In this article we lay the foundations for project work in music by looking at elementary aspects of western music, and exploring the capabilities of Java's built in synthesizer. Our goal is to build a domain specific language which allows convenient specification, performance, and display of musical pieces.

When working with the Java sound API, please use headphones or earbuds so as not to disturb other people in the lab.

1 Musical instruments

Broadly speaking, music is a form of structured sound. Sound itself is our perception of vibrations in the air which create sympathetic vibrations of our ear drums, and via the workings of the inner ear into changes in brain activity. A young human can perceive frequencies between about 20 and 20,000 cycles per second (written 20-20kHz) though maximum sensitivity is between 2kHz and 5kHz. We perceive different frequencies as different pitches: an increase or decrease in frequency is perceived as an increase or decrease in pitch.

A musical instrument is a device for creating structured sound. Physical objects display frequencies at which they 'want' to vibrate: we call these their resonant frequencies. If you have ever pushed somebody on a swing you will understand resonance: the swing has a frequency at which it naturally arcs back and forth, and if you give a little push just at the top of the arc then you can maintain steady smooth motion with little effort. If on the other hand you shove the swing before it reaches the high point, then the motion can become very irregular, and even cause the person on the swing to be thrown off. It turns out that you can change the resonant frequency of a swing by changing the length of the ropes holding the seat: longer ropes give a slower swing frequency.

It is a general rule that large physical objects resonate at lower frequencies than small physical objects, and that if you want to keep an object vibrating whilst using as little energy as possible, then you should stimulate it at its resonant frequency. This is, perhaps, why tiny humming birds' wings move so quickly that we perceive the disruption in the air as a hum, whilst an albatross with a 3m wingspan beats its wings slowly.

If we tension a string and pluck it, then it will vibrate at its resonant frequency. If we shorten the string, or increase its tension. or replace it with a lighter string, then the resonant frequency will increase. These are the principles behind stringed instruments including guitars, violins and pianos.

If we take an open bottle and blow across the top of it then the air in it will resonate. By using a smaller bottle (or perhaps two identical bottles with one half full of water) we can try smaller mass of air which will resonate at a higher frequency. This is the operating principle of woodwind instruments.

There are many variations on these basic themes: strings may be plucked, hit with hammers or excited with a bow. Air resonators may be fed with pumped air (as in a pipe organ) or blown into; their resonant frequencies may be modified by opening and closing valves between air spaces or by deforming the resonator, as in a trombone. For stringed instruments in particular, it is common to provide a coupled mass of air in a hollow *sound box* that will resonate in sympathy with the primary resonator, strengthening the amplitude of the air vibrations. Some modern instruments (such as the electric guitar) directly convert resonator vibrations into electrical signals which can be amplified, transmitted long distances and fed to a loudspeaker; the microphone is a general device for converting air pressure waves into an electrical signal.

Even the highest perceivable sound frequency is very low compared to the instruction execution frequencies of current computers. At 20kHz, each sound cycle lasts for $50\mu s$. A modern desktop processor can achieve instruction execution frequencies of more than 2×10^9 Hz, and can thus execute more than 100,000 instructions during each audio cycle. This makes it feasible to use software to generate quite complicated musical waveforms in real time. Such a device is called a digital music synthesizer, and the standard Java distribution ships with libraries for music synthesis.

Ensuring the correct synchronisation between multiple synthesized instruments and input devices such as keyboards requires careful coding and protocol design: in 1982 the *Musical Instrument Digital Interface* standard was published, and this has become a very broadly implemented system for controlling musical instruments. The Java distribution contains classes which may be used with MIDI controllers (such as keyboards) to make entirely electronic music. We shall give examples of the use of this library; and we shall write a very simple Domain Specific Language is to give the non-Java programmer access to these facilities using a small self-contained notation.

2 The perception of pitch

When presented with a complex repetitive waveform, the ear resolves it into multiple pitches, that is we can perceive the separate frequencies individually. In this respect, the ear is fundamentally different to the eye which 'averages' frequencies: when presented with a patch of green light overlaying a patch of red light, we perceive yellow. The separation of a waveform into its constituent frequencies is called Fourier Analysis: the ear effectively performs Fourier Analysis whereas the eye merges frequencies.

As we increase frequency, we hear an ascending pitch. Interestingly, and fundamentally, the ear perceives frequencies which are integer multiples of one another as 'the same but different' and nearly all music systems use this observation to split the frequency spectrum up into a sequence of 'octaves': one octave is the range of musical pitches between some frequency f and the frequency $2f$. This perception of frequency doubling naturally leads to a log-style description of pitch.

