Trabajo Práctico 1
Programación Funcional

Paradigmas de Lenguajes de Programación — 2^{do} cuat. 2011

Fecha de entrega: 8 de Septiembre

1. Introducción

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

1.1. Reglas del juego

La variante del juego con la que trabajaremos se juega en un tablero de 8×8 escaques. Este es un juego en el que participan dos jugadores: las blancas y las negras. Inicialmente, cada jugador tiene solamente dos fichas de su color. La disposición inicial de las fichas en el tablero es la indicada en la Figura 1. Comienzan las negras.

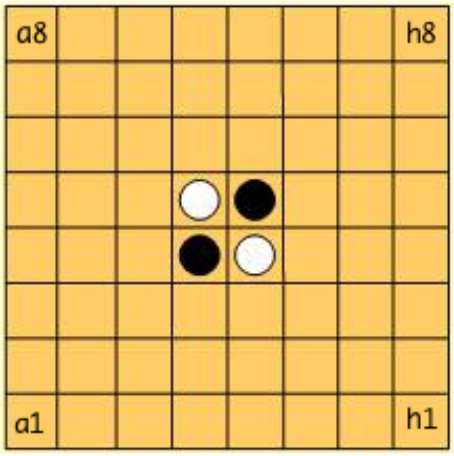
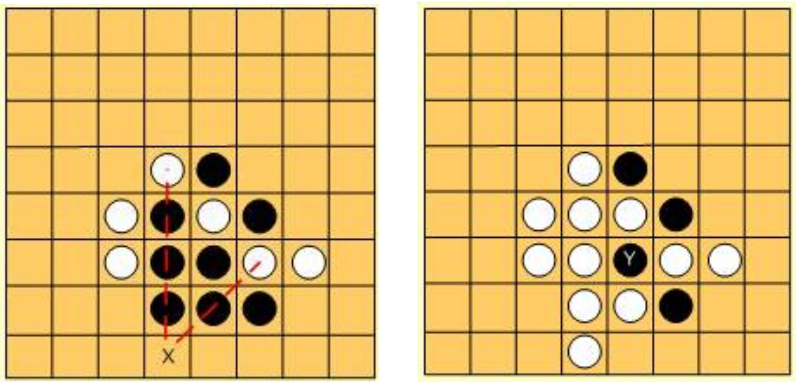


Figura 1: Disposición inicial

Una jugada válida es aquélla que permite rodear a al menos una pieza del oponente. Como resultado, las fichas del oponente que han sido rodeadas se transforman en piezas del jugador que ha efectuado la jugada. De no ser posible hacer una jugada de dichas características, el jugador deberá pasar. La Figura 2 ejemplifica una jugada válida. Nótese que las piezas pueden ser rodeadas tanto en forma horizontal, vertical y en diagonal.

El juego finaliza cuando ninguno de los jugadores puede mover, y el ganador es el que tiene más fichas.



(a) Antes (b) Después

Figura 2: Ejemplo de una movida válida

2. Implementación

2.1. Tipos de datos utilizados

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

```
data Color = Blanco | Negro
type Posicion = (Char, Int)
data Tablero = T (Posicion → Maybe Color)
```

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 color si el escaque contiene una ficha.

2.2. Módulo **Tablero**

En el módulo `Tablero.hs` se ubicarán las funciones relacionadas con el manejo del tablero. Se encuentra ya definida la función show.

Ejercicio 1

Definir los valores vacio :: Tablero y tableroInicial :: Tablero. Donde vacio corresponde a un tablero sin fichas y tableroInicial a uno en donde se situen fichas negras en las posiciones ('d',4) y ('e',5) y fichas blancas en las posiciones ('e',4) y ('d',5).

Ejercicio 2

Definir las funciones
contenido :: Posicion → Tablero → Maybe Color
poner :: Posicion → Color → Tablero → Tablero
de tal manera que contenido devuelva el contenido del tablero en la posición indicada y poner ubique una ficha en 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.

Ejercicio 3

Definir las siguientes funciones para recorrer posiciones.
`desplazarFila :: Int → Posicion → Posicion`
`desplazarColumna :: Int → Posicion → Posicion`

Ambas funciones reciben un parámetro entero que indica la tamaño del salto y una posición. Devuelven la posición desplazada según indica el nombre: Fila o Columna. Nota: no es necesario verificar que el resultado esté en rango.

