AutoComp Documentation

Imports:

```
module AutoComp where
import Haskore hiding (chord, Key)
import Data.List
import Data.Maybe
```

Data Structures:

A key (consisting of a pitch class like C, D or E and a Major/Minor mode) denotes the key the song is played in

```
type Key = (PitchClass, Mode)
```

Although many more 'chord' triads exist we've restricted ourselves to a Major and Minor chord triad. These structures are used to find the right notes for a particular chord voicing.

```
data Triad = TriMaj | TriMin deriving (Eq)
```

In a chord the PitchClass denotes the root note and the Triad will change the accompanying notes or chord voicing.

```
type Chord = (PitchClass, Triad)
```

Our bass style makes us swing in different ways.

```
data BassStyle = Basic | Calypso | Boogie deriving (Eq)
```

A chord progression is used to describe a song where each Chord is played for Dur duration

```
type ChordProgression = [(Chord,Dur)]
```

A ChordPitch is a more specific representation of a chord when compared to a general Chord data type. It explicitly states the pitches of all three notes (we only have triad chords) and is used as an intermediary step between a general Chord and a final Music data structure.

```
type ChordPitch = (Pitch, Pitch, Pitch)
```

Helper Functions:

This function was skilfully copied from the supplied code and thus requires no further explanation. The only adjustment that has been made is a reduced volume to enhance the final result.

```
fd d n = n d v
vol  n = n   v
v      = [Volume 60]
```

Again another beautiful example of 'better stolen well than thought of poorly'. This function is maily used to explicitly state repetition instead of implicitly repeating chords.

```
-- repeat something n times
times 1     m = m
times n m = m :+: (times (n - 1) m)
```

Although small shift is used throughout both the bass line and chord voicing generation. It's role is to iteratively append the head to the tail (removing the head) 0..inf times. We then simply pick the n-th iteration and due to Haskells lazy evaluation this should be the only iteration actually calculated. Thank you Haskell!

```
shift :: Eq a => Int -> [a] -> [a]
shift n l= (iterate f l)!!n
      where f [] = []
            f (x:xs) = xs ++ [x]
```

Bass Line:

The scalePattern returns all the notes for a particular key. For instance a C Major key is made up of CDEFGAB where each note is obtained by transposing the root note. Since this function is only used for generating the baseline we hardcoded a root note in the 3rd octave.

```
scalePattern :: Key ->[PitchClass]
scalePattern (pc, Major) = fst.unzip.zipWith trans [0, 2, 4, 5, 7, 9, 11] $ repeat (pc,
scalePattern (pc, Minor) = fst.unzip.zipWith trans [0, 2, 3, 5, 7, 8, 10] $ repeat (pc,
```

The chordMode function is used to generate a list of fitting notes based on the key of the song and a particular chord. This daunting looking function achieves it's greatness in a very simple manner. In the first step we shift the scale pattern

(of the song key) to the root note of the chord. For instance the scale C3-D3-E3-F3-G3-A3-B3 when applied to a GMaj chord becomes G3-A3-B3-C3-D3-E3-F3. Notice that that the pitches aren't increasing correctly since we simply shift and aren't transposing the notes. Therefore the octaveSplit and cleanOctave functions are used to first identify where this indescrepency happens and then to fix it. So G3-A3-B3-C3-D3-E3-F3 finally becomes G3-A3-B3-C4-D4-E4-F4.

```
chordMode :: Key -> Chord -> [Dur->[NoteAttribute]-> Music]
chordMode key (pc,_) = map Note $ cleanOctave $ zip transformedPitches (repeat 3)
                       where f (Just a) = shift a (scalePattern key)
                             f _  = take 12 $ repeat pc
                             transformedPitches = f (elemIndex pc (scalePattern key))
                             octaveSplit scale = partition (octaveSplitTest scale) scale
                             octaveSplitTest scale p = absPitch p >= (head $ map absPitch
                             cleanOctave scale = (fst.octaveSplit) scale ++ map (trans 1
```

