Battlefield Simulator

IA - Simulación - Compilación

Juan Marrero Valdés-Miranda
David Campanería Cisneros
Aylín Alvarez Santos

Universidad de La Habana

Índice

1. Mapa

El mapa donde se lleva a cabo la simulación consiste en una matriz de n filas y m columnas. Cada elemento de esta matriz es un componente de tipo *Terrain* que describe las características de cada casilla del mapa con los siguientes atributos:

- *height*: Valor entero positivo que representa la altura de la casilla.
- *m restriction*: Valor entero que representa el costo de movimiento que supone moverse desde esta casilla a una casilla adyacente.
- *camouflage*: Valor entre 1 y 2 que afecta las probabilidades que tiene un soldado que ocupe la casilla de ser detectado y también de que un disparo enemigo hacia este soldado falle.
- *terrain object* Referencia al objeto que se encuentra en la casilla, en caso de que no se encuentre ningún objeto el parámetro será *None*

El mapa se crea instanciando la clase *Map* con los parámetros *rows*: cantidad de filas y *cols*: cantidad de columnas. Con estos parámetros se crea una matriz de *Terrain* con propiedades predefinidas para ser modificados luego antes de comenzar la simulación.

1.1. Distancia en el mapa

Para definir las cordenadas de una posición en el mapa se trata este como un espacio cuadriculado. Considerando las filas como el eje x y las columnas como el eje y. Se define el área de una casilla en la coordenada (x, y) como la intersección del área comprendida entre x y $x + 1$ y el área comprendida entre y y $y + 1$. El espacio que ocupa cada casilla de terreno es de $1m^2$, para calcular la distancia entre dos casillas se utiliza la distancia euclideana entre las coordenadas de ambas casillas.

El elemento que define las dificultades que presenta el clima durante las acciones de un soldado es la clase *Weather*. Sus propiedades son estado, velocidad del viento, dirección del viento, afectación a la visión, temperatura, humedad. Estas propiedades son iguales en todas las regiones del mapa, un cambio en el estado del clima se representa a través de una modificación de estas propiedades.

2. Armas

Las armas que utilizan los soldados en la simulación se definen a traves de la clase *Weapon* con las siguientes propiedades:

- *name*: String con el nombre del arma que funciona como identificador de esta dentro de las armas en el inventario de un soldado. Un soldado no puede tener dos armas con el mismo nombre.
- *weight*: Valor entero que representa el peso que suma el arma a la capacidad de carga de un soldado.
- *w effective precision*: Valor entre 0 y 1 que representa el valor óptimo de precisión del arma.

- *max range precision*: Valor entre 0 y 1 que representa la precisión del arma fuera de su rango óptimo.
- *w effective range*: Valor entero que representa el rango (en metros) en que tiene efecto la precisión efectiva del arma.
- *w max range*: Valor entero que representa el rango (en metros) máximo que alcanza un disparo del arma.
- *damage*: Valor que representa el daño a la salud que ocasiona cada disparo individual del arma.
- *fire rate*: Valor entero que indica la cantidad de disparo que puede realizar el arma en una acción.
- *ammunition capacity*: Valor entero que indica la cantidad de munición máxima que puede tener equipada el arma.
- *current ammo*: Valor que indica la cantidad de munición actualmente equipada en el arma.

2.1. Acciones del arma

El arma solo tiene definida una acción fundamental, la acción de disparar a través del método *fire* que determina la probabilidad de acertar en base a los siguientes parámetros:

- Precisión del soldado.
- Precisión del arma.
- Parámetro de ocultamieno del objetivo.

La precisión del arma está influenciada por la distancia entre el arma y el objetivo, obtiene su valor óptimo si la distancia es menor o igual que el rango efectivo del arma, en caso de que el valor de la distancia se encuentre entre el límite del rango y el efectivo y el rango máximo se emplea el segundo valor de precisión del arma, orientado a ser menos efectivo.

La precisión del soldado puede ser altera según la postura del este y la afinidad con el arma.

El ocultamiento del objetivo se ve afectado por un mayor número de factores. Además de su valor de base se toma en cuenta la postura del objetivo, el camuflaje de la casilla del mapa en que se encuentra, el parámetro de afectación de la visión del clima y si el objetivo tiene un objeto con el cual cubrirse.

Por uĺtimo cada soldado tiene también una probabilidad de ocacionar un golpe crf́ico por el doble de daño. Esta probabilidad se aplica individualmente a cada disparo en la acción.

3. Soldados

La clase *Soldier* representa los soldados, que son agentes de la simulación. De cada soldado posee las siguientes propiedades: *id* único, su vida total, su vida actual, su rango de visión, su precisión, velocidad de movimiento, probabilidad de golpe crítico, afinidad con diferentes armas, orientación, postura, carga máxima, ocultamiento, equipo al que pertenece, armas en el inventario y arma equipada.

- *id*: Valor entero que representa un identificador del soldado en la simulación.

- *health*: Valor entero que indica la vida máxima del soldado.
- *current health*: Valor entero que indica la vida actual del soldado.
- *vision range*: Valor entero que representa el rango (en metros) de la visión del soldado.
- *precision*: Valor entre 0 y 1 que representa la precision base de un soldado. Esta se ve afectada luego por diferentes factores durante la acción de disparo.
- *move speed*: Valor entero que indica la velocidad de movimiento del soldado. Este valor se ve afectado luego por la restricción de movimiento del terreno.
- *crit chance*: Valor entre 0 y 1 que indica la probalidad del soldado de ocacionar un daño crítico con un disparo.
- *weapon affinities*: Diccionario que usa el nombre de un arma como llave y contiene un valor entre 1 y 2 que representa un bono a la precisión del soldado con ciertas armas de su preferencia.
- *stance*: String de 3 posibles valores *crouching*, *lying*, *standing* que representan las diferentes posturas que puede tomar un soldado en una casilla. Estás afectan la precision y el ocultamiento del soldado.
- *max load*: Valor que indica la capacidad máxima de carga del soldado.
- *concealment*: Valor entre 0 y 1 que representa el ocultamiento del soldado, que influye que lo detecten y la probabilidad de que diparos enemigos fallen.
- *team*: Valor entero que corresponde al identificador de la facción a la que pertenece el soldado.
- *weapons*: Lista con las armas en el inventario del soldado.
- *equipped weapon*: Referencia al arma que el soldado tiene equipada actualmente.
- *weapon ammo*: Diccionario que toma como llaves el nombre de un arma y contiene un valor que representa la munición de repuesto para cada arma en el inventario del soldado.
- *melee damage*: Valor númerico que representa el daño que realiza en soldado en un ataque cuerpo a cuerpo.

