Análisis de Lenguajes de Programación

Trabajo Práctico Final: Clairvoyance

Tomás Fernández de Luco, F-3443/6

Índice

1 Presentación del proyecto

1.1 Introducción

Los juegos de rol (*tabletop role-playing games* en inglés) son una categoría de juegos de mesa en la que los participantes narran colaborativamente los eventos de una historia, cada jugador describiendo las acciones que realiza un personaje bajo su control. El éxito o fracaso de dichas acciones se determina con un conjunto de reglas, usualmente involucrando tiradas de dados que pueden recibir bonificaciones por características del personaje que las realiza.

En este tipo de juegos existe un jugador designado, llamado usualmente *Dungeon Master* o *D.M.*, el cual oficia como narrador de la historia y presenta a los demás participantes con los desafíos que deberán superar. Tareas del *D.M.* incluyen la descripción del entorno, resolución de las acciones de los personajes e interpretación de personajes secundarios no controlados por ningún jugador.

Una parte importante en juegos de este género es el combate, en donde los jugadores se enfrentan a un conjunto de enemigos controlados por el *D.M.*. El combate se desarrolla por turnos sobre un tablero, con ambos bandos tratando de derrotar a sus oponentes hasta que sólo un equipo quede en pie.

1.2 Motivación

El amplio abanico de responsabilidades del *Dungeon Master* torna muy complicado recordar cada pequeño detalle que tiene preparado para un combate. En consecuencia, frecuentemente puede olvidarse de sorpresas o tiene que interrumpir el ritmo de juego con anticlimáticas revisiones de los manuales para buscar las estadísticas de algún enemigo.

Clairvoyance es un lenguaje de dominio específico que permite definir la configuración inicial de un combate en un juego de rol usando un conjunto de reglas simplificado de la versión 3.5 del juego *Dungeons and Dragons*[1]. La característica más interesante del lenguaje radica en la posibilidad de definir el comportamiento y estrategia de las unidades no controladas por los jugadores, permitiendo al *Dungeon Master* establecer con antelación las acciones que le interesa que cada unidad realice sin tener que preocuparse por ejecutar cada una durante el encuentro.

Una vez interpretado un archivo fuente, el programa almacena el tablero y unidades, permitiendo avanzar los turnos, informar las acciones realizadas por cada unidad y modificar el estado mediante un intérprete de comandos. De esta manera, se provee con un asistente de combate que permite tener un control más rápido y eficaz del juego, desligando acciones mecánicas en el programa para poder centrarse en tareas narrativas o cualquier otro detalle importante.

2 Definición del lenguaje

En esta sección se presenta la gramática de una sintaxis concreta para el DSL. La definición de un combate necesita un tablero y una lista de descripciones para las unidades, comportamientos y equipos.

$$Combat ::= Board\ Units\ AIs\ Teams$$

Para las definiciones siguientes se consideran dados un tipo *String* para cadenas de caracteres, *Int* para números enteros y *Nat* para naturales.

2.1 Tablero

La descripción de un tablero requiere dos componentes: una especificación de su contorno y una lista opcional de casillas ocupadas por paredes. La primera puede darse de dos maneras:

[1] Las reglas de juego pueden encontrarse en el Anexo. Se recomienda leerlas antes de continuar a la definición del lenguaje.

- Como un rectángulo de cierta cantidad de columnas y filas.
- Como una sucesión de direcciones. Cada dirección indica en qué sentido y cantidad de casillas debería moverse un marcador que delimita el contorno exterior del tablero. Necesariamente el recorrido debe iniciar y terminar en la misma casilla.

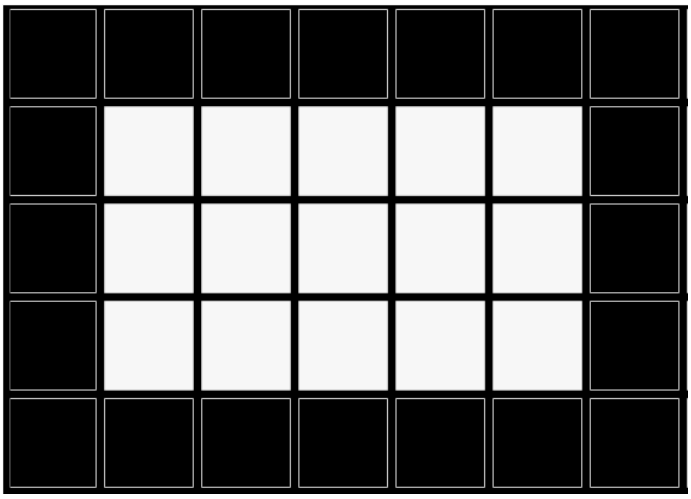

Figura 1: Tablero definido con *Rectangle 5, 3*.

