■ 連載

Haskell プログラミング

ペンシルパズルを解く

山下 伸夫（(株) タイムインターメディア）
nobsun@sampou.org

ペンシルパズル

鉛筆を使って解くパズルをペンシルパズル☆1 という．代表的なものにクロスワードパズルがあるが，今回はペンシルパズルの中でも数理パズルという分類に属するパズルを解くプログラムを取り上げる．「数理パズル」は日常言語や生活習慣に依存する知識を必要とせず，「初期条件とルールのみから演繹で解けるパズル」かつ「解が一意に定まるパズル」という意味である．

数独 (Sudoku)

2005 年の春頃に英国で火が付いて，またたく間に大ブームになったパズルである．この英国でのブームが報道されたりもしたので聞いたことがある方もいるかもしれない．もっとも，日本のペンシルパズルファンにとってはすでに 20 年も慣れ親しんだパズルである．まずこの数独 [1), 2)] を取り上げる．このパズルのルールは簡単だ．**図 -1** のように

8				3	4		5	
		2						1
	1		9					
		8			9			6
5				1				8
6			4			7		
					1		7	
2						1		
	9		5	6				2

8	6	7	1	3	4	2	5	9
9	5	2	6	7	8	3	4	1
4	1	3	9	5	2	6	8	7
7	4	8	3	2	9	5	1	6
5	3	9	7	1	6	4	2	8
6	2	1	4	8	5	7	9	3
3	8	6	2	4	1	9	7	5
2	7	5	8	9	3	1	6	4
1	9	4	5	6	7	8	3	2

（ニコリ刊：『激辛数独1』裏表紙より許可を得て引用）

図-1　数独の問題（左）と解（右）

☆1　ペンシルパズルにどのようなものがあるかはパズル専門の出版社「ニコリ」のサイト [4)] を参照．

9×9の空いているマス目の中に次の3つのルールに従って1から9までの数字のどれかを入れる．

1. タテの列（9列ある）のどの列にも1から9までの数字が1つずつ入る
2. ヨコの行（9行ある）のどの行にも1から9までの数字が1つずつ入る
3. 3×3のボックス（9ボックスある）のどのボックスにも1から9までの数字が1つずつ入る

このパズルのソルバをHaskellで書く[☆2]．

数独パズルの構成要素

通常数独といえば9行，9列，9ボックスであるが，16行，16列，16ボックスというものや，25行，25列，25ボックスなどというものも存在する．ここでは9行，9列，9ボックスのものに固定するが，分かりやすいようにサイズには名前を付けておく．

```
sudokuBase, sudokuSize :: Int
sudokuBase = 3
sudokuSize = sudokuBase ^ 2
```

マス目に入れる数字（Sudoku）の型はIntとする．その範囲は，1～9である．また，マス目が空白であることを0で表すことにする．

```
type Sudoku = Int
sudokuElm :: [Sudoku]
sudokuElm = [1..sudokuSize]
vacant :: Sudoku
vacant = 0
```

パズルの盤面は数字のリストのリストで表現する．

```
type SudokuBoard = [[Sudoku]]
```

たとえば，図-1の数独の問題盤面は，

```
sample :: SudokuBoard
sample = [[8,0,0,0,3,4,0,5,0]
         ,[0,0,2,0,0,0,0,0,1]
         ,[0,1,0,9,0,0,0,0,0]
         ,[0,0,8,0,0,9,0,0,6]
         ,[5,0,0,0,1,0,0,0,8]
         ,[6,0,0,4,0,0,7,0,0]
         ,[0,0,0,0,0,1,0,7,0]
         ,[2,0,0,0,0,0,1,0,0]
         ,[0,9,0,5,6,0,0,0,7]]
```

である．次に，「マス目の位置」を列と行とで表現するPositionを用意する．

```
type Position = (Int,Int) -- (列，行)
```

図-1の問題において，一番上のヨコ1行に現れている4つの数字のうち3が入っている「マス目の位置」は(4,0)である．Haskellのリストでは要素のインデックスは0から始まるので，件の3の位置は，4列目，0行目とする．