El soldado tiene definidos también una serie de acciones capaces de realizar como moverse a otra casilla, disparar, recargar, cambiar de arma y cambiar de postura. También es capaz de analizar su entorno como buscar soldados enemigos en su campo de visión, buscar aliados cercanos y detectar objetos cercanos.

La acción de moverse a otra casilla está afectada por la velocidad de movimiento del soldado y las restricciones de movimiento de las casillas del camino que se encuentre hacie el objetivo. El camino se obtiene a través de un algoritmo A^* que busca el camino óptimo, definiendo óptimo como el camino cuya suma de restricciones sea la menor.

Las acciones de detectar objetos o soldados utilizan un algoritmo de detección de colisiones entre caja y segmento. Si una casilla en la linea de visión entre el soldado y el objetivo a detectar, está ocupada por un objeto, no se detecta el objetivo.

Las acciones de cambio de postura afectan la precisión y ocultamiento del soldado según la postura que adopte.

3.1. Métodos complementarios de la clase Soldier

La clase soldado implementa una serie de métodos complementarios que son usados constantemente a lo largo de la simulación para apoyar y adquirir el conocimiento necesario para realizar algunas de sus acciones.

Métodos de detección: Los métodos de detección implementabos se basan en la simulación del sentido de visión de un soldado. Se considera el área de visión de un soldado como un cuadrado compuesto por casillas del mapa con lados de longitud igual al doble del rango de visión del soldado más una unidad, tomando como centro del cuadrado la casilla en la que se encuentra el soldado.

Para determinar si el soldado es capaz de ver el objeto o soldado en una casilla dentro de su área de visión se comprueba que no exista un obstáculo que obstruya la visión del soldado en la línea de visión. Para lograr esto se utiliza un algoritmo de detección de colisiones basado en considerar una casilla como un *Axis Aligned Bounding Box* (*AABB*) y determinar si existe una colisión con la línea de visión del soldado representada por un vector. Se considera que la visión del soldado está obstruida si existe un objeto en alguna de la casillas con las que colisiona su línea de visión hacia el objetivo a detectar.

Cuando se intentar detectar soldados enemigos se toma en cuenta una probabilidad de fallo basada en los párametros de ocultamiento del soldado objetivo y los factores que influyen sobre él. Los métodos de detección se aplican para soldados enemigos, soldados aliados y objetos en el mapa.

Método de toma de daño: Método que debilita al soldado según el daño que recibe. Si la salud actual de un soldado es inferior a la mitad de su salud máxima, la precisión y velocidad de movimiento de este pasan a ser menos efectivas.

3.2. Acciones del soldado

Las posibles acciones de los soldados marcan todo el desarrollo de la simulación, dado que son los únicos agentes que actúan sobre ella. Todos los resultados obtenibles partiendo un estado de la simulación dependen del conjunto de acciones que un soldado puede realizar, en ese conjunto están definidas las siguienes acciones:

Acciones de movimiento: Se define cualquier intento de traslado de un soldado desde una casilla A a una casilla B como una acción de movimiento. Existen 2 parámetros fundamentales que definen el resultado de una acción de movimiento hacia una casilla objetivo. La velocidad de movimiento del soldado que realiza la acción y la restricción de movimiento de las casillas por las que el soldado pase. La velocidad de movimiento consiste en un valor entero que representa de cierta forma la cantidad de movimiento que el soldado puede utilizar durante la acción. La restricción de movimiento de una casilla representa el costo de movimiento que gasta un soldado para moverse a una casilla adyacente. Antes de comenzar el traslado del soldado el primer objetivo a cumplir es obtener el camino más corto, o mejor dicho es menos costoso desde el punto de partida hasta el objetivo. Definimos este camino óptimo como la secuencia de casillas con inicio en el punto y partida y final en el objetivo cuya sumatoria de valores de restricción de movimiento sea la menor. Para hallar el camino óptim se utiliza un algoritmo un algoritmo de $A^\star$ sobre la matriz de terrenos utilizando como heurística las distancia euclidieanas entre las casillas y el objetivo. Una vez decido el camino el soldado comienza a avanzar descontado la restricción de movimiento de su capacidad de moverse cada vez que se mueve a una casilla adyacente. Si la

casilla en la que se encuentra supera su velocidad de movimiento restante el soldado se detiene y concluye su acción.

Acciones de disparo: La acciones de disparo se basan en el empleo de método *fire* del arma equipada descrito anteriormente. Recibiendo los parámetros de distancia, visibilad y ocultamiento del objetivo se llama al método *fire* que retorna el daño total realizado por el arma.
Acciones de combate cuerpo a cuerpo: Este tipo de acción se toma en cuenta cuando existe un enemigo en una casilla adyacente a la casilla que ocupa el soldado.

Acciones de cambio de postura: Conjunto de acciones orientado a alternar entre las 3 posibles posturas de un soldado, ajustando los parámetros de precisión y ocultamiento del soldado según la postura tomada.

Acciones de cambio de arma: Conjunto de posibles acciones que cambian el arma equipada de un soldado por una arma diferente que tenga en el inventario.

Acción de recargar: Acción sencilla que recarga las balas del arma equipada del soldado según las balas restantes en el inventario del soldado.