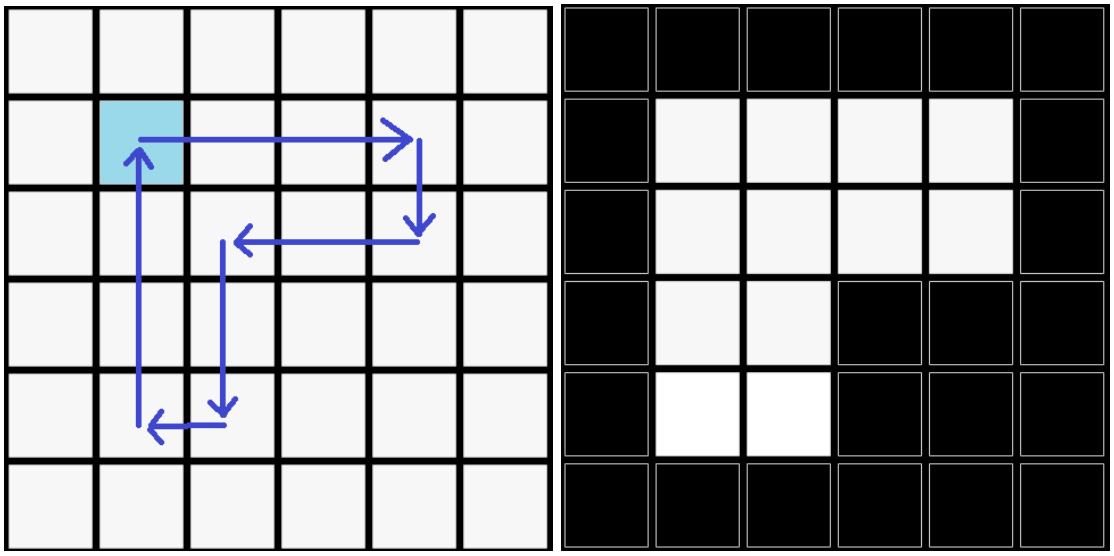

Figura 2: Recorrido *Outline 3R - 1D - 2L - 2D - 1L - 3U* y tablero resultante.

Las posiciones de obstáculos y unidades en el resto del archivo se darán en coordenadas de la forma *(columna, fila)*. El origen de coordenadas *(0, 0)* estará en la casilla libre en la esquina superior izquierda para tableros rectangulares y en el inicio de recorrido para tableros hechos por contorno. Consideramos que el eje de las columnas crece hacia la derecha y el de las filas, hacia abajo.

La lista de obstáculos especifica un conjunto de casillas internas al tablero que se marcarán como paredes y serán intransitables. Cada obstáculo se especifica con una coordenada, permitiendo también dar un intervalo para las columnas y/o filas, pudiendo definir una línea horizontal, vertical o rectángulo contiguo de obstáculos con una única entrada.

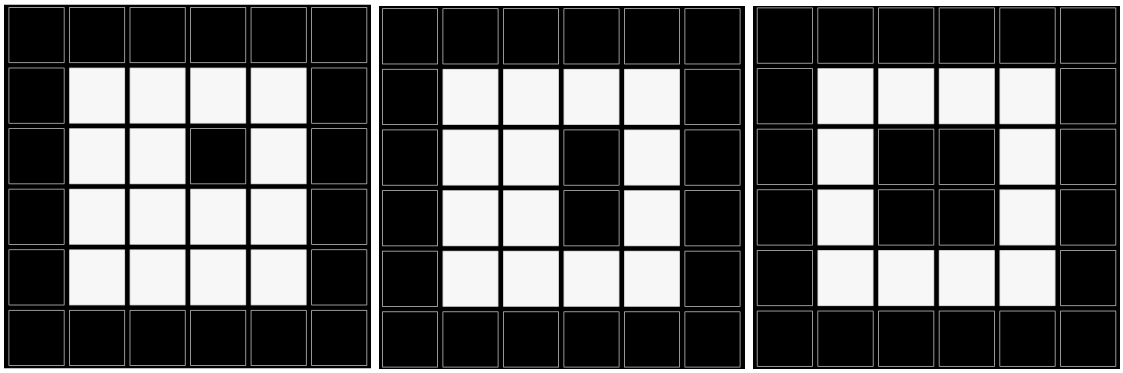

Figura 3: Ejemplos de obstáculo en *(2, 1)*, *(2, 1-2)* y *(1-2, 1-2)*.

Por lo tanto, la gramática para la definición del tablero es:

$$
\begin{aligned}
\mathit{Board} &::= \text{`}MAP\text{'}\ \text{`\{'}\ \text{`}\mathit{Layout}:\text{'}\ \mathit{LayoutDesc}\ \text{`;'}\ \text{`}\mathit{Obstacles}:\text{'}\ \mathit{Obstacles}\ \text{`;'}\ \text{`\}'} \\
\mathit{LayoutDesc} &::= \text{`}\mathit{Rectangle}\text{'}\ \mathit{Nat}\ \text{`,'}\ \mathit{Nat} \\
&\quad \mid \text{`}\mathit{Outline}\text{'}\ \mathit{Directions} \\
\mathit{Directions} &::= \mathit{Nat}\ \mathit{Direction} \\
&\quad \mid \mathit{Nat}\ \mathit{Direction}\ \mathit{Directions} \\
\mathit{Direction} &::= \text{`}U\text{'}\ \mid\ \text{`}R\text{'}\ \mid\ \text{`}D\text{'}\ \mid\ \text{`}L\text{'} \\
\mathit{Obstacles} &::= \mathit{Obstacle}\ \mathit{ObstacleRest}\ \mid\ \varepsilon \\
\mathit{ObstacleRest} &::= \text{`,'}\ \mathit{Obstacle}\ \mathit{ObstacleRest}\ \mid\ \varepsilon \\
\mathit{Obstacle} &::= \text{`(`}\ \mathit{IntRange}\ \text{`,'}\ \mathit{IntRange}\ \text{`)'} \\
\mathit{IntRange} &::= \mathit{Int}\ \mid\ \mathit{Int}\ \text{`}-\text{'}\ \mathit{Int}
\end{aligned}
$$

2.2 Unidades

Cada descripción de una unidad debe tener un nombre único y un bloque de estadísticas[2]. La información que hay en ese bloque depende del tipo de unidad:

- Los personajes controlados por los jugadores son identificados con la etiqueta *PLAYER*. Para este tipo de unidades sólo se requiere su modificador de iniciativa.
- Las unidades controladas por el programa son identificadas con la etiqueta *UNIT*. Su bloque de estadísticas es más completo, pues requiere los puntos de vida, iniciativa, velocidad, clase de armadura y descripciones de ataque estándar y ataque completo. Estas entradas pueden darse en cualquier orden, pero en el bloque debe haber exactamente una de cada tipo.

