■ 連載

Haskell プログラミング

数当てゲームを解く

尾上 能之（東京大学情報基盤センター）
onoue@ecc.u-tokyo.ac.jp

数当てゲーム MOO

前回は木を扱ったので，今回はリストを用いて遊ぶこととしよう．数当てゲーム MOO は，名前は聞いたことがなかったとしても誰でも一度は遊んだことのあるゲームではないだろうか．ルールは簡単で，対戦する 2 人のうち片方が各桁に重複のない 4 桁の数（**MOO 数**と呼ぶ）を 1 つ決め，もう一方がその数を当てようとする．もちろん 1 回で当たることは稀で，出題者は解答者側の予想がどの程度正解に近いかヒント情報（**MOO 積**と呼ぶ）を提供し，解答者はヒント情報を元に新たな予想をたてる．これを正解が出るまで繰り返し，その予想回数の少なさを競うものである．

MOO 積は，正解と質問の数を各桁ごとに比較して，同じ数字が同じ桁に現れる回数 (Bull) と同じ数字が別の桁に現れる回数 (Cow) の 2 つ組とする．たとえば正解が 7412 のとき，予想した数が 0123 なら MOO 積は (0, 2) (= 2 Cows), 予想した数が 1470 なら (1, 2) (= 1 Bull, 2 Cows) となる．この場合のゲーム進行の例を**表 -1** に示す．

正解 : 7412

回数	予想	MOO 積 (Bulls,Cows)
1	0123	(0, 2)
2	1245	(0, 3)
3	2671	(0, 3)
4	4712	(2, 2)
5	7412	(4, 0)

表 -1　ゲームの進行例

このゲームは古くから存在し海外でも広く知られていて，イギリスで作られ学校の生徒の間で広まったとの話もある [1]．海外ではいろいろな名前で呼ばれていて，以下のような呼び方が知られている．

MOO, Bulls & Cows, Hit & Blow, Mastermind

数当てゲームにはルールが少し異なるバリエーションが存在し，1973 年にイギリスのインビクタ社から発売され大ヒットしたマスターマインド (Mastermind) というボードゲームが有名である．このゲームでは，0 ～ 9 の数字 4 桁を用いる代わりに 6 色のピン 4 本を用い，色の選択肢が減る分同じ色の重複を許して正解を決める．

このように数当てゲームでは，桁数，数字の種類・個数，同じ数字の繰り返しを許すか否か，を変えることで違うルールとして遊ぶことができるが，本稿では n 桁で各桁に重複のない 0, 1,..., 9 の数字を用いた数当てゲームを考えることにする．

まずは手始めに，出題者側であるプログラムを作成する．すなわちこれは正解を 1 つ定め，解答者である人間が入

力する予想に応じてヒント情報を対話的に返し，正解が入力されるまでこの手順を繰り返すプログラムである．次に，簡単な戦略を用いた解答者側のプログラムを示す．こちらは人間が指定する任意の数に対し，コンピュータが正解を求めるまでの予想の過程を出力するプログラムである．

出題プログラム

まずは出題者側にたって，乱数でMOO数を決め，解答者からの予想に応じてMOO積を返すプログラムを生成することにしよう．手始めに，データ構造として以下を用意する[☆1]．

```
import Random
import List ((\\), nub)     -- Listのモジュールを取り込み (\\), nub を使う
import Char (isDigit)

type PackedInt = Int
type UnpackedInt = [Int]
type MOOProduct = (Int, Int)

showUnpackedInt :: UnpackedInt -> String
showUnpackedInt ds = concatMap show ds
    -- 定義:concatMap f = concat . map f

showMOOProduct :: MOOProduct -> String
showMOOProduct (bulls, cows) =
  "  Bulls: " ++ show bulls ++ ", Cows: " ++ show cows
```

type宣言は，前号にも現れた既存の型に別名を付ける構文で，ここではn桁以下の整数を型PackedIntで，その整数を各桁に分割しn個の$0 \leq x \leq 9$の整数のリストで表したものをUnpackedIntとする．

