Proyecto Final de ALPII

Ezequiel Postan

Contents

1 Introducción

Este es el informe del proyecto final de ALPII **FChess** (**F**unctional**Chess**). El informe consta en:

- Descripción del programa entregado
- Aspectos de diseño
- Algoritmos
- Estructura del código
- Descripción general del código
- Relfexiones
- Instrucciones de Compilación

2 Descripción

FChass es un programa de consola que permite al usuario jugar un juego de ajedrez contra la máquina.

El juego cumple con todas las reglas de movimientos del juego de ajedrez exceptuando la captura "*en passant*". Las jugadas se introducen por medio del teclado.

Cada casilla tiene asignada una fila y una columna. Por ejemplo, avanzar el peón rey blanco dos casillas desde su posición inicial se denota "e2e4". Para coronación se escribe al final de la jugada una letra extra ("d" para dama, "t" para torre, "a" para alfil y "c" para caballo), por ejemplo "a7a8d" corona el peón de la columna "A" por una dama.

El enroque se denota moviendo el rey a su posición final. Por ejemplo "e1g1" para el enroque corto de blancas.

El programa ignorará toda entrada no válida. Para ingresar la entrada se escribe la jugada como se explicó anteriormente y se oprime "Enter"

La idea del proyecto surgió del interés de sobre el área de Inteligencia Artificial, originalmente pensé en implementar un juego d adversario más pequeño como el Ta-Te-Ti. Por cuestiones de tamaño el programa se extendió a un juego más extenso, ajedrez.

El programa entero fue diseñado desde cero. No hubo un programa modelo por diversos motivos. En primer lugar, la mayor parte de los juegos similares estaban escritos en otros lenguajes. Los dos programas que sí estaban en Haskell eran o incompletos (como el programa hsChess) o tenían implementaciones demasiado avanzadas (como Lambda Chess, implementaba una interfaz gráfica própia), ambos se encuentran en:

http://www.haskell.org/haskellwiki/Applications_and_libraries/Games

para su libre descarga.

Preferí diseñarlo completamente en lugar de usar el tiempo entendiendo otro programa existente. Si bien podría haber encontrado ideas para solucionar algunos problemas que tuve durante la implementación, fue motivador pensarlo por mi cuenta. Afortunadamente los resultados fueron aceptables.

3 Aspectos de diseño

El diseño presenta dos problemas fundamentales en cuanto a estructuras.

El primero es la estructura del tablero. El tablero es el corazón del programa como estructura. Debe ser de rápida actualización y lectura. Originalmente pensé en realizar una lista de listas de casillas (una intiutiva matriz). Sin embargo tras escribir algunas funciones que trabajaban sobre el tablero decidí que no era una buena estructura. Opté entonces por usar una lista de casillas, esta representación me pareció mucho más cómoda. Acceder a una casilla era simplemente indexar un elemento de la lista.

El segundo desafío era el entorno. Hay datos que deben llevarse a lo largo de la partida, el turno del jugador que mueve, la posición actual, algunos datos sobre las reglas del enroque. Opté por usar una tupla para estos datos.

Las estructuras solucionaron los problemas que se planteaban, pero creo que hoy lo haría distinto. (Ver sección 7 del informe)

4 Algoritmos

El algoritmo fundamental del juego es conocido MiniMax. La descripción a grandes rasgos del mismo fue tomda del paper **Why Functional Programming Matters** [John Hughes].

5 Estructura del código

El código se ha separado en diversos módulos según su funcionalidad. Entre estos se encuentran:

Main: Contiene la estructura central que modela el comportamiento del programa

Estructuras: Definiciones de todos los tipos de datos usados a lo largo del código y algunas funciones vinculadas a estos

Movimientos: Funciones de manipulación del tablero y entorno

Validacion: Conjunto de funciones que prmiten establecer si una jugada ingresada por el usuario es válida

Maquina: Funciones que generan los movimientos de la computadora

Parser: Parser de entrada

6 Descripción general del código

Esta sección describe las funciones de cada módulo.

6.1 Estructuras

Se definen los tipos de datos usados.