数独ソルバの構成

パズルの解は再帰で考えるのがやさしい．

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

1. 盤面に空白のマス目がなければ，その盤面が解盤面
2. 空白のマス目があれば，そこに適切な数字を入れて作った新しい盤面を解いて得られる盤面が解盤面

数独パズルとして公開されている問題盤面は唯一解を持つように作成されているが，一般に 1 つの盤面の解盤面は複数になる．そこで数独ソルバ sudoku の型は数独盤面を引数とし，解盤面のリストを返す関数とする．

```
sudoku :: SudokuBoard -> [SudokuBoard]
sudoku b
 = case vacantPositions b of
    [] -> [b]                              -- 空白のマス目が無い → 終了
    ps -> case nextVacant b ps of          -- 空白のマス目が有る
            (p,xs) ->                      -- 空白のマス目の位置 p と候補リスト xs
                      concatMap sudoku     -- 新しい盤面をそれぞれ解いて集める
                    $ map (putCell b)      -- 各ペアで更新した盤面を作る
                    $ [(p,x) | x <- xs]    -- 位置と候補のペアのリスト
```

sudoku はいくつかの関数から構成されている．

vacantPositions 「盤面」を取り，数字が入っていない「マス目の位置」のリストを返す
nextVacant 「盤面」と，数字が入っていない「マス目の位置」のリストとを取り，空白の「マス目の位置」とその位置の空白のマス目に入る候補の「数字」のリストのペアを返す
putCell 「盤面」と，「マス目の位置」と「数字」のペアとを取り，更新した「盤面」を返す

これらの関数の型を考えると，

```
vacantPositions :: SudokuBoard -> [Position]
nextVacant      :: SudokuBoard -> [Position] -> (Position,[Sudoku])
putCell         :: SudokuBoard -> (Position,Sudoku) -> SudokuBoard
```

となる．この型宣言は上の関数の仕様の一部を表現している．Haskell の型システムを利用すると，sudoku が上の 3 つの関数を型の上で誤って使っていないかチェックすることができる．それには，関数の実装を以下のように書く．

```
vacantPositions = error "(vacantPositions) is not yet implemented"
nextVacant      = error "(nextVacant) is not yet implemented"
putCell         = error "(putCell) is not yet implemented"
```

こうしておいて，ここまでのコードを sudoku0.hs というファイル名で保存する．対話モードインタプリタ ghci の環境にロードして，以下のように無事ロードできれば，少なくとも型チェックはパスしていることになる．すなわち，上位の関数と下請関数との間で型として表現された仕様に矛盾がないことが，下請関数の実装を完成させる前に分かる．この手法はトップダウン指向でインクリメンタルにプログラムを書くときに有用である．

```
% ghci
   ___         ___ _
  / _ \ /\  /\/ __(_)
 / /_\// /_/ / /  | |      GHC Interactive, version 6.4.1, for Haskell 98.
/ /_\\/ __  / /___| |      http://www.haskell.org/ghc/
\____/\/ /_/\____/|_|      Type :? for help.

Loading package base-1.0 ... linking ... done.
Prelude> :load sudoku0.hs
Compiling Main             ( sudoku0.hs, interpreted )
Ok, modules loaded: Main.
*Main>
```

もちろん数独ソルバ関数 sudoku を数独問題 sample に適用して評価はまだできない（評価するとエラーになる）．

```
*Main> solve sample
*** Exception: (vacantPositions) is not yet implemented
*Main>
```

上の 3 つの関数の実装は以下のようにする.