この2通りの整数表記は，以下の関数により互いに変換できる．関数unpackで桁数nを与えているのは，元の数が10^{n-1}未満だった場合，上位の位に0をパディングする必要があるからである．

```
pack :: UnpackedInt -> PackedInt
pack ds = foldl (\ x y -> 10*x + y) 0 ds

unpack :: Int -> PackedInt -> UnpackedInt
unpack 1 d = [d]
unpack (n+1) d = unpack n q ++ [r]
  where
    (q, r) = d `divMod` 10
```

packで用いたfoldlはたたみ込み演算子*fold*の一種で，リストを扱う多くの関数定義に用いられている．*fold*はリストを辿る方向に応じて，以下のように*foldr*, *foldl*として定義される．

$$foldr\ (\oplus)\ a\ [x_1, x_2, \ldots, x_n] = x_1 \oplus (x_2 \oplus (\cdots\ (x_n \oplus a)\ \cdots))$$
$$foldl\ (\oplus)\ a\ [x_1, x_2, \ldots, x_n] = (\cdots\ ((a \oplus x_1) \oplus x_2)\ \cdots) \oplus x_n$$

この演算子を用いたリストを扱う関数定義の例を以下に示す．実際に処理系の標準プレリュードに含まれる定義は，*foldr*の代わりに評価順を考慮した同等の関数で置き換えられていることもあるが，大まかな処理の内容は変わらない．

```
{- fold を用いたリスト関数の例
  sum    = foldr (+) 0
  and    = foldr (&&) True
  concat = foldr (++) []
```

☆1 このプログラムはhttp://www.sampou.org/haskell/ipsj/から取ることができる．

```
  length = foldr (\ _ n -> n+1) 0
-}
```

ここでユーティリティ関数として以下を用意する．disjoint は，n 桁の各数字が異なっているか判定する関数，mooNum は与えられた MOO 積から正解が見つかったか否かを返す関数である．

```
disjoint :: UnpackedInt -> Bool
disjoint xs = length xs == length (nub xs)

mooNum :: Int -> MOOProduct -> Bool
mooNum n (bulls, cows) = bulls == n && cows == 0
```

nub は List ライブラリに含まれ，リストの重複する要素を除く関数である．手順としては，リストの要素を先頭から検査し同じ要素が複数回現れた場合，初出のもの以外をすべて除去する．以下に動作の例を示す．

```
? nub "abracadabra" ↵
"abrcd"
```

nub の定義は省略するが，難しいものではないので自分で考えてみるのもよいであろう．

また関数 mooNum に現れる式 `bulls == n && cows == 0` は，&& が短絡評価 (short circuit) されるため，`cows == 0` の判定は `bulls == n` が真のときだけに限られる．このゲームの場合，`bulls == n` が真なら常に `cows == 0` も真となるので条件の後半は実は不要なのだが，mooNum の呼び出しごとに cows の比較も毎回行われ効率を落としているわけではないことを明記しておく．

乱数の生成には，Random ライブラリを用いることにする．関数型言語には参照透明性 (referencial transparency) といい，同じ式は常に同じ値を返す性質がある．これは乱数の持つ性質と矛盾するものであるが，Haskell では入出力モナド (IO monad) を用いることで，これを解決している．モナドについては今後本連載で詳しく取り扱うことにし，今回はその使い方だけ紹介しておく．

```
{- 乱数の使い方の例
  import Random

  sampledo :: IO ()
  sampledo = do
    randomGen <- newStdGen
    let randoms = randomRs (0, 10^n-1) randomGen
-}
```

do 記法は，その直後に I/O アクションを示す文を並べて配置することで，そのアクションを順番に行うことを指定する．I/O アクションとして以下の 3 種類の文が用意されている．