Declaración	Descripción
data Color = White | Black | Grey	Indica si una casilla contiene una pieza blanca, negra o está vacía
data Square = Empty |WP |WN |WB |WR |WQ |WK |BP |BN |BB |BR |BQ |BK	Representa las casillas del tablero.
type Board = [Square]	Representa el tablero
type Pos = Int	Representa una posición en el tablero
type Move = (Pos,Pos,Square)	Representa un movimiento
type Env = (Color , Board , Int , Int)	Información durante el juego, turno, posición, estado de enroque
data GenTree a = Gen a [GenTree a]	Para generar un árbol de jugadas

También se definen:

- **prettyBoard :: Color -> Env -> String** Se encarga de convertir el tablero en una cadena para mostrar al jugador tras realizar cada movimiento.
- **initPos :: Board:** El tablero en su posición inicial

```
initPos = [WR , WN , WB , WQ , WK , WB , WN , WR,
           WP , WP , WP , WP , WP , WP , WP , WP,
           Empty, Empty, Empty, Empty, Empty, Empty, Empty, Empty,
           Empty, Empty, Empty, Empty, Empty, Empty, Empty, Empty,
           Empty, Empty, Empty, Empty, Empty, Empty, Empty, Empty,
           Empty, Empty, Empty, Empty, Empty, Empty, Empty, Empty,
           BP , BP , BP , BP , BP , BP , BP , BP,
           BR , BN , BB , BQ , BK , BB , BN , BR
          ]
```

- **initEnv :: Env**: El entorno inicial

```
initEnv = (White, initPos, 0, 0)
```

6.2 Main

Main simplemente coordina las funciones del programa. Las funciones principales son:

- **presentacion :: IO ()**: Imprime un mensaje de inicio
- **config :: IO Color**: Da a elegir con qué color jugar
- **jugar :: Color -> IO ()**: Inicia el juego
- **humanMove :: Env -> IO Env**: Lee un movimiento del teclado y verifica si és válido
- **humano :: Color -> Env -> IO ()**: Realiza un movimiento dado por el jugador, verifica si el juego terminó y llama al siguiente movimiento de la máquina
- **maquina :: Color -> Env -> IO ()**: Genera un movimiento de la máquina y si no se llegó al final del juego llama al siguiente movimiento del adversario

6.3 Movimientos

Funciones vinculadas a la manipulación y actualización del tablero y entorno después de cada jugada

- **toColor :: Char -> Color**: Transforma una caracter previamente analizado en un color
- **toPos :: String -> Pos**: Transforma una cadena previamente analizada en una posición del tablero
- **toMove :: String -> Move**: Transforma una cadena previamente analizada en un movimiento

- **getSquare :: Pos -> Board -> Square**: Dada una posición y un tablero devuelve el contenido de la casilla correspondiente
- **setSquare :: Square -> Pos -> Board -> Board**: Coloca una pieza en una casilla de un tablero dado
- **delSquare :: Pos -> Board -> Board**: Setea en vacía una casilla dada de un tablero
- **updateBoard :: Move -> Board -> Board**: Actualiza el tablero, no analiza la validez de las jugadas
- **opPiece :: Square -> Square**: Dada una pieza blanca retorna la equivalente en el bando opuesto. Esta función se usa sólo para coronación
- **color :: Square -> Color**: Nos da el color de la pieza en una casilla
- **opColor :: Color -> Color**: Dado un color, nos da el opuesto
- **move' :: Env -> Move -> Env**: Actualiza el tablero y el estado de los enroques. También cambia el color del turno
- **move :: Env -> Move -> IO Env**: Encapsula lo que hace "move"' en la mónada IO
- **castleHandler :: Color -> Pos -> Int**: Funci¿ón auxiliar para manejar los enroques
- **updateCastle :: Int -> Int -> Int**: Actualiza el estado de los enroques. Toma el handler del último movimiento generado por castleHandler, el estado actual y devuelve el resultado

6.4 Validacion

Este módulo agrupa las funciones necesarias para verificar si un movimiento parseado es válido.

