DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico Funcional

today

Paradigmas de lenguajes de programación

Integrante	LU	Correo electrónico
Bianchi, Mariano	92/08	marianobianchi08@gmail.com
Brusco, Pablo	527/08	pablo.brusco@gmail.com
Höss, Emiliano	664/04	emihoss@gmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

Índice

1. Módulo Tablero

```
module Tablero where

import Char
import Maybe

data Color = Blanca | Negra deriving (Show, Eq)
data Ficha = Simple Color | Reina Color deriving (Show, Eq)
type Posicion = (Char, Int)
data Tablero = T (Posicion -> Maybe Ficha)
data Direccion = TL | TR | BR | BL deriving (Eq, Ord, Show)


---- Igualdades ----
instance Eq Tablero where
  t1 == t2 = foldr (\pos r -> r && ((contenido pos t1) == (contenido pos t2)))
                   True
                   posicionesValidas

---- Funciones de regalo ----

tableroInicial :: Tablero
tableroInicial = T f
  where
        f (i,j)
          | j `elem` [1..3] && negro = Just (Simple Blanca)
          | j `elem` [6..8] && negro = Just (Simple Negra)
          | otherwise = Nothing
          where negro = (ord i - ord 'a' + j) `mod` 2 /= 0

instance Show Tablero where
  show (T tablero) =
      "   a b c d e f g h  \n" ++
      "  ------------------ \n" ++
      concatMap showFil [8,7..1] ++
      "  ------------------ \n" ++
      "   a b c d e f g h  \n"
    where
      showFil fil = show fil ++ " " ++
                    concatMap (showCol fil) ['a'..'h'] ++ "| " ++
                    show fil ++ "\n"
      showCol fil col = "|" ++ p (tablero (col,fil))
      p Nothing = " "
      p (Just (Simple Blanca)) = "b"
      p (Just (Reina Blanca)) = "B"
      p (Just (Simple Negra)) = "n"
      p (Just (Reina Negra)) = "N"

---- Ejercicios ----

{- Generamos un tablero vacio, el cual es una funcion que para cualquier posicion
devuelve Nothing -}
-- Ejercicio 1
-- vacio :: Tablero
vacio:: Tablero
vacio = T g where g _ = Nothing

{- Dada una posicion y un tablero, usamos la funcion definida dentro de tablero para
devolver una ficha o Nothing en caso que no exista una en esa posicion. -}
```

```
-- Ejercicio 2
--contenido :: Posicion -> Tablero -> Maybe Ficha
contenido::Posicion -> Tablero -> Maybe Ficha
contenido pos (T f) = f pos

{- Tanto para la funcion 'poner' como para 'sacar' utilizamos la funcion
'cambiarFuncion' la cual dada una 'Posicion pos' y una 'Maybe Ficha ficha'
devuelve una funcion modificada con el cambio en la 'Posicion' pos por la
'Maybe Ficha' ficha. -}
--poner :: Posicion -> Ficha -> Tablero -> Tablero
poner pos fich (T f) = T (cambiarFuncion f pos (Just fich) )

--sacar :: Posicion -> Tablero -> Tablero
sacar pos (T f) = T (cambiarFuncion f pos Nothing )

--cambiarFuncion:: (Posicion-> Maybe Ficha) -> Posicion ->
--                                  Maybe Ficha -> (Posicion-> Maybe Ficha)
cambiarFuncion f pos fich = (\posx -> if (posx==pos) then fich else f posx)

{-enRango devuelve verdadero si la posicion pasada por parametro esta
en rango o falso si no-}
--enRango :: Posicion -> Bool
enRango p = elem p posicionesValidas

--devuelve verdadero si la posicion esta vacia en el tablero
--estaVacia :: Posicion -> Tablero -> Bool
estaVacia p t = contenido p t == Nothing

posicionesValidas::[(Char,Int)]
posicionesValidas = [(c,n) | c <- ['a'..'h'], n <- [1..8]]


---------------------- OBSERVADORES PARA TABLERO ----------------------
--func :: Tablero -> (Posicion -> Maybe Ficha)
func (T f) = f

---------------------- OBSERVADORES Y AUX PARA FICHA ----------------------

--colorF :: Ficha -> Color
colorF (Simple c) = c
colorF (Reina c) = c

--esReina :: Ficha -> Bool
esReina (Simple _) = False
esReina (Reina _) = True

--esSimple :: Ficha -> Bool
esSimple (Simple _) = True
esSimple (Reina _) = False
```

2. Módulo Damas

```
module Damas where

import Tablero
import Char
import Maybe

data Juego = J Color Tablero
data Movimiento = M Posicion Direccion
  deriving Show

data Arbol a = Nodo a [Arbol a] deriving Show
type ArbolJugadas = Arbol ([Movimiento], Juego)

type Valuacion = Juego -> Double

---- Igualdades ----


{-Para la igualdad de arboles, estamos teniendo en cuenta el orden de los
hijos de cada arbol. Es decir, si a1 y a2 fueran arboles de tipo "a" y e un
elemento de tipo "a", el arbol (Nodo e [a1,a2]) NO es igual a (Nodo e [a2,a1]).