4. Facciones

Se definen las facciones como los diferentes equipos de soldados. La simulación se basa en el enfrentamiento de dos facciones. Las funciones de una facción consisten en listar sus soldados en el mapa además de mantener un registro de los logros de sus soldados durante la simulación.
Simulación
El objetivo a simular es el enfrentamiento entre dos facciones de soldados en el mapa definido hasta que uno de los bandos se quede sin soldados vivos. El desarrollo de la simulación se basa en rondas y turnos, definiendo una ronda como el tiempo que demoran todos los soldados de la simulación en realizar una acción. Un soldado no puede tomar una acción 2 veces en la misma ronda. Luego las 2 facciones que se enfrentan se alternan los turnos, un turno consiste en la realización de una acción por parte de un soldado de la facción. Si todos los soldados de la facción que le toque moverse ya se movieron esta ronda pero faltan soldados por moverse, la facción pasa el turno sin realizar ninguna acción.

5. Estados de la simulación

Una herramienta que definirá el transcurso de la simulación son los estados de esta. Definimos un estado de la simulación a través de la clase *State*. Una instancia de esta clase consiste en una serie de componentes que describen la situación en que se encuentra la simulación con todo detalle. Estos componentes son los siguientes:

- *soldier variables*: Diccionario que usa como llave el *id* de un soldado y contiene una tupla de valores que caracterizan al soldado en ese momento concreto de la simulación. Estos valores son: cantidad enemigos en rango de visión, cantidad aliados en rango, cantidad enemigos en rango efectivo, cantidad enemigos dentro del rango máxmimo, cadencia de fuego, munición actual equipada, capacidad de munición del arma equipada, rango efectivo del arma, daño del arma, ocultamiento, vida actual y precisión.

- *soldier str variables*: Diccionario que usa como llave el *id* de un soldado y contiene una tupla de *strings* para describir los siguientes valores: postura del soldado, si se encuentra junto a un objeto, nombre del arma equipada.
- *soldier extra variables*: Diccionario que usa como llave el *id* de un soldado y contiene una tupla de valores númericos que representan el rango de visión, la velocidad de movimiento, la probabilidad de crítico, la carga máxima y el daño cuerpo a cuerpo.
- *soldier positions*: Diccionario que usa como llave el *id* de un soldado y contiene la coordena de la casilla del mapa en la que se encuentra.
- *soldier reverse positions*: Diccionario que usa como llave una coordenada y contiene el soldado que se encuenre en ella, en caso de existir un soldado en esa casilla en ese estado de la simulación.
- *soldiers in map*: Diccionario que usa como llave el *id* de un soldado y contiene una referencia a la instancia del soldado.
- *alive soldiers*: Diccionario que usa como llave el *id* de una facción y contiene la cantidad de soldados vivos actualmente que pertenecen a esta.
- *team variables moved*: Diccionario que usa como llave el *id* de una facción y contiene la cantidad de soldados de esa facción que ya se han movido esta ronda.
- *soldier ammo per weapon*: Diccionario que usa como llave el *id* de un soldado y contiene un segundo diccionario que usa como llave el nombre del arma y contiene la cantidad de munición restantes en el inventario del soldado.
- *soldier weapons current ammo*: Diccionario que usa como llave el *id* de un soldado y contiene un segundo diccionario que usa como llave el nombre del arma y contiene la cantidad de munición actualmente equipada en las armas inventario del soldado.
- *soldier weapons*: Diccionario que usa como llave el *id* de un soldado y contiene un segundo diccionario que usa como llave el nombre del arma y contiene una referencia a la instancia del arma.
- *soldier moved*: Diccionario que usa como llave el *id* de un soldado y contiene un *bool* que indica si ya se movió esta ronda.
- *soldier died*: Diccionario que usa como llave el *id* de un soldado y contiene un *bool* que indica si el soldado está muerto en este estado de la simulación.

La para manejar la simulación se define clase *SimulationManager* con el objetivo de controlar los turnos y rondas de la simulación.

SimulationManager se encarga de crear el estado inicial de la simulación basado en las características iniciales de los soldados, el mapa y las posiciones de los soldados en el mapa. Maneja los resultados de las acciones y la evaluación de los estados, tema que se tratará más adelante.

6. Manejo de acciones

Para simular las acciones descritas anteriormentes se define la clase *ActionManager* con el objetivo de manejar los parámetros necesarios para la ejecución de una acción y generar el

nuevo estado de la simulación que resultante. Debido a la naturaleza de la simulación existen diferentes posibles estados resultantes de realizar una acción concreta partiendo de un estado de la simulación.

Como una herramienta para poder indentificar las mejores acciones a tomar durante un estado de la simulación, el *ActionManager* define una manera de llevar a cabo una acción concreta sin realizar cambios "físicos.en la simulación, este método se basa en revertir estos cambios al estado incial y quedarse solamente con el estado de la simulación resultante. Un estado de la simulación tiene toda la información necesaria para realizar cualquier acción definida.

7. Comportamiento de los soldados

Durante un turno de la simulación se requieren conocer todas las posibles acciones que los soldados de la facción pueden realizar, con ese objetivo se define la clase *ActionBuilder* que se encarga de recoger todas estas posibles acciones y los parámetros necesearios para llevarse a cabo cada una. Las posibles acciones de un soldado se dividen en acciones de disparo a los enemigo en rango de su arma equipada, ya sea efectivo o no, acciones de cambio de arma, cambio de postura, recargas de arma y acciones de movimiento. En esta últimas se consideran como opciones principales movimiento que te acerquen a un soldado en su campo de visión, moverte hacia un objeto cercano para buscar más probabilidades de supervivencia y un movimiento que siempre consideran es simplemente dirigirse hacia el centro del mapa.

