Universidad de La Habana
Facultad de Matemática y Computación

Battle-sim: Simulador de enfrentamientos bélicos

Proyecto de Inteligencia Artificial, Compilación y Simulación

Equipo de desarrollo:
Rocio Ortiz Gancedo - C311
Carlos Toledo Silva - C311
Ariel A. Triana Pérez - C311

2022

Índice general

1. Introducción 1
2. Core, centro de procesamiento de la simulación de enfrentamientos bélico 2
   - 2.1. El mapa 2
   - 2.2. Los objetos 4
   - 2.3. El simulador 8
3. Battle Script, el lenguaje de dominio específico para ejecutar el proyecto 10
   - 3.1. Reglas del lenguaje 10
   - 3.2. Gramática 10
   - 3.3. Compilador 14
   - 3.4. Tokenizador 14
   - 3.5. Parser 19
   - 3.6. Análisis Semántico 22
   - 3.7. Generación de código 24
4. Ejemplos 25
5. Reflejo de las asignaturas en el proyecto 28
   - 5.1. Simulación 28
   - 5.2. Compilación 28
   - 5.3. Inteligencia Artificial 28
6. Aplicaciones 29
7. Conclusiones 29
8. Recomendaciones 30
9. Referencias 31

1. Introducción

A lo largo de la historia, los conflictos bélicos han estado fuertemente ligados al desarrollo de la humanidad. Ejemplo de esto es la "caza de cabezas" y otras prácticas de lucha endémica en las sociedades agrarias en la prehistoria en las que los hombres se disputaban la tierra aprovechable. También, los maoríes de Nueva Zelanda se caracterizaron por la construcción de fortificaciones que aumentaban el prestigio de cada grupo en las luchas. Otro ejemplo es la Guerra de las Galias, un conflicto militar librado entre el procónsul romano Julio César y las tribus galas entre el año 58 a. C. y 51 a. C. En el curso de las mismas la República Romana sometió a la Galia, extenso país que llegaba desde el Mediterráneo hasta el Canal de la Mancha.

La Guerra de los Cien Años (Guerre de Cent Ans en francés, Hundred Years' War en inglés) fue un prolongado conflicto armado que duró en realidad 116 años (1337-1453) entre los reyes de Francia y los de Inglaterra. Esta guerra fue de raíz feudal, pues su propósito no era otro que dirimir quién controlaría las enormes posesiones de los monarcas ingleses en territorios franceses desde 1154. Más recientes, se tienen los ejemplos de las Guerras Mundiales en el siglo pasado. Con el paso del tiempo, los hombres fueron evolucionando, y así también lo hicieron los objetivos de los conflictos bélicos, los armamentos y estrategias utilizados en estos conflictos.

El objetivo de este proyecto es el desarrollo de un programa que permita la simulación de diferentes batallas que se hayan producido en un pasado distante, en épocas más recientes e incluso simular batallas futuristas o con elementos de fantasía. Además, poder simular batallas entre diferentes épocas, por ejemplo enfrentar 20 soldados armados con AK-47 contra 100 soldados armados con espadas y escudos, en un terreno montañoso.

El proyecto consiste en dos grandes módulos:

- **Core:** comprende todo lo relacionado con la simulación de los enfrentamientos bélicos: la definición y generación de mapas y terrenos, el funcionamiento de las unidades básicas y el funcionamiento del simulador.
- **Battle Script:** comprende la definición de la gramática del lenguaje, y la implementación del compilador, entiéndase el tokenizador, el parser, el análisis semántico, la generación de código, entre otras funcionalidades.

2. Core, centro de procesamiento de la simulación de enfrentamientos bélicos

Como se plantea en el capítulo anterior, se quiere desarrollar un programa que permita la simulación de enfrentamientos bélicos entre dos o más bandos en un terreno determinado.

Se fijaron las siguientes reglas para cada una de las simulaciones:

1. La existencia de un mapa o terreno donde se produce el enfrentamiento. Este tendrá propiedades que serán modificables como las dimensiones, el relieve, la hidrografía, etc. La representación más abstracta de un mapa o terreno es una matriz de alturas.
2. La simulación ocurre por turnos. Durante un turno cada unidad hace una y solo una acción ya sea atacar, moverse o mantener la posición. El objetivo de las unidades de un bando es destruir las unidades de los bandos enemigos. Cada unidad funciona de manera independiente.

2.1. El mapa

El mapa consiste en una matriz bidimensional de m filas y n columnas. Cada objeto de la matriz es un objeto `Cell`. Un objeto `Cell` representa cada una de las casillas que conforman el mapa y tiene los siguientes atributos:

- `passable`: Valor entre 0 y 10 que indica cuan accesible es una celda. Las unidades tienden a buscar las celdas que tengan este parámetro lo más alto posible, pues mientras mayor sea este valor, pueden hacer ataques más poderosos. Una interpretación de esta parámetro es cuan estratégica es una determinada posición.
- `row`: Este parámetro es un número entero que indica la fila en la que se encuentra la celda.
- `col`: Este parámetro es un número entero que indica la columna en que se encuentra la celda.
- `height`: Este parámetro es un número entre 0 y 1 que indica la altura de la casilla. En dependencia de este parámetro la casilla será terrestre o naval.
- `bs_object`: Este parámetro hace referencia al objeto que se encuentra en la casilla. Si en la casilla no se encuentra ningún objeto entonces este parámetro es `None`.

El mapa para una simulación se crea instanciando una clase `LandMap` con los siguientes argumentos:

- Número de filas m.
- Número de columnas n.
- Un array bidimensional de m filas por n columnas tal que cada posición i, j del array es un número entre 0 y 10 que indica cuan accesible es la celda i, j.
- Un array bidimensional de m filas por n columnas tal que cada posición i, j del array es un número entre 0 y 1 que indica la altura de celda i, j
- Un número entre 0 y 1 que indica el nivel del mar. Todas las celdas cuya altura sea menor a este número serán consideradas como celdas navales y todas las celdas superior a este número serán consideradas terrestres.

Generación aleatoria de mapas La representación abstracta de un mapa es una matriz de alturas, o sea, una matriz de valores en el intervalo $[0, 1]$, donde la noción de nivel del mar es 0.45, o sea, todo valor $x > 0{,}45$ es una elevación y todo valor $x < 0{,}45$ es una depresión cubierta por agua. Es importante poder generar un mapa de alturas de forma aleatoria con un porcentaje de relieve determinado por el usuario que sea lo más realista posible. Por tal razón, queda totalmente descartado el enfoque que va por generar una matriz M donde $M[i, j]$ es un número totalmente al azar, pues se obtendrían matrices de ruido similares a la estática en las señales de televisión, además se violaría la restricción del porciento de elevaciones.

Luego de una consulta bibliográfica, se encontraron algunos procedimientos para la generación de mapas de alturas muy interesantes:

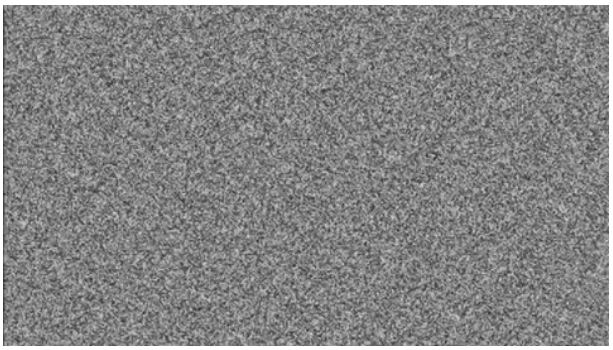

Figura 1. Estática en la señal de TV

- **Algoritmo de Perlin Noise:** debe su nombre a su creador, Ken Perlin, que durante el rodaje de la película Tron creó el algoritmo con el fin de crear texturas procedimentales para los efectos generados por el ordenador. La idea tras el algoritmo es generar una nueva matriz de dimensión menor a la requerida y superponerla a la requerida escalando de forma que encajen perfectamente. A cada esquina de cada celda de la nueva matriz se le asigna un vector gradiente pseudoaleatorio. Además se determinan los valores offset que representan la posición de cada punto de la matriz original dentro de la celda de la nueva matriz. Luego, para cada esquina de la celda que encierra un punto se determina el vector que va desde ella hasta el punto y se hace el producto escalar de ese vector con el vector gradiente. Se realiza una interpolación lineal entre todos los resultados de los productos escalares y este es el valor del punto. El comportamiento de este algoritmo no puede ser modificado pero aún así fue de gran significación en la época y hoy día.
- **Algoritmo Voronoi:** consiste en dividir el espacio en diagramas de Voronoi, que son, simplemente, particiones de un plano en regiones basadas en la distancia a ciertos puntos específicos del plano. Ese conjunto específico de puntos se denomina "semillas, sitios o generadores". Se suelen especificar de antemano y para cada uno de estos sitios se genera una región del plano que consiste en todos los puntos que están más cerca de este sitio que de ningún otro.
- **Algoritmo de cortes:** tiene un comportamiento poco intuitivo pero interesante, se selecciona un corte en la matriz, se define como corte una linea recta que divida el plano o terreno en dos mitades, luego se decide elevar una de las mitades y bajar o no modificar las otras. Este mismo proceso se repite miles de veces y se obtienen resultados aceptables. El algoritmo es ineficiente, por la cantidad de iteraciones necesarias para obtener buenos resultados.
- **Algoritmos de erosión térmica e hídrica:** algoritmos que simulan los procesos naturales de erosión térmica e hídrica. En el caso de la térmica se fija un valor T que representa el valor a partir del cual se va a erosionar, y se recorre la matriz buscando los puntos que cumplen que la diferencia de altura con sus vecinos es superior a T. En dichos puntos el algoritmo resta valor de altura a sus vecinos cuya distancia sea mayor que T de forma que todas las distancias queden menor que T. En el caso de la erosión hídrica se define una matriz de agua y una matriz de sedimentos la cual representa un porciento de material que puede ser movido por la erosión. Y se realiza un procedimiento parecido a la erosión térmica.

