Proyecto de Programación. Battle Cards

Nombres:

1. **Ana Paula González Muñoz**
2. **Dennis Daniel González Durán**

Grupo: 113

Jerarquía de clases namepace Cards_Game.Logic

Class GAME: consiste en un IEnumerable de estados que controla las acciones del juego.
Consta de un motor basado en el método *Actual* que realiza la acción que desea el jugador y devuelve un nuevo estado del juego.

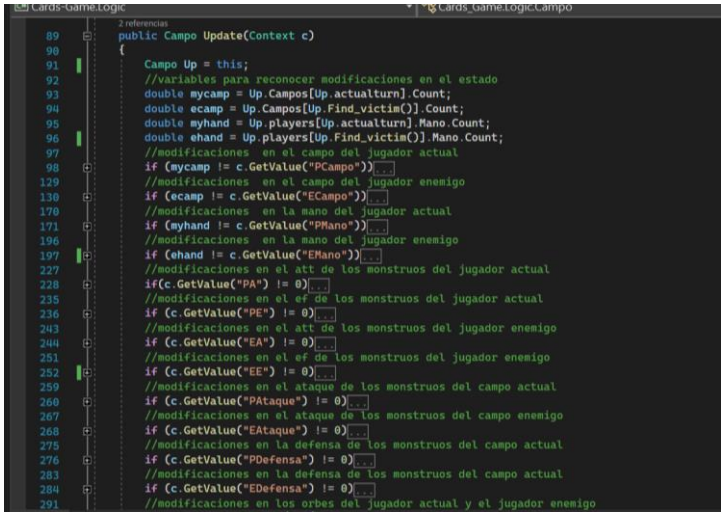
```
IEnumerable<Campo> Actual()
{
    Campo Status = Camp;
    //Variaciones del estado
    while (Status.players[0].vida > 0 && Status.players[1].vida > 0 && Status.Mazo.Count != 0)
    {
        //accion que se quiere ejecutar
        string action = Status.players[Status.actualturn].PLAY(Status);
        //array que almacena los parametros de esas acciones
        string[] decision = action.Split(' ');

        switch (decision[0])
        {
            case "Invocar":
                Actions.Invocar(Status.players[Status.actualturn].Mano[int.Parse(decision[1])], Status);
                break;

            case "Efecto":
                if(Status.Campos[Status.actualturn][int.Parse(decision[1])].ef < 1)
                {
                    Context con = Union.InterpretEffect(Union.CreateContext(Status), Status.Campos[Status.actualturn][int.Parse(decision[1])].code);
                    Status = Status.Update(con);
                    Status.Campos[Status.actualturn][int.Parse(decision[1])].ef++;
                }
                break;

            case "DAtacar":
                Status = Actions.Atacar(Status.Campos[Status.actualturn][int.Parse(decision[1])], Status.Campos[Status.actualturn][int.Parse(decision[1])], Status);
                break;
            case "Atacar":
                Status = Actions.Atacar(Status.Campos[Status.actualturn][int.Parse(decision[1])], Status.Campos[Status.Find_victim()][int.Parse(decision[2])], Status);
                break;
            case "Fin":
                Status = Actions.Fin_Turno(Status);
                break;
            case "Magia":
                Context cont = Union.InterpretEffect(Union.CreateContext(Status), Status.players[Status.actualturn].Mano[int.Parse(decision[1])].code);
                Status = Status.Update(cont);
                Status.players[Status.actualturn].Mano.Remove(Status.players[Status.actualturn].Mano[int.Parse(decision[1])]);
                break;
        }
        yield return Status;
    }
}
```

Class Campo: agrupa los estados actuales del juego.
Consta de colecciones como:

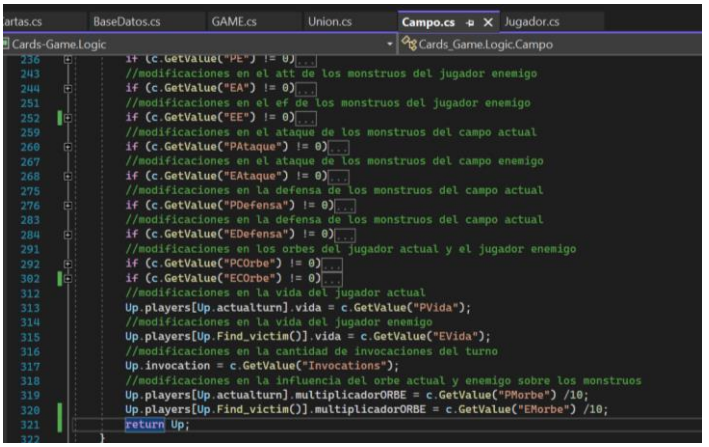
- *players*: almacena las características de los jugadores que se enfrentan al iniciar una partida.
- *Campos*: representación del tablero del juego. Contiene a los monstruos invocados por ambos jugadores.
- *Mazo*: almacena todas las cartas que estarán a disposición de los jugadores a lo largo de la partida.
- *Elements*: almacena los posibles elementos de una carta.
- *Multiplicadores*: almacena la relación de daños provocados de una carta a otra según su tipo.

Consta funciones como:

- *Update:* actualiza el estado del juego a partir de un contexto.



```
Cards-Game.Logic                                   Cards_Game.Logic.Campo
2 referencias
public Campo Update(Context c)
{
    Campo Up = this;
    //variables para reconocer modificaciones en el estado
    double mycamp = Up.Campos[Up.actualturn].Count;
    double ecamp = Up.Campos[Up.Find_victim()].Count;
    double myhand = Up.players[Up.actualturn].Mano.Count;
    double ehand = Up.players[Up.Find_victim()].Mano.Count;
    //modificaciones  en el campo del jugador actual
    if (mycamp != c.GetValue("PCampo"))...
    //modificaciones  en el campo del jugador enemigo
    if (ecamp != c.GetValue("ECampo"))...
    //modificaciones  en la mano del jugador actual
    if (myhand != c.GetValue("PMano"))...
    //modificaciones  en la mano del jugador enemigo
    if (ehand != c.GetValue("EMano"))...
    //modificaciones en el att de los monstruos del jugador actual
    if(c.GetValue("PA") != 0)...
    //modificaciones en el ef de los monstruos del jugador actual
    if (c.GetValue("PE") != 0)...
    //modificaciones en el att de los monstruos del jugador enemigo
    if (c.GetValue("EA") != 0)...
    //modificaciones en el ef de los monstruos del jugador enemigo
    if (c.GetValue("EE") != 0)...
    //modificaciones en el ataque de los monstruos del campo actual
    if (c.GetValue("PAtaque") != 0)...
    //modificaciones en el ataque de los monstruos del campo enemigo
    if (c.GetValue("EAtaque") != 0)...
    //modificaciones en la defensa de los monstruos del campo actual
    if (c.GetValue("PDefensa") != 0)...
    //modificaciones en la defensa de los monstruos del campo actual
    if (c.GetValue("EDefensa") != 0)...
    //modificaciones en los orbes del jugador actual y el jugador enemigo
```