[2]El significado de cada una de las estadísticas también se explica en detalle en el Anexo.

$$
\begin{aligned}
Units &::= UnitDec \mid UnitDec\ Units \\
UnitDec &::= \text{`}PLAYER\text{'}\ String\ \text{`}\{\text{'}\ \text{`}Initiative:\text{'}\ Modifier\ \text{`}\}\text{'} \\
&\mid \text{`}UNIT\text{'}\ String\ \text{`}\{\text{'}\ Stats\ \text{`}\}\text{'} \\
Stats &::= Stat\ \text{`};\text{'} \mid Stat\ \text{`};\text{'}\ Stats \\
Stat &::= \text{`}HP:\text{'}\ Nat \\
&\mid \text{`}Initiative:\text{'}\ Modifier \\
&\mid \text{`}Speed:\text{'}\ Nat \\
&\mid \text{`}AC:\text{'}\ Int \\
&\mid \text{`}Attack:\text{'}\ Attacks \\
&\mid \text{`}FullAttack:\text{'}\ Attacks \\
Attacks &::= Attack \mid Attack\ \text{`},\text{'}\ Attacks \\
Attack &::= AttackRange\ Modifier\ DieRoll \mid AttackRange\ DieRoll \\
AttackRange &::= \text{`}Melee\text{'} \mid \text{`}Ranged\text{'}\ \text{`}(\text{'}\ Nat\ \text{`})\text{'} \\
DieRoll &::= DieExp \mid DieExp\ Modifier \\
DieExp &::= Nat\ \text{`}d\text{'}\ Nat \\
Modifier &::= \text{`}+\text{'}\ Nat \mid \text{`}-\text{'}\ Nat
\end{aligned}
$$

2.3 Comportamientos

2.3.1 Sintaxis

La descripción de un comportamiento se compone de un nombre único y una secuencia de acciones que tomará una unidad en el combate. La unidad básica de las acciones es la descripción de un turno, que consiste de cualquier combinación posible de acciones de movimiento, estándar y de turno completo según las reglas del juego. También existen estructuras de control de flujo de tipo *if-then-else* y *while*. Por último, las acciones pueden componerse secuencialmente para generar planes que se llevan a cabo en el transcurso de múltiples turnos.

Las acciones de movimiento definidas son acercarse a un objetivo o intentar escapar de unidades enemigas. Por su lado, las acciones estándar y de turno completo permiten hacer un ataque estándar o completo, respectivamente, a un objetivo. El significado de cada una de estas acciones se explica en la sección siguiente.

Un objetivo puede ser una unidad en específico (ya sea quien realiza la acción o una descripción única para cualquier otra unidad a partir de su nombre de equipo, nombre de unidad y número identificador), o una descripción más genérica que permite casos como *"el aliado más cercano"* o *"el último enemigo atacado"*.

Las condiciones utilizadas en los controladores de flujo son un predicado sobre el estado del combate. La mayoría revisan la cantidad de unidades en juego que satisfacen cierto predicado, y luego determinan si existe alguna que lo haga, o cuentan la cantidad que haya y lo comparan a un número dado. También hay una que permite realizar comparaciones con respecto a la cantidad de turnos que hayan pasado en el combate. Las condiciones pueden combinarse con los operadores lógicos *and* y *or*.

Por último, algunas condiciones utilizan una expresión de rango a la hora de evaluarse. Una expresión de rango es una manera de describir la cantidad máxima de casillas de distancia alrededor de la unidad que realice la acción en la que se busca para determinar la validez del predicado, ignorando todo lo que quede por fuera. Posibles rangos son cualquier número natural, rango cuerpo a cuerpo o valores que dependen de las estadísticas de la unidad en particular, como su velocidad o el rango mínimo de un ataque en su sucesión de ataque estándar o completo. Las expresiones básicas también pueden combinarse sumándolas o multiplicándolas por un número natural.

$$
\begin{aligned}
AIs &::= \text{`}AI\text{'}\ String\ \text{`}\{\text{'}\ ActionSeq\ \text{`}\}\text{'} \\
&\mid \text{`}AI\text{'}\ String\ \text{`}\{\text{'}\ ActionSeq\ \text{`}\}\text{'}\ AIs \\
ActionSeq &::= Action \mid Action\ ActionSeq \\
Action &::= Turn\ \text{`};\text{'} \\
&\mid \text{`}if\text{'}\ Condition\ \text{`}then\text{'}\ \text{`}\{\text{'}\ ActionSeq\ \text{`}\}\text{'}\ \text{`}else\text{'}\ \text{`}\{\text{'}\ ActionSeq\ \text{`}\}\text{'} \\
&\mid \text{`}while\text{'}\ Condition\ \text{`}\{\text{'}\ ActionSeq\ \text{`}\}\text{'} \\
Turn &::= \text{`}Pass\text{'} \\
&\mid MoveAction \\
&\mid StandardAction \\
&\mid FullAction \\
&\mid MoveAction\ \text{`}-\text{'}\ MoveAction \\
&\mid MoveAction\ \text{`}-\text{'}\ StandardAction \\
&\mid StandardAction\ \text{`}-\text{'}\ MoveAction \\
MoveAction &::= \text{`}Approach\text{'}\ Target \mid \text{`}Disengage\text{'} \\
StandardAction &::= \text{`}Attack\text{'}\ Target \\
FullAction &::= \text{`}Full\ Attack\text{'}\ Target \\
Target &::= \text{`}self\text{'} \\
&\mid String\ \text{`}:\text{'}\ String\ \text{`}:\text{'}\ Nat \\
&\mid Adjective\ Description \\
Adjective &::= \text{`}closest\text{'} \mid \text{`}furthest\text{'} \mid \text{`}last\text{'} \\
Description &::= \text{`}ally\text{'} \mid \text{`}enemy\text{'} \mid String \mid String\ \text{`}:\text{'}\ String \\
Condition &::= Description\ \text{`}count\text{'}\ IntComparison \\
&\mid Description\ \text{`}in\,range\text{'}\ \text{`}(\text{'}\ RangeExp\ \text{`})\text{'} \\
&\mid Description\ \text{`}count\,in\,range\text{'}\ \text{`}(\text{'}\ RangeExp\ \text{`})\text{'}\ IntComparison \\
&\mid String\ \text{`}:\text{'}\ String\ \text{`}:\text{'}\ Nat\ \text{`}in\,range\text{'}\ \text{`}(\text{'}\ RangeExp\ \text{`})\text{'} \\
&\mid \text{`}total\,turn\,count\text{'}\ IntComparison \\
&\mid Condition\ \text{`}and\text{'}\ Condition \\
&\mid Condition\ \text{`}or\text{'}\ Condition \\
&\mid \text{`}(\text{'}\ Condition\ \text{`})\text{'} \\
IntComparison &::= Comparator\ Int \\
Comparator &::= \text{`}>\text{'} \mid \text{`}>=\text{'} \mid \text{`}<\text{'} \mid \text{`}<=\text{'} \mid \text{`}=\text{'} \mid \text{`}!=\text{'} \\
RangeExp &::= Nat \mid \text{`}Melee\text{'} \mid \text{`}Attack\text{'} \mid \text{`}Full\ Attack\text{'} \mid \text{`}Speed\text{'} \\
&\mid RangeExp\ \text{`}+\text{'}\ RangeExp \\
&\mid RangeExp\ \text{`}*\text{'}\ Nat \\
&\mid \text{`}(\text{'}\ RangeExp\ \text{`})\text{'}
\end{aligned}
$$