The autoBass function is where the low-register magic happens. First the key of the Chord is transformed to fit the key of the song generating a 'list of fitting notes to pick from'. By performing pattern matching on the BassStyle for each Chord in the ChordProgression a unique picking pattern is generated. The picking pattern consists of the semitone differences between the root note. This picking pattern further depends on the duration of the chord where the only cases handled are where the chord is a 'wn' (Whole Note) or a 'hn' (Half Note).

```
autoBass :: BassStyle-> Key -> ChordProgression -> Music
autoBass style key [] = Rest 0
autoBass Basic key (chord:[])
  | (snd chord == hn) = line $ zipWith fd [hn] [t!!0]
  | (snd chord == wn) = line $ zipWith fd [hn,hn] [t!!0,t!!4]
  | otherwise = Rest (snd chord)
  where t =  chordMode key (fst $ chord)

autoBass Calypso key (chord:[])
  | (snd chord == hn) = bar
  | (snd chord == wn) = times 2 bar
  | otherwise = Rest (snd chord)
  where bar = Rest qn :+: (line $ zipWith fd [en,en] [t!!0,t!!2])
        t   = chordMode key (fst $ chord)

autoBass Boogie key (chord:[])
  | (snd chord == hn) = bar
  | (snd chord == wn) = times 2 bar
  | otherwise = Rest (snd chord)
  where bar = line $ zipWith fd [en,en,en,en] [t!!0,t!!4,t!!5,t!!4]
        t =  chordMode key (fst $ chord)
autoBass style key (c:cs) = autoBass style key [c] :+: autoBass style key cs
```

Chord Voicing:

The toChord function transforms a ChordPitch into music by applying a specific duration and transforming each individual Pitch into a corresponding Note. Since the ChordPitch is a much more appropriate datatype than Music to use during calculations this function bridges the abstraction layer it imposes.

```
toChord :: Dur -> ChordPitch -> Music
toChord dur (a,b,c) = foldl (\acc x -> acc :=: (fd dur $ Note x) ) (Rest 0) [a,b,c]
```

This operator (notice the fitting notation) takes the absolute value of the difference between two pitches. For instance $C,3| - |C,4 = C,4| - |C,3 = 12$.

```
(|-|) :: Pitch -> Pitch -> Int
(|-|) a b = abs (absPitch a - absPitch b)
```

The distance function returns the sum of the distance between a chords corresponding notes. Since we always represent chords in a uniform matter (lowest notes to the left) this function returns a correct value.

```
distance :: ChordPitch -> ChordPitch -> Int
distance (a1, b1, c1) (a2, b2, c2) = (a1|-|a2) + (b1|-|b2) + (c1|-|c2)
```

This nifty little minimize function picks a chord from a list of chords where the distance (as defined previously) between it and another supplied 'base-chord' is minimized.

```
minimize :: ChordPitch -> [ChordPitch] -> ChordPitch
minimize prev (x:xs) = foldl (\acc y -> if distance prev y < distance prev acc then y el
```

The permutate function returns all the possible chord voicings within a limited range of pitches. Since this function is only used for chord voicing the root note of is fixed to the 4th octave. It enforce that the first rule in the lab-requirements is respected. The third rule is respected also because the hardcoded patterns always generate the closest voicing within the triad.

```
permutate :: Chord -> [ChordPitch]
permutate (p, t)
          | t == TriMin = map toTuple (zipWith (zipWith trans) minPat rootNote)
          | otherwise = map toTuple (zipWith (zipWith trans) majPat rootNote)
          where toTuple [a,b,c] = (a,b,c)
                minPat = [[0,3,7],[3,7,12],[7,12,15]]
                majPat = [[0,4,7],[4,7,12],[7,12,16]]
                rootNote = (repeat.repeat) (p,4)
```

Finaly the autoChord function combines the power of 'minimize' and permutate by for each chord in the ChordProgression picking the voicing that lies closest to the previously played chord (just guaranteeing rule 2). The first chord played will always have its root in the lowest note since we select the first item from all available permutations. The combine helper function recusively applies this chord selection algorithm to the end of the song and then the main foldl uses the chosen chords to transform into the final Music.

```
autoChord :: Key -> ChordProgression -> Music
autoChord _ cp@((c,_):_) = foldl (\acc x -> acc :+: x) (Rest 0) (combine ((permutate c)!
                         where combine last ((c,d):[]) = toChord d (minimize last (permu
                               combine last ((c,d):cp) = (toChord d (minimize last (perm
```

Final Thoughts

We haven't implemented the autoComp function because during the experiments with different volumes and instruments we found that we needed the freedom to choose these individually for the bass and chord lines. Therefore we chose to simply generate and combine these lines in the music files 'twinkle.hs' and 'clocks.hs'.

\end{document}