

Haskell プログラミング

対戦型 n 目並べ

尾上 能之 (東京大学情報基盤センター)

onoue@ecc.u-tokyo.ac.jp



対戦型ゲームの設計

子供の頃、よくノートの片隅に 3×3 のマス目を引き、 \circ と \times を埋めて先に 1 列揃えた方が勝つゲームで遊んだ記憶はないだろうか。今回取り上げるのは、ルールの説明が不要であるくらいお馴染みの三目並べである。ただせっかく計算機を用いてプログラミングするので、人間同士が対戦するのではなく機械に最善手を求めさせ、計算機と人間が対戦できるようなプログラムを作る。またマス目も 3×3 に限定せず任意の長方形を仮定し、そこに n 個同じ印を並べた方が勝ちとなるよう拡張する。また盤が広いとどうしても先手有利になり面白みに欠けるので、盤には障害物を置けるようにもしておこう。

一般的に計算機が相手の対戦型ゲームを設計する場合、ある局面から次に指す手を計算機に求めさせなければならない。ゲームの処理の流れは以下ようになる。

1. 最初の局面を表示
2. 手番に応じて (a), (b) を交互に選択
 - (a) ユーザ番: ユーザの入力待ち, 入力に応じて次の局面へ
 - (b) 計算機番: 計算機側の最善手を求めて次の局面へ
3. 移動後の局面を表示
4. 勝敗が決まる \Rightarrow ゲーム終了. そうでなければ 2. へ戻る

印の定義

Haskell にはモジュールの概念があって、プログラムの一部を機能のまとまったかたまりとして、外から取り込んだり (インポート) 外に見せたり (エクスポート) できる。これは、データ構造の操作手法だけを公開しその内部構造を隠す抽象データ型 (Abstract Data Type) を扱うために便利な機能である。今回はこの機能を用いて、 n 目並べのプログラムを (1) 印, (2) 盤面, (3) 対戦型ゲームの処理を行う 3 つの異なるモジュールから構成してみる。まずは印を表す `Mark` モジュールから定義する。

モジュールを定義するには、各モジュールごとに別々のファイルとしてプログラムを用意し、予約語 `module` の後に大文字から始まるモジュール名と、そのモジュールがエクスポートする関数やデータ構造のリストを明記する。これまで連載で扱ったプログラムは先頭に `module` 行がなかったことを覚えている読者もいると思うが、その場合 `module Main (main) where` の 1 行が補われ、関数 `main` のみをエクスポートする `Main` モジュールとして扱うよう定められている。

```

-- 各モジュールはファイル名を「モジュール名.hs」として保存
module Mark ( Mark(..), isEmpty, next ) where
    -- Mark(..) データ型名とその構築子名をエクスポート
    -- Mark      データ型名のみエクスポート

data Mark = O -- O, 先手の印
          | X -- X, 後手の印
          | U -- 空き, 何も置かれていない
          | B -- 障害物, 何も置けない
          deriving Eq

```

各マスの初期状態は何も置かれていないUか障害物が置かれたBとし、状態Uのマスには手番に応じてOやXを置くものとする。

```

instance Show Mark where
    show O = "O"
    show X = "X"
    show U = "."
    show B = "*"

isEmpty :: Mark -> Bool
isEmpty = (U ==)

next :: Mark -> Mark      -- 手番を1つ進める
next O = X
next X = O

```

局面の定義

次に局面を表す Position モジュールを定める。局面 (position) は、1) O と X 次にどちらの手番 (turn) か、2) 現在の盤面 (table)、を組み合わせたとし、Position データ型として定義する。たとえば以下に挙げた局面 pos0 は、右下の図で次がOの番であることを示している。

```

pos0 =
  Pos O [ [X, U, U],      +-----+
          [U, O, U],      |X . .|
          [U, X, O] ]    ≡  |. O .|
                          |. X O|  で次は O の番
                          +-----+

