AJEDREZ POR CONSOLA EN HASKELL

uvus: <*jescelgar*>

Nombre y Apellidos: *Jesús Celada García*

Correo electrónico: *jescelgar@alum.us.es*

Índice

1.PRESENTACIÓN

Para el presente trabajo he decidido realizar un ajedrez por consola en Haskell. El ajedrez es un juego por turnos de dos jugadores que consiste en eliminar al rey de tu oponente. Hay distintas maneras para jugar al ajedrez. Todas las maneras difieren principalmente en el tiempo de la partida. En mi caso he optado por codificar el ajedrez clásico sin tiempo límite de tiempo.

Para llevar a cabo nuestro propósito, primero debemos de diseñar ciertos elementos conceptuales para empezar a construir el código. En primer lugar, definimos el tablero como una matriz de 8x8 donde desde las perspectivas de las blancas la casilla de abajo a la izquierda será el (1,1) y la de arriba a la derecha será el (8,8). Esta división se detalla en la Figura 1.

Luego, representaremos con la inicial de la pieza en inglés cada pieza en el tablero donde las blancas irán en mayúsculas y las negras en minúscula. Además, representaremos las casillas no ocupadas con el siguiente carácter '·'. En cada turno el tablero se dará la vuelta para que cada vez que le toque mover a un jugador vea el tablero como si estuviese sentado enfrente de su rival. Para denotar los inputs de los usuarios tomaremos como referencia la notación algebraica la cual es la notación más popular en los movimientos en ajedrez. En nuestro caso, en lugar de indicar la casilla de destino y la pieza que se mueve, indicaremos la casilla de origen y de destino. Esta decisión es tomada para evitar confusiones entre las distintas piezas. Primero se indica la columna en la que se encuentra usando una letra de la 'A' a la 'H' y luego se indica la fila correspondiente que va del 1 al 8. Por ejemplo, el input 'd2d3' mueve la pieza que esté en la casilla d2 a la casilla d3 siempre y cuando este movimiento sea válido. Esto se detallará más a continuación. El usuario tendrá una ayuda visual en todo momento para que le sea más fácil identificar las casillas y no se hará distinción entre mayúscula y minúscula. El resultado final de como se interpreta el tablero viene en la figura 2.

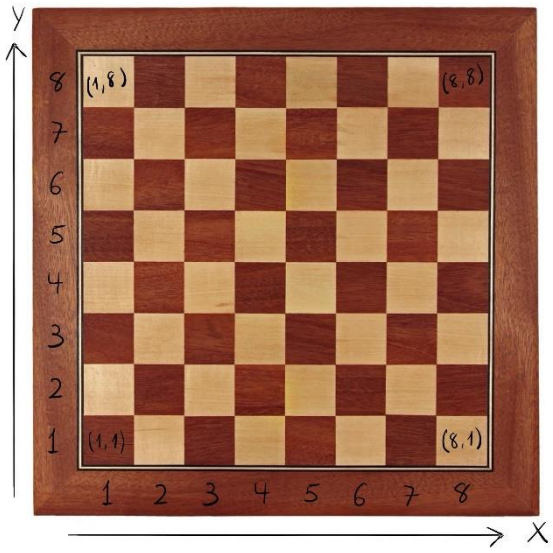

Figura 1: división teórica del tablero

```
8 |  r  n  b  q  k  b  n  r
7 |  p  p  p  p  p  p  p  p
6 |  ·  ·  ·  ·  ·  ·  ·  ·
5 |  ·  ·  ·  ·  ·  ·  ·  ·
4 |  ·  ·  ·  ·  ·  ·  ·  ·
3 |  ·  ·  ·  ·  ·  ·  ·  ·
2 |  P  P  P  P  P  P  P  P
1 |  R  N  B  Q  K  B  N  R
--+------------------------
  |  A  B  C  D  E  F  G  H
```

Figura 2: tablero final por consola (blancas)

Por otro lado, el usuario tiene la opción de consultar los posibles movimientos que puede hacer con la pieza elegida, simplemente introduciendo la casilla donde se encuentra. A su vez, el usuario tiene una opción para deshacer un movimiento previamente realizado pulsando la tecla 'ENTER'. Por último, el usuario también dispone de una opción para salir del juego pulsando 'e' o 'E'.