Es necesario que el tipo "a" tenga definida la igualdad. -}
instance (Eq a) => Eq (Arbol a) where
        a1 == a2 = foldArbol (\nodo rec tree ->
                                (nodo == (vNodo tree)) &&
                                (todos (zipWith ($) rec (hijos tree)))
                             )
                             a1
                             a2
                             --rec :: [(Arbol a -> Bool)]
                where
                        todos = foldr (\b rec -> b && rec) True

instance Eq (Movimiento) where
        a1 == a2 = ((posMov a1) == (posMov a2)) && (dirMov a1 == dirMov a2)

---- Funciones de regalo ----

instance Show Juego where
  show (J turno tablero) = "\n--Juegan las " ++ show turno ++ "s--\n"
                                            ++ show tablero

instance Eq Juego where
  j1 == j2 = tablero j1 == tablero j2 && colorJ j1 == colorJ j2

arbolDeJugadas :: Juego -> ArbolJugadas
arbolDeJugadas j = Nodo ([], j) $ zipWith agmov movs hijos
  where agmov m (Nodo (ms, r) hs) = Nodo ((m:ms), r) (map (agmov m) hs)
        movsJuegos = movimientosPosibles j
        movs = map fst movsJuegos
        hijos = map (arbolDeJugadas . snd) movsJuegos
```

```
---- Ejercicios ----

-- Ejercicio 3

{- Para programar esta funcion, la idea fue separar en casos a los distintos
movimientos. Por un lado, verificamos todos los posibles movimientos invalidos.
En caso que sea invalido, se devuelve directamente "Nothing". En caso de ser
valido, se verificaba si era una captura o no, y en base a esto, se realizaba el
movimiento indicado (previa eliminacion de la ficha capturada, si ese fuera el caso).