■ putCell　指定した「マス目の位置」に指定した「数字」を置く.

```
putCell b ((i,j), x) = ls0++(xs0++x:xs1):ls1
                       where
                         (ls0,l:ls1) = splitAt j b
                         (xs0,_:xs1) = splitAt i l
```

■ nextVacant 「マス目の位置」とそこに入る候補の「数字」のリストのペアのうち, 候補リストが一番短いものの, 位置と候補リストの組を返す.

```
nextVacant b ps = minimumBy cmp [(p,candidate b p) | p <- ps]
  where
    (_,xs) `cmp` (_,ys) = length xs `compare` length ys
```

nextVacant が利用する candidate 関数が数独ソルバの肝である. 指定した「マス目の位置」に入る「数字」の候補リストを求める. col, row, box はそれぞれ指定された「マス目の位置」にあるマス目が所属する「列」「行」「ボックス」にあるマス目（に入っている「数字」）のリストを求める関数である. 自分が属しているグループですでに使われている数字をマス目に入れ得る数字のリスト (sudokuElm) から除いたものが, 候補リストである.

```
candidate :: SudokuBoard -> Position -> [Sudoku]
candidate b p = sudokuElm \\ nub (concat $ map (\ c -> c b p) [col,row,box])

col, row, box :: SudokuBoard -> Position -> [Sudoku]
col b (i,_) = transpose b !! i
row b (_,j) = b !! j
box b (i,j) = concat
            $ map (take sudokuBase) $ map (drop $ (i `div` sudokuBase) * sudokuBase)
            $ take sudokuBase $ drop ((j `div` sudokuBase) * sudokuBase) b
```

■ vacantPositions

「盤面」にあるすべての「マス目の位置」とそこに入っている「数字」のペアのリストを生成し, 数字が入っていない (0 が入っている) ものだけを選び, その位置のリストを返す.

```
vacantPositions b = map fst
                  $ filter (\ (p,x) -> vacant == x)
                  $ concatMap (\ (j,xs) -> zipWith (\ i x -> ((i,j),x)) [0..] xs )
                  $ zip [0..] b
```

ソルバの実行例

ここまでのコードを sudoku1.hs というファイル名で保存し（先頭部分で List モジュールをインポートしておくこと), 例題 (sample) を解く（この例題は人間が解くと相当むずかしい問題の部類に入る).

```
Prelude> :load sudoku1.hs
Compiling Main             ( sudoku1.hs, interpreted )
Ok, modules loaded: Main.
*Main> :set +s
*Main> sudoku sample
```

```
[[[8,6,7,1,3,4,2,5,9],[9,5,2,6,7,8,3,4,1],[4,1,3,9,5,2,6,8,7],[7,4,8,3,2,9,5,1,6]
,[5,3,9,7,1,6,4,2,8],[6,2,1,4,8,5,7,9,3],[3,8,6,2,4,1,9,7,5],[2,7,5,8,9,3,1,6,4]
,[1,9,4,5,6,7,8,3,2]]]
(5.04 secs, 303990824 bytes)
```

Show クラスと Read クラス

数独は解けるようになったが，文字列として問題を与えたり，1 行ずつ表示したりできた方が便利である．そこで，数独盤面データの読み書きのための関数を用意する．

■ showSudokuBoard　数独盤面データを文字列に変換する．

```
showSudokuBoard :: SudokuBoard -> String
showSudokuBoard b = unlines $ map (unwords . map show) b
```

ここで，show は Show クラスのメソッドで，Show クラスのインスタンスである型のデータを文字列に変換する．この show に渡るデータは数独のマス目に入る数字 Sudoku でこれは Int である．Int はプレリュードで Show クラスのインスタンスであることが宣言されているので，show メソッドで Sudoku が文字列に変換される．unwords はプレリュード関数で，文字列のリストを取り，各文字列の間に空白を 1 つ挟んで 1 本の文字列に変換する．unlines もプレリュード関数で，文字列のリストを取り，各文字列の間に改行を 1 つ挟んで 1 本の文字列に変換する．