パターンへの束縛 $pat \leftarrow exp$
局所定義 let $v_1 = e_1 ; \cdots ; v_n = e_n$
式 exp

先の例では，変数 randomGen に新たな乱数生成器を割り当て，これを元に関数 randomRs で 0 以上 10^n 未満の乱数の無限リストを生成している．ここで乱数を 1 つではなく無限に生成しているのは，その中から各桁に同じ数字が含まれるものを除いて正解候補を定めているためである．

さて，出題プログラムに戻ることにしよう．

```
main :: IO ()
main = moo 4              -- コンパイルして生成されるのは4桁版
```

```
moo :: Int -> IO ()
moo n = do
  randomGen <- newStdGen
  let randoms = randomRs (0, 10^n-1) randomGen
        -- 0 <= r < 10^n を満たす乱数 r の無限列を生成
      answer = (head . filter disjoint . map (unpack n)) randoms
        -- n 桁の数字に重複のないものを1つ抽出
  loop n answer []

loop :: Int -> UnpackedInt -> [UnpackedInt] -> IO ()
loop n answer history = do
  putStr "Your guess? "                          -- 入力を促す
  guessStr <- getLine                            -- 改行までの文字列を入力
  if not (legal guessStr)
    then loop n answer history                   -- redo
    else let guess = unpack n (read guessStr :: Int) --  readでStringからIntへ変換
             mooProduct = score answer guess in
         if mooNum n mooProduct
           then do putStrLn ("  You got it in " ++
                             show (1+length history) ++ " guesses!")
           else do putStrLn (showMOOProduct mooProduct)
                   loop n answer (guess:history)
  where
    legal str = length str == n && and (map isDigit str)
```

関数 moo は桁数を引数にとり，桁数，正解，予想の履歴を実引数として関数 loop を呼び出す．関数 loop は標準入力から文字列として予想を受け取り，数として正しくない場合は再入力を求める．入力された予想は UnpackedInt へと変換され，正解との間で MOO 積が求められる．そこで正解が見つかったときはメッセージを表示して終了し，その他の場合は予想を履歴に追加し次の入力を受け取るため再度 loop を呼び出す．

mooProduct を求める際に用いた関数 score は以下のように定義される．

```
score :: UnpackedInt -> UnpackedInt -> MOOProduct
score xs ys = (bulls, cows)
  where
    (xs', ys') = unzip [ (x,y) | (x,y) <- zip xs ys, x /= y ]
    bulls = length xs - length xs'
    cows = length xs' - length (xs' \\ ys')
```

\\ はリストの差を求める関数である．すなわち ys の各要素 y に対し，xs の中で最初に現れる y を取り除いたリストが xs \\ ys となる．

```
? "abracadabra" \\ "abrcd" ↵
"aaabra"
```

"aaabra" は元の文字列の第 3,5,7,8,9,10 要素が残されたリストである（先頭の 'a' は第 0 要素とする）．この性質により，(xs ++ ys) \\ xs == ys が常に成立することは自明であろう．

ここまでのプログラムにより，moo 4 のように引数に桁を指定することで，数当てゲームの対戦を行うことができるようになる．MOO 数はすべての位が異なることを条件としていることから，桁数 n は 11 以上にできないことに注意されたい．またゲームを途中で止めるには Ctrl-C を入力すればよい．

```
Prelude> :l Moo.hs ↵
Main> moo 4 ↵
Your guess? 0123
  Bulls: 0, Cows: 2
Your guess? 1245
```

```
  Bulls: 0, Cows: 3
Your guess? 2671
  Bulls: 0, Cows: 3
Your guess? 4712
  Bulls: 2, Cows: 2
Your guess? 7412
  You got it in 5 guesses!
```