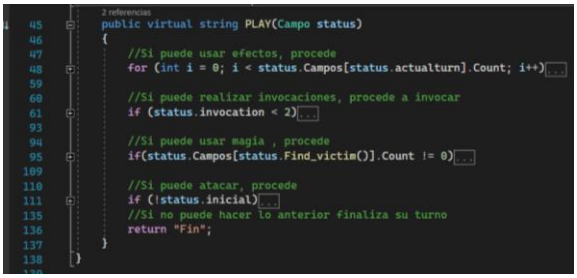
```
artas.cs   BaseDatos.cs   GAME.cs   Union.cs   Campo.cs   Jugador.cs
Cards-Game.Logic                                   Cards_Game.Logic.Campo
    if (c.GetValue("PE") != 0)...
    //modificaciones en el att de los monstruos del jugador enemigo
    if (c.GetValue("EA") != 0)...
    //modificaciones en el ef de los monstruos del jugador enemigo
    if (c.GetValue("EE") != 0)...
    //modificaciones en el ataque de los monstruos del campo actual
    if (c.GetValue("PAtaque") != 0)...
    //modificaciones en el ataque de los monstruos del campo enemigo
    if (c.GetValue("EAtaque") != 0)...
    //modificaciones en la defensa de los monstruos del campo actual
    if (c.GetValue("PDefensa") != 0)...
    //modificaciones en la defensa de los monstruos del campo actual
    if (c.GetValue("EDefensa") != 0)...
    //modificaciones en los orbes del jugador actual y el jugador enemigo
    if (c.GetValue("PCOrbe") != 0)...
    if (c.GetValue("ECOrbe") != 0)...
    //modificaciones en la vida del jugador actual
    Up.players[Up.actualturn].vida = c.GetValue("PVida");
    //modificaciones en la vida del jugador enemigo
    Up.players[Up.Find_victim()].vida = c.GetValue("EVida");
    //modificaciones en la cantidad de invocaciones del turno
    Up.invocation = c.GetValue("Invocations");
    //modificaciones en la influencia del orbe actual y enemigo sobre los monstruos
    Up.players[Up.actualturn].multiplicadorORBE = c.GetValue("PMorbe") /10;
    Up.players[Up.Find_victim()].multiplicadorORBE = c.GetValue("EMorbe") /10;
    return Up;
}
```

Class Jugador: define a un jugador
Consta de propiedades como:

- *Vida*
- *Orbe*
- *Mano*
- *action,* en esta clase es el string vacío

Contiene las propiedades y métodos del *jugador virtual*:

- *PLAY*: recibe el estado actual del juego y decide que acción va a realizar el jugador virtual, esta acción será recibida por el motor de juego que se encargará de ejecutarla si es válida



```
2 referencias
public virtual string PLAY(Campo status)
{
    //Si puede usar efectos, procede
    for (int i = 0; i < status.Campos[status.actualturn].Count; i++)...

    //Si puede realizar invocaciones, procede a invocar
    if (status.invocation < 2)...

    //Si puede usar magia , procede
    if(status.Campos[status.Find_victim()].Count != 0)...

    //Si puede atacar, procede
    if (!status.inicial)...
    //Si no puede hacer lo anterior finaliza su turno
    return "Fin";
}
}
```

Class Person: hereda de la clase jugador. Tiene las mismas características que el jugador virtual.
Modifica por override el método *PLAY* de la clase Jugador y en dependencia de las acciones que realice la persona en la interfaz gráfica, se modifica el string *action* para reconocer la acción deseada. Dicha acción será posteriormente ejecutada por el motor del juego.

Class Cartas: clase abstracta que contiene las propiedades comunes de los tipos de cartas existentes en el juego
Consta de propiedades como:

- *name*: nombre de la carta

- *category*: tipo de la carta (monstruo / magia)
- *efecto*: descripción del efecto de la carta
- *code*: código del efecto de la carta en el lenguaje a interpretar
- *id*: número único que identifica cada carta
- *imagen*: ruta de la imagen de la carta

Class Monstro: hereda de Cartas y caracteriza a todos los monstruos del juego
Consta de propiedades como:

- *ataque*: ataque del monstruo
- *defensa*: defensa del monstruo
- *element*: elemento del monstruo
- *fathers*: elementos necesarios para invocar el monstruo
- *att*: regula la cantidad de veces que puede atacar un monstruo
- *ef*: regula la cantidad de veces que un monstruo puede usar su efecto

Para crear un monstruo se necesita añadir al constructor todos los parámetros de la clase carta y de la propia clase

Class Magic: hereda de Cartas y caracteriza a todos las cartas mágicas del juego
Consta únicamente de las propiedades definidas en la clase Cartas