El comportamiento de un soldado, está afectado principalmente por los intereses de la facción a la que pertenece. Un soldado solo puede definir acciones a tomar basado en su conocimiento individual de su entorno, pero la decisión sobre que acción realizará el soldado la toma la facción en su turno correspondiente.

7.1. ¿Cómo se encuentra la mejor acción a tomar?

La acción que más beneficie a la facción que corresponde su turno se busca aponyándose en un algoritmo de búsqueda adversarial *MinMax*.

Basándose en el sistema de turnos de la simulación el algoritmo el algoritmo utiliza simulaciones de la posibles acciones a tomar, alternando los turnos de las facciones hasta explorar todas las combinaciones de acciones llegando a una profundidad definida previamente. Utilizando un método para evaluar los estados que resulten de simular estas series de acciones se obtiene la acción que más beneficia a la facción que corresponde actuar y que al mismo tiempo permita beneficiarse lo menor posible a la facción enemiga.

Debido a la naturaleza de la simulación, el resultado obtenido puede no ser definitivo, la simulación de una acción tiene diferentes posibles estados como resultado pero el algoritmo solo evalúa el resultado obtenido de una simulación concreta. Esto abre paso a que se el caso de que aunque una acción A tenga en sus posibles resultados una media de evaluación más beneficiosa para la facción que una acción B si el resultado de la acción B fue mejor que el de la acción A en el momento de evaluación, se tomará la acción B. Aunque luego durante la ejecución real"de la acción no se obtenga este resultado.

7.2. Acciones definidas por un usuario

A los usuarios se les brinda la posibilidad de definir una acción para que un soldado realize. Estás acciones se vinculan a los soldados que el usuario desee. Para incorporar estás acciones al sistema se definieron unos métodos genéricos de generación de estados de la simulación. No

existe diferencia en la función que realizan este tipo de acciones a lo largo de la simulación. Estas son consideradas también por el algoritmo de *MinMax* con la ayuda de los métodos genéricos de creación de nuevos estados a través de los cambios que estas acciones ocasionan sobre las entidades de la simulación, dichos cambios también se revierten durante el *MinMax* al igual que las acciones predefinidas. Después de la ejecución de una de estas acciones se comprueba que no existan valores ilegales en las propiedades de las entidades de la simulación, como por ejemplo un solado con precisión de valor 50. En caso de ser detectado un valor ilegal se detiene la simulación. Las instrucciones sobre como definir una acción se brindarán en el *Manual de Usuario*.

8. ¿Cómo evaluar un estado?

Para ser capaces de evaluar que tan beneficioso es una estado para una facción se define una heurística manejada por la clase *HeuristicManager* que consiste en una serie de valores asignados a diferentes aspectos de la simulación. Estos valores son los siguientes:

- *damage hvalue*: Valor asignado al daño que es capaz de ocasionar un soldado. Toma como referencia las características del arma equipada y la precisión y afinidad del soldado con el arma. Un mayor valor de este parámetro probablemente cree una tendencia a mantener a los soldados de la facción con las mejores condiciones antes de un combate. Tomando medida como el cambio a la que considere mejor arma, recargar con frecuencia, etc.
- *allies hvalue*: Valor asignado a la cantidad de aliados cerca del soldado. Un mayor valor de este parámetro puede crear una tendencia a los soldados de la facción a andar en grupos.
- *enemies in range hvalue*: Valor asignado a la cantidad de enemigos en el rango efectivo del arma equipada del soldado.
- *enemies in sight hvalue*: Valor asignado a la cantidad de enemigos en el rango de visión del soldado. Este valor se evalúa de forma negativa. Puede ser compensado por los enemigos dentro del rango del arma.
- *low ammo hvalue*: Valor asignado a la cantidad de munición restante en el arma equipada. Este valor inlfuye la tendencia a mantener el arma cargada.
- *concealment hvalue*: Valor asignado al ocultamiento del soldado. Un mayor valor de este parámetro puede crear una tendencia a los soldados a priorizar la supervivencia. Puede lograrlo con diferentes tipos de acciones como cambiar de postura, moverse a zonas con mayor camuflaje, etc.
- *remaining health hvalue*: Valor asignado a la cantidad de salud restante del soldado.
- *dead soldier hvalue*: Valor asignado a la cantidad de soldados enemigos muertos. Aunque matar soldados enemigos sea el objetivo fundamental de una facción un valor bajo en este parámetro puede crear tendencias a priorizar acciones relacionadas con otros aspectos.
- *damage dealt hvalue*: Valor asignado a la cantidad de daño realizado a los soldados de la facción enemiga. Al igual que el parámetro anterior este valor influye la agresividad de una facción.

Estas heurísticas están vinculadas a una facción. Durante el algoritmo de *MinMax* se evalúan los estados en base a los criterios de la heurística de la facción a la que le corresponde el turno.

Se evalúan sus movimientos según los estados resultados más beneficiosos. A la hora de evaluar estados resultados de una acción enemiga se utiliza el mismo método lo cual permite que cada facción tenga su propia interpretación de que acciones enemigas resultan más problemáticas.

Esto permite influenciar los comportamientos de las 2 facciones de forma independiente. Permitiendo que surjan nuevas estrategias basadas en el enfrentamiento de los intereses de ambas facciones.

Esto permite influenciar los comportamientos de las 2 facciones de forma independiente. Permitiendo que surjan nuevas estrategias basadas en el enfrentamiento de los intereses de ambas facciones.

9. Compilación

Para el manejo del sistema se implementó un lenguaje de dominio específico (DSL, por sus siglas en inglés Domain Specific Language) con el nombre de BSL.