-}
mover :: Movimiento -> Juego -> Maybe Juego
mover m j =     if (elMovimientoDirectoEsInvalido || laCapturaEsInvalida)
                        then Nothing
                        else moverSegunSiEsSimpleOCaptura

        where
                elMovimientoDirectoEsInvalido = ( (not origenEnRango) ||
                                                  (not destino1EnRango) ||
                                                  (not hayFichaEnOrigen) ||
                                                  (not mueveElJugadorCorrespondiente) ||
                                                  (not mueveEnDireccionCorrecta) )

                laCapturaEsInvalida = (not elMovimientoDirectoEsInvalido) &&
                                      ( (not destino1Vacio) &&
                                                  (seVaAAutoCapturar ||
                                                  (not destino2EnRango) ||
                                                  (not destino2Vacio)) )

                moverSegunSiEsSimpleOCaptura =
                        if (destino1Vacio)
                                then realizarMovimiento origen destino1 j
                                else realizarMovimiento origen destino2
                                                        juegoSinLaFichaAComer
                origen = posMov m
                destino1 = posDeMoverDirecto m
                destino2 = posDeMoverDirecto (M destino1 (dirMov m))
                fichaOrigen = fromJust (contenido origen (tablero j))
                fichaDestino1 = fromJust (contenido destino1 (tablero j))
                origenEnRango = enRango origen
                destino1EnRango = enRango destino1
                hayFichaEnOrigen = not (estaVacia origen (tablero j))
                mueveElJugadorCorrespondiente = colorF fichaOrigen == colorJ j
                destino1Vacio = estaVacia destino1 (tablero j)
                seVaAAutoCapturar = colorF fichaDestino1 == colorJ j
                destino2EnRango = enRango destino2
                destino2Vacio = estaVacia destino2 (tablero j)
                mueveEnDireccionCorrecta = (esNegraSimple && mueveHaciaAbajo) ||
                                           (esBlancaSimple && mueveHaciaArriba) ||
                                            esReina fichaOrigen
                esNegraSimple = (colorF fichaOrigen == Negra) &&
                                (not (esReina fichaOrigen))
                esBlancaSimple = (colorF fichaOrigen == Blanca) &&
                                 (not (esReina fichaOrigen))
                mueveHaciaAbajo = (dirMov m == BL) || (dirMov m == BR)
                mueveHaciaArriba = (dirMov m == TL) || (dirMov m == TR)
                juegoSinLaFichaAComer = J (colorJ j) (sacar destino1 (tablero j))


--dado un movimiento, devuelve la posicion en donde "desembocaria" ese movimiento
posDeMoverDirecto :: Movimiento -> Posicion
posDeMoverDirecto (M pos TL) = ( chr (ord (fst pos) - 1), (snd pos) + 1 )
```

```
posDeMoverDirecto (M pos TR) = ( chr (ord (fst pos) + 1), (snd pos) + 1 )
posDeMoverDirecto (M pos BL) = ( chr (ord (fst pos) - 1), (snd pos) - 1 )
posDeMoverDirecto (M pos BR) = ( chr (ord (fst pos) + 1), (snd pos) - 1 )

--Dado un color, devuelve el opuesto
cambiaColor :: Color -> Color
cambiaColor Blanca = Negra
cambiaColor Negra = Blanca

{-Devuelve un nuevo juego igual al pasado como parametro pero en el que mueve
una ficha desde la posicion origen hasta la posicion destino,cambiando de turno
de jugador-}
--PRE: el movimiento a realizar es un movimiento valido
realizarMovimiento :: Posicion -> Posicion -> Juego -> Maybe Juego
realizarMovimiento origen destino j = Just (J nuevoColor tableroNuevo)
        where
                tableroViejo = tablero j
                fichaVieja = fromJust (contenido origen tableroViejo)
                tableroNuevo = sacar origen (poner destino fichaNueva tableroViejo)
                nuevoColor = cambiaColor (colorJ j)
                fichaNueva =    if (llegoAlFondo destino (colorJ j))
                                        then Reina (colorF fichaVieja)
                                        else fichaVieja

{-Esta funcion devuelve un booleano, el cual es verdadero cuando la posicion
pasada como parametro se corresponde con el fondo del tablero para el color
recibido como "segundo parametro"-}
llegoAlFondo :: Posicion -> Color -> Bool
llegoAlFondo p c = ((snd p == 1) && (c == Negra)) || ((snd p == 8) && (c == Blanca))


-- Ejercicio 4
{- Para esta funcion utilizamos la definida en el ejercicio 3 para conocer en
cada posicion y para cada direccion si el movimiento es valido o no. En caso de
serlo, se encola en la lista resultado. Por el orden elegido entre los selectores,
el orden del resultado estara dado primero por las posiciones del tablero por
columnas (es decir, primero la columna 'a' desde 1 hasta 8, luego 'a' y asi
sucesivamente). -}
movimientosPosibles :: Juego -> [(Movimiento, Juego)]
movimientosPosibles j = [((M pos dir), fromJust (mover (M pos dir) j)) |
                         pos <- posicionesValidas, dir <- [TL, TR, BL, BR],
                         esMovimientoValido (M pos dir) j]
        where
                esMovimientoValido mov = \game -> ((mover mov game) /= Nothing)



-- Ejercicio 5
foldArbol :: (a -> [b] -> b) -> Arbol a -> b
foldArbol f (Nodo x ys) = f x (map (foldArbol f) ys)


-- Ejercicio 6
podar :: Int -> Arbol a -> Arbol a
podar = flip podar'

{- Para esta funcion lo mas complicado fue notar que tipo tenia el paso recursivo.