Para crear una carta mágica se necesita añadir al constructor todos los parámetros de la clase carta

Class Union: clase estática encargada de relacionar el estado actual del juego con el intérprete
Consta de los siguientes métodos:

- *CreatContext*: recibe el estado actual de la partida y crea un objeto de tipo *Context* que agrupa una representación de las propiedades del estado necesarias para los efectos para que el intérprete pueda posteriormente modificarlas. Entre estas propiedades se encuentran:
    - PVida: vida del jugador actual
    - PCampo: cartas en el campo del jugador actual
    - PMano: cartas en la mano del jugador actual
    - PMorbe: influencia del orbe del jugador actual sobre sus cartas
    - PMA: mayor ataque entre los monstruos del campo del jugador actual
    - PMD: mayor defensa entre los monstruos del campo del jugador actual
    - PTA: total de ataque entre los monstruos del campo del jugador actual
    - PTD: total de defensa entre los monstruos del campo del jugador actual
      análogamente existen las representaciones de estas propiedades sobre el campo enemigo, como muchas otras representaciones no mencionadas que aumentan de forma exponencial la dimensión de los efectos del juego

```
2 referencias
public static Context CreateContext(Campo camp)
{
    Context con = new Context();
    //vida del jugador actual
    con.Add(new TOKENS("ENTERO", "PVida"), camp.players[camp.actualturn].vida);
    //vida del jugador enemigo
    con.Add(new TOKENS("ENTERO", "EVida"), camp.players[camp.Find_victim()].vida);
    //cantidad de monstruos en el campo del jugador actual
    con.Add(new TOKENS("ENTERO", "PCampo"), camp.Campos[camp.actualturn].Count);
    //cantidad de monstruos en el campo del jugador enemigo
    con.Add(new TOKENS("ENTERO", "ECampo"), camp.Campos[camp.Find_victim()].Count);
    //cantidad de cartas en la mano del jugador actual
    con.Add(new TOKENS("ENTERO", "PMano"), camp.players[camp.actualturn].Mano.Count);
    //cantidad de cartas en la mano del jugador enemigo
    con.Add(new TOKENS("ENTERO", "EMano"), camp.players[camp.Find_victim()].Mano.Count);
    //cuantas invocaciones se han realizado en el turno
    con.Add(new TOKENS("ENTERO", "Invocations"), camp.invocation);
    //influencia del orbe del jugador actual sobre sus cartas
    con.Add(new TOKENS("ENTERO", "PMorbe"), camp.players[camp.actualturn].multiplicadorORBE * 10);
    //influencia del orbe del jugador enemigo sobre sus cartas
    con.Add(new TOKENS("ENTERO", "EMorbe"), camp.players[camp.Find_victim()].multiplicadorORBE * 10);
    //variables que identifican si habra que modificar el ataque y defensa de los monstruos del campo
    con.Add(new TOKENS("ENTERO", "PAtaque"), 0);
    con.Add(new TOKENS("ENTERO", "EAtaque"), 0);
    con.Add(new TOKENS("ENTERO", "PDefensa"), 0);
    con.Add(new TOKENS("ENTERO", "EDefensa"), 0);
    //orbe del jugador actual
```

- *InterpretEffect*: recibe un objeto *Context* y el código a interpretar del efecto del monstruo. Luego de interpretar este efecto devuelve un nuevo contexto con modificaciones de las propiedades que son representaciones de las propiedades del campo. Luego de esto el método Update del campo recibe este contexto y se actualiza

Class Action: clase estática que contiene métodos que cambian el estado del juego, entre estos se encuentran:

- *Atacar*: dado dos monstruos y el estado actual del juego realiza la acción de atacar
- *Invocar*: dado una carta y el estado actual del juego invoca dicha carta
- *Destroy*: método para destruir cartas del juego
- *Fin_Turno*: finaliza el turno del jugador actual

Clase BaseDatos: se encarga de manejar los elementos en las bases de datos
Consta de las siguientes colecciones:

- *Monstruos*: se rellena cuando se crea un objeto de tipo monstruo
- *Magicas*: se rellena cuando se crea un objeto de tipo monstruo

Consta de los siguientes métodos:

- *LeerMonstruo/Leer Magia:* se encarga de leer las propiedades de los monstruos-magias creados
- *ExisteMonstruo/ExisteMagia*: retorna true o false en dependencia si un monstruo/una magia fue creada anteriormente
- *GuardarMonstruo/GuardarMagia*: se encarga de guardar los monstruos/ las magias en una lista para luego añadirlos al .json
- *CargarId*: se encarga de determinar un Id válido al crear una carta

```
110     public void GuardarMonstruo(Monstruo monstruo)
111     {
112         //lee los monstruos que hay en el json de la ruta
113         LeerMonstruos(RutaMonstruos);
114
115         //si el monstruo no existe lo agrega
116         if (!ExisteMonstruo(monstruo))
117             Monstruos.Add(monstruo);
118
119         //deserializar el json y almacenar los objetos en una lista de monstruos
120         var options = new JsonSerializerOptions { WriteIndented = true };
121         string jsonString = JsonSerializer.Serialize(Monstruos, options);
122
123         //si existe la ruta agrega el contenido al json
124         if (File.Exists(RutaMonstruos))
125             File.WriteAllText(RutaMonstruos, jsonString);
126     }
```

Jerarquía de clases namespace Cards_Game.Logic.Inter
Este namespace es el encargado de interpretar los efectos de las cartas en un lenguaje definido.

Características del lenguaje:

- Es capaz de crear variables de tipo enteras y booleanas.
- Es capaz de evaluar expresiones booleanas con los operadores de comparación, conjunción y disyunción
- Es capaz de evaluar expresiones algebraicas con los operadores algebraicos
- Es capaz de ejecutar ciclos con el operador *WHILE*

Para la gramática:

- Término:= factor ((MULT / DIV) factor)^n
- Expresión:= término ((PLUS / MINUS) término)^n
- Compuesto:= expresión (MENOR / MAYOR / IGUAL) expresión
- Logic:= compuesto (AND / OR) compuesto
- If:= IF(compuesto) BEGIN compuesto END

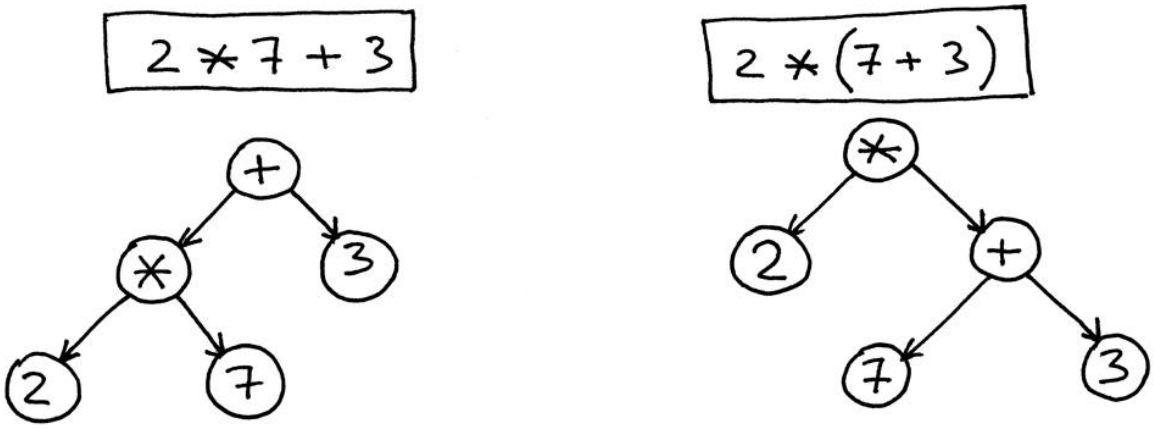
Para la sintaxis:

- Cada línea debe terminar con un ;
- Un bloque de código debe comenzar con un BEGIN y terminar con un END
- Los END intermedios deben ir seguidos por un ; y el END final por un .