9.1. Tokenizacion

Para poder realizar la transpilación del lenguaje de dominio específico a python, se necesita primeramente determinar a partir de un texto válido las palabras, números y símbolos en orden de aparición para luego ser analizados en el proceso de parsing. Para ello se creó un módulo dedicado al proceso de tokenización de una cadena de texto, con lo cual se obtendrá el conjunto de 'tokens' determinados por las características de la gramática establecida. Dicho módulo se divide en 4 submódulos:

- **Token:**
  Dentro del submódulo Token se encuentra la definición del objeto Token el cual cuenta con los atributos:

  - type: representa el tipo de token construido. Se definen 7 tipos de tokens: `Unknown`, `Number`, `Text`, `Keyword`, `Identifier`, `Symbol` y `EOF` (por sus siglas en ingles `End Of File`)
  - value: representa el valor del Token creado, es decir, la expresión literal `tokenizada`
  - lexeme: representa la unidad mínima con significado léxico sin morfemas gramaticales, que puede ser identificado dentro del DSL.

  Podremos encontrar además el conjunto de `TokenValues` que determinan el lexema de un token a partir de que se identifique su valor y tipo.

- **Lexical Analyser:**
  El submódulo `Lexical Analyser` contiene las clases y métodos que nos permitirán e análisis y tokenización de una cadena de textos. Para ello se definen dos clases:

  - **Token Reader**: Esta clase contiene métodos ,atributos que nos auxilian en el manejo del texto suministrado; es la clase que contiene el texto a tokenizar. Dentro de esta podemos encontrar:

    -¿ **método peek**: que nos permitirá obtener el caractér en la posición actual(el texto se irá leyendo caractér a caractér) siempre y cuando la posición sea válida

Listing 1: *Método peek*

```
        if (self.pos < 0 or self.pos >= len(self.code)):
            return
        st = self.code[self.pos]
        return st
```

-¿ **método Read Any**: nos permitirá obtener el caractér en la posición actual del 'stream' y avanzar una posición(siempre y cuando sea válido).

Listing 2: *Método read_any*

```
        self.read_any()
        return True
    return False


def read_until(self, end, allowLB, text):
    text[0] = ""
    while not self.match(end):
        if not allowLB and (self.eof() or self.eol()):
```

-¿ **Read Number**: nos permitirá definir si a partir del caractér en la posición actual se puede conformar una cadena de caracteres que constituyan un valor numérico, para ello se va leyendo del stream en tanto los caracteres sean numéricos. En caso de que encontremos un '' procederemos a determinar la parte decimal del mismo, siguiendo el mismo procedimiento. En caso de hallarse un elemento no numérico se determinará que la cadena de caracteres leída no es de tipo numérica.

Listing 3: *Método read_number*

```
def read_number(self, number):
    token = self.peek()
    number[0] = ""
    while not self.eol() and str.isnumeric(self.peek()):
        number[0] += self.read_any()
    if number[0] != '' and (not self.eol() and self.match('.')):
        number[0] += '.'
        while not self.eol() and str.isdigit(self.peek()):
            number[0] += self.read_any()
    if len(number[0]) == 0:
        return False

    while not self.eol() and str.isalnum(self.peek()):
        number[0] += self.read_any()
    return len(number[0]) > 0
```

-¿ **Read Id**: nos permitirá leer una cadena de caracteres que comienza por un caractér que pertenece al alfabeto, seguido por un conjunto de caracteres alfanuméricos.

Listing 4: *Método read_id*

```
def read_id(self, id):
    id[0] = ""
    while not self.eol() and self.is_valid_character(self.peek(), len(id[0])
        == 0):
        id[0] += self.read_any()
    return len(id) > 0
```

- **LexicalAnalyser**:
  Define la clase encargada de determinar los tokens a partir de un conjunto de diccionario de operadores definidos, palabras claves y agrupadores de texto.Dicha clase define los métodos que determinan si a partir del caractér en la posición actual del 'TokenReader' se puede formar un token que pertenezca a alguna de las categorías definidas(texto, número, símbolo) para ello cuenta con métodos para el análisis y categorización:

  -¿ **match_text**: comprueba si a partir del caractér actual se puede formar una cadena de texto agrupado entre los agrupadores de texto definido, en cuyo caso se define el token correspondiente.

Listing 5: *Método match_text*

```
def match_text(self, stream, tokens: List, errors):
    for start in sorted(self.texts.keys(), key=lambda start: len(start),
        reverse=True):
        text = [""]
        if stream.match(start):
            if not stream.read_until(self.texts.get(start),
                self.allowLB.get(start), text):
                errors.append(CompilingError(stream.location,
                    ErrorCode.expected, self.comments.get(start)))
            tokens.append(Token('SS', 'SS', TokenType.Symbol,
                stream.get_codelocation))
            tokens.append(Token("STRING", text, TokenType.Text,
                stream.get_codelocation))
            tokens.append(Token('SE', 'SE', TokenType.Symbol,
                stream.get_codelocation))
            return True
    return False
```

  -¿ **match_text**: comprueba si a partir del caractér actual se puede formar un símbolo definido en el diccionario de símbolos definidos en el lenguaje.

Listing 6: *Método match_symbol*

```
def match_symbol(self, stream, tokens: List):
    for op in sorted(self.operators.keys(), key=lambda op: len(op),
        reverse=True):
        if stream.match(op):
            tokens.append(Token(self.operators.get(op), op,
                TokenType.Symbol, stream.get_codelocation))
            return True
    return False
```

-¿ **get_tokens**: En este método construimos el conjunto de tokens a partir del texto. Primero comprobamos si el caractér actual es un espacio en blanco, en cuyo caso lo ignoramos.Luego comprobamos si se obtiene un comentario a partir del caractér actual.En caso contrario comprobamos si se obtiene una cadena de texto a partir del caractér actual. Comprobamos para el caso en que constituya un valor numérico o un símbolo, en dicho orden y por último comprobamos si es un identificador, es decir una cadena de caracteres alfanuméricos, teniendo en cuenta si constituyen una palabra clave definida en el respectivo diccionario, en cuyo caso se obtiene un token bajo esa categoría. Si a partir del caractér actual no se pudo categorizar una cadena de caracteres valida, entonces el sistema lo determina como token desconocido, para lo cual se generará un error. Cada token se guarda en una lista que será lo que se obtenga a partir de este método.