El principal reto a la hora de programar el código se trata de codificar las reglas del juego. Estas son llevadas a cabos en diferentes funciones como se detallará a continuación.

2.ESTRUCTURA, DETALLES DEL CÓDIGO Y CRITERIOS MÍNIMOS

Para llevar a cabo el juego he realizado dos archivos de código: Chess.hs y Main.hs. En el archivo Chess.hs definimos las funciones y elementos básicos para luego poder llevar al cabo la lógica del juego mientras que en el Main.hs importamos las funciones y definimos la función que ejecutará el juego correctamente.

Primero detallamos el contenido del archivo Chess.hs. En primer lugar, definimos los tipos que van a servir de base para representar todo el juego. Aquí tenemos las declaraciones de tipo de las casillas, y del tablero. Primeramente, pensé en usar una matriz para representar el tablero, pero luego me di cuenta de que no era necesario representar los espacios en blanco del tablero y que usando un diccionario tipo Map con clave la casilla y valor la pieza era más que suficiente. Además, las ventajas que tenemos de usar Map antes que una lista normal por ejemplo es que las búsquedas son más eficientes. Las casillas por otra parte serán definidas como detallamos en la presentación, donde indicaremos primera la columna y luego la fila que pertenece,

A continuación, pasamos a declarar nuevos tipos, mas concretamente el color, las piezas y el estado actual del juego. El color (*Colour*) será asociado a las piezas para saber a qué jugador corresponde cual y solo tendrá dos valores: *'White'* y *'Black'*. Las piezas (*Piece*) además de las clásicas del ajedrez incluyen '*Blank*' que representa lo que hay en una casilla vacía. Todas tienen asociado un color excepto el espacio en blanco. A su vez tenemos el estado actual de la partida (*CurrentGameState*) que será crucial a la hora de jugar. Este tipo representa la partida y nos indica el color del jugador al que le toca mover, el tablero actual y una pila con todos los "estados" anteriores al actual de la partida. Este es el tipo que exportaremos al Main.hs y funcionará como un tipo de dato abstracto ya que también exportaremos sus funciones.

Seguidamente, encontramos las funciones que exportaremos al Main para poder llevar a cabo el juego. Primero encontramos la función estado inicial (*initialState)* que inicializa un dato tipo *CurrentGameState,* que es el input de la función principal para empezar una partida de cero clásica. Además, defino otros tres estados más de la partida como son *drawByStaleMateState*, *checkMateState* y *pawnPromotionState* para que el usuario pueda probar distintas funcionalidades del juego. A continuación, tenemos la función *boardRender* que se encarga de transformar la información del programa en un tablero por consola del tipo explicado en la presentación para que el usuario pueda correctamente visualizar la partida. Esta función se construye principalmente utilizando la función *SquareRender* que traduce la información del tablero en el output y que es recursiva sin acumulador. A su vez, tenemos la función *renderPossiblesMoves* que dado un estado actual y una casilla calcula los posibles movimientos que esa pieza puede hacer. Esta función, que está definida por guardas, la utilizaremos para cuando el usuario