```

```

module Position where
    -- エクスポートリストを略すと、すべての値、型、クラスがエクスポートされる

-- 処理系に含まれるライブラリと同様、自前のモジュールも import 宣言で取り込む
import Mark      ( Mark(..), isEmpty, next )
import Data.List ( transpose )

-- Positon(局面) 次の手番と盤面からなる
-- Table(盤面) リストのリストで2次元の盤面を定義
--                内側の各リストの長さは等しいものとする
data Position = Pos { turn :: Mark, table :: Table }
type Table = [[Mark]]

mkPos :: Int -> Int -> Position      -- row1 * col1 の盤面を U で埋めた局面
mkPos row1 col1 = Pos O (replicate row1 (replicate col1 U))

```

```

gameLimit :: Int
gameLimit = 3 -- n 目並べの n

instance Show Position where
  show (Pos p pss) = "-----> " ++ show p ++
    concatMap showLine pss ++ "\t\t"
  where
    showLine cs = "\n" ++ unwords (map show cs)

```

2次元の盤面に対し、左上を(0,0)とする座標で指定されるマスの値を操作する関数 `getCell`, `putCell` を定義する。また局面と座標を受け取り、次の手番に応じてマスを更新した新しい局面を返す関数を `updatePosition` とする。

```

getCell :: (Int,Int) -> Table -> Mark
getCell (x,y) pss
  | x<0 || row<x || y<0 || col<y = error "Illegal move"
  | otherwise                      = pss !! x !! y
  where (row,col) = row_col pss

putCell :: (Int,Int) -> Mark -> Table -> Table
putCell (x,y) p pss
  | x<0 || row<x || y<0 || col<y = error "Illegal move"
  | not $ isEmpty p0              = error "Not empty"
  | otherwise                      = pss'
  where
    (row,col) = row_col pss
    (pss0, ps:pss1) = splitAt x pss
    (ps0, p0:ps1) = splitAt y ps
    ps' = ps0 ++ p:ps1
    pss' = pss0 ++ ps':pss1

-- リストのインデックス(0から始まる)の上限として用いるのでマイナス1しておく
row_col tab = (length tab - 1, length (head tab) - 1)

updatePosition :: (Int,Int) -> Position -> Position
updatePosition (x,y) (Pos p pss) = Pos (next p) (putCell (x,y) p pss)

```

関数 `allLines` は、盤面からすべての縦、横、斜めのマス目の並びを生成し、長さが `gameLimit` 以上のものをリストにして返す。関数 `allCells` は、盤面に含まれるすべてのマス目をリストにして返す。

```

allLines :: Table -> [[Mark]]
allLines tab = [ line | line <- rows++cols++diags, length line >= gameLimit ]
  where
    rows = tab
    cols = transpose tab
    diags = [ [ getCell (i,j) tab | i<-[0..row], let j=k+i, 0<=j && j<=col ]
              | k <- [-row..col] ] ++
            [ [ getCell (i,j) tab | i<-[0..row], let j=k-i, 0<=j && j<=col ]
              | k <- [0..row+col] ]
    (row,col) = row_col tab

allCells :: Table -> [Mark]
allCells tab = concat tab

```

ある局面から可能なすべての指し手 (move) をとったときの次の局面をリストにして返す関数 `allMoves`, ならば

にゲームが終わりかどうか判断するための関数 `winlosegame`, `drawgame` を定義する.

```
-- 局面 pos に対し、次のすべての局面候補のリストを返す
allMoves :: Position -> [Position]
allMoves pos@(Pos _ pss)
  | winlosegame pos || drawgame pos = []
  | otherwise = [ updatePosition (x,y) pos |
                  x <- [0 .. row],
                  y <- [0 .. col],
                  isEmpty (getCell (x,y) pss) ]
  where (row,col) = row_col pss

winlosegame :: Position -> Bool
winlosegame (Pos p pss) = or [ fin line | line <- allLines pss ]
  where
    fin :: [Mark] -> Bool
    fin qs = fin' 0 qs          -- qs の中に p' が n 個並んでいれば True
      where
        fin' n [] = False
        fin' n (q:qs)
          | q == p' = let n1 = n+1 in
                      if n1 >= gameLimit then True else fin' n1 qs
          | otherwise = fin' 0 qs
        p' = next p

drawgame :: Position -> Bool
drawgame (Pos _ pss)
  = and [ not $ isEmpty cell | cell <- allCells pss ]
```