Listing 7: *Método match_symbol*

```
def get_tokens(self, file_name, code, errors):
    tokens = []
    stream = token_reader(file_name, code)
    char = stream.peek()

    while not stream.eof():
        element = [""]
        char = stream.peek()

        if stream.read_blank():
            continue

        elif self.match_comment(stream, errors):
            continue

        elif self.match_text(stream, tokens, errors):
            continue

        elif stream.read_number(element):
            number = 0
            if not element[0].replace('.', '', 1).isdigit() and
                stream.pos > 0:
                errors.Add(CompilingError(stream.get_codelocation(),
                    ErrorCode.invalid, "Number format"))
            tokens.append(Token("NUMBER", element[0], TokenType.Number,
                                stream.get_codelocation()))
```

```
                continue

            elif self.match_symbol(stream, tokens):
                continue

            elif stream.read_id(element):
                if self.keywordsDic.get(element[0]) is not None:
                    tokens.append(Token(self.keywordsDic.get(element[0]),
                        element[0], TokenType.Keyword,
                                    stream.get_codelocation()))
                else:
                    tokens.append(Token('Identifier', element[0],
                        TokenType.Identifier, stream.get_codelocation()))
                continue

            unknown_str = stream.read_any()
            errors.Add(CompilingError(stream.get_codelocation(),
                ErrorCode.unknown, unknown_str))
        return tokens
```

- **TokenManager**: En este se define la clase del mismo nombre. Esta es la clase que contiene la lista de tokens obtenidos en el análisis léxico, y contiene los métodos auxiliares para manejar el consumo de tokens en el proceso de parsing.

- **Lexer**: En este se define la clase del mismo nombre. Esta clase es la encargada de definir la instancia del LexicalAnalyzer y registrar las palabras claves, símbolos y agrupadores de texto definidos en el lenguaje. Contiene el método con el que a partir del nombre del archivo y el texto que contiene se construye una instancia de TokenManager con los tokens obtenidos por el analizador.

Listing 8: *Método match_symbol*

```
def get_token_manager(self, filename, code):
    errors = []
    tokens = self.analyser.get_tokens(filename, code, errors)
    for error in errors:
        self.output_info.add_error(error)
    return Token_Manager(tokens)
```

9.2. Parser

El parser implementado es el LR(1) canónico, el cual es un analizador sintáctico LR (k) para $k = 1$, es decir, con un único terminal de búsqueda anticipada. El atributo especial de este analizador es que cualquier gramática LR (k) con $k > 1$ se puede transformar en una gramática LR (1). Para la implementación de este se definieron varias clases, entre ellas la clase `LR1Item` que representa la definición de Item LR(1) e Item SLR, para este último se permite la no entrada del parámetro `lookahead`.

La clase `LR1Item` tiene como atributos una producción, la posición del punto y el terminal lookahead. La posición del punto indica los símbolos que han sido recorridos y los que no, los que han sido recorridos son los símbolos cuyas posiciones son menores que la posición del punto y los que no son los símbolos cuyas posiciones son mayor o igual que la del punto.

A continuación se muestra la implementación de esta clase:

```
class LR1Item:
   def __init__(self, production: Production, dot_index: int, lookahead: Terminal = None):
   self._repr = ''
   self.production = production
   self.dot_index = dot_index
   self.lookahead = lookahead
   self._repr = f"{self.production.head} -> "
   self._repr += " ".join(str(self.production.symbols[i]) for i in range(self.dot_index))
   self._repr += " . "
   self._repr += " ".join(str(self.production.symbols[i]) for i in
       range(self.dot_index,len(self.production.symbols)))
   self._repr += f", {self.lookahead}"

   def __repr__(self) -> str:
      return self._repr

   def get_symbol_at_dot(self) -> Symbol:
      if self.dot_index < len(self.production.symbols):
      return self.production.symbols[self.dot_index]
      return None

   def __eq__(self, o):
      if isinstance(o, LR1Item):
      return self._repr == o._repr
      return False

   def __hash__(self):
      return hash(self.__repr__())
```

Otra de las clases implementadas es la clase `State` que representa un estado del autómata LR(1). El constructor de esta clase recibe como parámetro una lista de items LR1, a partir de los cuales el método `build` de la propia clase construirá el estado inicial. Otro de los métodos implementados en esta es el `set_go_to`, el cual tiene como función calcular todas las transiciones a partir de un conjunto de items y un símbolo.

Para la representación del autómata se implementó una clase con el mismo nombre. El constructor de esta clase recibe como parámetro una gramática. Esta clase al instanciarse crea una lista de estados como atributo de la clase, para llegar a construir esta lista primero se extendiende la gramática mediante el método `extended_grammar` de la propia clase el cual añade una nueva producción con el símbolo inicial como cabeza `''S''` el cual pasará a ser el nuevo comienzo de la gramática. Luego se obtiene una lista de no terminales de la gramática y por cada una de las producciones de este se generan los items iniciales, luego a partir de este, el estado inicial, y del estado inicial los próximos estados para cual se simula una cola utilizando los slices de Python partiendo del estado inicial.

Para el manejo de las clases anteriores y creación de las tablas `action_table` y `go_to_table` se

implementó la clase `LR1Table`. El constructor de esta clase recibe como parámetro una gramática a partir de la cual serán construidas las tablas mencionadas. El método `build_table` perteneciente a esta clase es el encargado de la construcción de las tablas, estas están representadas por una lista de diccionarios, donde cada posición *i* de la lista corresponde a un estado de la clase `Autómata`. Por cada uno de estados son creados dos diccionarios locales, uno de estos almacenará las acciones a realizar y número de estado próximo, dado un símbolo, la acción `SHIFT` se indicará por el caracter inicial de esta acción si el símbolo revisado es un terminal, en caso contrario (No terminal), este es añadido como llave del otro diccionario `go_to`, y el valor correspondiente será el número del estado de la proxima transición.