pregunte los posibles movimientos de una pieza. Luego, tenemos la función *makeMove* la cual es una función indispensable en nuestro juego ya que dado un estado actual una casilla de origen y una de destino ejecuta el movimiento que se le indica y devuelve el nuevo estado del juego. *makeMove* no posee internamente las comprobaciones necesarias para saber si el movimiento indicado es válido o no, esto se comprueba externamente en la función principal del juego antes de pasarle el input a *makeMove* usando la función *isValidMovement*. Esta función recibe lo mismo que *makeMove* y devuelve True si el movimiento es posible o no. La validación de las reglas se comprueba mediante *pieceLegalMoves* en dos partes. Por una parte, con *possibleMoves* nos aseguramos de que ninguna pieza haga un movimiento inválido del tipo mover fuera del tablero, saltarse una pieza sin ser posible, mover a una casilla ocupada con tu pieza o comer ilegalmente una pieza. A esto durante el código lo llamaremos los movimientos potenciales. Por otra parte, filtramos lo que llamaremos los movimientos legales que son aquellos movimientos que no dejen en jaque a tu rey. Luego comprobamos que el movimiento propuesto esté dentro de los posibles movimientos y así es como nos aseguramos de que solo se puedan realizar movimientos válidos. Todo esto puede verse ilustrado en las figuras 3, 4 y 5.

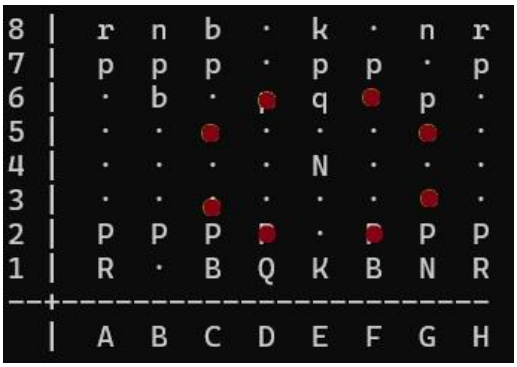

Figura 3: potenciales movimientos del caballo e4

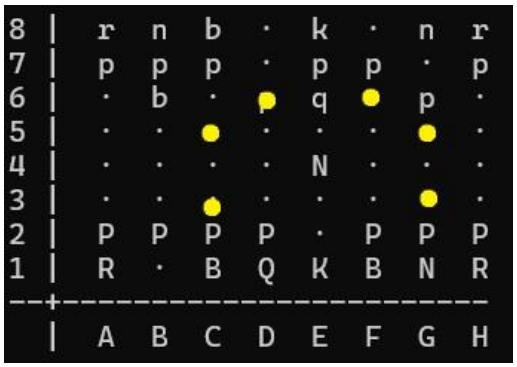

Figura 4: posibles movimientos del caballo e4

(caso ideal)

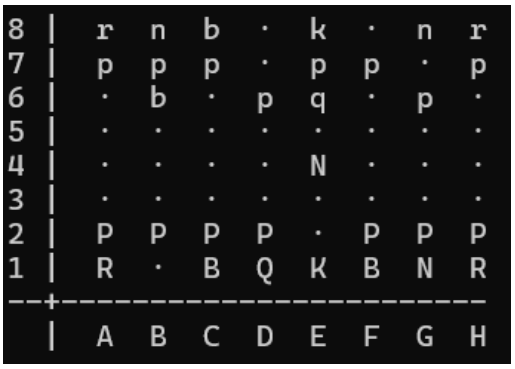

Figura 5: movimientos legales del caballo e4

El juego proporciona también la posibilidad de deshacer un movimiento. Esto se lleva a cabo con la función *undoLastMove* la cual devuelve el ultimo *CurrentGameState* guardado en la pila. Este estado contiene el estado anterior al completo con lo que no hace falta hacer nada más. A continuación, encontramos unas funciones que son las que nos indican

si la partida ha finalizado o no. Las dos formas de que termine la partida es que o un jugador caiga en jaque mate o que un jugador caiga en rey ahogado. Son cosas similares ya que en las dos al jugador no le quedan movimientos legales, pero tiene las diferencia de que si el jugador está en jaque es jaque mate y por tanto gana el oponente, y si por el contrario no está en jaque la partida queda en tablas. Estas cosas se comprueban con las funciones *playerInCheckMate* y *drawByStaleMate*. Por último, tenemos dos funciones las cuales servirán para coronar los peones. La primera, *isPawnPromoting,* comprueba que haya algún peón coronando. Como esto se comprueba en todos los movimientos, como mucho esta función va a encontrar un peón coronando ya que se mueve una pieza por turno. La segunda, *promotePawn*, se utiliza si la primera es verdadera y cambia el peón en el tablero por la pieza que le indique el usuario entre las posibles que son Reina, Torre, Alfil y Caballo.