解答プログラム

次に解答者側にたって，予想を与えるとMOO積を返す出題プログラムが存在すると仮定し，予想とMOO積の履歴から正解を求めるためのプログラムを作成することにする．

数当てゲームの中でもMOOやマスターマインドのように桁数が4と小さいものに対しては，完全探索によって総質問回数を最小にするための戦略がすでに求められている．詳しくは文献3）に解説されているので，興味のある読者は参照して欲しい．最初に挙げた表-1のゲーム進行例は，文献2)で示されたMOOの最小質問戦略を用いたもので，この戦略によれば平均質問回数5.213回で正解が得られることが示されている．

ここでは簡単な手法として，正解候補の一覧を作成しある時点までに得られたヒント情報と照らし合わせて不整合のないものを総当たり的に調べ上げる戦略をとることにする．

まずは以下の関数genによって，各桁に重複する数字のないn桁のUnpackedIntをすべて，リストの形で生成することができる．この関数genは，リストの内包表記を用いた4桁版のMOO数候補を生成する関数gen4を，n桁用に一般化したものである．

```
digits :: [Int]
digits = [0..9]

gen :: Int -> [UnpackedInt]
gen n = gens n []
  where
    gens :: Int -> [Int] -> [UnpackedInt]
    gens 0 rs = if disjoint rs then [rs] else []
    gens (n+1) rs = concatMap (\r -> gens n (rs++[r])) digits

{- 参考: gen4 == gen 4
gen4 = [ ds | p<-digits, q<-digits, r<-digits, s<-digits,
             let ds=[p,q,r,s], disjoint ds]
-}
```

リストの内包表記は，以下のルールにより内包表記を用いない別の定義に置き換えられることが知られていて，genはこのルールを用いて一般化されている．

1. [e | b, Q] = if b then [e | Q] else []
2. [e | $p \leftarrow l$, Q] = concatMap f l where f p = [e | Q]
3. [e | let $decls$, Q] = let $decls$ in [e | Q]

実際に関数genにより候補が生成される様子を以下に示す．

```
{- gen による候補の生成
  gen 4 = gens 4 []
        = concatMap (\ p -> gens 3 ([]++[p])) digits
        = concatMap [ gens 3 [p] | p <- digits ]
  {-
```

```
  gens 3 [p]       = concatMap (\ q -> gens 2 ([p]++[q])) digits
                   = concatMap [ gens 2 [p,q] | q <- digits ]
  gens 2 [p,q]     = concatMap (\ r -> gens 1 ([p,q]++[r])) digits
                   = concatMap [ gens 1 [p,q,r] | r <- digits ]
  gens 1 [p,q,r]   = concatMap (\ s -> gens 0 ([p,q,r]++[s])) digits
                   = concatMap [ gens 0 [p,q,r,s] | s <- digits ]
  gens 0 [p,q,r,s] = if disjoint [p,q,r,s] then [[p,q,r,s]] else []
  -}
        = concatMap [
            concatMap [
              concatMap [
                concatMap [
                  if disjoint [p,q,r,s] then [[p,q,r,s]]
                                        else []
                | s <- digits ]
              | r <- digits ]
            | q <- digits ]
          | p <- digits ]
  {-
  -- s,r,q,p の順に 0..9 の範囲をとり，disjoint なものだけ残す
  [0,0,0,0],[0,0,0,1],[0,0,0,2], .. ,[0,0,0,9],
  [0,0,1,0],[0,0,1,1],[0,0,1,2], .. ,[0,0,1,9],
     ..,
  [0,0,9,0],[0,0,9,1],[0,0,9,2], .. ,[0,0,9,9],
  [0,1,0,0],[0,1,0,1],[0,1,0,2], .. ,[0,1,0,9],
  [0,1,1,0],[0,1,1,1],[0,1,1,2], .. ,[0,1,1,9],
  [0,1,2,0],[0,1,2,1],[0,1,2,2],[0,1,2,3]  ==> 最初の disjoint !
  -}
        = [[0,1,2,3],[0,1,2,4],..,[0,1,2,9],[0,1,3,2],....,[9,8,7,6]]
-}
```