2.3.2 Semántica de las acciones

Una parte esencial del proyecto es la evaluación del comportamiento actual de una unidad para determinar la acción que debe realizar en su turno, además de cuál debe ser el comportamiento que se debe tener en cuenta para el turno siguiente. La idea fundamental detrás de la evaluación consiste en recorrer la secuencia de acciones hasta encontrar una descripción de un turno, ejecutarlo y preservar las acciones restantes para utilizarlas como el comportamiento para el turno siguiente.

Para formalizar estas nociones, definiré una función de evaluación *small-step* para acciones, la cual permitirá explicar en más detalle el avance para cada construcción. Primero consideraré la siguiente sintaxis abstracta para acciones, la cual posee todos los tipos descriptos anteriormente más una acción especial *None* que representa una acción vacía que no describe un turno y es simplemente salteada.

$$\begin{aligned} Action &::= None \\ &\mid Turn\ TurnDescription \\ &\mid If\ Condition\ Action\ Action \\ &\mid While\ Condition\ Action \\ &\mid Cons\ Action\ Action \\ TurnDescription &::= ... \\ Condition &::= ... \end{aligned}$$

Supongamos definido un tipo *Gamestate* que contiene toda la información necesaria del estado actual del juego, además de dos funciones $evalCondition : Gamestate \times Condition \to Bool$ y $evalTurn : Gamestate \times TurnDescription \to Gamestate \times Bool$. La primera evalúa una condición en el estado actual, mientras que la segunda ejecuta el turno y devuelve el estado de juego modificado, además de si todas las acciones pudieron llevarse a cabo.

Entonces, nuestra función de evaluación tendrá el tipo $\Rightarrow: Gamestate \times Action \to Gamestate \times Action \times Bool$, la cual toma un estado y una acción, devolviendo el estado actualizado al ejecutar un turno, la acción que deberá evaluarse en el turno siguiente y un valor booleando que representa si ya se ejecutó una acción de turno. La razón de este último valor es para que la función de evaluación sepa si tiene que seguir recorriendo el árbol de acciones buscando un turno para realizar o si simplemente debe devolver los valores nuevos.

$$\text{S-None}\ \frac{}{(s, None) \Rightarrow (s, None, False)}$$

$$\text{S-TurnT}\ \frac{evalTurn\ s\ t == (s', True)}{(s, Turn\ t) \Rightarrow (s', None, True)} \qquad \text{S-TurnF}\ \frac{evalTurn\ s\ t == (s', False)}{(s, Turn\ t) \Rightarrow (s', Turn\ t, True)}$$

$$\text{S-IfT}\ \frac{\begin{array}{c} evalCondition\ s\ cond == True \\ (s, trueAct) \Rightarrow (s', action, success) \end{array}}{(s, If\ cond\ trueAct\ falseAct) \Rightarrow (s', action, success)}$$

$$\text{S-IfF}\ \frac{\begin{array}{c} evalCondition\ s\ cond == False \\ (s, falseAct) \Rightarrow (s', action, success) \end{array}}{(s, If\ cond\ trueAct\ falseAct) \Rightarrow (s', action, success)}$$

$$\text{S-While}\ \frac{(s, If\ cond\ (Cons\ action\ (While\ cond\ action))\ None) \Rightarrow (s', action', success)}{(s, While\ cond\ action) \Rightarrow (s', action', success)}$$

$$\text{S-ConsT}\ \frac{(s, act1) \Rightarrow (s', action, True)}{(s, Cons\ act1\ act2) \Rightarrow (s', Cons\ action\ act2, True)}$$

$$\text{S-ConsF}\ \frac{\begin{array}{c} (s, act1) \Rightarrow (s', action, False) \\ (s, act2) \Rightarrow (s'', action', success) \end{array}}{(s, Cons\ act1\ act2) \Rightarrow (s'', action', success)}$$

Varias de las reglas son lo suficientemente directas como para que no haya nada interesante para comentar sobre ellas. Sin embargo, hay algunas excepciones. *S-TurnT* y *S-TurnF* evalúan la descripción de un turno, actualizan el estado del juego y devuelven que se completó un turno. Si se concretaron todas las acciones, se devuelve *None* como nueva acción pues la descripción del turno ya fue ejecutada. En el caso contrario, se devuelve el mismo turno que se intentó ejecutar. La razón por la que se hace esto se explica en la sección de acciones por defecto.