Some instruments offer a continuous range of frequencies (examples include the Theramin, fretless stringed instruments and the human voice) but most instruments provide a finite set of discrete pitches which may be based on individual tuned resonators (like the strings of a piano) or by discrete adjustments of an otherwise-continuous resonator (like the frets used in a guitar). When using a computer to generate audio waveforms, there are no constraints at all, but in practice we often use the computer to make sounds like traditional instruments.

3 The physics and psychology of pitch

Music appreciation is highly culturally conditioned, and perceptions of 'satisfying' music vary geographically and over time. As computer scientists, we understand how to systematise and implement behaviours which 'make sense' to our users by hiding low-level complexity and only providing control over high level concepts: for instance, when we connect our laptops to a network, we do not expect users to understand the complexities of the network protocol, or even to know the numeric address of the machine they are connecting to. Similarly, musicians organise the continuum of available frequencies into a set of conventions which 'feel right'; and growing up within a particular musical culture these conventions we may come to feel in some sense natural and fundamental. However, we must never lose sight of the fact that alternative conventions may be just as natural to people growing up in other cultures.

Most western music is organised around the twelve-tone-equal-temperament scale (12-TET) in which an octave is divided into twelve discrete frequencies. The ratio between octaves is 2; the ratio between two successive notes (called a *semitone* is thus $\sqrt[12]{2}$. Each discrete frequency is called a note. Western keyboard instruments such as the piano have individual resonators tuned to each note, and a key which when pressed causes that note to sound. Some instruments provide only a subset of the available notes in an octave, so to avoid these complications we shall use the keyboard as a reference instrument.

We noted before that the ear is particularly sensitive to frequencies up to about 5kHZ. If we start at, say, 20Hz and generate tones by multiplying by $\sqrt[12]{2}$, we get to 6kHz in around 100 steps, so we shouldn't be surprised to find that large concert pianos have 88 keys, and that nonstandard pianos have been constructed with 102 keys. Of course, we could have started at 25Hz or 19Hz. The particular mapping between the discrete notes and the continuum of frequencies is called the *tuning* of an instrument. For an equal tempered tuning such as 12-TET, it suffices to pick one particular frequency for one particular note: the other notes are then fixed by the $\sqrt[12]{2}$ ratio between semitones.

Current orchestral practice is to use 440Hz as a tuning standard and to tune the 49th key of a standard 88 note keyboard to to that frequency. This then results in the leftmost key generating 27.5Hz, and rightmost key generating 4186.01Hz. (A 102 key keyboard, rarely implemented, ranges from 16.3516Hz to 5587.65Hz.)

The MIDI standard defines 128 notes numbered from zero to 127, with twenty notes below and twenty notes above the standard piano keyboard. Key zero generates 8.17Hz and key 127 12,543.85Hz. Key 69 gives the 440Hz concert tuning standard.

3.0.1 A note on alternative tunings

There is nothing inherently perfect about this particular tuning. The 440Hz standard was internationally agreed in 1939, and became an ISO standard in 1955. However frequencies from 400Hz to 460hz are known to have been used historically, and those frequencies are more more than $\sqrt[5]{2}$ apart, which is greater than two semitones: playing an historical piece to modern tuning may therefore yield a performance that differs significantly from the composer's intent.

Apart from these shifts in reference frequency, equal temperament (the division of the octave using a constant ratio) is not the only way of mapping discrete notes on to the frequency continuum. Many musical traditions instead use ratios of small integers, since frequencies related in his way form harmonious combinations: these tunings are collectively called *just intonation* and arise naturally with certain classes of instrument. The ratio 3:2 forms the basis of the so-called *Pythagorean tuning*; many other systems exist.

The division of the octave into 12 notes is also merely a convention. Divisions into 15, 19, 34 and even 53 notes have been studied; one way of thinking about this is that by having more notes we can more closely approach the small-integer-ratio harmonics of just intonation. There is a vast literature on tunings which you can begin to explore through online articles: in the rest of this section we shall restrict ourselves to 12-TET.

4 Pure tones and instrument voices

The resonators used in musical instruments generate more than one frequency. Roughly speaking, the note that we hear is the lowest frequency produced, but there will usually be many related *overtones* present, usually integer multiples of the lowest frequency. The lowest frequency is called the fundamental; the integer-multiple frequencies are the harmonics. For a fundamental frequency f, the first harmonic has frequency $2f$, the second harmonic $3f$ and so on. Notice how these harmonics are at the octave intervals: hence a fundamental and its overtones together sound like a single note rather than resolving into several independent notes.

A pure single tone sounds rather other-worldly: a tuning fork or the human whistle is probably the closest thing to a pure tone resonator that most people hear. Of course, we can use the computer to generate pure tones, once we know the waveform.