■ readSudokuBoard　文字列を数独盤面データに変換する．

```
readSudokuBoard :: String -> SudokuBoard
readSudokuBoard s = map (map read) $ map words $ lines s
```

lines はプレリュード関数で，文字列を改行ごとに分割して文字列のリストにする．lines もプレリュード関数で，文字列を空白（連続する空白は 1 つの空白と見なす）ごとに分割して文字列のリストにする．read は Read クラスのインスタンス上に定義された関数で，文字列をその型のデータに変換する．この read が返すべきデータは数独のマス目に入る数字 Sudoku でこれは Int である．Int はプレリュードで Read クラスのインスタンスであることが宣言されているので，read 関数で文字列が Sudoku に変換される．
showSudokuBoard と readSudokuBoard は表裏の関係にあって，2 つの関数の合成
(readSudokuBoard . showSudokuBoard) は恒等写像になる．

```
*Main> sample
[[8,0,0,0,3,4,0,5,0],[0,0,2,0,0,0,0,0,1],[0,1,0,9,0,0,0,0,0],[0,0,8,0,0,9,0,0,6]
,[5,0,0,0,1,0,0,0,8],[6,0,0,4,0,0,7,0,0],[0,0,0,0,0,1,0,7,0],[2,0,0,0,0,0,1,0,0]
,[0,9,0,5,6,0,0,0,2]]
*Main> (readSudokuBoard . showSudokuBoard) sample
[[8,0,0,0,3,4,0,5,0],[0,0,2,0,0,0,0,0,1],[0,1,0,9,0,0,0,0,0],[0,0,8,0,0,9,0,0,6]
,[5,0,0,0,1,0,0,0,8],[6,0,0,4,0,0,7,0,0],[0,0,0,0,0,1,0,7,0],[2,0,0,0,0,0,1,0,0]
,[0,9,0,5,6,0,0,0,2]]
```

showSudokuBoard と readSudokuBoard を使って，SudokuBoard を Show クラスと Read クラスのインスタンスであることを宣言する☆3．

```
instance Show SudokuBoard where
  show = showSudokuBoard

instance Read SudokuBoard where
  readsPrec _ str = [(readSudokuBoard str, "")]
```

☆3 Haskell 98 の仕様範囲では型シノニムに対してインスタンス宣言はできない．これ以降のコードを試すには ghci を -fglasgow-exts および -fallow-overlapping-instances フラグを付けて起動すること．

SudokuBoard 型の定義は，Show クラスおよび Read クラスの定義とは独立していることに注意してほしい．

```
*Main> sample
8 0 0 0 3 4 0 5 0
0 0 2 0 0 0 0 0 1
0 1 0 9 0 0 0 0 0
0 0 8 0 0 9 0 0 6
5 0 0 0 1 0 0 0 8
6 0 0 4 0 0 7 0 0
0 0 0 0 0 1 0 7 0
2 0 0 0 0 0 1 0 0
0 9 0 5 6 0 0 0 2
```

Puzzle クラスによる抽象

前章で作成した数独を解くプログラムには，数独パズル特有の部分と，他のペンシルパズルにも共通する部分が含まれている．この2つをプログラムコード上で分離し，共通部分をモジュールとして抽象して再利用できるようにする．

```
module Puzzle where

import List

type Board a = [[a]]       -- 盤面
type Position = (Int,Int)  -- マス目位置

putCell :: Board a -> (Position, a) -> Board a
putCell b ((i,j),x) = ls0 ++ (cs0 ++ (x:cs1)):ls1
                      where (ls0,l:ls1) = splitAt j b
                            (cs0,_:cs1) = splitAt i l

class Eq a => Puzzle a where
  puzzleElm  :: [a]                          -- マス目に入れるものリスト
  vacant     :: a                            -- マス目に何も入っていないことを表すもの
  candidate  :: Board a -> Position -> [a] -- 指定したマス目の位置に入れるものの候補（リスト）

solve :: Puzzle a => Board a -> [Board a]
solve b = case vacantPositions b of
            [] -> [b]
            ps -> case nextVacant b ps of
                    (p,xs) -> concatMap solve
                            $ map (putCell b)
                            $ [ (p,x) | x <- xs ]

vacantPositions :: Puzzle a => Board a -> [Position]
vacantPositions b = map fst
                    $ filter (\ (p,x) -> vacant == x)
                    $ concatMap (\ (j,xs) -> zipWith (\ i x -> ((i,j),x)) [0..] xs )
                    $ zip [0..] b

nextVacant :: Puzzle a => Board a -> [Position] -> (Position,[a])
nextVacant b ps = minimumBy cmp [ (p, candidate b p) | p <- ps ]
   where
     (_,xs) `cmp` (_,ys) = length xs `compare` length ys

showBoard :: (Puzzle a, Show a) => Board a -> String
readBoard :: (Puzzle a, Read a) => String -> Board a
```

```
showBoard = unlines . map (unwords . map show)
readBoard = map (map read) . map words . lines
```