Operaciones algebraicas: + - / * ()
Operaciones booleanas: & || < > ==
Asignación: :=
Condicional: IF (condición) { }
Ciclos: WHILE (condición) { }

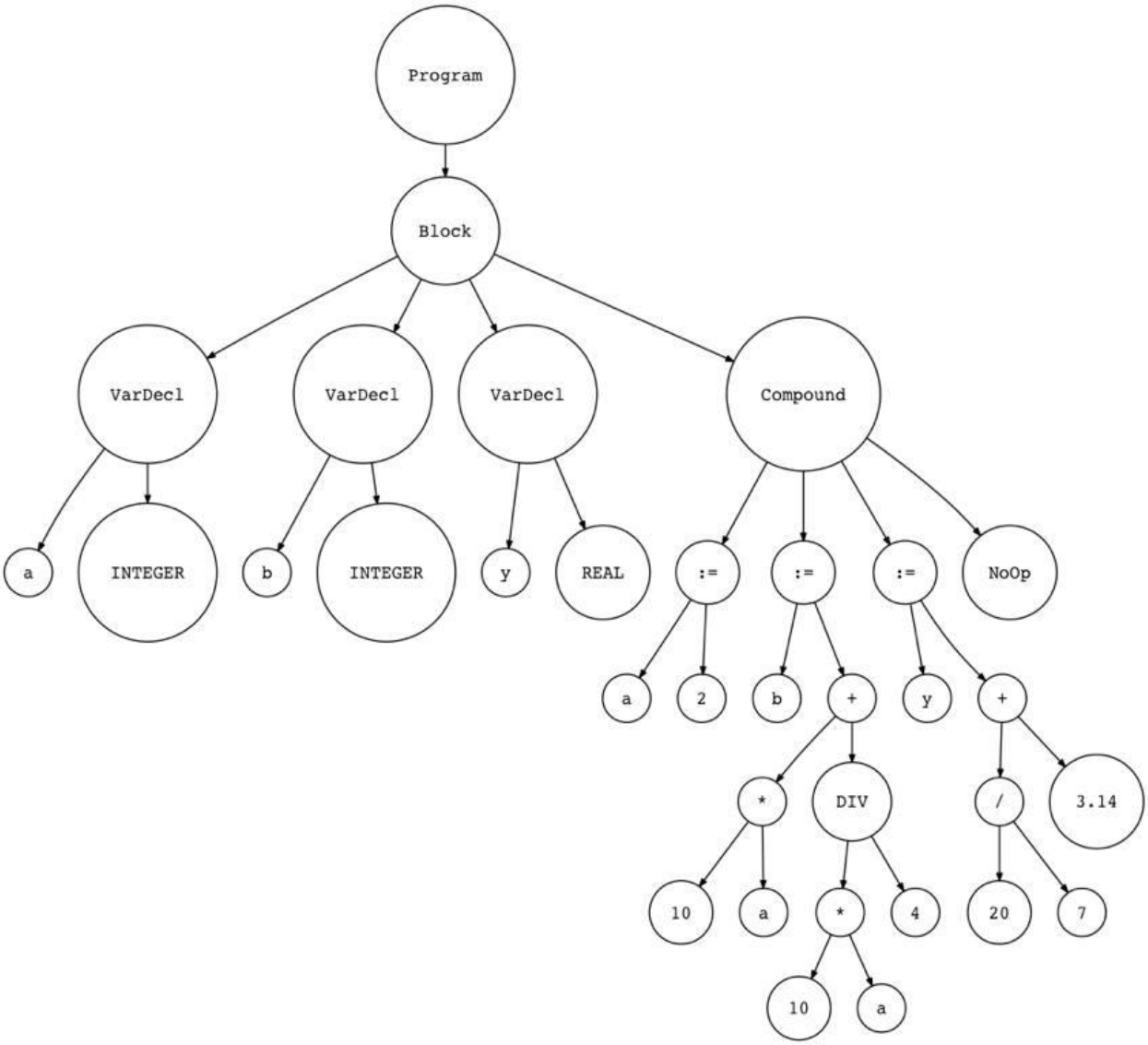
```
151     "Efecto": "Resta 50 puntos de vida al enemigo por cada carta en la mano de ambos jugadores",
152     "Category": "Monstruo",
153     "Imagen": "../images/earthshaker.png",
154     "Code": "BEGIN; EVida:= EVida - 50*(PMano+EMano); END."
```