Una vez acabamos con las funciones que se exportarán y se utilizarán en el Main encontramos todas las funciones auxiliares sobre las cuales se han apoyado todas las anteriores. Sobre estas funciones me voy a detener a explicar aquellas que generan los posibles movimientos de las piezas sin tener en cuenta si ese movimiento deja en jaque al rey o no. En primer lugar, tenemos *pawnMoves* la cual genera los posibles movimientos de los peones. Básicamente, esta función comprueba cuatro movimientos: mover una casilla hacia delante, mover dos casillas hacia delante, comer hacia la izquierda y comer hacia la derecha. Como las piezas blancas y las piezas negras están enfrentadas todas las comprobaciones relativas que se hacen sobre los peones de un color se pueden aprovechar para hacerlas sobre el otro color simplemente multiplicando el factor por (-1) sobre el movimiento relativo lo que indica que es en sentido contrario. Lo mismo tendremos que hacer con los movimientos resultantes. En segundo lugar, tenemos *knightMoves* y *knightMoves'* que calculan los posibles movimientos de los Caballos. Para ello calcula en primera instancia todos los posibles movimientos ideales y luego utiliza una recursión donde se usan las guardas para comprobar que estos movimientos cumplen las reglas del juego. En esta función también se utiliza la función de Data.List *head*. A continuación, tenemos los posibles movimientos de los alfiles *bishopMoves* y de las torres *rookMoves*. Estas dos piezas comparten una lógica similar a la hora de moverse por eso he decidido incluir una función común que la computase. La diferencia está en que los alfiles se mueven en diagonal y las torres en vertical y horizontal. En el caso del alfil, los casos ideales los genero con una lista por compresión y, además, en la función común llamada *bAndRMoves*, usamos funciones de orden superior como *map*, *break* y *filter*. En el caso de las torres utilizamos las funciones básicas del prelude *zip* y *repeat* para generar sus movimientos potenciales. Relacionado con estas dos tenemos los movimientos de la Reina *queenMoves* que básicamente puede actuar como un alfil y como una torre. En última instancia tenemos los movimientos del Rey *kingMoves* los cuales son todos aquellos movimientos que cumplan las normas y que sean adyacentes a la casilla de partida. Estos movimientos se calculan usando una lista por compresión donde se calculan los movimientos potenciales y se filtran.

Adicionalmente quiero comentar que en la función *getEnemyColour* encontramos un caso de uso de *case of* y también me gustaría añadir que cuando me refiero a estado me refiero al output del programa que luego cogerá la función principal como entrada. En Haskell no existen estados ya que todo son funciones y dados unos inputs devuelven unos outputs.

Pasamos a detallar el archivo Main.hs. Este archivo importa las funciones anteriormente explicadas del Chess.hs y las utiliza para crear el funcionamiento del juego. La función principal se maneja esencialmente con un solo input ya que en este podemos, según lo que pongamos, ejecutar todas las acciones posibles. Luego, internamente, existe otro input que solo se da cuando se corona un peón y se necesita saber por qué pieza la quiere cambiar el usuario. Además de todo esto, también incluyo aquí varias funciones que sirven para controlar el input, convertir el input y para borrar la pantalla. Tanto en las funciones de control de input como en las de conversión de este usamos patrones para controlar el numero de elementos que puede tener el input y para desglosar con mas facilidad este. A su vez, usamos funciones de la librería Data.Char que nos ayudan con estos propósitos. Por último, tenemos una función que borra la terminal para que en esta sea más fácil de ver la partida actual y no se emborrone con las anteriores. Esta función vale para múltiples sistemas operativos como Linux, MacOS y Windows ya que la función *os* de System adapta el output a cada caso. Esta función no sabía como hacerla y la he sacado de internet, mas concretamente la ha generado la IA ChatGPT.