Por otro lado, *S-ConsT* y *S-ConsF* muestran la razón de ser del valor booleano en la imagen de $\Rightarrow$. Si ya se ejecutó un turno evaluando *act1*, entonces solamente hay que tomar la acción resultante como primer elemento del *Cons* y seguir reconstruyendo la secuencia. Por otro lado, si no se pudo ejecutar nada, entonces se procede a evaluar la segunda acción.

2.3.3 Acciones por defecto

A la hora de evaluar la descripción de un turno, surge la pregunta de qué hacer cuando una acción no pueda llevarse a cabo. Si bien el lenguaje provee estructuras de control para revisar condiciones del estado antes de decidir realizar una acción, preguntar por algunas situaciones en particular constantemente puede resultar tedioso en algunos casos, o no expresable en el lenguaje en algunos otros. Es por eso que decidí implementar un sistema de acciones por defecto en los casos en los que una acción no pueda realizarse en el turno en el que se la considera, pero podría llegar a realizarse en el futuro.

Primero hay que hacer una distinción entre qué acciones podrían ser realizables en un turno futuro y cuáles no van a ser realizables jamás. Si una unidad debe atacar a su enemigo más cercano pero su rango de ataque es menor a la distancia que haya con su objetivo, una alternativa es usar su movimiento en el turno para acercarse y poder atacarlo en el turno siguiente. Sin embargo, si no queda ningún enemigo vivo que pudiera ser un objetivo válido, jamás podría realizar su ataque. Entonces, se agrega una condición extra al valor de retorno de la función *evalTurn*: devolverá *True* si todas las acciones del turno que no pudieron realizarse jamás podrían concretarse. De esta manera, los turnos que se consumen en la evaluación de acciones son los que se ejecutaron completamente, o los que quedaron incompletos y jamás podrían completarse. Para cualquier otro caso, se devuelve *False* para reintentar las acciones en el turno siguiente. Las acciones por defecto dan alternativas conservadoras a acciones no realizables actualmente. En varios casos la unidad tendrá un turno subóptimo pero que lo dejará en una posición más propicia para poder ejecutar su turno descripto en la siguiente ronda.

Para las acciones de movimiento siempre se hacen el mejor esfuerzo: ya sea acercarse todo lo que la velocidad permita a un objetivo dado (siempre y cuando exista y haya un camino hacia él), o escaparse a la casilla que esté lo más alejada posible de los enemigos. Por esta razón, dichas acciones siempre se consideran exitosas. Por otro lado, si se desea atacar a un objetivo válido que esté fuera de rango o de línea de visión, se buscará la casilla más cercana desde la que se pueda concretar el ataque y la unidad se moverá en dirección a la misma.

2.4 Equipos

Por último, cada equipo tiene un nombre único y una lista no vacía de descripciones de sus miembros. Para unidades controlados por jugadores, se da su nombre (el que debe haberse definido previamente con un bloque de tipo *PLAYER*). Para unidades controladas por el programa, debe darse una combinación de su nombre (definido en un bloque *UNIT*) y un nombre de comportamiento (definido en un bloque *AI*). En ambos casos, luego se escribe un multiplicador del tipo "x N" para indicar que hay N unidades de este tipo en el equipo, seguido de una lista de N coordenadas en donde inicia el combate cada una.

Todas las unidades con el mismo nombre en un equipo reciben un número identificador único. De esta manera, la tripla nombre de equipo - nombre de unidad - identificador permite determinar de manera única a toda unidad en el combate. Esos tres valores son los utilizados en algunas

descripciones de objetivos de las acciones. Además, al inicio un combate también se le asigna un índice numérico único a cada unidad, pero eso será utilizado más adelante.

$$
\begin{aligned}
Teams &::= Team \mid Team\ Teams \\
Team &::= \text{`}TEAM\text{'}\ String\ \text{`}\{\text{'}\ TeamMembers\ \text{`}\}\text{'} \\
TeamMembers &::= TeamMember\ \text{`};\text{'} \mid TeamMember\ \text{`};\text{'}\ TeamMembers \\
TeamMember &::= String\ \text{`,'}String\ \text{`}\times\text{'}\ Nat\ \text{`:'}Positions \\
&\mid String\ \text{`}\times\text{'}\ Nat\ \text{`:'}Positions \\
Positions &::= \text{`(`}\ Int\ \text{`,'}\ Int\ \text{`)'} \\
&\mid \text{`(`}\ Int\ \text{`,'}\ Int\ \text{`)'}\ \text{`,'}\ Positions
\end{aligned}
$$

3 Manual del programa

3.1 Instalación

Un requisito previo para utilizar el programa es tener instalado *stack*[3]. Luego, los pasos a seguir son:

- Descargar el código fuente del proyecto:

```
$ git clone https://github.com/Tomasfdel/Clairvoyance.git
```

- Acceder a su carpeta:

```
$ cd Clairvoyance/
```

- Inicializar *stack*:

```
$ stack setup
```

- Ejecutar el script que genera los parsers:

```
$ ./generateParsers.sh
```

- Construir el entorno:

```
$ stack build
```

3.2 Uso

Para utilizar el programa se debe ejecutarlo pasándole como argumento un archivo fuente con las definiciones del encuentro:

```
$ stack exec Clairvoyance-exe <archivo_fuente>
```

Una vez que se interpretó un archivo fuente correcto, el programa imprime el orden de iniciativa de las unidades en el combate junto con una representación simple del estado inicial del tablero.