4.0.1 Fourier analysis and synthesis

In 1822, Jean-Baptiste Joseph Fourier published a claim that any periodic waveform could be decomposed into a (possibly infinite) set of sinusoidal waveforms, which when added together would reconstruct the original waveform. This observation has turned out to have many applications in physics and engineering. It is of direct relevance to sound synthesis, since it suggests that if we can write a program that generates sine waves at different frequencies and then add them together in various proportions we can numerically construct any audio waveform. This is the principle behind music synthesizers. Interestingly, there is a procedure by which we can take an arbitrary waveform and numerically decompose it into its constituent sine waves. We call a display of the strength of each frequency a *spectrum analysis.*

If our musical instrument resonator produces only a fundamental and harmonics, then we can characterise the sound of, say, a piano string by listing the proportions of sine waves of frequencies f, $2f$, $3f$,... kf where k is chosen to be beyond the limit of human hearing. Even for a very low fundamental note such as 20Hz, the tenth harmonic exceeds 20kHz. Hence we have the prospect of being able to accurately encode the detailed sound of a piano note into eleven real numbers, and then reconstituting the sound in realtime using software.

The conversion of a waveform to a spectrum display of frequencies is called Fourier Analysis; the reverse process of converting a series of proportions of sine waves to a single waveform is called Fourier Synthesis. We sometimes refer to operations on waveforms as 'working in the time domain' and operations on frequency proportions as 'working in the frequency domain'.

4.0.2 The Nyquist-Shannon criterion and making recordings

An ability to encode a single note of a single instrument accurately is clearly important for synthesis. However, it does not tell us how to capture the behaviour of an entire orchestra, or indeed how to encode non-musical sounds.

We can use a computer as a sound recorder by connecting a microphone and then measuring its output, say every microsecond. We use an Analogue to Digital Converter (ADC) to convert the voltage developed by the microphone into a number. Typical high quality digital audio systems use around 16 bits to represent each sample. By storing the resulting sequence of integer numbers, we can have a permanent record of a sound experience which may be reconstructed by converting the numbers back into voltages and applying the resulting waveform to a loudspeaker.

We can see that if we sample the original sound too slowly, then we may lose information. On the other hand, if we sample at very high speed then we shall require extra storage. A fascinating result which arises from Fourier analysis is that we can capture the full detail of any waveform, no matter how complex, up to some bounding frequency f by sampling the waveform at no more than $2f$. This is why CD quality audio samples waveforms at 44.1kHz: since the human ear can perceive sounds up to 20kHz, a sample rate of greater then 40kHz ensures that no information is lost. (In detail, the figure of 44.1kHz arises as a result of early experiments using video recorders to store audio information: you can read the story online.) This $2f$ requirement is called the Nyquist-Shannon criterion.

5 Tempo, rhythm and articulation

It is unusual for a note to be played continuously (although bagpipes and some other instruments have a *drone* which sounds continuously during a performance).

Instead, the playing time is divided up into discrete beats which set the duration of the basic note.

In the western tradition, a piece of music will have an indicated *tempo*, sometimes expressed as *beats per minute* or bpm. As computer scientists, we might prefer to use Hz to specify the tempo, that is, beats per second. A very slow piece would be below 30bpm and a very fast piece above 200bpm, from which we can see that beat frequencies range from around 0.5 to 3.5Hz.

**** Todo: Rhythms**

Much of the character of a performance is embedded in the detailed way in which a performer uses the tempo. A straightforward approach is to leave a very short silence at the end of each beat period, and to sound each note uniformly throughout the rest of the beat period. This very simplistic approach is easy to program but sounds, well, synthetic.

A human performer even when attempting uniformity will display some small variations in the length of notes. More significantly, humans players deliberately vary the details of note timing within the basic rhythm framework, a technique known as *articulation*. For instance, some notes may be run together into a smoothly connected frequency shift whereas other are deliberately shortened so as to create a jumpy effect. In wind instruments articulation is achieved by controlling airflow with the tongue; in stringed instruments by dampening the vibrations with the hand. Other forms of articulation include rapid periodic changes in amplitude (called tremolo) and rapid periodic changes in pitch (called vibrato). A slower shift in pitch is often called a *pitch bend*: some electric guitars (such as the fender Stratocaster) have an arm which allows the tension and length of the strings to be varied – this device is often called a tremolo bar (although really it is a vibrato bar).

6 Musical terminology for pitch

Musicians use a very large number of technical terms, and this can be rather overwhelming at a first encounter. However, we are interested in Domain Specific Languages, and musician's terminology certainly represents a very widely used language which is extremely domain specific, and as such can be the basis of some interesting case studies.

Musical nomenclature has grown up over a long historical period, and can seem rather arbitrary to outsiders even though there is usually an underlying logic. For instance, one might imagine that the divisions of an octave into 12-semitones might be represented by twelve unique names. In fact, in the western tradition there are seven unique names (the letters A through G inclusive), and two modifiers ♯ and ♭ (spoken sharp and flat) which raise (lower) by a semitone the note represented by a name.