Ahora, estos algoritmos generan mapas de alturas con apariencia realista, pero no son suficientes para cumplir las restricciones de relieve. Por tanto, nuestra propuesta es la implementación de un algoritmo evolutivo que genere un mapa de alturas realista y satisfaga la restricción anteriormente mencionada.

Los algoritmos evolutivos son estrategias de optimización y búsqueda de soluciones que toman como inspiración la evolución en distintos sistemas biológicos. La idea fundamental de estos algoritmos es mantener un conjunto de individuos que representan posibles soluciones del problema. Estos individuos se mezclan y compiten entre sí, siguiendo el principio de selección natural por el cual sólo los mejores

adaptados sobreviven al paso del tiempo. Esto redunda en una evolución hacia soluciones cada vez más aptas.

El algoritmo evolutivo en cuestión inicia con una población de individuos generados utilizando el algoritmo de Perlin Noise implementado en la biblioteca de Python bajo el nombre `perlin-noise`. Luego se realizan una cantidad de iteraciones seleccionando los individuos más adaptados, mezclándolos y mutándolos. Se considera individuo mejor adaptado aquel cuyo valor `fit` sea mayor. Este valor se calcula como sigue:

```
def fit_func(self, heightmap: HeightMap):
    count = sum(sum(heightmap.__map__ > self.__sea__))
    per = count / prod(self.__shape__)
    return per if per < self.percentage else 0
```

Esto es, se cuentan la cantidad de posiciones que su valor de altura supera el nivel del mar, y se calcula el porciento que representan estas del total, si este porciento es superior al porciento de la restricción del problema entonces se le asocia a dicho individuo el valor 0. Por tanto, el valor `fit` máximo que puede tener cualquier individuo es el porciento de la restricción y el menor el 0.

Luego, el proceso de selección de individuos se realiza utilizando el procedimiento de Selección competitiva o por torneos, se seleccionan 4 individuos de la población de forma aleatoria, y se realiza una competencia 2 a 2 seleccionando el mejor en cada pareja y estos son los individuos mejor adaptados. El proceso de mezcla de los individuos seleccionados se realiza utilizando una suma ponderada, con valores 0.95 y 0.05 respectivamente.

El proceso de mutación del individuo resultante de la mezcla es el siguiente:

1. Se genera una matriz temporal `tmp` que representará una matriz de suavizado para la mutación. Esta matriz es una matriz de alturas generada igualmente por el algoritmo de Perlin Noise.
2. Si el porciento de relieve de la restricción es superior a 0.4 se realiza el procedimiento siguiente:
   - La matriz `tmp` se multiplica por el escalar -0.3.
   - La matriz `tmp` se suma al individuo proveniente de la mezcla. Dando como resultado un individuo mutado donde de forma general se le baja altura a todas las posiciones.
3. Si el porciento de relieve de la restricción es inferior o igual a 0.4 entonces se realiza el siguiente procedimiento:
   - Se recorren todas las posiciones del individuo resultante de la mezcla y se mutan aquellas que están por encima del nivel mar. Se calcula el valor `w` que representa cuán por encima del nivel del mar está dicha posición.
   - A las posiciones que están por encima del nivel del mar se le resta el resultado de multiplicar el valor `w` por el resultado de la suma de la posición en el individuo mutado y la posición en `tmp`

El algoritmo evolutivo en cuestión tiene ciertos parámetros que definen las condiciones de paradas. Se definió la cantidad de iteraciones en 100 iteraciones. Se definió un nivel de tolerancia al porciento de relieve de la restricción en 0.03, de forma que cuando exista un individuo cuyo valor `fit` esté en el intervalo $[p - 0{,}03, p + 0{,}03]$, donde p es el valor de la restricción, el algoritmo se detiene.

2.2. Los objetos

Un objeto es toda entidad que se puede poner en el mapa; cada objeto ocupa una y solo una casilla del mapa. Estos tienen propiedades como el `id` que es el identificador único para cada objeto, los puntos de vida (`life_points`) que determinan el estado de un objeto y la defensa (`defense`) un parámetro que indica cuan resistente es un objeto a los daños que puede sufrir durante la simulación. Todos estos parámetros son números en el intervalo de números enteros $[1, 10]$. Cuando la vida de un objeto llegue a 0, este se destruye desapareciendo del mapa.

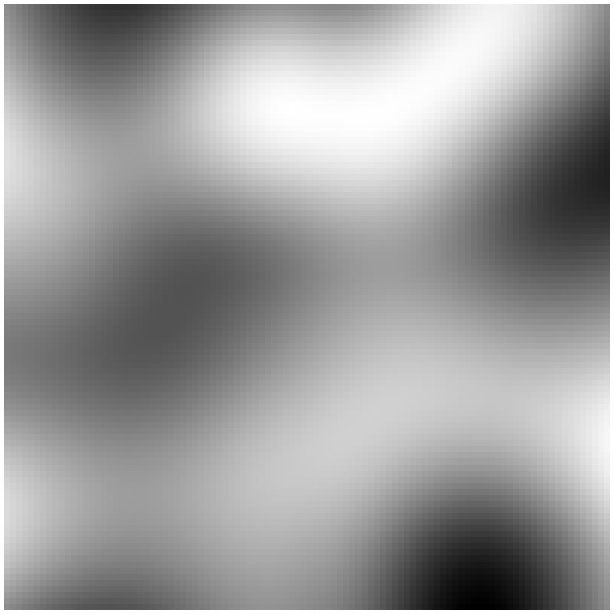

Figura 2. Mapa de alturas generado por el algoritmo evolutivo con porciento de relieve de 80 %

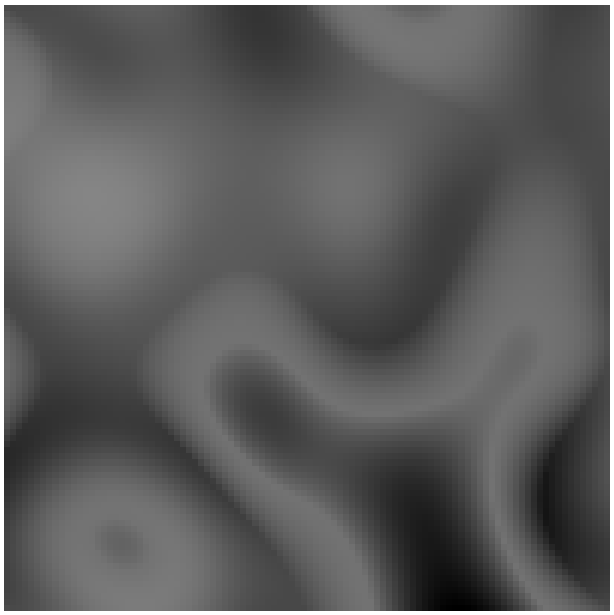

Figura 3. Mapa de alturas generado por el algoritmo evolutivo con porciento de relieve de 10 %

Los objetos se clasifican en dos tipos: unidades y objetos estáticos. Los objetos además tienen definidas dos funciones `put_in_cell` cuya función es colocar al objeto en alguna posición de un mapa. Si se intenta poner un objeto en una posición diferente a su tipo este automáticamente se destruye. La otra función `take_damage`, a partir de un ataque sufrido indica como se reducen los puntos de vida del objeto.

Objetos estáticos Los objetos estáticos se definen como objetos propios del ambiente. Estos no pertenecen a ningún bando y no pueden realizar acciones pero si pueden ser afectados por las acciones que realicen las unidades. Estos solo pueden ser puestos en celdas terrestres. Ejemplos de estos objetos pueden ser árboles, rocas, muros, etc.

Unidades Las unidades son los agentes de la simulación. Estas se dividen en dos tipos fundamentalmente: unidades terrestres (`LandUnit`) y unidades navales (`NavalUnit`). El objetivo de cada una de las unidades es destruir a las unidades enemigas (las que no pertenecen a su mismo bando), y para ello cada unidad podrá analizar parte del ambiente en el que se encuentra y tomar la decisión que sea más conveniente según sean las circunstancias.

Dado que el ambiente cambia constantemente, las unidades serán agentes casi puramente reactivos. Un agente puramente reactivo es un sistema computacional situado en un ambiente, el cual es capaz de percibir y responder de modo oportuno a los cambios que ocurran en el mismo con acciones autónomas y sin tomar en cuenta el pasado para lograr sus objetivos. En este caso las unidades son agentes casi puramente reactivos pues para el movimiento de las mismas a través del mapa se tiene en cuenta si una unidad ya estuvo en una posición, con el fin de estimular la exploración del mapa por parte de las unidades.

La arquitectura empleada para definir el comportamiento de las unidades es la Arquitectura de Brooks (de categorización o inclusión) la cual es una arquitectura de control surgida en los años 80 y 90 del siglo pasado. Esta en lugar de guiar el comportamiento mediante representaciones mentales simbólicas del mundo, acopla la información sensorial a la selección de la acción de manera interna. La arquitectura de Brooks tiene las siguientes características:

- La toma de decisión se realiza a través de un conjunto de comportamientos para lograr objetivos (reglas de la forma situación $\rightarrow$ acción).
- Las reglas pueden dispararse de manera simultánea por lo que debe existir un mecanismo para escoger entre ellas.

Dado esto se definió un sistema experto que actuará como la función del agente. Este será descrito posteriormente.

Propiedades de las unidades

Las unidades además de las descritas anteriormente que tienen todos los objetos cuentan con las siguientes propiedades:

- `side`: Una instancia de la clase `Side` que indica el bando al que pertenece la unidad
- `attack`: Valor entre 1 y 10 que marca la capacidad de causar daños a sus oponentes.
- `moral`: Valor entre 1 y 10 que marca la moral con la unidad encara la batalla. Cuanto mayor es ese valor más efectivos son sus ataques y sus defensas.
- `ofensive`: Valor entre 1 y 10 que indica cuan ofensiva es una unidad. Un valor alto la hace más ofensiva y un valor bajo la hace más defensiva.
- `min_range`: Valor entero entre 1 y 10 que indica el rango mínimo al que se debe encontrar un enemigo para que la unidad pueda atacarlo.
- `max_range`: Valor entero entre 1 y 10 que indica el rango máximo al que se debe encontrar un enemigo para que la unidad pueda atacarlo. Lógicamente el rango máximo debe ser mayor o igual que el rango mínimo.

