Trabajo Práctico 1
Programación Funcional

Paradigmas de Lenguajes de Programación — 1^{er} cuat. 2011

Fecha de entrega: 19 de abril

1. Introducción

El objetivo de este trabajo es implementar un programa en Haskell capaz de jugar a una variante del juego de damas.

1.1. Reglas del juego

La variante del juego con la que trabajaremos se juega en un tablero de 8×8 escaques. Hay dos tipos de fichas: simples y reinas. Inicialmente, cada jugador tiene doce fichas simples de su color. La disposición inicial de las fichas en el tablero es la indicada en la Figura 1. Comienzan las blancas.

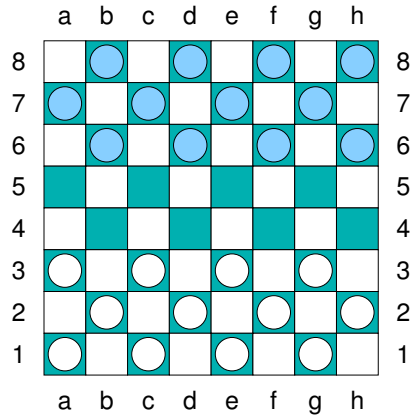
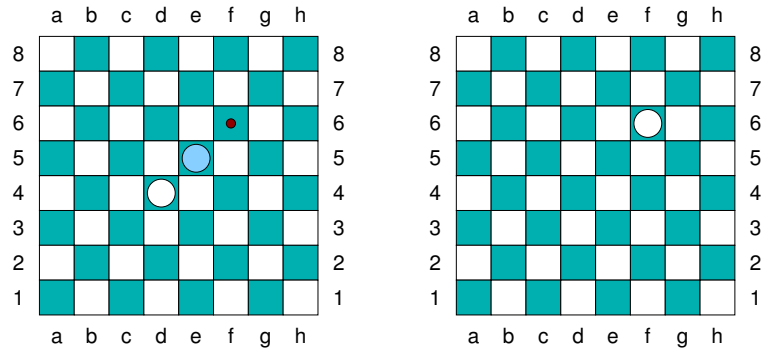


Figura 1: Disposición inicial

En cada turno, un jugador mueve una ficha:

- Las fichas simples mueven de a una casilla en dirección diagonal, siempre hacia adelante. Es decir, las blancas hacia la fila 8 y las negras hacia la fila 1.
- Las reinas mueven de a una casilla en cualquier dirección diagonal.

- Ambos tipos de fichas pueden capturar a una ficha adversaria, "saltando" por sobre ella. Las fichas simples sólo pueden capturar saltando hacia adelante. Las reinas pueden capturar saltando en cualquier dirección diagonal.



(a) Antes (b) Después

Figura 2: Captura de piezas mediante un salto

Si en un movimiento una ficha simple alcanza la primera fila del color adversario[1], automáticamente es coronada, convirtiéndose en reina. Gana el jugador que captura todas las fichas del adversario. Adicionalmente, si un jugador no puede mover en su turno, pierde.

Se remarcan algunas peculiaridades de esta variante:

- No se pueden encadenar movimientos.
- No hay obligación de comer, y por lo tanto no hay "soplo" de fichas.

2. Implementación

2.1. Tipos de datos utilizados

Se introducen los siguientes tipos para representar algunos conceptos del juego:

```
data Color = Blanca | Negra
data Ficha = Simple Color | Reina Color
type Posicion = (Char, Int)
data Tablero = T (Posicion → Maybe Ficha)
data Direccion = TL | TR | BR | BL
```

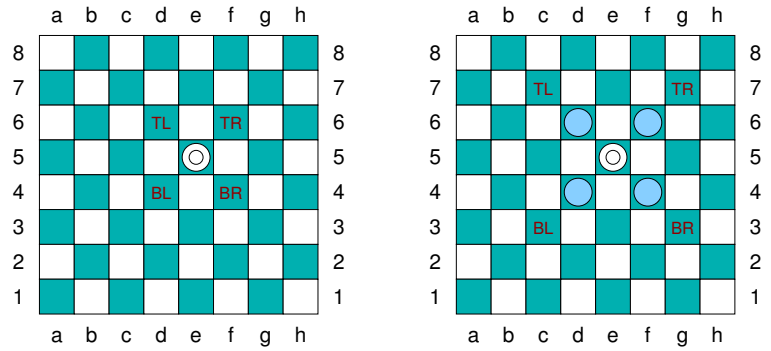
Una posición en el tablero se representa con un valor del tipo `Posicion`; por ejemplo `('b',5)` representa el escaque en la segunda columna y la quinta fila. Las posiciones válidas tienen su primera componente en el rango `['a'..'h']` y su segunda componente en el rango `[1..8]`.