Puzzle クラスは，マス目に入れるものの「データ型を規定するクラス」である．この規定を満たすデータのリストのリストがパズル盤として成立するための条件を規定している．

```
class Eq a => Puzzle a where
```

の部分は Puzzle クラスが，Eq クラスのインスタンスであるようなデータ型に対して定義されることを示す．データ型が Puzzle クラスのインスタンスであるためには，まず，Eq クラスのインスタンスでなければならない．定義の後の部分では，このクラスのメソッド名とその型を定義する．

puzzleElm　マス目に入れるもののリスト
vacant　マス目に何も入っていないことを示すもの
candidate　盤面とマス目の位置を引数としてとり，指定された盤面において指定された位置にパズルのルールを満たして入れることのできるものの候補（リスト）を返すメソッド

実際のクラスのメソッド定義は，メソッド名と型だけを規定している．メソッドの意味の一部は型で表現できるが，正確な意味までは規定できないことに注意すること．

上のモジュールで定義したパズルソルバ solve の型宣言

```
solve :: Puzzle a => Board a -> [Board a]
```

は，マス目に入れるものの型が Puzzle クラスのインスタンスであるという文脈で solve が定義されることを示している．すなわち，このモジュールを使えば，マス目に入れるものとして定義した型が Puzzle クラスのインスタンスであるということを宣言する（上の 3 つのメソッド puzzleElm，vacant，candidate の実装を提供する）だけで，ソルバを改めて定義しなくてもよい．また，パズル盤面から文字列への変換関数 showBoard，文字列からパズル盤面への変換関数 readBoard も同様でマス目に入れるものの型が Puzzle クラスのインスタンスであり同時に Show クラス，Read クラスのインスタンスであれば，このモジュールの実装を利用できる．