- `radio`: Valor entero entre 1 y 9 que indica el número de casillas que son afectadas por un ataque de la unidad. Si es 1 se afecta solo a la casilla seleccionada para el ataque. Si es 9 se afectan la casilla seleccionada y las 8 adyacentes a esta. Si es $1 <$ `radio` < 9, entonces se toman como casillas afectadas la seleccionada y `radio` $- 1$ casillas aleatorias adyacentes a esta.
- `vision` : Valor entre 1 y 10 que indica la cantidad de celdas en una determinada dirección, que la unidad puede "ver" (saber que objetos están en dicha celda). La visión debe ser mayor o igual que el rango máximo pues una unidad no debería ser capaz de atacar una unidad que no pueda ver.
- `intelligence`: Valor entre 0 y 10 que indica la inteligencia de la unidad. Mientras más inteligente sea una unidad con mayor precisión puede calcular los atributos de sus enemigos.
- `recharge_turns`: Turnos que demora la unidad en recargar después de hacer un ataque. Mientras esté recargando la unidad no podrá atacar pero si puede moverse.
- `solidarity`: Valor booleano que indica si la unidad es solidaria o no.
- `movil`: Valor booleano que indica si la unidad puede desplazarse por el mapa.

Sistema experto A continuación se explicará el sistema experto implementado que actúa como función del agente la cual describe el comportamiento del agente durante un turno.

Lo primero que la función hace es buscar si existe un enemigo al que se pueda atacar. Para eso primero se comprueba si la unidad no está recargando, si lo está se reduce en uno los turnos que debe esperar para atacar, si no lo está se busca el mejor enemigo para atacar.

El mejor enemigo para atacar se determina de la siguiente forma:

Por cada casilla atacable (casilla que se encuentra a una distancia de la unidad entre su rango mínimo y su rango máximo), se chequea si en la casilla hay un enemigo. Si el radio de ataque es mayor que 1 y hay una unidad amiga cerca del enemigo que pudiera verse afectada por el ataque, se ignora este enemigo. Esto se definió así para evitar el daño ocasionado por fuego amigo.

Entonces si el enemigo no es ignorado debido a lo anterior, se calcula el costo de atacar al enemigo. Para dicho cálculo la unidad estima la vida y la defensa del enemigo y con estas estimaciones, y teniendo en cuenta el daño que le puede causar al enemigo, calcula cuantos turnos podría tomarle destruir a dicho enemigo. Mientras mayor sea la inteligencia de la unidad más precisas serán las estimaciones.

Luego de haber analizado todos los enemigos, el seleccionado por la unidad para atacarlo es aquel cuyo costo es menor. Si no se detecta ningún posible enemigo a atacar entonces la unidad no realiza un ataque en el turno.

Si la unidad no realiza un ataque, entonces, en caso de que sea móvil, chequea si se puede mover a alguna casilla adyacente a la que se encuentra. Esto se hace de la siguiente forma:

Se fija un costo en infinito. Luego por cada una de las celdas que la rodean en todos los puntos cardinales posibles (NW, N, NE, W, E, SW, S y SE), la unidad calcula el costo de moverse a dicha celda, y se queda con la celda cuyo costo es el menor. Luego si el menor costo detectado es menor a infinito la unidad se mueve a dicha celda en caso contrario la unidad mantiene su posición.

Ahora, ¿cómo calcular el costo que tiene para una unidad moverse de una celda a otra?

Si la celda nueva es intransitable (tiene el parámetro `passable` en 0), el tipo de la celda es diferente al tipo de la unidad o si en la celda hay algún objeto se devuelve infinito. Estos son todos los casos en los que la unidad no puede moverse a dicha celda. Para las unidades terrestres si la diferencia de alturas entre las dos celdas es muy grande (mayor a 0.3), tampoco pueden avanzar a dicha celda, retornándose infinito.

Entonces en un primer momento se fija el costo en 10 - `passable`/2, de esta manera se premia ir a celdas más transitables. A continuación se comprueba si esa celda está entre las que la unidad recuerda como ya visitadas. Si así es, el coste se incrementará en `passable`/3. Así logramos incentivar que las unidades visiten celdas no visitadas con anterioridad

Ahora se comprueba si la celda se encuentra en zona "amiga", es decir, si esa celda es adyacente a alguna celda en la que se encuentre algún compañero de su bando. Si así es y nuestra unidad protagonista es solidaria, el coste se reducirá a la mitad. Si es una zona amiga pero la unidad no es solidaria, el coste solo se reducirá dividiéndose por la raíz cuadrada de 2. De esta manera se incentiva que las unidades tiendan a permanecer en grupo y más cuanto más solidarias son.

Si la celda que se está estudiando es una celda en la que nuestra unidad tendrá al alcance un enemigo, su coste se reducirá en el valor del parámetro `ofensive` multiplicado por la inversa de la raíz cuadrada de la distancia mínima hasta el enemigo. Además la unidad observa las celdas cercanas a la celda que se está estudiando que estén dentro de su rango de visión. Si detecta que al moverse a dicha celda se encontrará en rango de algún enemigo el costo se aumenta en 1.1 por cada enemigo que pudiera atacar a la unidad. De esta manera se está diciendo que, cuanto más ofensiva sea nuestra unidad, más se aproximará al enemigo, aunque con cierta precaución.

Métodos de las unidades

A continuación se hace una breve explicación de las funciones implementadas a las unidades. Además de las ya mencionadas, implementadas en `BSObject`, un agente `BSUnit` tiene implementadas las siguientes funciones:

- `calculate_distance(self, cell1, cell2) -> int` con la que la unidad calcula la distancia entre dos celdas. La distancia se calcula como la cantidad mínima de celdas que tiene recorrer la unidad para ir de una celda a otra.
- `nearby_friend(self, cell) -> bool` con la que la unidad determina si moviéndose a la celda de entrada estará cerca de un compañero.
- `enemy_in_range(self, cell)` con la que la unidad determina si al moverse a esa celda, tiene algún enemigo al alcance.
- `in_range_of_enemy(self, cell) -> int` con la que la unidad determina la cantidad de enemigos que pudieran atacarla, de los que puede ver.
- `move_cost_calculate(self, cell, type) -> float` con la que la unidad determina el costo de moverse a una determinada celda.
- `enemy_cost_calculate(self, enemy) -> float` con la que el unidad calcula el costo de atacar a un enemigo determinado.
- `friend_in_danger(self, cell) -> bool` con la que la unidad detecta si atacando a una celda determinada, pone en peligro a una unidad aliada.
- `enemy_to_attack(self)` con la que la unidad determina, de ser posible, cual es el mejor enemigo para atacar.
- `take_damage(self, damage)` se redefine el método de la clase `BSOBject` para tener en cuenta la moral de la unidad.
- `attack_enemy(self, enemy)` que es la acción de atacar a un enemigo. Para darle mayor realismo a la simulación, se simula que las unidades además puedan fallar los ataques. En la precisión de un ataque influyen factores como la distancia entre la unidad que ataca y la unidad atacada y los objetos adyacentes a la unidad atacada que se interpongan entre ambas unidades.
- `move_to_cell(self, cell)` que es la acción de moverse a una determinada celda.
- `turn(self, type)` que es la función que permite a la unidad realizar un turno.

Las funciones `put_in_cell` y `turn` se redefinen en las clase `LandUnit` haciendo un llamado a la respectiva función de la clase padre tal que el valor del parámetro `type` es “earth”. De igual forma ocurre con las unidades navales (`NavalUnit`), pero el valor del parámetro `type` es “water”.

Ejemplos Si por ejemplo se quisiera definir una clase `Soldier` que es una unidad terrestre, simplemente a la hora de la definición de la misma se hace heredar a esta clase de la clase `LandUnit`. Si se quiere por ejemplo definir una unidad naval, por ejemplo `Boat`, se hace heredar a la misma de la clase `NavalUnit`.

2.3. El simulador

Los bandos Dentro de la simulación de batalla las unidades se agrupan por bandos que se enfrentarán durante la batalla. Dos unidades aliadas pertenecen al mismo bando y dos unidades enemigas pertenecen a bandos distintos.

Para este proceso se implementó la clase `Side` que consta de los siguientes atributos y métodos:

- Atributos
  - `id`: Valor numérico único que se le asocia al bando, se utiliza entre otras cosas para hallar el Hash
  - `name`: Nombre del bando
  - `units`: Unidades que se encuentran en el bando
  - `no_own_units_defeated`: Cantidad de unidades del bando que han sido derrotadas.
  - `no_enemy_units_defeated`: Cantidad de unidades enemigas que han sido derrotadas por unidades del bando.
- Métodos
  - `add_unit`: Este método permite agregar una unidad al bando.
  - `remove_unit`: Este método posibilita quitar una unidad del bando.
  - `get_units`: Devuelve todas las unidades del bando.
  - `__iter__`: Para iterar las unidades
  - `__eq__`: Comparar si dos bandos son el mismo bando
  - `__hash__`: Calcular el hash de un bando

Los eventos Para la simulación de la batalla se implementó la clase `Simulator` encargada de simular los eventos que ocurrirán durante el enfrentamiento. Se le llamará evento a la acción de alguna de las unidades activas en el terreno en un tiempo determinado, sabiendo que una unidad selecciona la acción a realizar de acuerdo al sistema experto implementado.