ゲーム木を用いた最善手の探索

最後に用意するモジュールとして、対戦型ゲームを作る上で欠かせない最善手の探索ルーチンや、人間との対話処理部分を含む `TicTacToe` モジュールを定めよう. まず始めに木のデータ構造を操作するいくつかのユーティリティ関数を定義する.

```
module TicTacToe where

import Data.Char      ( digitToInt )
import Data.Tree      ( Tree(..) )

import Mark           ( Mark(..), isEmpty, next )
import Position

size :: Tree a -> Int          -- 節の個数を求める
size (Node x ts) = 1 + sum (map size ts)

depth :: Tree a -> Int        -- 木の深さを求める
depth (Node x []) = 0
depth (Node x ts) = 1 + maximum (map depth ts)

-- すべての部分木に f を適用したものから構成される新たな木を作る
mapSubTree :: (Tree a -> b) -> Tree a -> Tree b
mapSubTree f n@(Node x ts) = Node (f n) (map (mapSubTree f) ts)

-- x を元に、繰り返し f を適用していったものを子とする木を作る
```

```
repTree :: (a -> [a]) -> a -> Tree a
repTree f x = Node x (map (repTree f) (f x))
```

2人の対戦者が交互に手を指し続け勝敗を決定するゲームは、ゲーム木を用いて局面の推移を表すことができる。このとき木とゲームの間には以下の関係が成り立つ。

木	ゲーム
節 (ノード) 情報	局面 (position)
根 (ルート)	最初の局面
枝	指し手

関数 `gameTree` はある局面を引数にとり、その後続くすべての局面をノードに格納したゲーム木を返す。また、規模が大きく全体を処理しきれないような木を扱う際、関数 `prune` を用いて根から `m` 段を超えるノードを刈り込んだ木を用いる。

```
type GTree = Tree Position

gameTree :: Position -> GTree
gameTree pos = repTree allMoves pos

prune :: Int -> Tree a -> Tree a
prune 0 (Node x _) = Node x []
prune (m+1) (Node x ts) = Node x (map (prune m) ts)
```

局面に対し、その局面が次の手番方に有利か不利かを静的に判定する評価関数 `static` を定義する。この評価関数の出来がゲームの強さを左右するが、以下に定義した `static` はまず局面を各並び（縦、横、斜め）に分解し、各行から `eval` で求めた得点を合計したものを評価値とする。

```
static :: Position -> Int
static (Pos p pss) = sum [ eval p line | line <- allLines pss ]
```

関数 `eval` は、まず行に含まれる `O, X` の個数を数え上げる。次に以下の規則によって行あたりの得点を決める。

状態	得点
<code>O, X</code> が1つも含まれない、または <code>O, X</code> が混在	0点
自分の印が a 個で、相手の印が 0 個	10^a 点
相手の印が b 個で、自分の印が 0 個	-10^b 点

この定義は $n \times n$ の盤面で n 目並べをするときは良い手を選択できるが、それ以外の場合は最善手を選ぶとは限らないので注意が必要である。このほかにも評価関数はさまざまな定義が考えられるので、強いプログラムを作るにはこの部分を工夫するとよいであろう。

```
eval :: Mark -> [Mark] -> Int
eval p qs = eval' (count p qs)
  where
    count :: Mark -> [Mark] -> (Int,Int)
    count p [] = (0,0)
    count p (q:qs)
      | q == p    = (a+1,b)
      | q == p'  = (a,b+1)
      | otherwise = (a,b)
      where (a,b) = count p qs
            p' = next p
    eval' :: (Int,Int) -> Int
```

```

eval' (0,0) = 0
eval' (a,0) = 10^a
eval' (0,b) = -10^b
eval' (_,_) = 0