Una vez detectado eso, no hubo mayores problemas. Como aclaracion, tomamos a un
arbol que solo tiene un nodo y ningun hijo como arbol con 0 niveles de profundidad.-}
podar' :: Arbol a -> Int -> Arbol a
podar' = foldArbol (\val rec n -> if (n==0)
```

```
                                    then Nodo val []
                                    else Nodo val (map (flip ($) (n-1)) rec))


-- Ejercicio 7
mejorMovimiento :: Valuacion -> ArbolJugadas -> Movimiento
mejorMovimiento v aj = head (snd (minimax v aj))

{-Para esta funcion se presento el problema de como alternar los maximos y minimos
entre los distintos niveles del arbol. Una de las ideas que surgieron fue la de
utilizar siempre el minimo de los valores negados, es decir: en vez de calcular
max (x1, x2, x3) se puede calcular -min (-x1, -x2, -x3).
        El problema con esto fue que no encontramos una manera de lograr un tipo de
recursion que comenzara a evaluar desde la raiz del arbol alternadamente y no desde
las hojas con el fold que definimos.
        La segunda propuesta consistio en darnos cuenta si habia que evaluar un
maximo o minimo de acuerdo al nivel del arbol en el que se encuentra el nodo. Aqui
salieron dos ideas, la primera fue utilizar un contador agregando un parametro extra
a minimax' y para los niveles impares calcular minimo y para los pares maximo.
        Finalmente la idea que se implemento fue la de agregar un parametro a la
funcion de evaluacion, el cual es el color del jugador que tiene el turno en la raiz,
asi se puede comparar con el color del jugador que tiene el turno en cada nodo
y evaluar apropiadamente.-}

minimax :: Valuacion -> ArbolJugadas -> (Double, [Movimiento])
minimax fVal arbol = foldArbol
                        (\movs_juego listaRec ->
                        if (null listaRec)
                                then (valuacion (snd movs_juego),fst movs_juego)
                                else (minimaValuacion listaRec, movimientos listaRec)
                        ) arbol --listaRec :: [(Double, [Movimiento])]
        where
                valuacion juego = valuacionConveniente  (colorJ (snd (vNodo arbol)))
                                                        fVal
                                                        juego
                movimientos l_V_lM = ((dameSeconds l_V_lM) !! indiceDelMinimo l_V_lM)
                indiceDelMinimo l_V_lM = dameIndice     (minimaValuacion l_V_lM)
                                                        (dameFirsts l_V_lM)
                minimaValuacion l_V_lM = minL (dameFirsts l_V_lM)

--Devuelve el minimo elemento de una lista
minL :: Ord a => [a] -> a
minL = foldr1 (\x rec -> if (x<=rec) then x else rec)

--Devuelve una lista con los primeros elementos de cada par en la lista
--dameFirsts :: [(a,b)] -> [a]
dameFirsts = map (fst)

--Devuelve una lista con los segundos elementos de cada par en la lista
--dameSeconds :: [(a,b)] -> [b]
dameSeconds = map (snd)

--Devuelve el indice del elemento e en la lista
--PRE dameIndice: e debe estar en la lista
--dameIndice :: Eq a => a -> [a] -> Int
dameIndice e = foldr (\x rec -> if (e == x) then 0 else 1 + rec) 0



{-Usando esta valuacion, cuando haga minimax, siempre tengo que tomar el minimo
```

```
valor de las valuaciones en cada paso.-}
valuacionConveniente :: Color -> Valuacion -> Juego -> Double
valuacionConveniente c v j = if ( (colorJ j) == c) then -(v j) else v j


{- Para esta funcion reutilizamos 'movimientosPosibles' en la cual devolvemos
una lista de los movimientos posibles a realizarse para un juego determinado.