La clase `Simulator` consta de los siguientes atributos y métodos:

- Atributos
  - `earth_map`: Mapa del terreno donde ocurre la simulación
  - `sides`: Bandos que participan en la simulación
  - `units`: Unidades que participan en la simulación. Incluye a todas las unidades de todos los bandos
  - `time_beg`: Momento en que se empieza la simulación.
  - `interval`: Tamaño del intervalo que dura un turno
  - `turns`: Cantidad de turnos a simular.
  - `no_enemies`: Con este atributo se conoce si ya terminó la simulación, pues almacena si queda más de un bando con unidades vivas.
- Métodos
  - `event_is_pos`: Revisa si un evento es posible, principalmente comprueba que la unidad que realizará la acción está viva.
  - `get_events`: Toma todas las unidades que aún están vivas y les asigna un tiempo de forma aleatoria dentro del intervalo dado. Para seleccionar el tiempo de forma aleatoria se utilizó la distribución Beta que es una distribución de variable aleatoria continua, a diferencia de la uniforme, es más flexible gracias a los parámetros α y β, además de no tener problemas de pérdida de memoria como la exponencial. Una vez asociado un tiempo en el que cada unidad viva realizará una acción, estos pasan a ser los eventos a simular.
  - `simulator_by_turns`: Se van a sacar los eventos a partir de un intervalo dado y se ejecutarán por orden uno a uno según el tiempo que se le asignó cuando se hizo el evento.
  - `simulating_k_turns`: Se realizan los turnos, a partir del atributo `turns`, cada turno en un intervalo de tamaño `interval`

3. Battle Script, el lenguaje de dominio específico para ejecutar el proyecto

El usuario final del proyecto necesita un medio para específicar las condiciones del enfrentamiento a simular, entiéndase como son las unidades, el mapa, el movimiento de las unidades, etcétera. Para ello se implementó un lenguaje de dominio específico (DSL, por sus siglas en inglés *Domain Specific Language*) con el nombre de Battle Script, que permite la creación de nuevas unidades, la creación de obstáculos, la creación de mapas, la inserción de unidades y obstáculos en el mapa, la creación de bandos, y la ejecucción de la simulación.

3.1. Reglas del lenguaje

Reglas sintácticas:

- Un programa escrito en Battle Script es una secuencia de clases y a continuación una secuencia de instrucciones.
- Una clase se define con un nombre y el nombre de la clase de la que hereda (herencia simple), un costructor donde se definen los atributos de la clase y una secuencia o no de funciones.
- Las instrucciones permitidas son la definición de función, todas las variantes de if-elif-else, la definición de ciclos while, declaración y asignación de variables y las instrucciones return, break y continue.
- Una función se define con un tipo de retorno, un nombre, una serie de argumentos con sus respectivos tipos y un cuerpo que es una serie de instrucciones.
- Un ciclo while se define con una condición y una secuencia de instrucciones.
- Las instrucciones if y elif se definen con una condición y una secuencia de instrucciones, y además pueden tener a continuación una instrucción elif o una instrucción else. La instrucción else se define con una secuencia de instrucciones; y tanto esta como la instrucción elif deben ir precedidas de una instrucción if o elif.
- Las declaraciones se definen con un tipo, un nombre y una expresión.
- Las asignaciones se definen igual que las declaraciones lo que sin especificar un tipo.
- La definición de atributo es muy parecida a la declaración, pero solo se puede hacer en el constructor de una clase y además se usa la palabra clave self.
- Las expresiones pueden ser de varios tipos:
  - Expresiones atómicas como variables, números, llamado a funciones, etc.
  - Expresiones binarias que pueden ser lógicas u aritméticas.
  - Expresiones ternarias.
  - Llamado a atributos y métodos de una clase.
  - Combinaciones de las anteriores.

Reglas semánticas:

- Existe un contexto global.
- Las instrucciones definición de clase, definición de función, ciclo while y las instrucciones condicionales if - elif - else genera cada una su propio contexto.
- Una variable declarada en un contexto no puede volverse a declarar en el mismo contexto o en un contexto descendiente de este.
- Una función se considera como un caso particular de variable por lo que no pueden compartir nombre según lo planteado en el punto anterior.
- Dos argumentos en una función no pueden tener el mismo nombre.
- Toda función y variable deben haber sido definidas antes de ser usadas; y todos los tipos de estas (en el caso de la función el tipo de retorno y de los argumentos) deben haber sido creados. anteriormente (Salvo las funciones y tipos pre-definidos).

- Un llamado a función se debe hacer de tal forma que la cantidad y cada uno de los tipos de los argumentos con que se llama coincida con lo especificado en la definición de la función.
- Una variable puede ser asignada tantas veces como se desee, siempre que la asignación tenga el mismo tipo con que se declaró.
- Las operaciones lógicas y condicionales solo admiten objetos de tipo **bool**.
- Las operaciones aritméticas solo admiten números.

3.2. Gramática

Una gramática libre del contexto es una tupla $G = \langle T, N, S, P \rangle$ donde:

- T es un conjunto de símbolos terminales (el vocabulario).
- N es un conjunto de símbolos no terminales.
- $S \in N$ es el símbolo inicial.
- P es un conjunto de producciones de la forma $A \to \alpha$ donde $A \in N$ y $\alpha \in \{T \times N\}^+$ o $\alpha = \epsilon$.

Se diseñó la siguiente gramática para el lenguaje:

```
bs_file ->  classes '&' statements EOF
        |   EOF

classes -> class_def ';' classes
        |  class_def ';'

statements ->   statement ';' statements
            |   statement ';'

statement ->    func_def
            |   if_def
            |   while_def
            |   decl
            |   assign
            |   return_stat
            |   'break'
            |   'continue'
            |   expression


func_def ->     'function' return_type NAME '(' params ')' '->' '{' statements '}'
            |   'function' return_type NAME '(' ')' '->' '{' statements '}'

if_def ->   'if' expression '->' '{' statements '}' elif_def
        |   'if' expression '->' '{' statements '}' else_def
        |   'if' expression '->' '{' statements '}'

elif_def ->     'elif' expression '->' '{' statements '}' elif_def
            |   'elif' expression '->' '{' statements '}' else_def
            |   'elif' expression '->' '{' statements '}'

else_def -> 'else' '->' '{' statements '}'

class_def ->    'class' NAME 'is' NAME '->' '{'  constructor_def ';' functions '}'
        |       'class' NAME 'is' NAME '->' '{'  constructor_def ';' '}'
```

```
functions -> func_def ';' functions
           | func_def ';'

constructor_def -> 'constructor' '(' params ')' '->' '{' attributes '}'
             | 'constructor' '(' ')' '->' '{' attributes '}'
             | 'constructor' '(' ')' '->' '{' '}'


attributes -> attr_def ';'  attributes
            | attr_def ';'

attr_def ->  type 'self' '.' NAME '=' expression


while_def -> 'while' expression '->' '{' statements '}'

return_type ->  'void'
            |   type

type ->   'number'
      |   'bool'
      |   'List'
      |   NAME

assign -> NAME '=' expression

decl ->  type NAME '=' expression

return_stmt ->  'return' expression
            |   'return'

params ->   type NAME ',' params
        |  type NAME


expressions ->  expression ','  expressions
            |   expression

expression ->   disjunction 'if' disjunction 'else' expression
            |   disjunction

disjunction ->  conjunction 'or' disjunction
            | conjunction

conjunction ->  inversion 'and' conjunction
            |   inversion

inversion ->    'not' inversion
            |    comparision
```

```
comparision ->  sum 'eq' sum
            |   sum 'neq' sum
            |   sum 'lte' sum
            |   sum 'lt' sum
            |   sum 'gte' sum
            |   sum 'gt' sum
            |   sum


sum ->  sum '+' term
    |   sum '-' term
    |   term

term -> term '*' factor
    |   term '/' factor
    |   term '%' factor
    |   factor

factor ->   '+' factor
        |   '-' factor
        |   pow

pow ->  primary '^' factor
    |   primary

primary ->  primary '.' NAME
        |   primary '(' expressions ')'
        |   primary '(' ')'
        |   atom
        | '(' expression ')'

atom -> NAME
    |   'True'
    |   'False'
    |   'None'
    |   NUMBER
    |   list

list -> '[' expressions ']'
    |   '[' ']'
```

Se creó un submódulo de Python para implementar gramáticas de forma más fácil en las secciones siguientes del compilador. Se implementó la jerarquía de clases que se aprecia en la Fig 4.

Para inicializar una gramática es necesario tener creado todos los `NonTerminal` con sus producciones agregadas, y se le pasa el `NonTerminal` inicial al constructor de `Grammar`.

Luego de crear dicho submódulo se implementó la gramática en el archivo `bs_grammar.py` de forma que esté accesible para todo el compilador en la variable `GRAMMAR`.

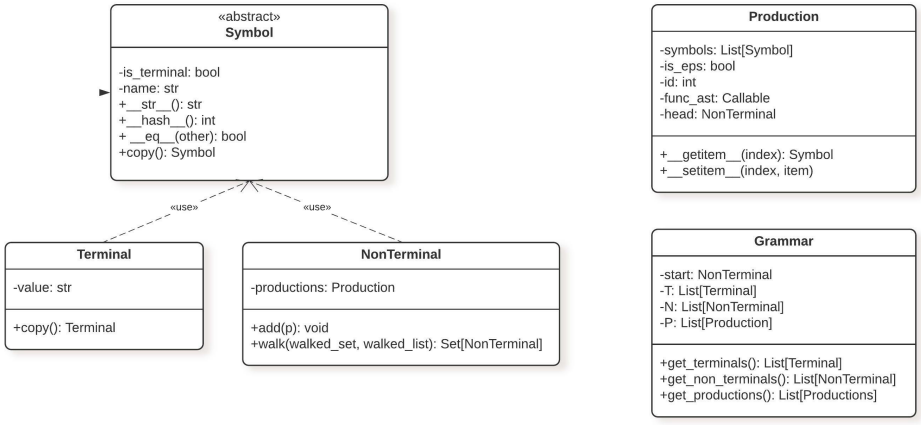

Figura 4. Diagrama de clases del módulo grammar

3.3. Compilador

El compilador implementado es un transpilador del lenguaje de dominio específico Battle Script hacia Python. Para ello se implementaron las siguientes etapas del proceso de compilación: tokenización, parsing, análisis semántico y generación de código.