[3] https://docs.haskellstack.org/en/stable/README/

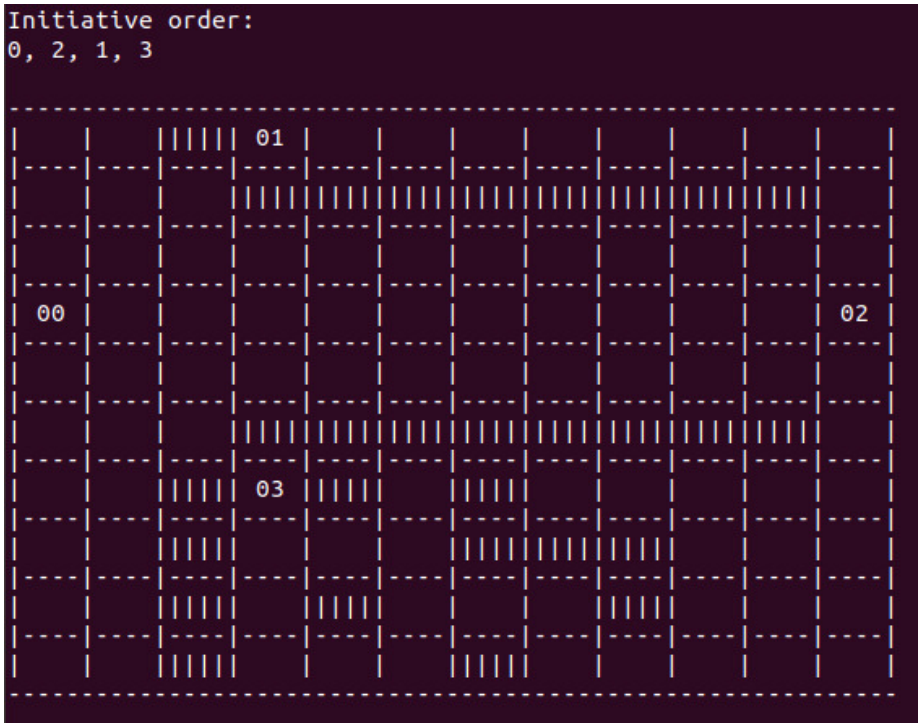

Figura 4: Ejemplo de estado inicial del programa.

Luego, se abrirá un intérprete de comandos que permitirá al *Dungeon Master* avanzar los turnos y modificar unidades conforme los jugadores jueguen sus turnos. Las tablas debajo explican los comandos con cada uno de los argumentos que se les puede pasar.

Comando	**Efecto**
next	Ejecuta el turno de la siguiente unidad.
show board *show UNIT*	Muestra en pantalla el tablero. Muestra en pantalla el bloque de estadístidas de la unidad *UNIT* (ver más abajo).
move UNIT (col, row) *move UNIT MOVEMENT*	Mueve la unidad *UNIT* a la coordenada *(col, row)*. Mueve la unidad *UNIT* según las direcciones dadas en *MOVEMENT*.
attack UNIT atk dmg	Realiza un ataque contra la unidad *unit* con un valor de ataque *atk* y un valor de daño *dmg*.
kill UNIT	Mata a la unidad *UNIT*.

Las formas de referenciar a una unidad en los comandos de arriba es mediante su índice único o por su tripla *equipo:nombre:identificador*. Por otro lado, *MOVEMENT* es una secuencia de direcciones de paso a una casilla adyacente, separados por guiones, que en conjunto definen el camino a tomar por la unidad. Las cadenas de texto para cada dirección son:

Cadena	Movimiento
U	Arriba
UR *RU*	Diagonal arriba-derecha
R	Derecha
DR *RD*	Diagonal abajo-derecha
D	Abajo
DL *LD*	Diagonal abajo-izquierda
L	Izquierda
UL *LU*	Diagonal arriba-izquierda

También se imprimirán en pantalla algunas líneas que relaten los eventos ocurridos cuando se ejecute un comando que pueda tener efecto sobre unidades y en los turnos de los personajes controlados por el programa.

4 Organización del código

El código fuente del proyecto está dividido en módulos dentro del directorio *src/*. Cada uno de ellos contiene los archivos que manejan alguna de las funcionalidades principales del programa. A continuación se da una breve descripción de los mismos:

- *AIActions*: Contiene la lógica de evaluación de los comportamientos de unidades controladas por el programa, además de los algoritmos de búsqueda de caminos, escape y determinación de líneas de visión.
- *Auxiliary*: Posee funciones extra que son utilizadas en varios de los demás módulos y tratan más con manejo de mónadas que funcionalidad del programa.
- *Commands*: Contiene el manejador del intérprete de comandos, así como los archivos necesarios para el parsing de los mismos.
- *FileParser*: Archivos utilizados por el parser de archivos fuente.
- *Game*: Contiene la lógica de la construcción del estado inicial del juego, el manejador de turnos y el código de *pretty printing* para los comandos de impresión.
- *parsing_sources*: Aquí se encuentran los archivos utilizados por Alex y Happy para generar los parsers de archivos fuente y de comandos.

5 Bibliografía y fuentes

[1] Wizards of the Coast, Inc. *Dungeons and Dragons Player's Handbook, Core Rulebook I v.3.5.* July 2003.