Para terminar, me gustaría comentar ciertas cosas a añadir que podrían mejorar el juego. La primera es que no está codificado el enroque. EL enroque es un movimiento de ajedrez que cumpliendo unas determinadas condiciones te permite hacer un doble movimiento entre tu torre y tu rey. La segunda es que cuando la partida se termina, a los jugadores no les aparece un mensaje detallando el motivo de finalización de la partida y quien ha ganado a quien (si es que esto ocurre). Cabe recalcar también que cuando se llegan a estas situaciones en ajedrez los dos jugadores suelen saber con bastante claridad los motivos de finalización de la partida.

3. EJECUCIÓN

Para ejecutar el programa es muy sencillo:

1. Cargar el archivo Main.hs

```
ghci> :l Main.hs
[1 of 4] Compiling PilaTA           ( PilaTA.hs, interpreted )
[2 of 4] Compiling Chess            ( Chess.hs, interpreted )
[3 of 4] Compiling Main             ( Main.hs, interpreted )
Ok, three modules loaded.
```

Figura 6: ejemplo de carga del archivo Main.hs

2. Ejecutar la partida clásica con el comando ‘main’

```
ghci> main

8 |  r  n  b  q  k  b  n  r
7 |  p  p  p  p  p  p  p  p
6 |  .  .  .  .  .  .  .  .
5 |  .  .  .  .  .  .  .  .
4 |  .  .  .  .  .  .  .  .
3 |  .  .  .  .  .  .  .  .
2 |  P  P  P  P  P  P  P  P
1 |  R  N  B  Q  K  B  N  R
--+------------------------
  |  A  B  C  D  E  F  G  H

Si desea saber los posibles movimientos de alguna pieza introduzca la casilla la cual se encuentra la pieza [ln]
Si desea hacer un movimiento indica la casilla de partida y la casilla final que desea modificar [lnln]
Si desea deshacer el ultimo movimiento pulse ENTER
Si desea salir teclee [e]
Turno de White:
```

Figura 7: ejemplo ejecución partida clásica

Y ya aquí intentar mover una ficha (ej. ‘d2d3’), consultar los movimientos de una ficha (ej. ‘d2’). Cuando lleves algún movimiento podrás también deshacer algún movimiento pulsando ‘ENTER’ o salir de la partida pulsando ‘e’/’E’