Un tablero se representa mediante una función que, dada una posición, devuelve `Nothing` si el escaque en cuestión está vacío, o `Just ficha` si el escaque contiene una ficha.

Los valores de tipo `Direccion` representan posibles direcciones, de acuerdo con lo que se muestra en la Figura 3. Por ejemplo, `TL` representa la dirección "hacia arriba a la izquierda".

[1] La fila 8 para las blancas y la fila 1 para las negras

Cuando se utilicen direcciones para representar movimientos de las fichas, representarán un *movimiento simple* en esa dirección si el escaque vecino está vacío y un *salto* en esa dirección si el escaque vecino está ocupado por una pieza del adversario.



(a) Movimientos simples (b) Saltos (capturas)

Figura 3: Direcciones representadas por los valores del tipo Direccion

2.2. Módulo **Tablero**

En el módulo `Tablero.hs` se ubicarán las funciones relacionadas con el manejo del tablero. Se encuentran ya definidos la función show y el valor tableroInicial :: Tablero, que representa el tablero en la disposición de la Figura 1.

Ejercicio 1

Definir el valor vacio :: Tablero, que corresponda a un tablero sin fichas.

Ejercicio 2

Definir las funciones
contenido :: Posicion → Tablero → Maybe Ficha
poner :: Ficha → Posicion → Tablero → Tablero
sacar :: Posicion → Tablero → Tablero
de tal manera que contenido devuelva el contenido del tablero en la posición indicada, poner ubique una ficha y sacar quite una ficha de la posición indicada. En caso de haber una ficha en la posición dada, poner debe reemplazar el contenido de esa posición por la nueva ficha.

2.3. Módulo **Damas**

En el módulo `Damas.hs` se ubicarán las funciones relacionadas con las reglas del juego. Observar que el estado actual del juego queda unívocamente determinado por:

- el color del jugador al que le toca mover
- el estado del tablero

Y que, por otra parte, un *movimiento* en el juego queda unívocamente determinado por:

- la posición de la ficha que se mueve
- la dirección hacia la cual realiza el movimiento

Teniendo esto en cuenta, se definen los siguientes tipos para representar el estado actual del juego y un movimiento:

```
data Juego = J Color Tablero
data Movimiento =M Posicion Direccion
```

Ejercicio 3

Definir la función `mover :: Movimiento → Juego → Maybe Juego`
Si el movimiento es inválido, devuelve `Nothing`. Si el movimiento es válido, devuelve el estado del juego después de efectuar el movimiento dado. Para que un movimiento sea válido se debe controlar:

- Que las casillas de origen y destino no caigan fuera del tablero.
- Que haya una ficha en la casilla de origen.
- Que el color de la ficha movida coincida con el color del jugador al que le toca mover.
- Que la ficha se mueva en la dirección correcta, si no es una reina.
- Que la casilla de destino esté libre.
- En caso de tratarse de una captura, que la casilla intermedia contenga una ficha del color del adversario.

Si el movimiento es válido, observar la captura de fichas y la coronación de las fichas simples en reinas.

Ejercicio 4

Definir la función `movimientosPosibles :: Juego → [(Movimiento,Juego)]`
que devuelva la lista de todos los movimientos válidos que se pueden realizar en el juego dado, acompañados del estado que resulta al efectuar ese movimiento. □

Considerando los siguientes tipos:

```
data Arbol a = Nodo a [Arbol a]
type ArbolJugadas = Arbol ([Movimientos], Juego)
```

el módulo `Damas` ya provee la función: `arbolDeJugadas :: Juego → ArbolJugadas`
que devuelve el árbol de juegos alcanzables mediante movimientos válidos, comenzando desde el juego dado. Cada nodo del árbol consta de un juego, acompañado por la lista de movimientos que se deberían realizar desde la raíz del árbol para alcanzar dicho estado. (En la raíz del árbol, la lista de movimientos es vacía). Notar que el árbol generado por esta función es infinito.

Ejercicio 5

Dar el tipo y definir la función foldArbol que implemente un esquema de recursión *fold* para el tipo Arbol. Se permite utilizar recursión explícita para definir esta función.

Ejercicio 6

Definir la función podar :: Int → Arbol a → Arbol a
de tal forma que (podar n) sea la función que se queda con los primeros n niveles de un árbol, reemplazando a la lista de hijos por [] cuando se alcanza el enésimo nivel.

Definirla utilizando foldArbol, sin recursión explícita.

Sugerencia: definir primero la función podar' :: Arbol a → Int → Arbol a

Ejercicio 7

Considerando el tipo type Valuacion = Juego → Double, definir la función
mejorMovimiento :: Valuacion → ArbolJugadas → Movimiento
que, dada una función de valuación y un árbol de jugadas, devuelva el movimiento más conveniente para el jugador en la raíz.