3.4. Tokenizador

Un Tokenizador, comúnmente llamado Lexer, es un ente encargado de dividir la cadena de texto de entrada del compilador en tokens del alfabeto del lenguaje Battle Script, identificando el tipo del token y envíandolo a la siguiente etapa del proceso de compilación.

Para la implementación del tokenizador fue necesario un sistema de expresiones regulares, una clase para representar un token, así como su tipo y la definición de los tokens del lenguaje, además de la clase propia del tokenizador.

Sistema de expresiones regulares Una expresión regular es una definición recursiva de un lenguaje donde a es la expresión regular para $L(a) = \{a\}$ y ϵ es la expresión regular para $L(\epsilon) = \{\epsilon\}$. Si s y r son expresiones regulares entonces:

- $(s)|(r)$ es la expresión regular para la unión de lenguajes $L(s) \cup L(r)$
- $(s)(r)$ es la expresión regular para la concatenación de lenguajes $L(s)L(r)$
- $(s)*$ es la expresión regular para la clausura del lenguaje $L(s)* = \bigcup_{k=0}^{\infty} L(s)^k$

Se implementó un sistema de expresiones regulares que admite los siguientes operadores:

- | que hace la unión de dos expresiones regulares.
- La concatenación de expresiones regulares de la siguiente forma, si a y b son expresiones regulares entonces ab es la expresión de la concatenación.
- $*$ que hace la clausura del lenguaje que representa la expresión regular.
- ? que busca la coincidencia de la expresión regular una vez o ninguna.
- $+$ que busca la coincidencia de la expresión regular una o más veces.

- . que busca la coincidencia de ninguno o cualquier caracter.
- \ que permite la inclusión de los operadores anteriores en un expresión regular como un caracter.

La gramática para el lenguaje de las expresiones regulares que se implementó es la siguiente:

```
regex = exp

exp      = term '|' exp
         | term

term     = factor term
         | factor

factor   = primary '*'
         | primary '+'
         | primary '?'
         | primary

primary  = '(' exp ')'
         | '\' CHAR
         | CHAR
         | '.'
```

Una expresión regular en el sistema se implementó utilizando la clase `Regex`. Una expresión regular se construye con la cadena de texto que representa el patrón de la expresión regular. Este patrón se compila y se devuelve un Autómata Finito No Determinista (NFA) que se utiliza para los procesos de saber si una cadena pertenece al lenguaje representado por la expresión, o encontrar todas las coincidencias de la expresión en una cadena de texto.

```
class Regex:
        def __init__(self, pattern: str):
            self.pattern : str = pattern
            self.nfa : NFA = self.compile()

        def compile(self) -> NFA:
            lex: Lexer = Lexer(self.pattern)
            parser : Parser = Parser(lex)
            tokens : List[Token] = parser()
            handler: Handler = Handler()

            nfa_stack : List[NFA]= []

            for t in tokens:
                handler.handlers[t.name](t, nfa_stack)

            if len(nfa_stack) == 1:
                return nfa_stack.pop()
            raise Exception("Bad regex!")

        def match(self, string: str) -> bool:
            return self.nfa.match(string)

        def find_all(self, string: str) -> List[Match]:
            return self.nfa.find_all(string)
```

El proceso de compilación de una expresión regular es bastante sencillo, se tiene un tokenizador que recorre la cadena y cada vez que encuentra un operador o un caracter devuelve el token correspondiente, si encuentra el operador de escape (\) devuelve un token de tipo `CHAR` con el caracter siguiente. El proceso de Parsing consiste en recibir el conjunto de tokens provenientes del tokenizador y llamar al método `exp` que este parsea el no terminal del mismo nombre de la gramática y va parseando el resto de no terminales según corresponda, o sea, se implementó un Parsing Recursivo Descendente.

Luego del proceso de parsing y verificar que la secuencia de tokens es correcta, se recorre la secuencia de tokens y haciendo uso de la clase `Handler` se construye el autómata de la expresión regular.

La clase `Handler` es la encargada de dado un conjunto de tokens construir el autómata correspondiente, para ello implementa un método para cada operador de la gramática, estos métodos saben construir el autómata que resulta de aplicar la operación a uno, dos o más autómatas según el operador. La clase implementa el algoritmo Thompson's construction para la construcción del autómata.

- **Reconcimiento de un caracter:** el método de la clase `Handler` encargado de construir dicho autómata construye el estado inicial `s0` y el final `s1` y añade la transición con el caracter en cuestión, como se muestra en la figura siguiente:

Figura 5. Autómata finito no determinista para el reconocimiento de un caracter

- **Operación de Unión:** el método encargado de construir el autómata solo debe los estados iniciales y finales, además añadir ϵ-transiciones desde el estado incial hacia los autómatas y desde los autómatas hacia el estado final.

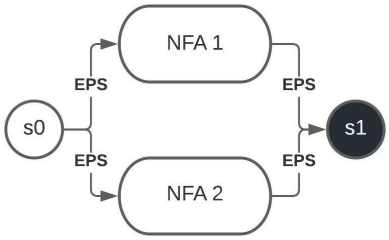

Figura 6. Autómata finito no determinista para el operador unión

- **Operación de Concatenación:** el método encargado de construir el autómata solo debe añadir una ϵ-transición desde el estado final del primer autómata hacia el estado incial del segundo autómata.
- **Operación `*`**: el método encargado de construir el autómata solo debe los estados iniciales y finales, además añadir ϵ-transiciones desde el estado incial hacia el autómata y desde el autómata hacia el estado final, así como una desde `s0` hacia `s1` para reconocer la no aparición del patrón, y una transición ϵ desde el estado final del autómata hacia el inicial para reconocer las repeticiones.

Figura 7. Autómata finito no determinista para el operador concatenación

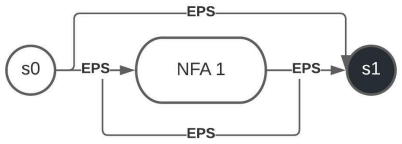

Figura 8. Autómata finito no determinista para el operador *

- **Operación +**: el método encargado de construir el autómata solo debe los estados iniciales y finales, además añadir ϵ-transiciones desde el estado incial hacia el autómata y desde el autómata hacia el estado final, y una transición ϵ desde el estado final del autómata hacia el inicial para reconocer las repeticiones.

Figura 9. Autómata finito no determinista para el operador +

- **Operación .**: el método encargado de construir dicho autómata construye los estados inciales y finales ya añade un transición para cada caracter y añade un ϵ-transición para reconocer la ocurrencia de ningún caracter.
- **Operación ?**: el método encargado de construir dicho autómata debe añadir una ϵ-transición desde el estado incial del autómata hacia el estado final del autómata para reconocer la no aparición del patrón.

Luego que se construye el autómata para una expresión regular el proceso de compilación ha terminado.

Entonces, ¿cómo saber si una cadena de texto pertence al alfabeto representado por una expresión regular? Para ello existen varios enfoques entre ellos están:

- **Enfoque inocente o tonto:** utilizando backtracking y revisando los operadores y la cadena. Este enfoque es bien costoso y no aprovecha las potencialidades de los autómatas anteriormente construidos.
- **Enfoque DFA:** se construye un Autómata Finito Determinista (DFA) a partir del no determinista anteriormente construido y se hace una pasada sobre él. Este enfoque es correcto, pero implica un procesamiento extra para construir el DFA.
- **Enfoque NFA:** este es el enfoque implementado en el proyecto en el método `match` de una `Regex`, que consiste en hacer una pasada sobre el autómata no determinista y revisar si coincide la

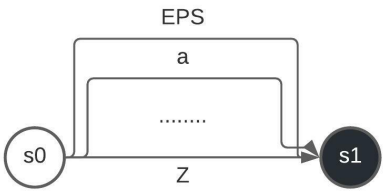

Figura 10. Autómata finito no determinista para el operador .

Figura 11. Autómata finito no determinista para el operador ?

cadena, pero el no determinismo trae un problema consigo, es que en un estado `x` no se sabe con total seguridad que transición aplicar. Para ello, la propuesta es ejecutar las transiciones a la vez y ver si por alguna se llega al estado final. Esto hace que el tiempo de reconocimiento sea lineal respecto al tamaño de la cadena de entrada.

Entonces, ¿cómo encontrar todas las coincidencias de un patrón dentro de una cadena de texto?. Para ello se modificó el método anterior y cada vez que no se podía continuar porque venía un caracter desconocido para la expresión regular se volvía al estado inicial. Además en cada posición se revisa si se ha llegado al estado final, se devuelve la coincidencia y se vuelve al estado inicial. Este procedimiento se implementó en método `find_all` de las `Regex`. Este procedimiento es lineal con respecto al tamaño de la cadena.

Luego de implementado el sistema de expresiones regulares se implementó la clase `Token` y el enum `TokenType`.