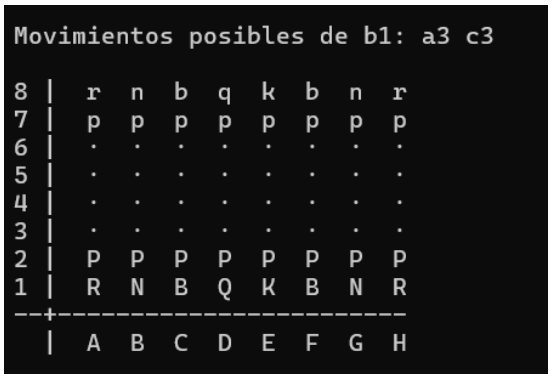

Figura 8: ejemplo posibles movimientos

3. Ejecutar el ejemplo de rey ahogado con el comando ‘main2’ y luego ‘g1g7’

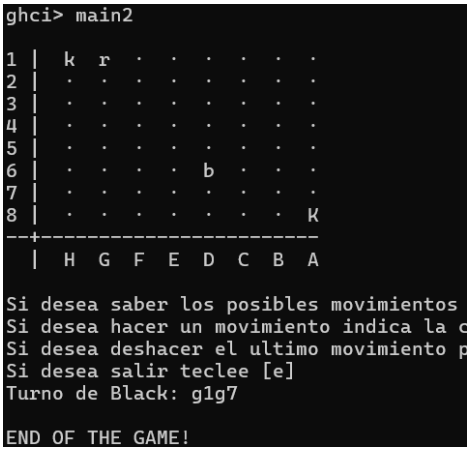

Figura 9: ejemplo rey ahogado

4. Ejecutar el ejemplo de jaque mate con el comando ‘main3’ y luego ‘g1a1’

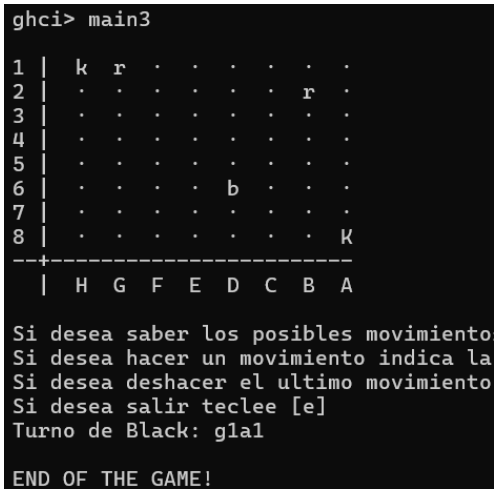

Figura 10: ejemplo jaque mate

5. Ejecutar el ejemplo para coronar un peón con el comando ‘main4’ y luego ‘a7a8’

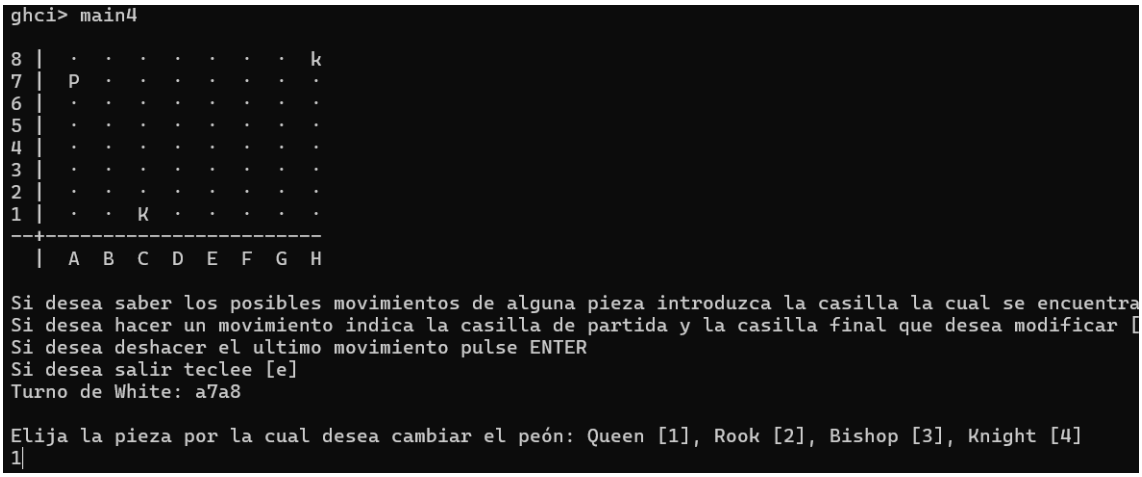

Figura 11: ejemplo de coronación de peón

4. LIBRERÍAS EXTERNAS

Como librerías externas no usado ninguna que haya que importar desde fuera. Únicamente he usado el módulo de la asignatura Pila TA para poder incluir una pila en mi trabajo, pero este archivo ya viene incluido en el zip. Por último, he usado el tipo *Maybe* que sirve para encapsular valores de funciones de los cuales no se puede asegurar una salida.