Luego se procede a analizar los lookaheads, y para esto se define un diccionario de Terminal como llave y lista de items LR1 como valor, luego se chequea si el lookahead está contenido en el diccionario mencionado, en caso de no encontrarse, este es añadido como llave al diccionario de acciones y es asignado la acción `REDUCE` representada por su caracter inicial `''R''` y la producción a la q se deberá reducir.

En caso de ser detectado el Terminal `$` y que la cabeza de la producción sea `S` (No terminal inicial) entonces se almacena, el terminal como llave y string `OK` como valor.

Para fines de optimización el resultado del método `build_table` se registra en un fichero .json, una vez estos almacenados en el directorio del proyecto, son detectados en un nuevo programa y se evita la reconstrucción de dichas tablas cargándolos desde el directorio.

9.3. Chequeo semántico y generación de código de python:

Dentro del lenguaje, todos los tipos son predefinidos y construidos antes de comenzar el proceso de parsing. Para ello se deifnió una clase específica `Type` que cuenta con los métodos y atributos que lo definen, entendiéndose que las clases predefinidas también se categorizan como tipos, y sus métodos y atributos definidos en la instancia de `Type`. Luego es necesario definir el entorno o contexto de ejecución en donde se definen las variables, los tipos y las funciones, además de contener los métodos auxiliares que nos permitan determinar las características de dichos elementos. Para ello se definió la clase context que es la encargada de definir dicho entorno de ejecución, entendiédose que la creación de nuevos contextos de ejecución se hacen a partir de un contexto `global` o `root` el cual no se deriva de ningún otro contexto y es quien realmente contiene los tipos definidos dentro del lenguaje. A partir de estas dos clases y el AST, se procede a realizar la traducción del código del DSL a lenguaje python y finalizar así el proceso de transpilación. Para ello recorremos el AST siguiendo el patrón `visitor` comenzando por su raíz que es el nodo que contiene el conjunto de `statements` definidos en el código al que se le aplicó el proceso de tokenización y `parsing`, visitando cada uno de ellos y generando el código a partir del generado por el nodo y los nodos expresiones que lo conforman, realizándose el chequeo semántico en la medida en que son visitados, resolviéndose sus tipos a partir del contexto de ejecución y si cumplen con las reglas propias de la expresión o del `statement`, para lo cual cada nodo posee una función 'check_semantic' encargada de dicha verificación. En cada `visita` se agrega la traducción a una variable `string` tomándose en cuenta la rígida indentación específica del lenguaje Python.

Listing 9: *Ejemplo de nodo de ast con su función de chequeo semántico definida*

```
def check_semantic(self, context: Context):
    if self.type == '':
        self.left.check_semantic(context)
        self.right.check_semantic(context)
        self.type = 'Number' if self.op not in context.logical_ops and self.op not in
            ['and', 'or'] else 'Bool'
```

```
typeL = self.left.type
typeR = self.right.type
if self.op == '+' and ((typeL != 'Number' and typeL != 'String') or (typeR !=
    'Number' and typeR != 'String')):
    raise Exception('Operation ' + ' is only valid for Number or String types"')
elif typeL != typeR:
    raise Exception(f'Invalid expression for operator "{self.op}"')

elif self.op in ['and', 'or'] and self.type != typeL:
    raise Exception(f'Invalid expression for operator "{self.op}"')

elif self.op in ['and', 'or'] and self.type != typeL:
    raise Exception(f'Invalid expression for operator "{self.op}"')
```

Listing 10: *Método de traducción de un nodo binario, en donde puede verse que no se ordena el chequeo semántico de este,dado que es una expresión*

```
@visitor(BinaryExpression)
def transpile(self, node: BinaryExpression, context: Context):
    left = self.translate(node.left, context)
    right = self.translate(node.right, context)
```

10. Reglas del lenguaje:

- El programa constituye un conjunto de instrucciones.
- Las instrucciones permitidas son definir funciones, la instrucción `return`, instrucciones `if-else`, ciclos `while`, instrucciones `break` y `continue`, así como la declaración de variables y su asignación.
- Toda instrucción debe terminar su declaración en `';'`, exceptuando una instrucción `if` a la que le siga una instrucción `else`.
- Definimos una función con la palabra clave `'def'` seguido del tipo de retorno, nombre de la función, argumentos y el tipo correspondiente entre parentésis, luego dos puntos y luego el conjunto de instrucciones de la función entre llaves.
- La instrucción `if` se compone por la palabra clave `'if'` seguida de una expresión condicional, dos putos y luego el cuerpo de instrucciones entre llaves.
- La instruccion `else` debe estar precedida por una instrucción `if`, comenzando por la palabra clave `else`, seguida de dos puntos, y el cuerpo de instrucciones entre llaves.
- La instrucción `while` se define de igual manera que la instrucción `if`.
- Para declarar una variable primeramente decimos su tipo, seguido de un identificador, seguido del operador de asignación `'='`, y luego la expresión a la que se quiere asignar, teniendo en cuenta que debe ser del mismo tipo que el declarado. En cuanto a la asignación, se escribe de igual forma obviando el tipo y teniendo en cuenta debe existir una variable con el mismo identificador asociado para ser asignada. La cantidad de asignaciones permitidas a

una misma variable son ilimitadas, siempre y cuando se cumplan las reglas semánticas de la asignación.

- Los tipos definidos cuentan con métodos y atributos propios que pueden ser accedidos agregando un punto y luego el nombre de la instrucción o el llamado al método correspondiente. Para modificar algún atributo se debe comprobar si el tipo posee algún método con dicho propósito, en caso contrario, será de solo lectura.

- Las listas pueden ser indexadas a través de describir la posición a la que se quiere acceder entre corchetes, contiguo a la lista correspondiente. Se debe tener en cuenta que a través de esta expresión se puede obtener el valor asociado a la posición, pero no se puede modificar.