En caso de que esta lista sea vacia el ganador sera el oponente a quien tiene el
turno de mover en el juego, en caso contrario se verifica si el oponente puede
realizar algun movimiento para saber si gano el jugador que tiene el turno del
juego. Cuando ninguna de estas dos cosas pasaron es porque el juego sigue en curso
y nadie gano, en estos casos se devuelve Nothing.-}
-- Ejercicio 8
ganador :: Juego -> Maybe Color
ganador j =     if (null (movimientosPosibles j))
                        then Just (cambiaColor (colorJ j))
                        else (  if (null (movimientosPosibles juegoOponente))
                                        then Just (colorJ j)
                                        else Nothing
                              )
                where juegoOponente = J (cambiaColor (colorJ j)) (tablero j)

{- Para esta funcion se realiza lo especificado en el enunciado del TP, chequeando
si gano o perdio el jugador que tiene el turno, y en caso de que no pase ninguna de
estas dos cosas, se evalua el estado de ambos jugadores.-}
-- Ejercicio 9
valuacionDamas :: Juego -> Double
valuacionDamas j =      if (noHayGanador)
                                then calculoValuacion
                                else    (beta ganaJugadorActual) * 1 +
                                        (beta ganaJugadorOponente) * (-1)
        where
                noHayGanador = ((ganador j) == Nothing) &&
                                ((ganador juegoOponente) == Nothing)
                calculoValuacion = 2*(numDelCalculo / denomDelCalculo) - 1
                numDelCalculo =  fromIntegral ( (2*cantReinasDelJugador j) +
                                                cantSimplesDelJugador j)
                denomDelCalculo = fromIntegral ( (2*cantReinasTotales j) +
                                                cantSimplesTotales j)
                juegoOponente = J (cambiaColor (colorJ j)) (tablero j)
                ganaJugadorActual = ( (ganador j) == (Just (colorJ j)) )
                ganaJugadorOponente = ( (ganador juegoOponente) ==
                                        (Just (cambiaColor (colorJ j))) )

--Recibe un booleano y devuelve 1 si es True o 0 si es False
beta b = if b then 1 else 0

cantReinasTotales j = cantFichaDeterminada (esReina) j

cantSimplesTotales j = cantFichaDeterminada (esSimple) j

cantReinasDelJugador j = cantFichaDeterminada (esReinaYDeColor) j
  where esReinaYDeColor ficha = (esReina ficha) && ((colorF ficha) == colorJ j)

cantSimplesDelJugador j = cantFichaDeterminada (esSimpleYDeColor) j
  where esSimpleYDeColor ficha = (esSimple ficha) && ((colorF ficha) == colorJ j)

cantFichaDeterminada :: (Ficha -> Bool) -> Juego -> Int
cantFichaDeterminada f j = foldr (\pos cantReinasParcial ->
                if ( (noHayFicha pos) || (noEsLaFichaBuscada pos))
                        then cantReinasParcial
```

```
                    else 1 + cantReinasParcial) 0 (posicionesValidas)
    where
            noHayFicha p = (contenido p (tablero j) == Nothing)
            noEsLaFichaBuscada p = not (f (fromJust (contenido p (tablero j))))
```

OBSERVADORES PARA TIPO JUEGO

```
tablero :: Juego -> Tablero
tablero (J col t) = t

colorJ ::Juego -> Color
colorJ (J col t) = col
```

OBSERVADORES PARA TIPO MOVIMIENTO

```
dirMov :: Movimiento -> Direccion
dirMov (M p d) = d

posMov :: Movimiento -> Posicion
posMov (M p d) = p
```

OBSERVADORES PARA TIPO ARBOL

```
vNodo :: Arbol a -> a
vNodo (Nodo x hs) = x

hijos :: Arbol a -> [Arbol a]
hijos (Nodo x hs) = hs
```

3. Módulo Tests

```
import Tablero
import Damas
import HUnit
import Maybe

-- evaluar t para correr todos los tests
t = runTestTT allTests

allTests = test [
        "tablero" ~: testsTablero,
        "damas" ~: testsDamasIniciales,
        "damas2" ~: testsDamasValidos,
        "podarArbol" ~: testPodar,
        "valuacion" ~: testValuacion,
  "mejorMov" ~: testMejorMovimiento
        ]

----------TABLERO----------
testsTablero = test [
  vacio ~=? T (\_ -> Nothing),
  sacar a_1 vacio ~=? T (\_ -> Nothing),
  poner a_1 _n vacio ~=? T (\pos -> if (pos == a_1) then (Just _n) else Nothing),
  poner a_1 _n (poner a_2 _n vacio) ~=?