Token y TokenType Un token en el proyecto se representa con la clase `Token` que tiene la siguiente implementación. La propiedad `regex` almacena la expresión regular que coincide con el token, la propiedad `name` representa el nombre del tipo de token, la propiedad `lexeme` almacena la cadena de texto que se extrajo de la cadena de entrada como token, es decir, si se tiene un token `t` entonces `t.regex.match(t.lexeme)` es verdadero. Además, un token almacena donde comienza y termina `lexeme` en la cadena de entrada.

```
class Token:
        @propertyclass Token:
        @property
        def regex(self) -> Regex:
            return self.type.value[0]
```

```
        @property
        def name(self) -> str:
            return self.type.value[1]

        def __init__(self, token_type: TokenType, lexeme: str, start: int, end: int):
            self.type: TokenType = token_type
            self.lexeme: str = lexeme
            self.start: int = start
            self.end: int = end


        @property
        def name(self) -> str:
            return self.type.value[1]

        def __init__(self, token_type: TokenType, lexeme: str, start: int, end: int):
            self.type: TokenType = token_type
            self.lexeme: str = lexeme
            self.start: int = start
            self.end: int = end
```

El tipo de los tokens se definió utilizando el enum `TokenType` que representa la expresión regular que lo representa y el nombre del token, por ejemplo: para el token que representa la palabra reservada `eq` el `TokenType` correspondiente es `Eq = (Regex("eq"), "eq")`

Algoritmo del tokenizador El algoritmo del tokenizador sigue la siguiente idea general, se le asocia a cada token de la gramática un nivel de precedencia que representa cuán relevante es un token sobre otro. Por ejemplo si se tienen los siguientes: `Token((Regex("eq"), "eq"), "eq", 1, 2)` y `Token((Regex("(a|b|....|Z)+"), "NAME"), 'eq', 1, 2)` donde el primer token tiene precedencia 1 y el segundo tiene 2, entonces el token más relevante es el primero. Entonces, se tiene una lista `tokens` y se recorre la lista de tokens de la gramática y a cada uno se le piden todas coincidencias en la cadena de entrada. Luego se recorren las posiciones `i` de la cadena de entrada, y se añade a `tokens` aquel token que comience en `i` y tenga menor precedencia.

```
class Tokenizer:
    def __call__(self, bs_content_file: str) -> Iterable[Token]:
        tokens: List[Token] = []

        matches = {}
        for token_def in TOKENS:
            matches[token_def] = token_def.type.value[0].find_all(bs_content_file)

        i = 0
        while i < len(bs_content_file):
            token_in_i: List[Tuple[TokenDefinition, Match]] = []

            # Get all matches
            for k, v in matches.items():
                for t in v:
                    if t.start == i:
                        token_in_i.append((k, t))
```

```
        # Get match with highest precedence
        if len(token_in_i):
            token_in_i = sorted(token_in_i, key=lambda tup: tup[0].precendece)
            token = Token(token_in_i[0][0], \
                          token_in_i[0][1].value,\
                          token_in_i[0][1].start,\
                          token_in_i[0][1].end)
            tokens.append(token)
            i = token.end

        i += 1
    return deque(tokens)
```

Este algoritmo tiene complejidad temporal $O(n * m)$ donde m es la cantidad de tokens, y n el tamaño de la cadena de entrada.

3.5. Parser

Construcción del Parser El parser implementado es un parser $LR(1)$ canónico, el cual es un parser $LR(k)$ con $k = 1$, es decir, con un solo terminal de anticipación. Una característica especial de este parser es que cualquier gramática $LR(k)$ con $k > 1$ puede transformarse en una gramática $LR(1)$. Para la implementación se definieron las clases `Item` e `ItemLR1` que representan la definición de Item LR(0) e Item LR(1).

La clase `Item` tiene como atributos una producción y un índice tal que los símbolos en las posiciones menores que el índice son los símbolos de la producción que se han detectado y los símbolos en las posiciones mayores o igual al índice son los símbolos de la producción que faltan por ver.

Como se dijo anteriormente la clase `ItemmLR1` es la definición de un item LR(1). Obsérvese que esta hereda de `Item` y además se agregan como atributos adicionales el `lookahead`, un string que representa al item y un valor de hash calculado a partir de este string. Además se implementan las funciones de `__hash__` y `__eq__`.

Además se implementó la clase `State` que representa un estado del autómata LR(1). Un estado tiene los siguientes atributos:

- `kernel` que es una lista de items LR(1) que son el kernel del estado.
- `string` un cadena que representa al estado, que no es más que la concatenación de los items kernel del estado delimitados por el caracter |.
- `items` que es el conjunto de items del estado (de ahí la necesidad de implementar la función `__hash__` para la clase `ItemLR1`).
- `nexts` es un diccionario tal que las llaves son los símbolos que pueden venir a continuación dado que el parser se encuentra en dicho estado; y el valor asociado al símbolo es el estado al que se mueve el parser si se detecta dicho símbolo.
- `expected_symbols` es un diccionario tal que las llaves son los símbolos que pueden venir a continuación dado que el parser se encuentra en dicho estado; y el valor asociado al símbolo es un conjunto de items del estado que tienen a dicho símbolo en la posición que marca su índice respectivo.
- `number` que es el número correspondiente al estado.

Esta clase tiene definida, entre otras, tres funciones interesantes: `add_item(self, item: ItemLR1)`, `build(self, initial_items)` y `go_to(self, sym: Symbol, states_dict, state_list, q: deque, initial_items)`. La primera añade un item LR(1) al conjunto de items del estado. La segunda se utiliza para construir el estado a partir de los items kernel del estado y los items iniciales de la gramática(los que tienen su índice en 0). La idea detrás de este segundo método es crear una cola a partir de los items kernel, luego mientras la cola no este vacía se extrae el primer item de la cola. Si el símbolo

en la posición que indica el índice es un no terminal se computan todos los items no kernel que este item genera y por cada uno de estos se comprueba si ya pertenece al conjunto de items del estado. En caso de no estarlo se añade el item al conjunto de items y a la cola de items.

La tercera función se utiliza para computar todas las transiciones desde el estado actual hacia otro estado dado un símbolo determinado. La idea seguida para esta función es por cada uno de los items que tienen al símbolo en la posición que indica su respectivo índice, generar un nuevo item con los mismos atributos que el item original excepto por el índice, que será el del original aumentado en 1. Fíjese que los nuevos items generados indican que ya se vio el símbolo analizado en sus respectivas producciones. Estos nuevos items se añaden a una lista, con la cual se crea un nuevo estado a partir de dicha lista. O sea, dicha lista es el kernel del nuevo estado. Luego se comprueba si este nuevo estado ya fue generado. Si no lo ha sido entonces se construye el nuevo estado por completo. Finalmente se añade la transición.

Luego se tiene la clase `Automaton` la cual se instancia con una gramática y tiene definida la función `build(self)`; la cual construye un autómata LR(1) a partir de la gramática. Esta función devuelve una lista de estados y las transiciones vienen dadas por el diccionario `nexts` que cada estado tiene como atributo. En este método primero se crea un símbolo inicial y se la añade una producción con el símbolo inicial de la gramática. Luego por cada una de las producciones se generan los items iniciales. Luego se crea el estado inicial y a partir de este se van generando el resto de los estados utilizando una cola de estados, que se inicializa con el estado inicial, y la función `go_to` de cada estado. Mientras la cola no este vacia se extrae el primer estado de la misma, se llama a la función `go_to` con cada uno de los posibles símbolos y cada vez que se encuentre un estado nuevo se añade a la cola. Los nuevos estados se van guardando en una lista que es la que devuelve el método.

Por último tenemos la clase `TableActionGoTo`, la cual se instancia a partir de una gramática y tiene un método `build(self, path: str)` la cual construye las tablas "Action" y "GoTo" y las guarda como objetos .json en la dirección especificada. Ambas tablas las representaremos como listas de diccionarios, tal que el elemento i de cada lista será un diccionario correspondiente al estado i del autómata. La lista `action` representará la tabla "Action" y la lista `go_to` representará la tabla "GoTo".

Entonces por cada estado se crean dos diccionarios: `state_action` y `state_go_to` y se analizan cada una de sus posibles transiciones. Si el símbolo con el que se hace la transición hacia otro estado es un terminal entonces, se agrega al diccionario `state_action` el símbolo como llave y una tupla de dos elementos como su valor correspondiente, de tal forma que el primer elemento de dicha tupla es el carácter 'S', que indica que la acción a hacer es Shift, y el segundo elemento es el número del estado al que lleva la transición. Si por el contrario el símbolo es un no terminal se añaden como llave y valor respectivamente el símbolo y el número del estado al que lleva la transición en el diccionario `state_go_to`. Luego se analizan todos los items del estado cuya producción ya fue vista por completo y si existe un par de items con igual lookahead se reporta la existencia de un conflicto Reduce - Reduce.

Por cada lookahead y su respectivo item detectados anteriormente, se chequea si el lookahead aparece entre las llaves del diccionario `state_action` y de ser así entonces ocurre un conflicto Shift-Reduce y se reporta el mismo. En caso contrario se añade al diccionario `state_action` el lookahead como llave y como su valor asociado una tupla de dos elementos, tal que el primer elemento es el carácter 'R', que indica que hay que hacer Reduce, y el segundo elemento es el id de la producción que se debe reducir. Si el lookahead detectado es el símbolo especial '$' y además la parte izquierda del item correspondiente es el símbolo 'S' definido como el inicial, entonces la cadena es válida y se añade al diccionario `state_action` dicho símbolo como llave y como su valor correspondiente una tupla con un solo elemento : la cadena 'OK'.

Luego de todos estos análisis sobre el estado se añaden los diccionarios `state_action` y `state_go_to` a las listas `action` y `go_to` respectivamente. Cuando se terminen de analizar todos los estados estas listas se guardan en sendos archivos de tipo json en la dirección especificada.

AST y Árbol de derivación Para la construcción del AST durante el proceso de parsing se implementaron las definiciones de sus nodos y de algunos nodos del árbol de derivación, para facilitar este

proceso. Los nodos del AST heredan de forma indirecta de la clase abstracta `Node`, dividiéndose en instrucciones y expresiones, excepto por los siguientes nodos que heredan directamente de `Node`:

- `BSFile`: Nodo raíz del AST.
- `AttrDef`: Definición de atributo de una clase.
- `ClassDef`: Definición de una clase.

Los nodos instrucciones son los siguientes:

- `FuncDef`: Definición de función.
- `If`: Instrucción **if**.
- `Branch`: Lista de instrucciones **if** y una instrucción **else**
- `WhileDef`: Instrucción **while**
- `Decl`: Declaración de variable.
- `Assign`: Asignación de variable.
- `Return`: Instrucción **return**.
- `Break`: Instrucción **break**.
- `Continue`: Instrucción **continue**.