```

次に求めた評価値を基にして、数ある次の手の候補から最善となる手を求めるための戦略を定める。ここでは、相手も自分と同じ最善手の推定手法を行うと仮定する。n目並べは一方からみた評価値がwであるとき相手からみた評価値は-wとなり、零和ゲームと呼ばれる性質を持つ。このようなゲームにおける戦略として、次の局面に対する評価値から相手の得点を最小にするような手を選択するものとし、この戦略はミニマックス法 (minimax) と呼ばれる。

局面から最善手を求めるための関数 dynamic の動作は以下のようになる。

1. 深さ m に刈り込んだゲームの木 n1 を生成
2. 関数 static により、木の各局面に対応する静的評価値を求める
3. 関数 minimax により、最善手に対応したミニマックス値 x2 を求める
4. 関数 select_pos により 3. で求めた値から次の局面を決定する

ミニマックス法を用いると、図-1のように2手先の局面に対し評価値を求めてから、b1, b2, a1 に対応する評価値を求め、 $-b1 > -b2$ であることから A は左下への枝に対応する手を選択する。

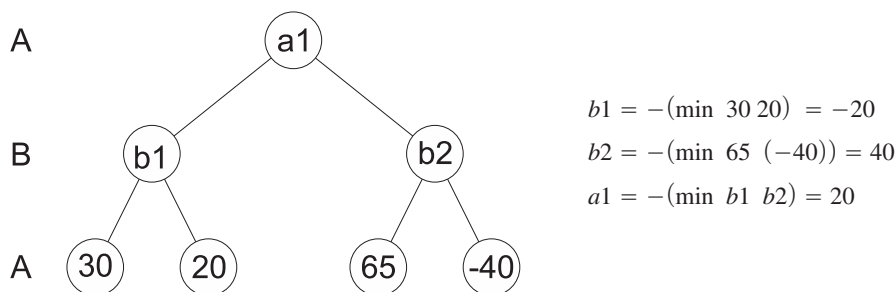


図-1 ミニマックス法による最善手の決定

```

dynamic :: Int -> Position -> Position
dynamic m pos = select_pos (-x2) ts2 ts1
  where n1@(Node x1 ts1) = (prune m . gameTree) pos
        Node x2 ts2 = (mapSubTree minimax . fmap static) n1 -- minimax 版
--      Node x2 ts2 = (mapSubTree minimax' . fmap static) n1 -- α-β 版 (後述)

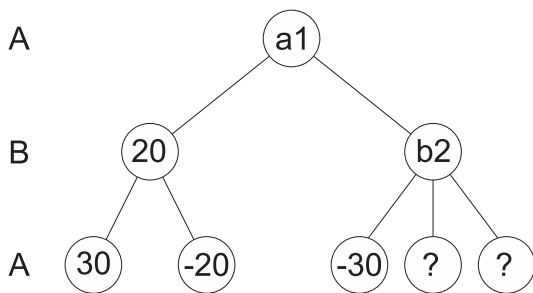
type ITree = Tree Int

select_pos :: Int -> [ITree] -> [GTree] -> Position
select_pos u (Node v _ : its) (Node p _ : gts)
  = if u==v then p else select_pos u its gts

minimax :: ITree -> Int -- 相手の得点を最小にするような手を選択
minimax (Node x []) = x
minimax (Node x ts) = -1 * minimum (map minimax ts)

```

ミニマックス法では、指定した手数に応じて刈り込んだ後のゲーム木におけるすべての末端の節に対し評価値を求めていたが、ミニマックスの性質から場合によっては評価関数の計算を省略できることもある。



左の局面では $b2 \geq 30$ が確定
 $\Rightarrow a1 = -(\min 20 \ b2) = -20$
 \Rightarrow 2カ所の?の局面の評価値は不要

図-2 評価値の計算を省略できる例

図-2の例では、左部分木のミニマックス値がそこから右への部分木に対する評価値の上限を与えている。このように、すでに計算した評価値を上限として、そこから先の計算の手間を減らしていく手法が α - β 法である。これを定式化するために、まず関数 `minimax` の範囲限定版 `bmx` を考える。`bmx` は以下の性質を持つ。