        T (\pos -> if (pos == a_1 || pos==a_2) then (Just _n) else Nothing),
  poner a_1 _n (sacar a_2  vacio) ~=?
        T (\pos -> if (pos == a_1) then (Just _n) else Nothing),
  sacar a_1 (poner a_2 _n vacio) ~=?
        T (\pos -> if (pos==a_2) then (Just _n) else Nothing),
  sacar a_1 (poner a_1 _n vacio) ~=? T (\_ ->  Nothing)
  --explicacion:
  --vacio == vacio
  --sacar al vacio mantiene vacio
  --poner una ficha al vacio se mantiene
  --poner 2 fichas las mantiene
  --sacar al vacio y luego agregar devuelve la agregada
  --poner una ficha en a2 y luego sacar una de a1 deja la de a2.
  --poner y sacar una ficha la quita.
  --tablero completo inicial coincide con el de la catedra.
  ]


----------DAMAS----------
juegoPrueba = J Blanca tableroInicial

----------CON TABLERO INCIAL----------
testsDamasIniciales = test (a_testear)

a_testear = map (\movimiento -> mover movimiento juegoPrueba ~=? Nothing)
                movs_a_nothing

movs_a_nothing =
  inicialmente_invalidos_por_turno ++
  inicialmente_invalidos_por_limites ++
  inicialmente_invalidos_por_origen ++
  invalidos_por_falta_ficha_origen juegoPrueba ++
  invalidos_por_color_del_turno juegoPrueba ++
  invalidos_por_direccion juegoPrueba ++
  inicialmente_invalidos_por_casilla_destino_ocupada
```

```
----------MOVIMIENTOS VALIDOS----------
testsDamasValidos = test (a_testear2)

juegoPrueba2 = J Blanca tableroPrueba

a_testear2 = [
  mover (M ('d',4) TL ) juegoPrueba2 ~=?
        Just (J Negra (poner c_5 _b (sacar d_4 tableroPrueba))),
  mover (M ('d',4) TR ) juegoPrueba2 ~=?
        Just (J Negra (poner f_6 _b (sacar d_4 (sacar e_5 tableroPrueba)))),
  mover (M h_7 TL) juegoPruebaCoronacionB ~=?
        Just (J Negra (poner g_8 _B (sacar h_7 tableroPruebaCoronacion))),
  mover (M b_2 BL) juegoPruebaCoronacionN ~=?
        Just (J Blanca (poner a_1 _N (sacar b_2 tableroPruebaCoronacion))),
  mover (M f_2 BL) juegoPruebaCoronacionB ~=?