Por ejemplo `desplazarColumna 1 ('c', 1)` devuelve `('d', 1)` y `desplazarFila −2 ('c', 4)` devuelve `('c', 2)`.

Ejercicio 4

Definir la función `generar :: Posicion → (Posicion → Posicion) → [Posicion]`
que dada una posición inicial y una función de desplazamiento, genera la lista de posiciones a las que se puede llegar aplicando la función de desplazamiento, hasta el final del tablero.

Por ejemplo:

`generar ('f', 5) (desplazarFila 1)` debería devolver
`[('f',5), ('f',6), ('f', 7), ('f', 8)]`

y por ejemplo:

`generar ('f', 5) (desplazarColumna 1).(desplazarFila 1)` debería devolver
`[('f',5), ('g',6), ('h', 7)]`

Ejercicio 5

Definir la función `posicionesAInvertir :: Posicion → Tablero → [Posicion]`
que dada una posición y un tablero, devuelve la lista de posiciones que deberían ser invertidas para finalizar la jugada. Asumir que el tablero tiene en la posición indicada la ficha "posicionada en la jugada que se está realizando".

Ejercicio 6

Definir la función `invertirTodas :: [Posicion] → Tablero → Tablero`
que dada una lista de posiciones y un tablero, devuelve el tablero que resulta de invertir todas las fichas en las posiciones indicadas. Asumir que en todas las posiciones hay alguna ficha.

2.3. Módulo **Othello**

En el módulo `Othello.hs` se ubicarán las funciones relacionadas con las reglas del juego. Obsérvese 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, una *jugada* en el juego queda unívocamente determinado por:

- la posición del tablero donde se coloca una ficha
- o por un identificador que represente la jugada de pasar

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

```
data Juego = J Color Tablero
data Jugada = M Posicion | Paso
```

Ejercicio 7

Definir la función `jugar :: Jugada → Juego → Maybe Juego`
Si la jugada es inválida, devuelve `Nothing`. Si la jugada es válida, devuelve el estado del juego después de efectuar la jugada. Para que una jugada sea válida se debe controlar si es `Paso` o:

- Que la casilla destino no caiga fuera del tablero.
- Que la casilla destino esté vacía.
- Que al colocar la ficha se rodeen fichas del jugador contricante.

Si la jugada es válida, tener en cuenta que las fichas que son rodeadas deben cambiar de color.

Ejercicio 8

Definir la función `jugadasPosibles :: Juego → [(Jugada,Juego)]`
que devuelva la lista de todas las jugadas válidas que se pueden realizar en el juego dado, acompañados del estado que resulta al efectuar esa jugada. Si no se puede colocar ninguna ficha en el tablero, la función deberá devolver la lista con el juego que resulta de realizar una jugada de `Paso`. □

Considerando los siguientes tipos:

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

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

Ejercicio 9

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 10

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 11

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

Definirla utilizando `foldArbol`, sin recursión explícita. Utilizar el algoritmo minimax[1], con la función de valuación dada, para determinar la mejor jugada. Asumir que el árbol de jugadas viene podado (y por lo tanto es finito).

Sugerencia: definir la función `minimax :: Valuacion → ArbolJugadas → (Double,[Jugada])` que devuelva la mejor valuación obtenida por el algoritmo minimax, acompañada de la secuencia de jugadas 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 ó 1 en el estado actual del juego, se debe devolver este valor, sin visitar los hijos del nodo.

Ejercicio 12

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 el juego finaliza cuando ambos jugadores no pueden jugar más.

Ejercicio 13

Definir la función `valuacionOthello :: 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 actualmente. La función de valuación está definida por:

$$\texttt{valuacionOthello juego} = \begin{cases} -1 & \text{si el jugador perdió} \\ 1 & \text{si el jugador ganó} \\ 2\frac{m}{tot} - 1 & \text{si el juego continúa} \end{cases}$$

donde:

- m es la cantidad de fichas del jugador que tiene el turno actual
- tot es la cantidad total de fichas en el juego

[1] `http://en.wikipedia.org/wiki/Minimax`

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

- Las funciones iterate :: (a → a) → a → [a] y takeWhile :: (a → Bool) → [a] → [a] definidas en el módulo estándar de Haskell List.
- 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[2] 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.

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