以下はこの Puzzle モジュールを使うように書き換えた数独の問題を解くプログラムである．

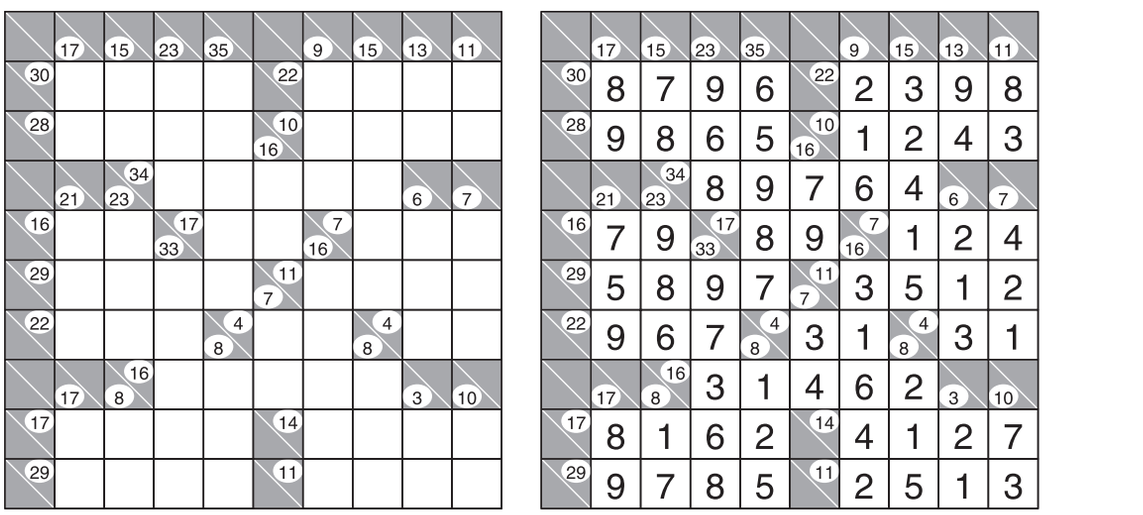
```
import List
import Puzzle

type Sudoku = Int

instance Puzzle Sudoku where
  puzzleElm     = [1..sudokuSize]
  vacant        = 0
  candidate b p = puzzleElm \\ nub (concatMap (\ c -> c b p) [col,row,box])

sudokuBase, sudokuSize :: Int
sudokuBase = 3
sudokuSize = sudokuBase ^ 2

col, row, box :: Board Sudoku -> Position -> [Sudoku]
col b (i,_) = transpose b !! i
row b (_,j) = b !! j
box b (i,j) = concat
            $ map (take sudokuBase) $ map (drop $ (i `div` sudokuBase) * sudokuBase)
            $ take sudokuBase $ drop ((j `div` sudokuBase) * sudokuBase) b
```

```
sudoku :: Board Sudoku -> [Board Sudoku]
sudoku = solve

showSudoku :: Board Sudoku -> String
showSudoku = showBoard
readSudoku :: String -> Board Sudoku
readSudoku = readBoard

instance Show (Board Sudoku) where
  show = showBoard
instance Read (Board Sudoku) where
  readsPrec _ s = [(readSudoku s,"")]

sample :: Board Sudoku
sample = read sampledata
sampledata = unlines
             ["8 0 0 0 3 4 0 5 0"
             ,"0 0 2 0 0 0 0 0 1"
             ,"0 1 0 9 0 0 0 0 0"
             ,"0 0 8 0 0 9 0 0 6"
             ,"5 0 0 0 1 0 0 0 8"
             ,"6 0 0 4 0 0 7 0 0"
             ,"0 0 0 0 0 1 0 7 0"
             ,"2 0 0 0 0 0 1 0 0"
             ,"0 9 0 5 6 0 0 0 7"]
```

カックロ (Kakro)

Puzzle モジュールは魔方陣を作成するようなパズルに適用できることは容易に推察できるが，ここでは，数独とは別のペンシルパズル，カックロ (Kakro)[3] に適用する．カックロは**図 -2** のように空いているマス目の中に次の 4 つのルールに従って 1 から 9 までの数字のどれかを入れるパズルである．

1. ナナメの線の右上に出ている数字は，その右の空白のマス目のつながりに入る数字の合計
2. ナナメの線の左下に出ている数字は，その下の空白のマス目のつながりに入る数字の合計
3. タテへの 1 つの空白のマス目のつながりには，同じ数字は入らない
4. ヨコへの 1 つの空白のマス目のつながりには，同じ数字は入らない