        Just (J Negra (poner e_1 _B (sacar f_2 tableroPruebaCoronacion)))
  ]

tableroPrueba = poner f_4 _b (poner d_6 _n (poner e_5 _n (poner d_4 _b vacio)))

juegoPruebaCoronacionB = J Blanca tableroPruebaCoronacion
juegoPruebaCoronacionN = J Negra tableroPruebaCoronacion
tableroPruebaCoronacion = poner h_7 _B (poner b_2 _n (poner f_2 _B vacio))



----------SECUENCIAS DE MOVIMIENTOS DE PRUEBA----------
--movimientos invalidos por limites del tablero
inicialmente_invalidos_por_limites = [  M ('a',1) BL, M ('a',1) BR, M ('b',8) TR,
                                        M ('b',8) TL,M ('h',8) TR, M ('h',2) TL]

--movimientos invalidos por turno (color)
inicialmente_invalidos_por_turno = [M ('b',6) BR,M ('d',6) BL]

--movimientos invalidos por origen invalidos
inicialmente_invalidos_por_origen = [   M ('z',9) BL, M ('a',9) BR, M ('s',8) TR,
                                        M ('i',8) TL,M (' ',0) TR, M ('h',-2) TL]

--movimientos invalidos por origen sin ficha
invalidos_por_falta_ficha_origen juego =
                        [M pos BL| pos <- posicionesSinFichas juego] ++
                        [M pos TL| pos <- posicionesSinFichas juego]

--movimientos invalidos por color (turno del oponente)
invalidos_por_color_del_turno juego =
                [M pos BR| pos <- posicionesConFichasDeColor juego Negra]

--movimientos invalidos por direccion incorrecta (blancas para arriba por ejemplo)
invalidos_por_direccion juego =
                [M pos TR| pos <- posicionesSimplesDeColor juego Negra] ++
                [M pos TL| pos <- posicionesSimplesDeColor juego Negra]

--movimientos invalidos por destino ocupado
inicialmente_invalidos_por_casilla_destino_ocupada =
                                [M ('a',1) TR, M ('c',1) TL, M ('h',2) TL]



----------FIN SECUENCIAS ----------
```

```
---------- TESTS PODAR ----------


arbol1 = Nodo 1 [Nodo 2 [Nodo 3 []]]
arbol2 = infinito (Nodo 0 [])


infinito ab = Nodo (vNodo ab) [ infinito (Nodo ((vNodo ab)+1) []) ]


testPodar = [
        podar 0 arbol1 ~=? Nodo 1 [],
        podar 0 arbol2 ~=? Nodo 0 [],
        podar 1 arbol2 ~=? Nodo 0 [Nodo 1 []],
        podar 2 arbol2 ~=? Nodo 0 [Nodo 1 [Nodo 2 []]],
        podar 3 arbol2 ~=? Nodo 0 [Nodo 1 [Nodo 2 [Nodo 3 []]]]
  ]

---------- TEST MEJOR MOVIMIENTO ----------


juegoPrueba3 = fromJust (mover (M ('d',6) BR)
                (fromJust (mover (M ('d',4) TR ) juegoPrueba2)))


mejorMovTest n = mejorMovimiento valuacionDamas
                                 (podar n (arbolDeJugadas juegoPrueba3))
mejorMovTest2 n = mejorMovimiento valuacionDamas
                                  (podar n (arbolDeJugadas juegoPrueba2))

testMejorMovimiento = [
  mejorMovTest 1 ~=? M ('f',4) TL,
  mejorMovTest 2 ~=? M ('f',4) TL,
  mejorMovTest 3 ~=? M ('f',4) TL,
  mejorMovTest2 1 ~=? M ('d',4) TR,
  mejorMovTest2 4 ~=? M ('d',4) TR
  ]


---------- TESTS VALUACIONES ----------

tabGananBlancas = poner f_6 _b (poner g_7 _b (poner h_8 _n vacio))
tabNoGanaNadieYEstanEmpatados = sacar g_7 tabGananBlancas
tabCon2ReinasBl1NeY1NeSimple =
                poner a_1 _B (poner c_1 _B (poner b_8 _N (poner d_8 _n vacio)))

testValuacion = [
                valuacionDamas (J Blanca tabGananBlancas) ~=? 1,
                valuacionDamas (J Negra tabGananBlancas) ~=? (-1),
                valuacionDamas (J Negra tabNoGanaNadieYEstanEmpatados) ~=? 0,
                valuacionDamas (J Blanca tabNoGanaNadieYEstanEmpatados) ~=? 0,
                valuacionDamas (J Blanca tabCon2ReinasBl1NeY1NeSimple) ~=?