[2] *Alex User Guide.*
https://www.haskell.org/alex/doc/html/index.html

[3] *Happy User Guide.*
https://www.haskell.org/happy/doc/html/index.html

[4] Miran Lipovača. *Learn You a Haskell for Great Good!.*
http://learnyouahaskell.com/

[5] *Dijkstra Maps Visualized.*
http://www.roguebasin.com/index.php/Dijkstra_Maps_Visualized

[6] *Mutual Visibility Field Of View.*
http://www.roguebasin.com/index.php?title=Mutual_Visibility_Field_Of_View

[7] *Respuesta en StackOverflow sobre combinación de State y StateT.*
https://stackoverflow.com/questions/4138671/combining-statet-and-state-monads

[8] *Data.List documentation.*
https://hackage.haskell.org/package/base-4.15.0.0/docs/Data-List.html

[9] *Data.Vector documentation.*
https://hackage.haskell.org/package/vector-0.12.3.0/docs/Data-Vector.html

[10] *Data.Map documentation.*
https://hackage.haskell.org/package/containers-0.4.0.0/docs/Data-Map.html

[11] *Data.Set documentation.*
https://hackage.haskell.org/package/containers-0.6.5.1/docs/Data-Set.html

[12] *Data.PSQueue documentation.*
https://hackage.haskell.org/package/PSQueue-1.1.0.1/docs/Data-PSQueue.html

[13] *Data.Sequence documentation.*
https://hackage.haskell.org/package/containers-0.6.5.1/docs/Data-Sequence.html

[14] *State and StateT documentation.*
https://hackage.haskell.org/package/mtl-2.2.2/docs/Control-Monad-State-Lazy.html

A Reglas de juego

A.1 Funcionamiento del combate

El combate de *Dungeons and Dragons* opera de manera cíclica: cada unidad toma su turno siguiendo un orden preestablecido en un ciclo regular de rondas. Las acciones del juego son las siguientes:

- Antes de iniciar el combate, cada unidad hace una tirada de iniciativa.
- Los combatientes que están vivos realizan sus turnos siguiendo el orden de iniciativa previamente establecido, desde el que tuvo el resultado más alto al más bajo.
- Cuando todos han podido actuar, la unidad con la iniciativa más alta actúa otra vez, repitiendo este paso y el anterior hasta que sólo queden en pie unidades de un solo equipo.

A.2 Tablero de juego

El tablero consiste en una grilla cuadriculada, en la que cada casilla puede estar libre o ser una pared. Cada unidad que participa en el combate ocupa una casilla libre, la cual puede contener a lo sumo una unidad. Por su parte, ninguna unidad puede ocupar la casilla de una pared ni atravesarla como parte de sus movimientos.

A.3 Nomenclatura para tiradas de dados

Muchas de las acciones realizadas en el combate requieren una tirada de dados, cuyo resultado es luego usado para determinar el éxito o fracaso de las mismas. Es por eso que se utiliza una notación sintética para describir cómo calcular el resultado de una tirada de dados. La forma de notarlas es con la expresión *"N d C ± M"*, siendo N, C y M números naturales. Dicha notación significa *"tira N dados de C caras, suma los resultados de cada uno y al resultado súmale (o réstale) M"*. Por simpleza, en el caso en que M sea 0, puede omitirse el modificador de la expresión.

A.4 Bloque de estadísticas

Toda unidad tiene un conjunto de estadísticas que determina su efectividad en combate y cómo se ejecutan algunas de sus acciones. La siguiente tabla da una breve descripción de cada una.

Estadística	**Descripción**
Puntos de Vida (HP)	Cada vez que una unidad recibe daño, pierde esa cantidad en puntos de vida. Una unidad que quede en puntos de vida negativos está muerta.
Iniciativa	Modificador que se usa para la tirada de iniciativa.
Velocidad	Cantidad de casillas que puede recorrer la unidad en un movimiento.
Clase de Armadura (AC)	Indica qué tan dificil de golpear es la unidad. Un ataque hace daño únicamente si su valor de ataque es al menos la Clase de Armadura de su objetivo.
Ataque Estándar	Secuencia de descripciones de los ataques que puede realizar una unidad en su ataque estándar. Cada descripción contiene un rango, modificador de ataque y una expresión para la tirada de daño.
Ataque Completo	Secuencia de descripciones de los ataques que puede realizar una unidad en su acción de ataque completo.

A.5 Iniciativa

Para determinar el orden en el que los combatientes realizarán sus turnos, cada uno hace una tirada de iniciativa, la cual es *1d20 + modificador de iniciativa*. Luego, se ordenan de mayor a

menor por los resultados. En caso de que haya un empate entre dos o más unidades, se realiza una tirada nueva únicamente entre ellas para definir el orden en el que actuarán entre sí.

A.6 Acciones

Cada acción tiene un tipo, que puede ser de movimiento, estándar o de turno completo. Durante un turno, una unidad puede realizar una acción estándar y una acción de movimiento, dos acciones de movimiento, o una única acción de turno completo. Siempre pueden realizarse menos acciones de las permitidas en un turno, o directamente pasar sin hacer nada.

A.6.1 Acción de movimiento

Este tipo de acciones es lo que permite a las unidades cambiar la casilla que están ocupando, desplazándose a otra parte del tablero. Como parte de una acción de movimiento, una unidad puede moverse hasta tantas casillas como su velocidad lo indique.

Desplazarse a una casilla adyacente horizontal o verticalmente cuenta como un movimiento. Sin embargo, los movimientos a una casilla adyacente diagonalmente operan levemente distinto. El primer movimiento diagonal de una acción cuenta como una casilla, pero el segundo vale como dos, el tercero vuelve a valer uno, y así sucesivamente. Efectivamente, puede considerarse que cada movimiento diagonal vale $1,5$ casillas, y la distancia total recorrida en una acción es la parte entera de la suma de cada uno de los desplazamientos a una casilla adyacente realizados.

Otra condición para poder hacer un movimiento en diagonal es que las dos casillas mutuamente adyacentes entre el origen y el destino tienen que estar libres y desocupadas. Eso significa que no es posible moverse en diagonal "a través de una esquina" o "a través de una unidad".

