```
Haskell test run started Tue Sep  5 11:39:11 AEST 2017
Proj1 testing
                          Test   1 ... PASSED 5.0
                          Test   2 ... PASSED 4.0
                          Test   3 ... PASSED 4.0
                          Test   4 ... PASSED 4.0
                          Test   5 ... PASSED 3.0
                          Test   6 ... PASSED 4.0
                          Test   7 ... PASSED 3.0
                          Test   8 ... PASSED 4.0
                          Test   9 ... PASSED 6.0
                          Test  10 ... PASSED 5.0
                          Test  11 ... PASSED 4.0
                          Test  12 ... PASSED 5.0
                          Test  13 ... PASSED 5.0
                          Test  14 ... PASSED 5.0
                          Test  15 ... PASSED 4.0
                          Test  16 ... PASSED 4.0
                          Test  17 ... PASSED 4.0
                          Test  18 ... PASSED 4.0
                          Test  19 ... PASSED 3.0
                          Test  20 ... PASSED 5.0
                          Test  21 ... PASSED 3.0
                          Test  22 ... PASSED 3.0
                          Test  23 ... PASSED 4.0
                          Test  24 ... PASSED 5.0
                          Test  25 ... PASSED 4.0
                          Test  26 ... PASSED 4.0
                          Test  27 ... PASSED 5.0
                          Test  28 ... PASSED 3.0
                          Test  29 ... PASSED 4.0
                          Test  30 ... PASSED 5.0
                          Test  31 ... PASSED 5.0
                          Test  32 ... PASSED 4.0
                          Test  33 ... PASSED 3.0
                          Test  34 ... PASSED 4.0
                          Test  35 ... PASSED 4.0
                          Test  36 ... PASSED 2.0
                          Test  37 ... PASSED 6.0
                          Test  38 ... PASSED 5.0
                          Test  39 ... PASSED 3.0
                          Test  40 ... PASSED 3.0
                          Test  41 ... PASSED 3.0
                          Test  42 ... PASSED 4.0
                          Test  43 ... PASSED 4.0
                          Test  44 ... PASSED 5.0
                          Test  45 ... PASSED 5.0
                          Test  46 ... PASSED 3.0
                          Test  47 ... PASSED 3.0
                          Test  48 ... PASSED 4.0
                          Test  49 ... PASSED 3.0
                          Test  50 ... PASSED 4.0
                          Test  51 ... PASSED 3.0
                          Test  52 ... PASSED 6.0
                          Test  53 ... PASSED 6.0
                          Test  54 ... PASSED 5.0
                          Test  55 ... PASSED 6.0
                          Test  56 ... PASSED 5.0
                          Test  57 ... PASSED 4.0
```

```
Test  58 ... PASSED 4.0
Test  59 ... PASSED 4.0
Test  60 ... PASSED 5.0
Test  61 ... PASSED 3.0
Test  62 ... PASSED 4.0
Test  63 ... PASSED 5.0
Test  64 ... PASSED 5.0
Test  65 ... PASSED 5.0
Test  66 ... PASSED 4.0
Test  67 ... PASSED 4.0
Test  68 ... PASSED 5.0
Test  69 ... PASSED 4.0
Test  70 ... PASSED 5.0
Test  71 ... PASSED 5.0
Test  72 ... PASSED 4.0
Test  73 ... PASSED 5.0
Test  74 ... PASSED 3.0
Test  75 ... PASSED 3.0
Test  76 ... PASSED 3.0
Test  77 ... PASSED 6.0
Test  78 ... PASSED 4.0
Test  79 ... PASSED 4.0
Test  80 ... PASSED 5.0
Test  81 ... PASSED 4.0
Test  82 ... PASSED 4.0
Test  83 ... PASSED 5.0
Test  84 ... PASSED 4.0
Test  85 ... PASSED 5.0
Test  86 ... PASSED 5.0
Test  87 ... PASSED 5.0
Test  88 ... PASSED 5.0
Test  89 ... PASSED 4.0
Test  90 ... PASSED 4.0
Test  91 ... PASSED 5.0
Test  92 ... PASSED 5.0
Test  93 ... PASSED 6.0
Test  94 ... PASSED 5.0
Test  95 ... PASSED 5.0
Test  96 ... PASSED 5.0
Test  97 ... PASSED 5.0
Test  98 ... PASSED 5.0
Test  99 ... PASSED 5.0
Test 100 ... PASSED 4.0
Test 101 ... PASSED 4.0
Test 102 ... PASSED 4.0
Test 103 ... PASSED 4.0
Test 104 ... PASSED 5.0
Test 105 ... PASSED 4.0
Test 106 ... PASSED 5.0
Test 107 ... PASSED 2.0
Test 108 ... PASSED 3.0
Test 109 ... PASSED 4.0
Test 110 ... PASSED 4.0
Test 111 ... PASSED 3.0
Test 112 ... PASSED 4.0
Test 113 ... PASSED 4.0
Test 114 ... PASSED 4.0
Test 115 ... PASSED 4.0
Test 116 ... PASSED 5.0
```

```
                          Test 117 ... PASSED 4.0
                          Test 118 ... PASSED 5.0
                          Test 119 ... PASSED 5.0
                          Test 120 ... PASSED 3.0
Total tests: 120.0
Tests successfully guessed: 120.0
Total guesses for successful tests: 511.0
Average guesses: 4.258333333333334
Points available: 70.0 * 120.0 / 120.0 = 70.0
Points: 70.0 / 70.0
Haskell test run ended Tue Sep  5 11:39:33 AEST 2017
Total CPU time used = 5682 milliseconds
```

```
--  File     : Proj1.hs
--  Author   : David Stern
--  Origin   : Fri Aug 25 07:30:02 2017
--  Purpose  : Project 1 Submission, COMP30020

-- This file implements functions that define an agent that plays 'ChordProbe'.