                                        (fromIntegral 8 / fromIntegral 7)-1,
                valuacionDamas (J Negra tabCon2ReinasBl1NeY1NeSimple) ~=?
                                        (fromIntegral 6 / fromIntegral 7)-1
        ]
```

```
-------FUNCIONES PARA TESTS----------

posicionesSinFichas juego = [pos|pos<-posicionesValidas,
                                contenido pos (tablero juego) == Nothing]
posicionesConFichasDeColor juego color = posicionesSimplesDeColor juego color ++
                                         posicionesReinasDeColor juego color
posicionesSimplesDeColor juego color = [pos | pos<-posicionesValidas,
                                contenido pos (tablero juego) == (Just(Simple color))]
posicionesReinasDeColor juego color = [pos | pos<-posicionesValidas,
                                contenido pos (tablero juego) == (Just(Reina color))]


----------AYUDAS-----------------------

-- idem ~=? pero sin importar el orden
(~~?) :: (Ord a, Eq a, Show a) => [a] -> [a] -> Test
expected ~~? actual = (sort expected) ~=? (sort actual)
        where
                sort = foldl (\r e -> push r e) []
                push r e = (filter (e<=) r) ++ [e] ++ (filter (e>) r)

(~~) :: (Ord a, Eq a, Show a) => [a] -> [a] -> Bool
expected ~~ actual = (sort expected) == (sort actual)
        where
                sort = foldl (\r e -> push r e) []
                push r e = (filter (e<=) r) ++ [e] ++ (filter (e>) r)

-- constantes para que los tests sean mas legibles
_n = Simple Negra
_N = Reina Negra
_b = Simple Blanca
_B = Reina Blanca

a_1 = ('a',1::Int)
b_1 = ('b',1::Int)
c_1 = ('c',1::Int)
d_1 = ('d',1::Int)
e_1 = ('e',1::Int)
f_1 = ('f',1::Int)
g_1 = ('g',1::Int)
h_1 = ('h',1::Int)

a_2 = ('a',2::Int)
b_2 = ('b',2::Int)
c_2 = ('c',2::Int)
d_2 = ('d',2::Int)
e_2 = ('e',2::Int)
f_2 = ('f',2::Int)
g_2 = ('g',2::Int)
h_2 = ('h',2::Int)

a_3 = ('a',3::Int)
b_3 = ('b',3::Int)
c_3 = ('c',3::Int)
d_3 = ('d',3::Int)
e_3 = ('e',3::Int)
f_3 = ('f',3::Int)
g_3 = ('g',3::Int)
h_3 = ('h',3::Int)
```

```
a_4 = ('a',4::Int)
b_4 = ('b',4::Int)
c_4 = ('c',4::Int)
d_4 = ('d',4::Int)
e_4 = ('e',4::Int)
f_4 = ('f',4::Int)
g_4 = ('g',4::Int)
h_4 = ('h',4::Int)

a_5 = ('a',5::Int)
b_5 = ('b',5::Int)
c_5 = ('c',5::Int)
d_5 = ('d',5::Int)
e_5 = ('e',5::Int)
f_5 = ('f',5::Int)
g_5 = ('g',5::Int)
h_5 = ('h',5::Int)

a_6 = ('a',6::Int)
b_6 = ('b',6::Int)
c_6 = ('c',6::Int)
d_6 = ('d',6::Int)
e_6 = ('e',6::Int)
f_6 = ('f',6::Int)
g_6 = ('g',6::Int)
h_6 = ('h',6::Int)

a_7 = ('a',7::Int)
b_7 = ('b',7::Int)
c_7 = ('c',7::Int)
d_7 = ('d',7::Int)
e_7 = ('e',7::Int)
f_7 = ('f',7::Int)
g_7 = ('g',7::Int)
h_7 = ('h',7::Int)

a_8 = ('a',8::Int)
b_8 = ('b',8::Int)
c_8 = ('c',8::Int)
d_8 = ('d',8::Int)
e_8 = ('e',8::Int)
f_8 = ('f',8::Int)
g_8 = ('g',8::Int)
h_8 = ('h',8::Int)
```