- **rightColor :: Color -> Square -> Bool**: Indica si el color de una pieza a mover coincide con el color del jugador que mueve
- **possible :: Square -> Pos -> [[Pos]]**: possible nos da el conjunto de casillas a las que se puede mover una pieza desde una casilla dada en el tablero vacío
  - Posible genera las posiciones en base a la estructura que usamos para el tablero. Ver archivo "Tablero.txt" adjunto
- **oneRow :: Pos -> [Pos] -> Bool**: Verifica la validez de los movimientos de peón
- **isPossible :: [[Pos]] -> Pos -> Bool**: Permite verificar que una pieza puede moverse a una casilla en un tablero vacío
- **findKing :: Color -> Board -> Pos**: Da la posición del rey de un jugador
- **check :: Color -> Env -> Bool**: Dice si el jugador dado está en jaque
- **hayRey :: Color -> Board -> Bool**: Verifica que no se haya tomado el rey
- **aux :: Color -> [[Square]] -> [[Square]] -> Bool**: Función auxiliar para verificar que un jugador está en jaque
- **firstRorQ, firstBorQ , pawn :: Color -> [Square] -> Bool**: Son otras funciones auxiliares para verificar jaque
- **inCheck :: Move -> Env -> Bool**: Toma el movimiento del jugador que movió y verifica que no se coloque a si mismo en jaque
- **freeWay :: Pos -> [[Pos]] -> Board -> Bool**: Verifica que una pieza no salte sobre otras en una jugada
- **isValid :: Env -> Move -> Bool**: Evalúa la validez de un movimiento
- **notKing :: Square -> Bool**: Función auxiliar, verifica que no se intente tomar el rey rival

6.5 Máquina

Agrupa funciones correspondientes a la gneración de los movimientos de la computadora.

- **checkMate :: Env -> Bool**: Verifica si terminó el juego en jaque mate
- **staleMate :: Env -> Bool**: Verifica si terminó el juego en ahogado (tablas)
- **value :: Square -> Int**: Otorga un valor a cada pieza
- **eval :: Board -> Int**: Le da un puntaje a un tablero dado
- **mate, material, center, others, develop :: Board -> Int**: Son funciones que asignan valores al tablero según diversos aspectos
- **machineMove :: Env -> IO Env**: Genera el siguiente movimiento de la máquina
- **moveGen :: Int -> Board -> Env -> [Move]** : Genera todos los movimientos habilitados en el turno actual
- **moveList :: Env -> [(Move,Int)]**: Genera una lista con todos los movimientos habilitados en el turno actual y el valor del tablero tras realizar cada movimiento
- **pieceGen :: Square -> Pos -> Env -> [Move]**: Genera todos los movimientos habilitados de una pieza en el turno actual
- **moveTree :: Int -> Env -> [GenTree (Move,Int)]**: Crea un árbol con todos los movimientos y respuestas a partir de un estado y la puntuación de cada movimiento hasta una profundidad dada
- **depth :: Int**: Profundidad del árbol de generación de jugadas
- **selectMove :: Env -> [GenTree (Move,Int)] -> (Move,Int)**: Selecciona un movimiento entre los mejores posibles que se han generado
- **nextMove :: Color -> [GenTree (Move,Int)] -> [(Move,Int)]**: Genera la lista de los mejores movimientos habilitados
- **minimax :: Color -> GenTree (Move,Int) -> (Move,Int)**: Teniendo un subárbol de respuestas aplica el algoritmo minimax y calcula el mejor valor del movimiento raíz considerando las mejores respuestas del adversario
- **best :: Color -> (Move,Int) -> [(Move,Int)] -> [(Move,Int)]**: Obtiene la lista de los mejores movimientos

6.6 Parser

Se definen los parsers usados para analizar la entrada tomada del teclado.

- **readMove :: Parser String**: Analiza si la entrada es un movimiento
- **side :: Parser Char**: Analiza si la entrada es una opción válida

7 Relfexiones: si lo empezara hoy...

Esta sección es una reflexión sobre el diseño y resultado.

Cosas que haría distinto de tener que hacer el proyecto de nuevo.

Usaría menos patern matching y crearía una abstracción sobre mis estructuras de datos.

Creo que ese fue mi error fundamental, después de medir rendimiento con profiling comprobé que gran parte del tiempo se pierde por el rendimiento de las estructuras que elegí (tiempo de acceso en listas por ejemplo) y cambiar esta estructura lleva a cambiar casi todo el código. Más importante aún considero el error de no hacer este tipo de abstracción con el entorno que llevan algunas funciones del programa. Agregar ahora campos al entorno también lleva hacer grandes cambios. Creo preferible crear funciones que me den acceso a las partes del entorno en lugar de usar el pattern matching para esa labor.

8 Instrucciones de Compilación

Simplemente sitúese en la carpeta FChess y ejecute

```
ghc -O2 --make Main.hs -o FChess;rm *.o;rm *.hi
```

Ejecute el programa con

```
./FChess
```