課題 8

次の路発見問題を解くプログラムを作りなさい。

問題　「農夫と狼と羊とキャベツ」(内容は省略)

解答及び解説

この問題における状態は、農夫、狼、羊、キャベツがそれぞれどちらの岸にあるかで表すことができる。岸を表す型 Bank と状態 State を次のように定義する。

```
data Bank = North | South deriving (Eq, Show)

type State = (Bank, Bank, Bank, Bank)
newtype Path = Path [State] deriving Show
```

状態を表す型 `State` は、4 つ組でそれぞれが、農夫、狼、羊、キャベツの位置 (北岸か南岸か) を示す。路 (型 `Path`) は、状態のリストで表わす。すると、最初は、皆な北岸にあり、最後には、皆な南岸にある。従って、とくべき問題は、次の solve となる。

```
solve = search (North, North, North, North) (South, South, South, South)
```

次に、状態遷移をどのように表すか。農夫に可能な行動は、単独で川を渡るか、狼、羊、キャベツのいずれか一つと一緒に川を渡ることである。但し、一緒に渡るには同じ岸になくてはならない。このことから、可能な動作は以下の 8 通りである。

1. 農夫が北岸から、南岸へ渡る。
2. 農夫が南岸から、北岸へ渡る。
3. 農夫と狼が北岸から、南岸へ渡る。
4. 農夫と狼が南岸から、北岸へ渡る。
5. 農夫と羊が北岸から、南岸へ渡る。
6. 農夫と羊が南岸から、北岸へ渡る。
7. 農夫とキャベツが北岸から、南岸へ渡る。
8. 農夫とキャベツが南岸から、北岸へ渡る。

これらを、状態を使って表現することを考える。例えば「農夫と羊が南岸から、北岸へ渡る」であれば、

```
(South, w, South, c) -> (North, w, North, c)
```

と表せる。ここで、w と c は、それぞれ狼とキャベツがどちらの岸にいるかを表すが、農夫が羊を連れて渡るとき、狼やキャベツの位置は変わらないので、状態遷移の前と後で同じでなくてはならない。

またこの例のように必ず、北岸 (North) からは南岸 (South) へ、そして南岸からは北岸へ渡るので、対岸を計算する関数 opposite を用意しておく。

```
opposite :: Bank -> Bank
opposite North = South
opposite South = North
```

そして、これらの農夫の行動により遷移した状態を、到達可能な状態のリストに加える関数を作成する。例えば、羊と一緒に渡る行動であれば、

```
sheep (from, w, s, c) ss | from == s = (to, w, to, c):ss
                         | otherwise = ss
    where to = opposite from
```

と表現できる。from は、North でも South でも良く、to は、その反対 opposite となる。さらに、「羊と一緒に渡る」には遷移の前の羊の位置 s が、農夫の位置 from と一緒でなくてはならない。そこで、from と s が等しい場合は、羊と一緒に渡ることが可能なので、到達可能なリスト ss に遷移後の状態を加える。(そうで無いときは、到達可能なリストはそのままにする。)

このとき、注意すべき点が二つある。

1) 無限の繰り返しを防ぐため、一度通った状態をリストに加えないこと

2) 狼が羊を、羊がキャベツを食べてしまうような状態も加えないこと

前者は、これまでの経路を引き数に加えて、そこに新しい状態が含まれていないかをチェックすると良い。後者は、述語 legal を使って、「食べられてしまう」状態をリストに加えないようにする。これらは、農夫の動作に共通のことなので、次のように、リストに状態を加えるための関数を用意し、1) 2) の場合には、状態を追加しないようにしておく

```
addState :: State -> [State] -> Path -> [State]
addState cs ss path | passed cs path = ss
                    | legal cs       = cs:ss
                    | otherwise      = ss


passed :: State -> Path -> Bool
passed cs (Path []) = False
passed cs (Path (s:ss)) | cs == s   = True
                        | otherwise = passed cs (Path ss)
```

```
legal:: State -> Bool
legal s | wolfEatSheep    s = False
        | sheepEatCabbage s = False
        | otherwise         = True
```

このとき、農夫に可能な行動のなかで、狼が羊を食べてしまう状態やキャベツを羊が食べてしまう状態になるようなものを、不可能な行動として、はじめから除外するのでは、この「問題」をプログラムで解くのではなく、プログラムを考えている人が (問題の一部を) 解いていることになるので、良くない。