Estructura:
El interprete toma el texto a interpretar y lo transforma en un árbol de sintaxis abstracta, el cual hace más sencilla su interpretación.



Class AST: de esta clase heredan todos los nodos que pertenecen a este árbol de sintaxis abstractas. Estos nodos son:

- While: Representa un ciclo. Consta de una operación binaria y un compuesto
- Var: Representa una variable. Contiene un token (átomo) y su valor
- UnaryOp: Representa una operación unaria. Contiene un token (átomo) que representa la operación y una expresión
- Num: Representa un número. Contiene el valor de este número
- Condition: Representa una condición. Contiene un compuesto y una operación binaria
- Compuesto: Representa un grupo de nodos. Contiene a este grupo de nodos
- Bool: Representa un booleano. Contiene su valor
- BinOp: Representa una operación binaria. Contiene dos nodos y un átomo que representa la operación binaria entre los nodos
- Asignar: Representa una asignación. Contiene una variable, un token de asignación y un nodo asignado a la variable



Class Lexer: Analizador léxico del interprete
Consta de propiedades como:

- *text*: contiene el texto a analizar
- *current_char*: char analizado actualmente

Consta de métodos como:

- *id*: identifica el tipo de token (átomo) actual y devuelve su id
- *avance*: hace que el análisis se centre en el char siguiente
- *peek*: para analizar el char siguiente sin moverse a él
- *número*: devuelve el número multidígito actual en forma de string
- *gnt*: se encarga de obtener el token siguiente

Class Analizador: analizador semántico del interprete
Consta de métodos como:

- *eat*: identifica si el token actual es el esperado
- *factor*: devuelve el factor que se analiza actualmente
- *term*: maneja la multiplicación y la división en los nodos
- *program*: crea el compuesto de nodos
- *compout_statement*: crea el compuesto de declaraciones
- *statement_list*: crea la lista de declaraciones
- *declaración*: crea la declaración
- *assignment_statement*: crea los nodos de asignaciones
- *variables*: crea variables
- *expression*: maneja la suma y la resta en los nodos
- *parse:* crea y analiza el árbol de sintaxis abstracta
- *Conditional*: crea el nodo de las condiciones
- *Logic*: maneja los & y las || de las expresiones booleanas
- *Comparer*: maneja los términos de comparación de las expresiones booleanas
- *Ciclo*: analiza los ciclos

Class Context: es un contexto para la interpretación de texto. Tiene variables definidas para que el intérprete las use.
Consta de un diccionario que contiene tokens y sus valores
Consta de los siguientes métodos:

- *Add*: agregar tokens al diccionario
- *ContainsId*: comprueba si el diccionario contiene un token
- *GetValue*: obtener el valor de un toquen del diccionario
- *Update*: cambiar el valor de un token de un diccionario

Class Interprete: interpreta el árbol de sintaxis abstracta
Consta de los siguientes métodos:

- *Visit_Num*
- *Visit_Binop*
- *Visit_UnaryOp*
- *Visit_Compuesto*
- *Visit_Asignar*
- *Visit_Var*
- *Visit_While*
- *Visit_Condition*
- *Visit_Bool*
  Todos estos métodos Visit se encargan de evaluar del tipo en que visitan
- *Interpretar*: interpreta el árbol de sintaxis abstracta