課題 9

○×ゲームを作りなさい。

解答及び解説

はじめに、このプログラムでは対話方式でゲームを進めるため入出力を伴う。このとき、数字を数に変換する必要から、

```
isDigit :: Char -> Bool
digitToInt :: Char -> Int
```

という関数を利用する。それには、プログラムのはじめにモジュール Data.Char をインポートする必要がある。

さてまずは、プレイヤーの型 (Player) と、盤面の型 (Board) を用意する。プレイヤーは、マルとペケ。盤面は、3x3 マスを、要素 3 つのリストが 3 つあるリストのリストで表現する。その要素には、Maybe Player 型を使い、まだマルもペケも無いところは、Nothing とする。従って、始まる前の盤面 empty は下のようになる。

```
import Data.Char

data Player = Maru | Peke deriving (Eq, Show)

type Row = [Maybe Player]
type Board = [Row]

empty :: Board
empty = [[Nothing,Nothing,Nothing],
         [Nothing,Nothing,Nothing],
         [Nothing,Nothing,Nothing]]
```

次に、このボードを表示する関数 display を考える。

```
display :: Board -> IO ()
display [] = return ()
display (r:rs) = do putStrLn $ row r
                    display rs
    where row [] = ""
          row (c:cs) | c == Nothing   = '-':row cs
                     | c == Just Maru = 'o':row cs
                     | c == Just Peke = 'x':row cs
```

さて、ゲームの進行を箇条書きにすると次のように定義できる。

1) マルペケを書く場所を決める
2) その場所に、マルペケを書いた盤面を作る
3) その盤面を表示する
4) 勝ち・引き分けの判断をし、
   4-a) 決まれば、終了
   4-b) 決まらなければ、プレイヤーを交代し、1) へ戻って繰り返す

この進行を制御する関数 game は、補助関数 nextGame を用いて、次のようになる。

```
game :: Player -> Board -> IO ()
game p b = do pos <- getPosition ("turn "++(show p)++"> ")
              let nb = update p b pos
              display nb
              nextGame p nb

nextGame :: Player -> Board -> IO ()
nextGame p b | win p b   = putStrLn ("Winner "++(show p))
             | draw b    = putStrLn "Draw"
             | otherwise = game (opposite p) b

opposite :: Player -> Player
opposite Maru = Peke
opposite Peke = Maru
```

関数 game は、プレイヤーと現在の盤面を受け取り、上記の手順を進める。補助関数 nextGame では、「勝ち」または「引き分け」が決まれば、その結果を表示する。決まらなければ、game を再度呼び出して、この手順を繰り返す。関数 opposite は、プレイヤーを交代するために使う。

そして、ゲームを開始する gameStart は、次のようなプログラムになる。(但し、慣例によりマルからゲームを開始する。)

```
gameStart :: IO ()
gameStart = display empty >> game Maru empty
```

(注) 問題文には無かったが、ゲームの始めに、空っぽの盤面を表示するようにした。

次に、それぞれの手順の中身を作成していく。まずは、「1) マルペケを書く場所を決める」のであるが、これには問題文に書いてある関数 getPosition を使う。但し、入力を促すプロンプトを引き数とするように改良した。

```
type Position = (Int, Int)

getInteger :: String -> Int -> (Int, String)
getInteger [] x = (x, [])
getInteger (c:cs) x | isDigit c = getInteger cs (10*x + digitToInt c)
                    | otherwise = (x, cs)

makePosition :: String -> Position
makePosition s1 = (x, y)
        where (x, s2) = getInteger s1 0
              (y, s3) = getInteger s2 0

valid :: Position -> Bool
valid (x, y) = (0 <= x && x <= 2) && (0 <= y && y <= 2)

getPosition :: String -> IO Position
getPosition prompt = do putStr prompt
                        str <- getLine
                        let (x,y) = makePosition str
                        validate (x, y)
    where validate (x, y) | valid (x, y) = return (x, y)
                          | otherwise    = getPosition prompt
```

次に、「2) その場所に、マルペケを書いた盤面を作る」のは、関数 update である。これは、プレイヤーと盤面、それから場所 (Position 型) を受け取り、場所 (r, c) の情報から r 番目のリスト (Row 型) を取り出し、補助関数 updateRow に渡す。updateRow では、リストの c 番目の要素を、プレイヤー p で置き換える。(但し、Just をつけて Maybe Player 型にする。)

```
update :: Player -> Board -> Position -> Board
update p (row:rows) (0,c) = (updateRow p row c):rows
update p (row:rows) (r,c) = row:update p rows ((r-1),c)

updateRow :: Player -> Row -> Int -> Row
updateRow p (col:cols) 0 = (Just p):cols
updateRow p (col:cols) c = col:updateRow p cols (c-1)
```

そして、「3) その盤面を表示する」のは、display を使い、「4) 勝ち・引き分けの判断をし」のところは、補助関数 nextGame で行う。さらに、nextGame の中では、勝ち・引き分けを判断するために述語 win と draw を使う。

