para el nivel 2 y el 2_. Teniendo en cuenta el orden operacional de la aritmética, las expresiones de nivel 1 solo se intentarán parsear una vez tengamos una expresión de nivel 2 o superior y las de nivel 2 solo cuando tengamos una de nivel 3.

Ahora un ejemplo de cómo quedaría el AST para el siguiente código de ejemplo

```
Conditions{
   TargetMonsterType == Planta
   TargetMonsterLife >= 100
}
Actions{
   Attack()
   Attack()
   Heal(PlayerMonsterLife/10)
}
```



Durante todo el proceso de parseo cada vez que no es posible parsear alguno de los tokens requeridos es agregado un error a la lista de errores solo que no se hablo de todos para no hacer más extenso el informe.