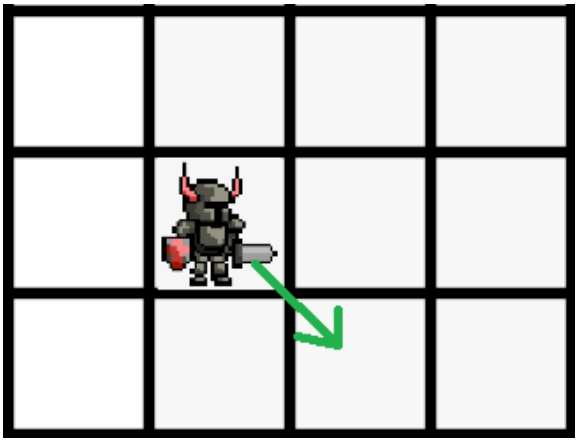

Figura 5: El movimiento diagonal es válido si las dos casillas adyacentes están libres.

Figura 6: Si una pared o unidad ocupa una casilla adyacente, el movimiento diagonal no es váilido.

A.6.2 Acción estándar

Una unidad puede ejecutar todos los ataques de su Ataque Estándar como una acción estándar. Para cada ataque individual, se realiza una tirada de ataque, la cual es *1d20 + modificador de ataque*. Si el resultado es al menos la Clase de Armadura de su objetivo, entonces el ataque impacta e inflige daño. Entonces, se realiza la tirada de daño que haya en la descripción y su resultado se resta de los Puntos de Vida del objetivo. Para que una unidad sea un objetivo válido de un ataque, debe estar dentro del rango del ataque y estar en línea de visión del atacante.

Cada ataque tiene un rango asociado, que indica las casillas en las que debe estar un objetivo válido del mismo. El rango puede ser *cuerpo a cuerpo*, que corresponde a las ocho casillas adyacentes al atacante, o *a rango N*, que son todas las casillas a distancia a lo sumo N del atacante.

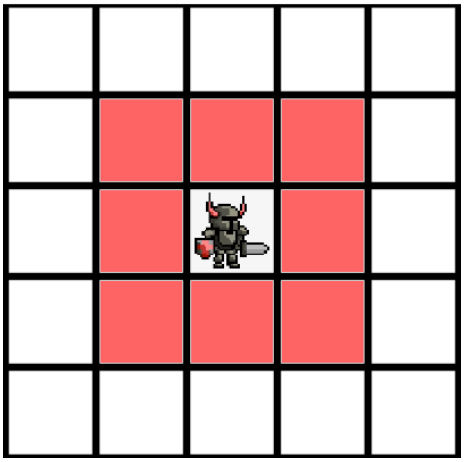

Figura 7: Casillas a rango cuerpo a cuerpo.

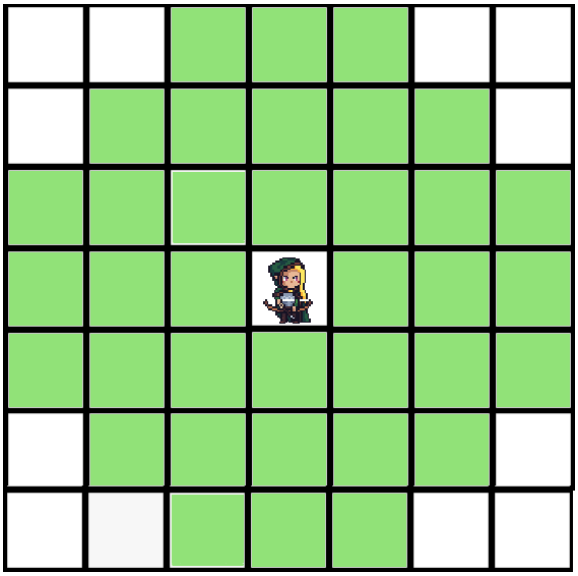

Figura 8: Casillas a rango 3.

Que dos unidades estén en línea de visión una de la otra significa que ambas pueden verse de manera directa sin obstáculos en el camino. Para definir si dos unidades comparten línea de visión, se trazan rectas imaginarias desde cada esquina de la casilla donde está una a cada esquina de la casilla de la otra. Si al menos una recta no toca ninguna casilla de una pared, entonces existe línea de visión entre las mismas. Notar que sólo las paredes obstruyen línea de visión, no las unidades.

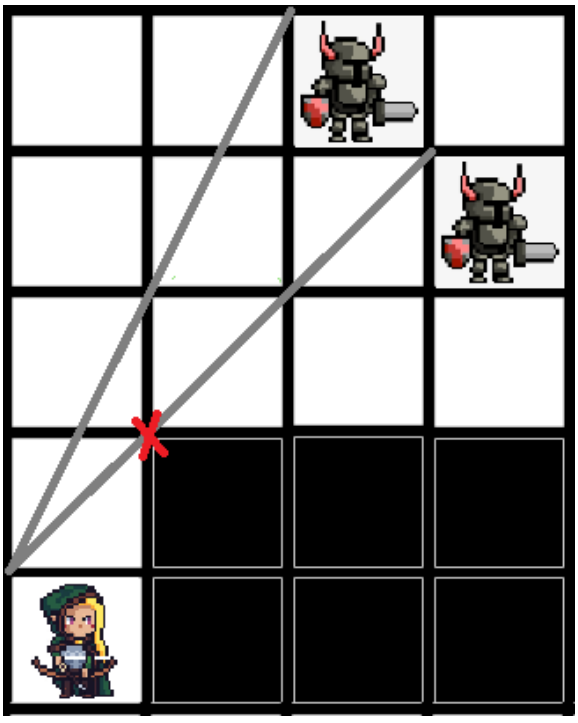

Figura 9: El caballero de la izquierda está en línea de visión de la arquera. El de la derecha, no.

A.6.3 Acción de turno completo

La última opción que tiene disponible una unidad es la de prescindir sus acciones de movimiento y estándar por una acción de turno completo. Realizar un Ataque Completo es un ejemplo de ello. Todas las reglas sobre ataques descriptas en la sección anterior siguen aplicando a un ataque completo, aunque la entrada de Ataque Completo de una unidad suele ser más poderosa que su Ataque Estándar.