このとき、述語 win は、横並び、縦並び、斜め並びがあるかを調べれば良い。プログラムは、次のようになる。

```
win :: Player -> Board -> Bool
win p b = or [winLine p b (0, 0) (0, 1),
              winLine p b (1, 0) (0, 1),
              winLine p b (2, 0) (0, 1),
              winLine p b (0, 0) (1, 0),
              winLine p b (0, 1) (1, 0),
              winLine p b (0, 2) (1, 0),
              winLine p b (0, 0) (1, 1),
              winLine p b (2, 0) (-1, 1)]
```

ここで、winLine は、始点から差分に従って、順にプレイヤーが並んでいるかを調べることができる。つまり、1 行目は、始点 (0,0) から、差分 (0,1) で調べるというものであり、横に一列プレイヤー p が並んでいるかどうかを表す。2 行目は、始点が (1,0) だから、その下の一列を調べている。以下同様に、横、縦、斜めの合計 8 つの列を順に調べて、結果をリストにする。述語 or は、リストの要素に一つでも True があれば True となる。

そして、述語 winLine は次のように定義される。ここで、i 行 j 列がマルかペケかを調べる関数 at を使っている。(後で定義する)

```
winLine :: Player -> Board -> Position -> (Int, Int) -> Bool
winLine p b (3, j) diff = True
winLine p b (i, 3) diff = True
winLine p b (i, j) (di, dj) = maybe False next (at b (i, j))
    where next q | p == q    = winLine p b (i+di, j+dj) (di, dj)
                 | otherwise = False
```

最後に、引き分けの判定をする draw を定義する。win と同様にして、drawLine を使って、8 つの列を調べる。

```
draw :: Board -> Bool
draw b = and [drawLine b (0, 0) (0, 1),
              drawLine b (1, 0) (0, 1),
              drawLine b (2, 0) (0, 1),
              drawLine b (0, 0) (1, 0),
              drawLine b (0, 1) (1, 0),
              drawLine b (0, 2) (1, 0),
              drawLine b (0, 0) (1, 1),
              drawLine b (2, 0) (-1, 1)]
```

drawLine では、その列にマルとペケの両方が出てくると True となる。

```
drawLine :: Board -> Position -> (Int, Int) -> Bool
drawLine b pos diff = drawLine' Maru b pos diff
                      && drawLine' Peke b pos diff

drawLine' :: Player -> Board -> Position -> (Int, Int) -> Bool
drawLine' p b (3, j) diff = False
drawLine' p b (i, 3) diff = False
drawLine' p b (i, j) (di, dj) = maybe draw next (at b (i, j))
    where draw = drawLine' p b (i+di, j+dj) (di, dj)
          next q | p == q    = True
                 | otherwise = draw
```

勝ち・負けが決まらずに、盤面のすべてが埋まったときを「引き分け」と考えた人が多かったが、実際には、最後のマスが埋まる前に引き分けが確定する場合もある。

最後に、盤面の特定の場所がマルやペケで埋まっているか、まだ何も書かれていないかを調べる関数 at を定義すれば完成である。

```
at :: Board -> Position -> Maybe Player
at (row:rows) (0, j) = atRow row j
at (row:rows) (i, j) = at rows ((i-1), j)

atRow :: Row -> Int -> Maybe Player
atRow (col:cols) 0 = col
atRow (col:cols) j = atRow cols (j-1)
```

さて、これで一応ゲームはできるが、盤面にマルまたはペケを書くとき、そのマスが空いているかどうかをチェックしていないので、このままでは相手のマスに上書きできてしまう。

そこで、きちんと空いているかをチェックし、空いていない場合には、エラーとなり「1) 書く場所を決める」からやり直すように変更してみよう。

これには、関数 update で、マルペケを書くまえに、at を使いチェックする方法も考えられるが、今回は、update を変更し、マルペケを書くときにすでにどちらかが書かれていれば、更新した盤面の代わりに Nothing を返すようにした。

```
update :: Player -> Board -> Position -> Maybe Board
update p (row:rows) (0,c) = maybe Nothing makeRow (updateRow p row c)
    where makeRow nrow = Just (nrow:rows)
update p (row:rows) (r,c) = maybe Nothing makeBoard (update p rows (r-1, c))
    where makeBoard nrows = Just (row:nrows)
```

```
updateRow :: Player -> Row -> Int -> Maybe Row
updateRow p (Nothing:cols)  0 = Just ((Just p):cols)
updateRow p ((Just q):cols) 0 = Nothing
updateRow p (col:cols) c = maybe Nothing makeRow (updateRow p cols (c-1))
    where makeRow ncols = Just (col:ncols)
```

関数 update の変更に伴って、関数 game も変更が必要になる。

```
game :: Player -> Board -> IO ()
game p b = do pos <- getPosition ("turn "++(show p)++"> ")
              maybe (game p b) go (update p b pos)
    where go :: Board -> IO ()
          go b = do display b
                    nextGame p b
```