

**Section A**

**Tests (1/1)**

**jlk21**

Username: jlk21

Compilation: 1 / 1

**Section A****Output (1/1)**

jlk21

```
1: cleanUp: 1 / 1
2:
3: split2: 4 / 4
4:
5: split3: 4 / 4
6:
7: uninsert: 4 / 4
8:
9: matches: 8 / 8
10:
11: evaluate: 8 / 8
12:
13: parseSynonym: 3 / 3
14:
15: parseAnagram: 4 / 4
16:
17: parseReversal: 3 / 3
18:
19: parseInsertion: 3 / 3
20:
21: parseCharade: 3 / 3
22:
23: extraParseCases: 2 / 2
24:
25: > parseClueLengths 13 = 286
26:   test case expected: 412
27:
28: > parseClueLengths 16 = 45
29:   test case expected: 72
30:
31: parseClueLengths: 5 / 7
32:
33: solveAll: 1 / 1
34:
```

]  
nice!  
] part 4 parse[redder word.

## Section A

## Solver.hs (1/3)

```

1: module Solver where
2:
3: import Data.List
4: import Data.Char
5:
6: import Types
7: import WordData
8: import Clues
9: import Examples
10:
11: -----
12: -- Part I
13:
14: punctuation :: String
15: punctuation
16: = "';.,-!?"
17:
18: cleanUp :: String -> String
19: cleanUp s
20: = [toLowerCase c | c <- s, `notElem` punctuation]
21:
22: split2 :: [a] -> [[a], [a]]
23: split2 xs
24: = map ('splitAt' xs) [1..length xs - 1]
25:
26: split3 :: [a] -> [[[a], [a], [a]]]
27: split3 xs
28: = nonempty ++ empty
29: where
30: nonempty = [(xs1', xs1'', xs2) | (xs1, xs2) <- split2 xs,
31: (xs1', xs1'') <- split2 xs1]
32: empty = map (\(xs1, xs2) -> (xs1, [], xs2)) (split2 xs)
33:
34: uninserit :: [a] -> [[[a], [a]]]
35: uninserit xs
36: = [(xs2, xs1 ++ xs3) | (xs1, xs2, xs3) <- split3 xs, (not . null) xs2]
37:
38: -- Uncomment these functions when you have defined the above.
39: split2M :: [a] -> [[[a], [a]]]
40: split2M xs
41: = sxs ++ [(y, x) | (x, y) <- sxs]
42: where
43: sxs = split2 xs
44:
45: split3M :: [a] -> [[[a], [a], [a]]]
46: split3M xs
47: = sxs ++ [(z, y, x) | (x, y, z) <- sxs]
48: where
49: sxs = split3 xs
50:
51: -----
52: -- Part II
53:
54: matches :: String -> ParseTree -> Bool
55: matches s (Synonym s')
56: = s `elem` (synonyms s')
57: matches s (Anagram _ s')
58: = sort s == sort s'
59: matches s (Reversal _ t)
60: = matches (reverse s) t
61: matches s (Insertion _ t1 t2)
62: = any (\(s1, s2) -> matches s1 t1 && matches s2 t2) (uninsert s)
63: matches s (Charade _ t1 t2)
64: = any (\(s1, s2) -> matches s1 t1 && matches s2 t2) (split2 s)
65: matches s (HiddenWord _ s')
66: = s == s'
67:
68: evaluate :: Parse -> Int -> [String]
69: evaluate (def, _ t) n
70: = filter (\s -> length s == n && matches s t) (synonyms (unwords def))
71:
72: -----
73: -- Part III
74:
75: -- Given...
76: parseWordplay :: [String] -> [ParseTree]
77: parseWordplay ws
78: = concat [parseSynonym ws,
79: parseAnagram ws,
80: parseReversal ws,
81: parseInsertion ws,

```

well done! 29.5 → 30

style: -0.5

## jlk21

## Section A

```

82:     parseCharade ws
83:     -- parseHiddenWord ws
84:     ]
85:
86: parseSynonym :: [String] -> [ParseTree]
87: parseSynonym ws
88: = null (synonyms s) = []
89: | otherwise = [Synonym s]
90: where
91: s = unwords ws
92:
93: parseAnagram :: [String] -> [ParseTree]
94: parseAnagram ws
95: = [Anagram f1 (concat f2) | (f1, f2) <- split2M ws,
96: unwords f1 `elem` anagramIndicators]
97:
98: parseReversal :: [String] -> [ParseTree]
99: parseReversal ws
100: = [Reversal f1 t | (f1, t) <- split2M ws,
101: unwords f1 `elem` reversalIndicators,
102: t <- parseWordplay f2]
103:
104: parseBinary :: [String] -> (Indicator -> ParseTree -> ParseTree -> ParseTree)
105: --> [String] -> [String] -> [ParseTree]
106: parseBinary ws constructor preIndicators postIndicators
107: = pre ++ post
108: where
109: pre = [constructor ws' t t' | (arg, ws', arg') <- split3 ws,
110: unwords ws' `elem` preIndicators,
111: t <- parseWordplay arg,
112: t' <- parseWordplay arg']
113: post = [constructor ws' t' t | (arg, ws', arg') <- split3 ws,
114: unwords ws' `elem` postIndicators,
115: t <- parseWordplay arg,
116: t' <- parseWordplay arg']
117:
118: parseInsertion :: [String] -> [ParseTree]
119: parseInsertion ws
120: = parseBinary ws Insertion insertionIndicators envelopeIndicators
121:
122: parseCharade :: [String] -> [ParseTree]
123: parseCharade ws
124: = parseBinary ws Charade beforeIndicators afterIndicators
125:
126: parseHiddenWord :: [String] -> [ParseTree]
127: parseHiddenWord ws
128: = [HiddenWord f1 hw | (f1, hw) <- split2 ws,
129: unwords f1 `elem` hiddenWordIndicators,
130: hw <- (substrings . removeFirstLast . concat) f2,
131: (not . null . synonyms) hw]
132: where
133: removeFirstLast (c : cs)
134: = init cs
135: prefixes s
136: = take (length s) (iterate init s)
137: substrings ""
138: = []
139: substrings (c : cs)
140: = [c] : map (c :) (prefixes cs) ++ substrings cs
141:
142: -- Given...
143: parseClue :: Clue -> [Parse]
144: parseClue clue@(s, n)
145: = parseClueText (words (cleanUp s))
146:
147: parseClueText :: [String] -> [Parse]
148: parseClueText ws
149: = [(def, link, t) | (def, link, wp) <- split3M ws,
150: unwords link `elem` linkWords,
151: (not . null . synonyms . unwords) def,
152: t <- parseWordplay wp]
153:
154: solve :: Clue -> [Solution]
155: solve clue@(s, n)
156: = [(clue, p, s) | p <- parseClue clue, s <- evaluate p n]
157:
158: -----
159: -- Some additional test functions
160:
161: -- Returns the solution(s) to the first k clues.
162: -- The nub removes duplicate solutions arising from the

```

very nice!

this part of the logic  
should go under  
'matches' (with some  
changes)

all you need to do  
here is join the f2  
strings and that  
would be hw.

same name,  
not a good practice in general

## jlk21

```
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:
```