- Las instrucciones `if-ese`, declaración de funciones, y ciclos `while` generan un nuevo contexto de ejecución para el bloque de instrucciones que contienen.

- Los argumentos de la función deben tener nombres diferentes

- Las funciones se consideran consideran como variables dentro del contexto, por tanto, no puede existir una variable y una función con el mismo nombre.

- Si se desea declarar una lista de un tipo determinado se hará: `'List_NombreDelTipo'`.

- Toda función y variable debe estar definida antes de ser llamada.

- Las operaciones aritméticas solo se permiten entre elementos de tipo `'Number'`, y en el caso del tipo específico `'String'`, solo si el operador es el de suma. En el caso de las operaciones booleanas, los miembros deben ser del mismo tipo y en el caso específico de `'and'`, `'or'` y `'not'` deben ser booleanos.

11. Tipos definidos:

{Soldier, Map, Terrain, Weather, Number, Bool, String, AuxActions$^{(*)}$,List } .

(*) El tipo AuxActions solo contiene un conjunto de métodos auxiliares usados en la definición de acciones extras.

12. Operadores definidos:

12.0.1. Aritméticos:

$\{ +, -, *, /, \hat{} \}$

12.0.2. Lógicos:

$\{ ==, <=, >=, <, >, and, or, not\}$

13. Funciones predefinidas:

- **Soldier.get_map:**
  - Argumentos:
  - Tipo de los argumentos:
  - Tipo de retorno: Map
  - Descripción: Retorna el mapa asociado a la simulación, siempre y cuando se haya posicionado el soldado en el.
- **Soldier.set_weapons:**
  - Argumentos: 'weapons', 'magazines'
  - Tipo de los argumentos:'List Weapon', 'List Number'
  - Tipo de retorno:'Void'
  - Descripción: Asigna 'weapons' como armas al soldado, con el número de cargadores equivalente en la misma posición en magazines.
- **Soldier.add_extra_action:**
  - Argumentos: 'action'
  - Tipo de los argumentos:'fuction'
  - Tipo de retorno:'Void'
  - Descripción: Agrega al conjunto de acciones posibles a realizar por el soldado, la función action, teniendo en cuenta que esta función debe cumplir con las características de tener como únicos argumentos a un soldado y el mapa en dicho orden, en caso contrario se generará un error en tiempo de ejecución. Notar que el mapa de la simulación no debe ser modificado dentro de dicha función, para un comportamiento correcto de esta.
- **Soldier.remove_extra_action:**
  - Argumentos:'index'

- Tipo de los argumentos:'Number'
- Tipo de retorno:'Void'
- Descripción: Remueve de la lista de acciones posibles, aquella que se encuentre en la posición index.

- **Soldier.is_ally:**
  - Argumentos: 'soldier'
  - Tipo de retorno: 'Soldier'
  - Descripción: Determina si 'soldier' pertenece a la misma facción.

- **AuxActions.detect_enemies_within_eff_range:**
  - Argumentos: 'soldier', 'map'
  - Tipo de los argumentos: 'Soldier', 'Map'
  - Tipo de retorno: 'List Soldier'
  - Descripción: Devuelve una lista con los enemigos en el campo de visión dentro del rango de eficiencia del arma.

- **AuxActions.detect_enemies_within_max_range:**
  - Argumentos: 'soldier', 'map'
  - Tipo de los argumentos: 'Soldier', 'Map'
  - Tipo de retorno:'List Soldier'
  - Descripción: Devuelve una lista con los enemigos en el campo de visión dentro del rango máximo del arma.

- **AuxActions.move**:
  - Argumentos:'soldier', 'position'

- Tipo de los argumentos: 'Soldier', 'List Number'
- Tipo de retorno:'Void'
- Descripción: Realiza el movimiento de un soldado a la posición 'position'.

- **AuxActions.shoot**:
  - Argumentos:'soldierA', 'soldierB'
  - Tipo de los argumentos: 'Soldier', 'Soldier'
  - Tipo de retorno:'Void'
  - Descripción: Realiza el disparo por parte del 'soldierA' con objetivo 'soldierB'.

- **List.append**:
  - Argumentos: 'a'
  - Tipo de los argumentos ''
  - Tipo de retorno: 'Void'
  - Descripción: Agrega el elemento a al final de la lista. El tipo de este debe coincidir con el de la lista.

- **len**:
  - Argumentos:'list'
  - Tipo de los argumentos: 'List'
  - Tipo de retorno: 'Number'
  - Descripción: Devuelve el tamaño de la lista list

- **int**:
  - Argumentos: 'number'

- Tipo de los argumentos: 'Number'
- Tipo de retorno:'Number'
- Descripción: Elimina la parte decimal de number.

- **str**:
  - Argumentos:'number'
  - Tipo de los argumentos: 'Number'
  - Tipo de retorno:'String'
  - Descripción: Convierte number a 'String'.

- **print**:
  - Argumentos: 'text'
  - Tipo de los argumentos: 'String'
  - Tipo de retorno: 'Void'
  - Descripción: Imprime la cadena de texto 'text' en la consola.

- **run**:
  - Argumentos: 'map', 'weather', 'soldiers', 'ia_max_depth', 'heuristic'
  - Tipo de los argumentos: 'Map', 'Weather', 'List Soldier', 'Number', 'List HeuristicManager'
  - Tipo de retorno: Void
  - Descripción: Función que da comienzo a la ejecución de la simulación a partir de sus argumentos.

Luego existen métodos asociados a los tipos que nos permiten modificar sus atributos, siempre siendo llamados con el comando: Tipo.set_nombre_del_atributo(elemento_del_mismo_tipo). Aquellos atributos modificables son:

- Soldier:
  current_health, vision_range, precision, move_speed, crit_chance, max_load, concealment, melee_damage.

- Weapon:
  name, weight, w_effective_range, w_max_range, effective_range_precision, max_range_precision, damage, fire_rate, current_ammo.