さて準備が出来たので、reachable を定義してみよう。

```
reachable :: Path -> [State]
reachable path = farmer path $ wolf path $ sheep path $ cabbage path []

wolfEatSheep (f, w, s, c) = (w == s && f /= w)
sheepEatCabbage (f, w, s, c) = (s == c && f /= s)

farmer :: Path -> [State] -> [State]
farmer path@(Path ((f, w, s, c):ps)) ss = addState (op,w,s,c) ss path
  where op = opposite f

wolf :: Path -> [State] -> [State]
wolf path@(Path ((f, w, s, c):ps)) ss
    | f == w    = let op = opposite f in addState (op,op,s,c) ss path
    | otherwise = ss

sheep :: Path -> [State] -> [State]
sheep path@(Path ((f, w, s, c):ps)) ss
    | f == s    = let op = opposite f in addState (op,w,op,c) ss path
    | otherwise = ss

cabbage :: Path -> [State] -> [State]
cabbage path@(Path ((f, w, s, c):ps)) ss
    | f == c    = let op = opposite f in addState (op,w,s,op) ss path
    | otherwise = ss
```

これで、「農夫の問題」をプログラムで表すことが出来たので、あとは授業で用いた路発見問題を解くための手法を使えば良い。

まず、深さ優先による探索をするならば、次のようになる。

```
search :: State -> State -> Path
search current_state goal_state = depth current_state goal_state (Path [])

depth :: State -> State -> Path -> Path
depth cs gs (Path ss)
   | cs == gs  = Path (gs:ss)
   | otherwise = breadth (reachable (Path (cs:ss))) gs (Path (cs:ss))

breadth :: [State] -> State -> Path -> Path
breadth [] gs (Path ss)  = Path []
breadth (cs:css) gs path = if ss == [] then breadth css gs path else (Path ss)
    where (Path ss) = depth cs gs path
```

次に、幅優先による探索を行うには、次のようにする。

```
search :: State -> State -> Path
search current_state goal_state = breadth [Path [current_state]] goal_state

breadth :: [Path] -> State -> Path
breadth paths gs = if ss == [] then breadth (depth paths) gs else (Path ss)
    where (Path ss) = reached paths gs

reached :: [Path] -> State -> Path
reached [] gs = Path []
reached (p:ps) gs | reached' p gs = p
                  | otherwise     = reached ps gs
    where reached' :: Path -> State -> Bool
          reached' (Path (cs:css)) gs = cs == gs

depth :: [Path] -> [Path]
depth [] = []
depth (path:ps) = step (reachable path) path (depth ps)

step :: [State] -> Path -> [Path] -> [Path]
step []     path      paths = paths
step (s:ss) (Path ps) paths = (Path (s:ps)):(step ss (Path ps) paths)
```

この問題にも、解は複数ある。そこで、それらの解をすべて求めるようにするには、幅優先による探索プログラムの三つの関数 search, breadth, reached を、次のように変更すると良い。

```
search :: State -> State -> [Path]
```

```
search cs gs = breadth [Path [cs]] gs []

breadth :: [Path] -> State -> [Path] -> [Path]
breadth [] gs ans = ans
breadth paths gs ans = breadth (depth paths) gs (reached paths gs ans)

reached :: [Path] -> State -> [Path] -> [Path]
reached [] gs ans = ans
reached (p:ps) gs ans | reached' p gs = reached ps gs (p:ans)
                      | otherwise     = reached ps gs ans
    where reached' :: Path -> State -> Bool
          reached' (Path (cs:css)) gs = cs == gs
```

但し、この方法では一度目的地にたどり着いたパスについても、その先を探そうとする。

つまり、関数 breadth の 2 行目で関数 reached を使って、パスのリスト (paths) に含まれている目的地にたどり着いたパス (解の一つ) を、解のリスト (ans) に追加している。しかし、その解をパスのリストから取り除いていないため、そのまま関数 depth の引き数には、解のパスも含まれてしまっている。

実際には、その解のパスの先はすぐに行き止まりになるため、探索の対象からはその時点で排除されるので良いが、やはり一度目的地に着いたパスの先を探す必要はない。

関数 reached を次のように変更すれば、必要のない探索を避けることができる。

```
reached :: [Path] -> State -> ([Path], [Path]) -> ([Path], [Path])
reached [] gs ans = ans
reached (p:ps) gs ans = if reached' p gs then (n, p:a) else (p:n, a)
    where (n, a) = reached ps gs ans
          reached' (Path (cs:css)) gs = cs == gs
```

それに伴って、関数 breadth も若干の修正が必要である。

```
breadth :: [Path] -> State -> [Path] -> [Path]
breadth [] gs ans = ans
breadth paths gs ans = breadth (depth new_paths) gs new_ans
    where (new_paths, new_ans) = reached paths gs ans
```

さて、深さ優先による探索でも、すべての解を求めるようにすることはできる。それについては、自分で考えてみよう。