Los nodos expresiones son los siguientes:

- `BinaryExpression`: Expresión binaria.
- `AritmeticBinaryExpression`: Expresión binaria que solo se puede realizar entre dos números.
- `TernaryExpression`: Expresión ternaria
- `Inversion`: Expresión que es la negación de otra expresión.
- `Primary`: Engloba tanto el llamado a una función, como la acción de referirse a un atributo de la clase (incluyendo las funciones de dicha clase).
- `Variable`: Variable.
- `Number`: Número
- `Bool`: Expresión booleana
- `MyNone`: Expresión None
- `MyList`: Lista
- `PExpression`: Expresión entre paréntesis

El resto de clases implementadas en este módulo corresponden a los nodos del árbol de derivación, los cuales son utilizados para construir los nodos anteriormente mencionados.

Para construir al AST durante el proceso de parsing se atributaron algunas derivaciones de la gramática, utilizando las funciones implementadas en el módulo src.language.parser.engine_ast.

Parseo de una cadena Para esto se implementó la clase `Parser`, la cual se instancia con una gramática y las tablas de "Action" y "GoTo". En esta clase se implementó la función `parse`, la cual recibe una cola de tokens y de ser una cadena sintácticamente correcta, devuelve el AST generado a partir de la cadena. Primeramente se añade un token especial que se considera como el toquen final de la secuencia y se definen tres pilas:

- `tokens_stack`: En esta se van a ir guardando los lexemas de los tokens vistos.
- `states_stack`: En esta se van a ir guardando los estados por los que se ha transitado hasta el momento.
- `nodes`: En esta se van a ir guardando los nodos del AST y el árbol de derivación. De tal forma que al final del proceso de parsing, en esta solo quede la raíz del AST.

Entonces mientras queden tokens por ver de la secuencia, se toma el primer token de la misma y se cargan las acciones del estado actual (el que está en el tope de la pila de estados). Si no existe ninguna acción con dicho token hacia otro estado, entonces se reporta que se detectó un token inesperado en la secuencia. Pasado este punto se consulta por la acción que se debe realizar.

Si la acción es 'OK' es porque la cadena es válida y se retorna el objeto en la posición 0 de la pila `nodes` (el AST). Si la acción es hacer Shift se añade el estado a la pila de estados, se añade el lexema del token a la pila de los tokens y se saca el mismo de la secuencia.

Si lo que toca es hacer Reduce, entonces se obtiene la producción con la que se va a reducir y si además la producción es atributada, o sea si esta producción tiene asociada una función que construye un nodo del AST o del árbol de derivación, se llama a esta función pasándole como argumentos la pila de tokens y la pila de nodos. Estas funciones son las que modifican la pila de nodos. Cada una de ellas extrae o no un determinado número de nodos de la pila de nodos y agrega un nodo nuevo a la pila de nodos. Luego se extraen de la pila de tokens una cantidad de elementos igual a la longitud de la parte derecha de la producción que se reduce. De igual forma se hace con la pila de estados.

Entonces con el estado que queda en el tope de la pila se cargan utilizando la tabla "GoTo" las transiciones que se pueden hacer con no terminales en dicho estado. Si no existen transiciones en dicho estado con el no terminal que es parte izquierda de la producción que se redujo, entonces se lanza un error indicando que la secuencia de tokens es inválida. En caso contrario se añade el no terminal a la pila de tokens y se añade el estado al que se avanza con dicho token a la pila de estados.

3.6. Análisis Semántico

Definiendo tipos Para implementar nuevos tipos en el lenguaje se implementó una clase `Type` en la que cada clase del lenguaje será una instancia de esta.

`Type` consta de los siguientes atributos y métodos

- Atributos
  - `name`: Nombre de la clase o nuevo tipo
  - `parent`: Tipo inmediato del cuál hereda este tipo
  - `context`: Contexto en que se encuentran todos los atributos y métodos de este tipo.
  - `def_context`: Contexto en el que se encuentra este tipo
- Métodos
  - `define_attribute`: Define un atributo de la clase.
  - `define_method`: Define un método de la clase.
  - `is_attribute`: Revisa si el atributo existe.
  - `check_method`: Revisa que el método esté bien definido a partir de los atributos
  - `is_method`: Revisa si el método existe.
  - `is_child`: Revisa si el nombre dado es padre del tipo.

Estos métodos en su mayoría se apoyan en funciones del contexto para definir y revisar atributos dentro del contexto del tipo creado.

Los tipos admitidos en el lenguaje son los siguientes:

- `number`: Que engloba a los números.
- `bool`: Que engloba a las expresiones booleanas
- `List`: Que engloba a las listas. Una lista cuenta con dos funciones: `append` y `remove` para añadir y remover elementos de la misma respectivamente.
- Todos los tipos descritos durante la explicación de la simulación con sus respectivos métodos y atributos. Aquí vale aclarar que de estos los únicos tipos instanciables son `LandMap` y `Simulator`.
- Los nuevos tipos definidos por el usuario que se explican a continuación (que también son instanciables).

En nuestro lenguaje lo primero que se hace una vez que se crea un tipo es revisar si este tiene un padre. En este lenguaje todos los tipos definidos por el usuario deben heredar de `LandUnit`, `NavalUnit` o `StaticObject` en caso contrario se lanzará una excepción. Una vez que se conoce cuál es la clase de la cuál se hereda, se procede a copiar los atributos y tipos definidos en la clase padre. Las funciones heredadas pueden ser modificadas una única vez durante la definición de la clase. Además cada función tiene en su contexto definida la función `super()` que devuelve el tipo de la clase padre y de esta forma se puede llamar a la definición de la función en la clase padre directamente.

Además se tienen como funciones built-in del lenguaje las funciones `build_random_map` y `print`.

La función `build_random_map` es una función built-in que devuelve una instancia de `LandMap` y recibe como parámetros un número real entre 0 y 1 que indica el porcentaje de casillas de tierra que se desea tenga el mapa, y dos enteros que indican el número de filas y de columnas que se desea tenga el mapa.

La función `print` recibe un argumento e imprime en consola dicho argumento.

Context Para definir el contexto en el que se escribe un programa con el lenguaje, se implementó la clase `Context`, esta cuenta con los siguientes atributos y métodos

- Atributos
  - `name`: Nombre del contexto
  - `father`: Contexto padre de este contexto
  - `children`: Contextos hijos de este contexto
  - `_var_context`: Variables del contexto.
  - `_func_context`: Funciones del contexto
  - `_type_context`: Tipos definidos en el contexto
  - `If`: Cantidad de sentencias `If` definidas en el contexto
  - `Else`: Cantidad de sentencias `Else` definidas en el contexto
  - `While`: Cantidad de sentencias `While` definidas en el contexto
- Métodos
  - `check_var`: Revisa si la variable está definida en el contexto.
  - `check_var_type`: Revisa que la variable esté definida en el contexto y el tipo corresponda con el tipo de la variable.
  - `check_func`: Revisa que la función esté definida.
  - `check_func_args`: Revisa si la función está definida y los tipos de los argumentos coinciden con los tipos de la función.
  - `get_type`: Devuelve el tipo de una variable en el contexto.
  - `define_var`: Si la variable no existe en el contexto, la crea.
  - `define_func`: Define la función si no existe en el contexto. Una función solo puede definirse una vez. En caso de funciones heredadas estas pueden re-definirse una única vez. Cuando se crea una función se crea además un contexto, el contexto que ella define que viene a ser el cuerpo del método.
  - `create_child_context`: Crea un contexto hijo.
  - `create_type`: Crea un nuevo tipo en el contexto. Una vez que se crea un tipo se define una función dentro del contexto donde se crea el tipo y con el mismo nombre de este, la cual será el constructor de la clase. Además, una clase también define un contexto, donde estarán todos sus atributos. Se tiene además una propiedad que dice si el tipo es accesible. Por defecto todos los tipos son accesibles. Si un tipo no es accesible no se podrá instanciar pues no tendrá definida una función constructor. Usualmente esta propiedad es para algunos tipos built-in, por ejemplo `number`, pues no tendría sentido hacer: `number a = number();`.
  - `get_return_type`: Devuelve el tipo de retorno de una función si esta está definida en el contexto.
  - `get_type_object`: Devuelve la clase a la que pertenece un objeto dado.
  - `is_type_defined`: Analiza si el tipo está definido.

- `is_context_father`: Revisa si el contexto dado es ancestro de este contexto.
- `get_context_father`: Devuelve el contexto dado si es ancestro de este contexto.
- `is_in_while_context`: Revisa si el contexto actual está definido dentro de un contexto while. De esta forma `break` y `continue` no pueden ser usadas si no están dentro de un bucle.
- `is_in_func_context`: Revisa si el contexto actual está definido dentro de un contexto función pues sería incorrecto usar una expresión de tipo `return` fuera de un método.
- `check_type`: A este método se le pasan dos tipos y comprueba que el primero sea ancestro del segundo.

Se definió `Type` como tipo genérico, que no viene a ser un tipo en sí, es más bien una palabra reservada que significa que puede ser sustituida por cualquier tipo existente.

Pasadas por el AST Para verificar la corrección de la semántica se implementó el patrón `Visitor` que visitará el árbol de sintaxis abstracta en cuatro pasadas distintas

- `Type_Collector`: Es la primera pasada con el patrón `Visitor` en este se recogen todos los tipos que se van a definir, para crearlos en el contexto. Además se revisa que los tipos que se crearon sean hijos de `LandUnit`, `NavalUnit` o `StaticObject` en caso contrario lanza una excepción. Se guardarán además el contexto de que define la clase, el contexto donde se define la clase y el contexto que define el constructor de la misma.
- `Type_Builder`: Es la segunda pasada con el patrón, se toman todos los tipos que se definieron anteriormente y se definen sus métodos y atributos. Se guarda además el contexto que define cada función cuando se crea.
- `Type_Context`: Es la tercera pasada del visitor, se visita cada nodo para definir en qué contexto está. Para esto se usa el atributo `current_context` en el cual se guarda el contexto que se está visitando actualmente, actualizándose cada vez que se pasa por un nodo que define un contexto (clases, funciones, condicionales o bucles) y volviendo al valor anterior una vez que se termina ese camino del árbol de sintaxis abstracta.
- `Type_Checker`: Es la cuarta pasada con el patrón, en esta se revisan que los tipos de cada expresión no entren en contradicción. Para esto se revisa en los contextos donde se encuentran los nodos y se utilizan las funciones de la clase `Context`. Se chequea además que expresiones de `return` solo ocurran dentro de una función y expresiones como `break` y `continue` estén siempre dentro de un bucle.