解答プログラムである関数 solver は引数に桁と正解を受け取り，補助関数 solver1, solver2 を用いて戦略に基づいた予想を行う．

```
solver :: Int -> Int -> IO ()
solver n answer = putStr result
  where
    score1 = score (unpack n answer)          -- 予想からMOO積を返す関数
    candidates = gen n
    result = solver1 n score1 [] candidates

solver1 :: Int -> (UnpackedInt -> MOOProduct) ->
           [UnpackedInt] -> [UnpackedInt] -> String
solver1 n score1 history (guess:rs) = solver2 n score1 history rs guess
```

score1 は高階関数で，予想の候補を入力とし正解との間で MOO 積を求めるために用いる．このように 2 引数関数に引数を 1 つ指定したものを新たな関数として用いる例は関数プログラミングでは多く見受けられる．

```
solver2 :: Int -> (UnpackedInt -> MOOProduct) ->
           [UnpackedInt] -> [UnpackedInt] -> UnpackedInt -> String
solver2 n score1 history rs guess =
  let mooProduct = score1 guess in
  if mooNum n mooProduct
    then "Computer found " ++ showUnpackedInt guess
         ++ " in " ++ show (1+length history) ++ " guesses."
    else let rs' = [ r | r <- rs, score guess r == mooProduct ] in
         "Computer guesses " ++ showUnpackedInt guess ++ "\n"
         ++ showMOOProduct mooProduct ++ "\n"
         ++ solver1 n score1 (guess:history) rs'
```

関数 solver2 では，次の予想を guess とし MOO 数を求める．正解だった場合は何ステップ要したかを出力し終了する．不正解だった場合，候補の一覧から guess との MOO 積が mooProduct にならないものを除いて新たな候補の一覧とし，次の予想をたてにいく．

これにより数当てゲームを解くプログラムができたことになる．動かすには solver 4 7412 のように，桁数と正解を Int で指定すればよい．

```
Prelude> :l Moo.hs ↵
Main> solver 4 7412 ↵
Computer guesses 0123
  Bulls: 0, Cows: 2
Computer guesses 1045
  Bulls: 0, Cows: 2
Computer guesses 2354
  Bulls: 0, Cows: 2
Computer guesses 3406
  Bulls: 1, Cows: 0
Computer guesses 3517
  Bulls: 1, Cows: 1
Computer guesses 5207
  Bulls: 0, Cows: 2
Computer found 7412 in 7 guesses.
```

これまで見てきたように，数当てゲームはプログラミングの難易度としては手頃な問題なので，読者の方も一度使い慣れた言語で解いてみてはいかがだろうか．同じ問題を別の言語を用いて解くことによって，ここで挙げた関数型言語によるプログラムの特徴がより明らかになるであろう．

くしくも前号の「切符問題」と同様，簡単なゲームを解くプログラミングを紹介したが，これは関数プログラミングがトイ (toy) プログラミングしか扱えないということを示しているわけではない．筆者らの趣味という面もあるが，身近な例題を扱うことで内容の見通しを良くし，すでに用意されている組み込み関数を多用することで簡潔なプログラミングをするためのエッセンスを知ってもらうのが狙いだからである．実際，Haskell のコンパイラである GHC もそれ自身 Haskell で記述されているし，Web サーバ，メールサーバ (smtp)，テキストエディタ，Perl インタプリタなどを Haskell 言語で書く試みが行われていることも記しておく [4]．

参考文献

1) ℵ0, Computer Recreations, Software Practice and Experience, Vol.1, No.2, pp.201-204 (1971).
2) 田中哲朗: 数当てゲーム MOO の最小質問戦略と最強戦略, 第3回ゲームプログラミングワークショップ (1996). http://www.tanaka.ecc.u-tokyo.ac.jp/~ktanaka/moo/moo.html
3) 松原　仁, 竹内郁雄 編: bit別冊ゲームプログラミング, 共立出版 (1997).
4) Haskell in Practice, http://haskell.org/practice.html

（平成17年4月16日受付）