- 定義 $bm_x \ a \ b \ t = a \ \text{`max`} \ (\minimax \ t) \ \text{`min`} \ b$
- a, b には下限, 上限を指定する
- $a=b$ のとき $bm_x \ a \ a \ t = a$ であるので `minimax t` の評価が不要
- すべての局面 p に対して $-m \leq \text{static } p \leq m$ であるような定数 m が存在
 $\Rightarrow \minimax \ t = bm_x \ (-m) \ m \ t$

関数 `dynamic` の中で呼んでいた `minimax` を以下の `minimax'` で置き換えると、 α - β アルゴリズムを用いて効率良く最善手の探索が行えるようになる。

```
minimax' :: ITree -> Int
minimax' = bmx (-max_sval) max_sval
  where
    max_sval = 231 - 1      -- as Infinity

bmx :: Int -> Int -> ITree -> Int
bmx a b (Node x []) = (a `max` x) `min` b
bmx a b (Node x ts) = cmx a b ts

cmx :: Int -> Int -> [ITree] -> Int
cmx a b [] = a
cmx a b (t:ts) = if a' == b then a' else cmx a' b ts
  where a' = - bmx (-b) (-a) t
```

最後に、盤面と計算機側の手番を引数にとり対話的な処理を行う `tictactoe` と、全体の処理をまとめた `main`, `main'` を定義する。`main` で先手番の、`main'` で後手番の三目並べがスタートする。また変数 `prune_val` によって、最善手を推定する際の枝刈りの深さを指定することが可能で、ここでは5すなわち5手先まで読むようにしてある。盤面が3×3程度の大きさで α - β 法を用いるときはこのレベルでも実行可能であるが、盤面が大きいときは枝刈りの深さを小さくしないと計算時間が増え、計算機が次の手を指してくれなくなるので注意されたい。

```
main, main' :: IO ()
main = tictactoe (mkPos 3 3) O
main' = tictactoe (mkPos 3 3) X
```

```

tictactoe :: Position -> Mark -> IO ()
tictactoe pos p0
  = do putStr $ show pos
      (if p0 == 0 then loop0 else loop1) pos
  where
    loop0 pos                -- ユーザの手番
      = do putStr "XY : "
          (c1:c2:_) <- getLine
          let [x,y] = map digitToInt [c1,c2]
              pos' = updatePosition (x-1,y-1) pos
              putStr $ show pos'
              finGame loop1 pos'
    loop1 pos                -- 計算機の手番
      = do putStr "\n"
          let pos' = dynamic prune_val pos
              putStr $ show pos'
              finGame loop0 pos'
          where prune_val = 5
    finGame next_loop pos
      | winlosegame pos = putStrLn $ "You " ++ if p==p0 then "lose." else "win!"
      | drawgame pos   = putStrLn "Game is draw."
      | otherwise      = next_loop pos
    where p = turn pos

```

このプログラムを実行した例を以下に示す（紙面の都合により3列に改行）。これは先手が計算機の例であるが、後手である人間が誤った手（二手目の32x）をとると、きっちりプログラムに負けられる様子が見てとれるであろう。

```

$ hugs TicTacToe.hs          -----> O                -----> X
    (...)                   . . .                O . .
Type :? for help            . O .                . O .
TicTacToe> main'           . X .                O X X          XY : 21
-----> O                  -----> X                -----> O
. . .                      O . .                O . .
. . .                      . O .                X O .
. . .                      . X .                O X X          XY : 33
-----> X                  -----> O                -----> X
. . .                      O . .                O . O
. O .                      . O .                X O .
. . .                      . X X                O X X          You lose.
( 中側の列へ ↗ )          ( 右側の列へ ↗ )

```

参考文献

1) Bird, R. and Wadler, P.: Functional Programming, Prentice-Hall (1988). (邦訳：武市正人: 関数プログラミング, 近代科学社(1991)). (平成17年10月20日受付)