3.7. Generación de código

Para la generación de código se parte de la raiź del AST y haciendo uso del patrón "Visitor" se visitan cada uno de los nodos del AST. El código generado es una cadena en código Python, de tal forma que se importan todos los módulos necesarios para realizar una simulación y el código Python generado es equivalente a lo que el usuario programó en nuestro lenguaje. El código referente a este apartado se encuentra en el módulo src.language.code_generation.code_genration. En este aparece implementada la clase `CodeGenerate`, la cual tiene como atributos un `StringIO`, un entero que lleva la cantidad de tabuladores para generar el código Python correspondiente y un diccionario que traduce algunos de los símbolos de nuestro lenguaje a símbolos en código Python. Además para cada uno de los nodos del AST se implementó la función `visit`.

Un dato interesante es que para los nodos de tipo `ClassDef` y los nodos de tipo `Statement`, su método `visit` correspondiente escribe su código correspondiente en el atributo `StringIO` de la clase. Para los nodos de tipo `Expression` se retorna su código correspondiente. Esto se hace así dado que en Python casi todo es una instrucción y muchas de estas dependen de expresiones. En el método `visit` correspondiente al nodo `BSFile` se escriben todas las importaciones de los módulos necesarios para correr una simulación; y la cadena del código transpilado a Python es la que devuelve este método, que devuelve el valor del atributo de tipo `StringIO` de la clase `CodeGenerator`.

4. Ejemplos

A continuación se muestran algunos ejemplos de uso del compilador y de ejecución de la simulación.

El primer ejemplo que se define a continuación es la creación de la clase Soldier que es un LandUnit. En este caso solo se define el constructor de la clase, que recibe el parámetro `id` que representa el identificador de la unidad en la simulación y el parámetro `attack` que es el valor de ataque de la unidad.

```
class Soldier is LandUnit -> {
    constructor(number id, number attack) -> {
        number self.id = id;
        number self.attack = attack;
    };
};
```

Note que tras cualquier instrucción en el lenguaje Battle Script se necesita poner un punto y coma (;) para especificar el fin de la instrucción. En este caso solo se indican los parámetros `id` y `attack`. Por tanto el resto de parámetros de la unidad tomará los valores por defecto. Dentro de una clase se pueden definir funciones como se muestra en el siguiente ejemplo:

```
class Archer is LandUnit -> {
    constructor(number id, number max_range) -> {
        number self.id = id;
        number self.max_range = max_range;
    };

    function number plus_id() -> {
        return self.id + 1;
    };
};
```

En este caso se define una función que devuelve un número y no recibe parámetros. Obsérvese que para hacer referencia a los atributos de la clase se utiliza la palabra clave `self`.

Si se quisiera redefinir una función de clase heredada del padre de la clase y en esta utilizar la función del padre, se puede hacer utilizando la función super() como se muestra en el siguiente ejemplo:

```
    function void turn() -> {
        number a=2*(-1);
        if a lte 1 -> {
            super().turn();
        };
    };
```

En este caso se ha redefinido la función `turn`, utilizando además la función heredada del padre.

Las definiciones de clases deben ir todas al inicio del programa y cuando se terminen definir todas se debe poner el caracter '&' para indicar que ya se terminó la definición de las clases y ahora se van a comenzar escribir las instrucciones.

Para generar un mapa aleatorio se hace de la siguiente forma:

```
LandMap map = build_random_map(1, 5, 5);
```

Entonces pasemos a ver como crear las unidades:

```
Soldier sOne = Soldier(1, 10);
Soldier sTwo = Soldier(2, 9);

Archer aOne = Archer(3, 5);
Archer aTwo = Archer(4, 5);
```

En este ejemplo hemos creado dos unidades de tipo “Soldier” y dos unidades de tipo “Archer” utilizando las clases definidas anteriormente.

Para poner una unidad en el mapa se hace de la siguiente forma:

```
sOne.put_in_cell(map, 0, 0);
```

Se toma a la unidad y se hace un llamado a la función `put_in_cell` pasándole como argumentos el mapa y par de enteros que indican la fila y la columna de la celda donde se desea colocar la unidad.

Un bando se crean de la siguiente forma:

```
Side SOne = Side(1, [sOne, aTwo]);
```

Como se muestra en los ejemplos anteriores un bando se instancia pasándole un id y la lista de unidades que conforman el bando.

Para crear un simulador se hace de la siguiente manera:

```
Simulator sim = Simulator(map, [SOne, STwo], 20, 1);
```

Fíjese que un simulador se instancia con un mapa, la lista de los bandos, la cantidad de turnos máximos que se desea dure la simulación y un número que indica la duración de un turno.

Para echar andar la simulación lo hacemos de la siguiente forma:

```
sim.start();
```

Entonces un ejemplo completo de un programa podría ser el siguiente:

```
class Soldier is LandUnit -> {
    constructor(number id, number attack) -> {
        number self.id = id;
        number self.attack = attack;
    };
};

class Archer is LandUnit -> {
    constructor(number id, number max_range) -> {
        number self.id = id;
        number self.max_range = max_range;
    };
};
&
LandMap map = build_random_map(1, 5, 5);

Soldier sOne = Soldier(1, 10);
Soldier sTwo = Soldier(2, 9);

Archer aOne = Archer(3, 5);
Archer aTwo = Archer(4, 5);
```

```
sOne.put_in_cell(map, 0, 0);
aTwo.put_in_cell(map, 0, 1);

aOne.put_in_cell(map, 4,4 );
sTwo.put_in_cell(map, 4, 3);

Side SOne = Side(1, [sOne, aTwo]);
Side STwo = Side(2, [aOne, sTwo]);

Simulator sim = Simulator(map, [SOne, STwo], 20, 1);

sim.start();
```

En el lenguaje además se pueden definir instrucciones más complicadas tales como: if, while, declaraciones, asignaciones, etc. A continuación se muestran algunos ejemplos de como hacerlo:

```
function number W(number a) -> {
    if a lt 0 and a eq 0 -> {
        return -1 * a;
    }
    elif a gt 1 -> {
        return 4;
    }
    else -> {
        return a;
    };
};

function number A(number a) -> {
    while a lte 5 -> {
        a = a + 1;
    };
    return a;
};
```

Más ejemplos se pueden encontrar en el módulo test/examples.

5. Reflejo de las asignaturas en el proyecto

5.1. Simulación

- Implementación de las unidades como agentes casi puramente reactivos.
- Utilización de la Arquitectura de Brooks para los agentes.
- Implementación de un simulador de conflictos bélicos.

5.2. Compilación

- Diseño de la gramática del lenguaje.
- Implementación de un tokenizador.
- Implementación de un sistema de expresiones regulares.
- Implementación de un parser $LR(1)$ canónico.
- Implementación del análisis semántico.
- Implementación de la generación de código Python.

5.3. Inteligencia Artificial

- Implementación de un sistema experto que actúa como función de los agentes de la simulación.
- Implementación de un algoritmo evolutivo para la generación de mapas de alturas bajo la restricción del porciento elevaciones.

6. Conclusiones

Se requería implementar un medio para simular batallas entre bandos distintos. Para ello, se implementó un simulador de batallas el cual es personalizable a través de un lenguaje de dominio específico Battle Script. La simulación de batallas en un entorno controlado ayudaría a reducir el costo en vidas humanas en las guerras, así como ahorrar recursos económicos y tomar decisiones estratégicas.

Se implementaron unidades que funcionan como agentes casi puramente reactivos, así como bandos que representan los ejércitos, batallones, compañías, escuadrones, etc. Las unidades funcionan a través de un sistema experto que les aporta la inteligencia necesaria para actuar con el fin de eliminar a todas las unidades del o de los bandos contrarios. Se definió un módulo para la generación automática de mapas de alturas.

Se probó la simulación con distintos casos de prueba, con mapas con poco o mucho relieve, bandos grandes o pequeños, así como se modificó el comportamiento de nuevas unidades definidas a través del lenguaje. Cabe destacar, que la simulación se ejecuta de forma eficiente, de forma que el usuario final obtiene resultados en poco tiempo.

Se sugiere desarrollar las sugerencias o recomendaciones para que el entorno y las unidades se asemejen cada vez más a los soldados, maquinarias de guerra, o ejércitos reales.

7. Recomendaciones

Los autores recomiendan nuevas líneas de desarrollo para el proyecto, a continuación se enumeran algunas de ellas:

- Implementación de un mapa aéreo para poder incluir unidades aéreas y así simular el accionar de aviones y helicópteros.
- Definir objetivos a los bandos. Se pudieran definir objetivos como los siguientes: eliminar una unidad determinada del bando contrario, rescatar una unidad aliada, tomar una posición del mapa, destruir a todas las unidades (objetivo actual), defender una posición, entre otros. Se pudiera enfocar utilizando algoritmos de planificación, o convertir los agentes en proactivos en lugar de reactivos.
- Compartir conocimiento entre unidades aliadas, por ejemplo: dónde se encuentran enemigos, qué celdas son más cómodas para realizar ataques.
- Implementación de unidades jefes dentro de los bandos, las cuales influyen hasta cierto punto en el comportamiento del resto de las unidades del bando.
- La implementación de una interfaz visual para la ejecución de la simulación.

8. Referencias

- Conferencias de Compilación, Curso 2021-2022, Alejandro Piad.
- Conferencias de Inteligencia Artificial, Curso 2021-2022, Suilán Estévez.
- Conferencias de Simulación, Curso 2021-2022, Yudivián Almeida.
- Creación de mundos mediante la generación procedural en Unity. Bouza, C., Romero, A. Universidad Complutense de Madrid, 2019.
- SimuBattle: software de recreación y simulación de batallas históricas. Alejandro Alonso, 2009