Username: jlk21

Compilation: 1 / 1

```
cleanUp: 1 / 1

split2: 4 / 4

split3: 4 / 4

uninsert: 4 / 4

matches: 8 / 8

evaluate: 8 / 8

parseSynonym: 3 / 3

parseAnagram: 4 / 4

parseReversal: 3 / 3

parseInsertion: 3 / 3

parseCharade: 3 / 3

extraParseCases: 2 / 2

 > parseClueLengths 13 = 286
   test case expected: 412

 > parseClueLengths 16 = 45
   test case expected: 72

parseClueLengths: 5 / 7

solveAll: 1 / 1
```

nice!

part 4 parseHidden word.

```haskell
module Solver where

import Data.List
import Data.Char

import Types
import WordData
import Clues
import Examples

-------------------------------------------------------
-- Part I

punctuation :: String
punctuation
  = "';.,-!?"

cleanUp :: String -> String
cleanUp s
  = [toLower c | c <- s, c `notElem` punctuation]

split2 :: [a] -> [([a], [a])]
split2 xs
  = map (`splitAt` xs) [1..length xs - 1]

split3 :: [a] -> [([a], [a], [a])]
split3 xs
  = nonempty ++ empty
  where
    nonempty = [(xs1', xs1'', xs2) | (xs1, xs2) <- split2 xs,
                                     (xs1', xs1'') <- split2 xs1]
    empty    = map (\(xs1, xs2) -> (xs1, [], xs2)) (split2 xs)

uninsert :: [a] -> [([a], [a])]
uninsert xs
  = [(xs2, xs1 ++ xs3) | (xs1, xs2, xs3) <- split3 xs, (not . null) xs2]

-- Uncomment these functions when you have defined the above.
split2M :: [a] -> [([a], [a])]
split2M xs
  = sxs ++ [(y, x) | (x, y) <- sxs]
  where
    sxs = split2 xs

split3M :: [a] -> [([a], [a], [a])]
split3M xs
  = sxs ++ [(z, y, x) | (x, y, z) <- sxs]
  where
    sxs = split3 xs

-------------------------------------------------------
-- Part II

matches :: String -> ParseTree -> Bool
matches s (Synonym s')
  = s `elem` (synonyms s')
matches s (Anagram _ s')
  = sort s == sort s'
matches s (Reversal _ t)
  = matches (reverse s) t
matches s (Insertion _ t1 t2)
  = any (\(s1, s2) -> matches s1 t1 && matches s2 t2) (uninsert s)
matches s (Charade _ t1 t2)
  = any (\(s1, s2) -> matches s1 t1 && matches s2 t2) (split2 s)
matches s (HiddenWord _ s')
  = s == s'

evaluate :: Parse -> Int -> [String]
evaluate (def, _, t) n
  = filter (\s -> length s == n && matches s t) (synonyms (unwords def))

-------------------------------------------------------
-- Part III

-- Given...
parseWordplay :: [String] -> [ParseTree]
parseWordplay ws
  = concat [parseSynonym ws,
            parseAnagram ws,
            parseReversal ws,
            parseInsertion ws,
            parseCharade ws
            -- parseHiddenWord ws
            ]

parseSynonym :: [String] -> [ParseTree]
parseSynonym ws
  | null (synonyms s) = []
  | otherwise         = [Synonym s]
  where
    s = unwords ws

parseAnagram :: [String] -> [ParseTree]
parseAnagram ws
  = [Anagram f1 (concat f2) | (f1, f2) <- split2M ws,
                              unwords f1 `elem` anagramIndicators]

parseReversal :: [String] -> [ParseTree]
parseReversal ws
  = [Reversal f1 t | (f1, f2) <- split2M ws,
                     unwords f1 `elem` reversalIndicators,
                     t <- parseWordplay f2]

parseBinary :: [String] -> (Indicator -> ParseTree -> ParseTree -> ParseTree)
            -> [String] -> [String] -> [ParseTree]
parseBinary ws constructor preIndicators postIndicators
  = pre ++ post
  where
    pre  = [constructor ws' t t' | (arg, ws', arg') <- split3 ws,
                                   unwords ws' `elem` preIndicators,
                                   t <- parseWordplay arg,
                                   t' <- parseWordplay arg']
    post = [constructor ws' t' t | (arg, ws', arg') <- split3 ws,
                                   unwords ws' `elem` postIndicators,
                                   t <- parseWordplay arg,
                                   t' <- parseWordplay arg']

parseInsertion :: [String] -> [ParseTree]
parseInsertion ws
  = parseBinary ws Insertion insertionIndicators envelopeIndicators

parseCharade :: [String] -> [ParseTree]
parseCharade ws
  = parseBinary ws Charade beforeIndicators afterIndicators

parseHiddenWord :: [String] -> [ParseTree]
parseHiddenWord ws
  = [HiddenWord f1 hw | (f1, f2) <- split2 ws,
                        unwords f1 `elem` hiddenWordIndicators,
                        hw <- (substrings . removeFirstLast . concat) f2,
                        (not . null . synonyms) hw]
  where
    removeFirstLast (c : cs)
      = init cs
    prefixes s
      = take (length s) (iterate init s)
    substrings ""
      = [""]
    substrings (c : cs)
      = [c] : map (c :) (prefixes cs) ++ substrings cs

-- Given...
parseClue :: Clue -> [Parse]
parseClue clue@(s, n)
  = parseClueText (words (cleanUp s))

parseClueText :: [String] -> [Parse]
parseClueText ws
  = [(def, link, t) | (def, link, wp) <- split3M ws,
                      unwords link `elem` linkWords,
                      (not . null . synonyms . unwords) def,
                      t <- parseWordplay wp]

solve :: Clue -> [Solution]
solve clue@(s, n)
  = [(clue, p, s) | p <- parseClue clue, s <- evaluate p n]

-------------------------------------------------------
-- Some additional test functions

-- Returns the solution(s) to the first k clues.
-- The nub removes duplicate solutions arising from the
```

```
163: -- charade parsing rule.
164: solveAll :: Int -> [[String]]
165: solveAll k
166:   = map (nub . map getSol . solve . (clues !!)) [0..k-1]
167:
168: getSol :: Solution -> String
169: getSol (_, _, sol) = sol
170:
171: showAll
172:   = mapM_ (showSolutions . solve . (clues !!)) [0..23]
173:
```