Definirla utilizando foldArbol, sin recursión explícita. Utilizar el algoritmo minimax[2], con la función de valuación dada, para determinar el mejor movimiento. Asumir que el árbol de jugadas viene podado (y por lo tanto es finito).

Sugerencia: definir la función
minimax :: Valuacion → ArbolJugadas → (Double,[Movimiento])
que devuelva la mejor valuación obtenida por el algoritmo minimax, acompañada de la secuencia de movimientos que conducen desde la raíz del árbol hasta el estado del juego que tiene dicho valor.

Asumir que la función de valuación devuelve un número entre -1 y 1, que indica cuán favorable es el estado del juego para el jugador al que le toca mover. Si la función de valuación toma los valores -1 o 1 en el estado actual del juego, se debe devolver este valor, sin visitar los hijos del nodo.

Ejercicio 8

Definir la función ganador :: Juego → Maybe Color
que devuelva Just c si el color c es el ganador del partido y Nothing si todavía no se definió un ganador. Recordar que, si le toca el turno a un jugador que no puede mover ninguna ficha, ese jugador pierde.

Ejercicio 9

Definir la función valuacionDamas :: Valuacion
que devuelve un número entre -1 y 1, asignando un puntaje a un estado del juego, siendo -1 el valor más desfavorable y 1 el más favorable para el jugador al que le toca el turno

[2] http://en.wikipedia.org/wiki/Minimax

actualmente. La función de valuación está definida por:

$$\text{valuacionDamas juego} = \begin{cases} -1 & \text{si el jugador perdió} \\ 1 & \text{si el jugador ganó} \\ 2(2r+s)/(2r_{\text{TOT}}+s_{\text{TOT}})-1 & \text{si el juego continúa} \end{cases}$$

donde:

- r es la cantidad de reinas del jugador
- s es la cantidad de fichas simples del jugador
- r_{TOT} es la cantidad total de reinas en juego
- s_{TOT} es la cantidad total de fichas simples en juego

Funciones útiles

- Las funciones ord :: Char → Int y chr :: Int → Char definidas en el módulo estándar de Haskell Char convierten entre caracteres y enteros.
- El operador (/) tiene tipo Double → Double → Double. Para poder dividir números enteros y obtener un Double, utilizar la función fromIntegral:

  fromIntegral numer / fromIntegral denom

- Se provee un módulo `Interfaz.hs`. Una vez que todos los ejercicios del TP estén implementados, permite jugar a las damas interactivamente:

  ```
  Hugs> :l Interfaz.hs
  Interfaz> jugar Humano (Maquina 2) inicial
  ```

  La función jugar recibe tres parámetros:

  - Tipo de jugador que juega con las blancas.
  - Tipo de jugador que juega con las negras.
  - Estado inicial del juego.

  Los tipos de jugadores admitidos son Humano y (Maquina n), donde n es el número de niveles del árbol de jugadas que se analizarán con minimax.

Pautas de entrega

Se debe entregar el código impreso con la implementación de las funciones pedidas. Cada función debe contar con un comentario donde se explique su funcionamiento. Cada función asociada a los ejercicios debe contar con ejemplos que muestren que exhibe la funcionalidad solicitada. Además, se debe enviar un e-mail conteniendo el código fuente en Haskell a la dirección `plp-docentes@dc.uba.ar`. Dicho mail debe cumplir con el siguiente formato:

- El título debe ser "[PLP;TP-PF]" seguido inmediatamente del **nombre del grupo**.

- El código Haskell debe acompañar el e-mail y lo debe hacer en forma de **archivo adjunto** (puede adjuntarse un .zip o .tar.gz).

El código debe poder ser ejecutado en `Haskell98`. No es necesario entregar un informe sobre el trabajo, alcanza con que el código esté adecuadamente comentado.

Los objetivos a evaluar en la implementación de las funciones son:

- Corrección.
- Declaratividad.
- Reutilización de funciones previamente definidas (tener en cuenta tanto las funciones definidas en el enunciado como las definidas por ustedes mismos).
- Uso de funciones de alto orden, currificación y esquemas de recursión.
- Salvo donde se indique lo contrario, **no se permite utilizar recursión explícita**, dado que la idea del TP es aprender a aprovechar las características enumeradas en el ítem anterior. Se permite utilizar esquemas de recursión definidos en el preludio o pedidos como parte del TP. Pueden escribirse todas las funciones auxiliares que se requieran, pero estas no pueden ser recursivas (ni mutuamente recursivas).
- Pueden utilizar cualquier función definida en el preludio de Haskell y el módulo List.
- Se recomienda la codificación de tests. HUnit[3] permite hacerlo con facilidad.

  Importante: se admitirá un único envío, sin excepción alguna. Por favor planifiquen el trabajo para llegar a tiempo con la entrega.

[3] http://hunit.sourceforge.net/