（ニコリ刊：『カックロ17』裏表紙より許可を得て引用）

図-2　カックロの問題（左）と解（右）

```
import List
import Puzzle

data Kakro = K Int | W Int Int

instance Eq Kakro where
  (K x)   == (K y)   = x == y
  (W _ _) == (W _ _) = True
  _       == _       = False

instance Show Kakro where
  show (K x)   = show x
  show (W v h) = show v ++ "\\" ++ show h

instance Read Kakro where
  readsPrec _ s = case break ('\\'==) s of
                    (_,[])  -> [(K (read s), "")]
                    (v,_:h) -> [(W (read v) (read h),"")]

instance Puzzle Kakro where
  puzzleElm = map K [1 .. 9]
  vacant    = K 0
  candidate b (i,j) = intersect vcand hcand
    where vcand = nub (concat vcands) \\ vfixeds      -- タテ方向の候補
          hcand = nub (concat hcands) \\ hfixeds      -- ヨコ方向の候補
          vcands = filter (include vfixeds) $ kakroCalc vwa (length vgroup)
          hcands = filter (include hfixeds) $ kakroCalc hwa (length hgroup)
          vfixeds = filter (not . (vacant ==)) vgroup
          hfixeds = filter (not . (vacant ==)) hgroup
          (W vwa _, vgroup) = getGroup j (transpose b !! i)
          (W _ hwa, hgroup) = getGroup i (b !! j)

getGroup :: Int -> [Kakro] -> (Kakro,[Kakro])
getGroup i ks = case splitAt i ks of
                  (ss,ts) -> case break (W 0 0 ==) ts of
                               (us,_) -> case break (W 0 0 ==) (reverse ss) of
                                           (rs,w:_) -> (w,reverse rs++us)

include :: Eq a => [a] -> [a] -> Bool        -- 部分集合かを判定する述語
include xs ys = null (xs \\ ys)

comb :: [a] -> Int -> [[a]]                  -- 組合せの生成
comb _  0 = [[]]
comb [] _ = []
comb (x:xs) n = [ (x:xs') | xs' <- comb xs (n-1) ] ++ comb xs n

kakroCalc :: Int       -- 和
          -> Int       -- マス目の数
          -> [[Kakro]] -- 指定された和になる Kakro 数字（1 から 9 まで）の組合せ
kakroCalc s n = [ xs | xs <- comb puzzleElm n, s == sum (map (\ (K x) -> x) xs) ]

kakro :: Board Kakro -> [Board Kakro]
kakro = solve

instance Show (Board Kakro) where
  show = showBoard

instance Read (Board Kakro) where
  readsPrec _ s = [(readBoard s, "")]
```

```
sample :: Board Kakro
sample = read sampledata

sampledata :: String
sampledata
 = unlines
   ["0\\0  17\\0  15\\0  23\\0  35\\0   0\\0   9\\0  15\\0  13\\0  11\\0"
   ,"0\\30  0      0      0      0      0\\22  0      0      0      0   "
   ,"0\\28  0      0      0      0     16\\10  0      0      0      0   "
   ,"0\\0  21\\0  23\\34  0      0      0      0      0      6\\0   7\\0"
   ,"0\\16  0      0     33\\17  0      0     16\\7   0      0      0   "
   ,"0\\29  0      0      0      0      7\\11  0      0      0      0   "
   ,"0\\22  0      0      0      8\\4   0      0      8\\4   0      0   "
   ,"0\\0  17\\0   8\\16  0      0      0      0      0      3\\0  10\\0"
   ,"0\\17  0      0      0      0      0\\14  0      0      0      0   "
   ,"0\\29  0      0      0      0      0\\11  0      0      0      0   "]
```

sample は図 -2 の問題である．このコードを kakro.hs としてセーブし，解くと以下のようになる（出力はカラム位置をエディタで編集調整してある）．

```
Prelude> :l kakro.hs
Compiling Puzzle           ( ./Puzzle.hs, interpreted )
Compiling Main             ( kakro.hs, interpreted )
Ok, modules loaded: Main, Puzzle.
*Main> head $ solve sample
0\0  17\0  15\0  23\0  35\0   0\0   9\0  15\0  13\0  11\0
0\30  8     7     9     6     0\22  2     3     9     8
0\28  9     8     6     5    16\10  1     2     4     3
0\0  21\0  23\34  8     9     7     6     4     6\0   7\0
0\16  7     9    33\17  8     9    16\7   1     2     4
0\29  5     8     9     7     7\11  3     5     1     2
0\22  9     6     7     8\4   3     1     8\4   3     1
0\0  17\0   8\16  3     1     4     6     2     3\0  10\0
0\17  8     1     6     2     0\14  4     1     2     7
0\29  9     7     8     5     0\11  2     5     1     3
```


参考文献

1) Sudoku: http://en.wikipedia.org/wiki/Sudoku
2) 数独: http://www.nikoli.co.jp/puzzles/1/index.htm
3) カックロ: http://www.nikoli.co.jp/puzzles/2/index.htm
4) ニコリ: http://www.nikoli.co.jp/

（平成17年9月22日受付）