--
-- ChordProbe is a game where a 'composer' selects a 'Chord'. A Chord is a list
-- of three distinct 'pitches'. A pitch is a String with a note ('A'..'G'),
-- and an Octave ('1'..'3'). The 'performer', the role of this agent, is to
-- guess the Chord, via repeated guesses, informed by feedback in the form of
-- a 'Score'. The 'score' function details this feedback.
--
-- This agent solves this problem via generating every possible chord & pruning
-- this list via selecting guesses that prune this game state as much as
-- possible. This is explained above functions 'bestGuess' and 'getAvgRemCands'

module Proj1 (initialGuess, nextGuess, GameState) where

import Data.List

-- | Chord represents a list of 'pitches', GameState is simply a list of
--   remaining possible chords, and Score is the scoring format defined by the
--   project specification.
type Chord     = [String]
type GameState = [Chord]
type Score     = (Int, Int, Int)


-- | Generates an initial guess, and a new game state enumerating every
--   possible guess, minus the guessed chord.
--   The initial guess is an empirically tested first guess, designed to
--   eliminate the most possible guesses (on average).
initialGuess :: (Chord, GameState)
initialGuess = (guess, state)
    where
        notes = [[note, octave] | note <- ['A'..'G'], octave <- ['1'..'3']]
        allChords = [chord | chord <- subsequences notes, length chord == 3]
        guess = ["A2", "B1", "C1"]
        state = allChords \\ [guess]


-- | Prunes the game state such that only chords that would have resulted in
--   the same score for our previous guess are included.
--   Based on this updated state, the next guess is generated.
nextGuess :: (Chord, GameState) -> Score -> (Chord, GameState)
nextGuess (lastGuess, state) score = (nextGuess', state')
    where
        state' = delete lastGuess [ cord
                                  | cord <- state
                                  , getScore cord lastGuess == score]
        nextGuess' = bestGuess state'


-- | Generates the 'best' possible guess, given a game's state (the possible
--   remaining chords). 'Best' is defined as the guess that when chosen will
--   (on average) result in the lowest number of remaining candidates, if it's
--   incorrect. To clarify, it reduces the no. of remaining possible chords by
--   the greatest amount, and therefore provides the highest amount of
```

```
--   information.
bestGuess :: GameState -> Chord
bestGuess state = fst (bestguess !! 0)
    where
        targRemCands = [(targ, remcands)
                       | targ <- state
                       , let state' = state \\ [targ]
                       , let remcands = getAvgRemCands targ state']
        bestguess = sortBy sndEl targRemCands


-- | Generates the average number of candidates that will remain after the
--   guess is discovered to be incorrect. Thus, this provides a reflection of
--   how effective the guess is at reducing the number of possible states.
--
--   Process:
--   Given a possible target chord, and the existing state at the time the
--   guess is to be made, it first generates a list of possible scores for the
--   given possible target, then sorts and groups these scores, and counts
--   the total number of tested targets.
--   This information is used to calculate an average number of remaining
--   candidates after such a guess, the lower the number the better the guess
--   prunes the game state, on average.
--
--   Average = sum of all (chanceOfOutcome * Outcome's remaining candidates)
--   Chance of each outcome is merely the number times this outcome can occur,
--   out of all of the possible outcomes.
getAvgRemCands :: Chord -> GameState -> Double
getAvgRemCands possTarget state
    = sum [(numOutcomes / totalOutcomes) * numOutcomes
          | grp <- grouped
          , let numOutcomes = (fromIntegral . length) grp]
    where
        scores  = [ans | guess <- state, let ans = getScore possTarget guess]
        grouped = (group . sort) scores
        totalOutcomes = (fromIntegral . length) scores


--  | An Ordering that is used to order tuples of remaining guesses and their
--    average number of remaining candidates, by the average number of
--    remaining candidates. Polymorphic type, so works with other tuples where
--    the second element is orderable.
--    The less-than tuple's second element is <= the other tuple's 2nd element.
sndEl :: (Ord b) => (a, b) -> (a, b) -> Ordering
sndEl x y
    | (snd x) <= (snd y)  = LT
    | otherwise           = GT


-- | Calculate the accuracy, the 'score' of a given guess chord for a given
--   target chord. The first argument is the target chord, the second is the
--   guess cord.
--   Understanding the Score:
--   Correct Pitches: How many pitches in the guess are included in the target
--   Correct Notes: How many pitches have the right note but the wrong octave
--   Correct Octaves: How many pitches have the right octave but the wrong note
getScore :: Chord -> Chord -> Score
getScore target guess = (correctPitches, correctNotes, correctOctaves)
    where
```

```
        guess'          = (sort . nub) guess
        target'         = sort target
        correctPitches  = length $ intersect target' guess'
        correctNotes    = overlapN 0 target' guess' - correctPitches
        correctOctaves  = overlapN 1 target' guess' - correctPitches


-- | Takes an integer and two lists of lists. Returns the no. of elements in
--   s1 that share sub-elements at index n, with the elements in s2.
overlapN :: (Ord a, Eq a) => Int -> [[a]] -> [[a]] -> Int
overlapN n s1 s2 = numSame (sort [s !! n | s <- s1]) (sort [s !! n | s <- s2])


-- | Returns the number of elements in the first list that are in the second.
numSame :: (Eq a) => [a] -> [a] -> Int
numSame (x:xs) ys
    | ys' /= ys = 1 + numSame xs ys'
    | otherwise = numSame xs ys
    where
        ys' = delete x ys
numSame _ _ = 0
```