This initially surprising naming convention arises from the observation that certain sequences of seven notes sound harmonious, and that the majority of western musical melodies are *mostly* constructed around seven note selections.

By appending a ♯ or a ♭ symbol to the seven basic names we can name the 'missing' five semitones. A conventional way of writing an ascending sequence of 12 semitones in an octave is:

A A♯ B C C♯ D D♯ E F F♯ G G♯

and a conventional way to write a descending sequence is:

A A♭ G G♭ F E E♭ D D♭ C B B♭

Using the 12-TET tuning (but not necessarily for other tunings), A♯ and B♭ represent the same frequency, and we can enumerate the full set of notes in an octave as

A A♯/B♭ B C C♯/D♭ D D♯/E♭ E F F♯/G♭ G G♯

**** Todo: Octave numbers**

7 Major and minor scales

Our basic pitch palette, then, comprises octaves of 12 fundamental notes each separated by a semitone. When played, notes come with a variety of harmonics which allow us to distinguish, say, a violin note from a guitar note.

Music can generate emotional responses in humans. It is clear that these responses are culturally conditioned, but nevertheless within a culture such as our own in which individuals are exposed to many musical pieces, strong associations between particular musical progressions and particular emotional states seem to be almost universally recognised — in the western tradition the difference between a joyous and a sad piece is well understood by most listeners.

The first component of mood is harmony. Some sequences of notes sound harmonious and some do not: we say they are discordant (which literally means that they disagree with each other). We can test our response to sequences by playing subsets of the 12 tones in an octave in ascending or descending order: it turns out that some sound good and some are unpleasant. Such a sequenced subset of the tones is called a *scale*.

If we think of the 12 semitones laid out as a 12-bit vector representing the presence or absence of a note within some scale, it is easy to see that there are $2^{12} = 4096$ scales. The one with all twelve notes in it is called the chromatic scale; its dual with no notes in at all is simply silence (and therefore of little musical utility).

We have already noted that western music focuses on scales with seven notes. It turns out that only two families of such seven note scales find wide application in popular music. The *major* scales start with any of the twelve notes and then include notes according to the increments

+2 +2 +1 +2 +2 +2 +1

The *minor* scales begin with any note, and then include notes according to the increments:

+2 +1 +2 +2 +1 +3 +1

Scales are usually played so as to finish one octave above the root. Hence, we play a major scale rooted on keyboard key k by playing the keys

$$k, k+2, k+4, k+5, k+7, k+9, k+11, k+12$$

and a minor scale with

$$k, k+2, k+3, k+5, k+7, k+8, k+11, k+12$$

As a further example, consider the scale made up of two equally spaced notes. These are three whole tones apart and thus called a tritone. When played, this creates, at best, a sense of tension: some might even say it sounds wrong.

8 Chords

A chord is a set of notes played simultaneously. Just as with scales, particular combinations sound harmonious, and the most common way of forming a chord for a root note k is to take the first, third and fifth elements of either the major or minor scale rooted on k. We write CM (spoken C-major) for the chord rooted on C using the major scale, and Cm (C-minor) for the chord rooted on C using the minor scale.

Chords formed of notes $k, k+6, k+12$ sound particularly inharmonious, made up as they are of a pair of tritones.

We can play a simple melody by picking out single notes. If we replace each note by the major chord rooted at that note then we get a fuller sound. We can do the same with the minor chords; and in general a melody and chords based on the minor scale will sound darker and perhaps more gentle: the major scale sounds brighter.

9 Synthesizing music with Java and MIDI

**** Todo: Overview of MIDI**

```java
package uk.ac.rhul.cs.csle.artmusic;

import javax.sound.midi.MidiChannel;
import javax.sound.midi.MidiSystem;
import javax.sound.midi.Synthesizer;

public class ARTMiniMusicPlayer {
   private Synthesizer synthesizer;
   private MidiChannel[] channels;
   private int defaultOctave = 5;
   private int defaultVelocity = 50;
   private int bpm;
   private double bps;
   private double beatPeriod;
   private double beatRatio = 0.9;
   private int beatSoundDelay = (int) (1000.0 * beatRatio / bps);
   private int beatSilenceDelay = (int) (1000.0 * (1.0 - beatRatio) / bps);

   public ARTMiniMusicPlayer() {
      try {
         System.out.print(MidiSystem.getMidiDeviceInfo());
         synthesizer = MidiSystem.getSynthesizer();
         synthesizer.open();
         channels = synthesizer.getChannels();
      } catch (Exception e) {
         System.err.println("miniMusicPlayer exception: " + e.getMessage());
         System.exit(1);
      }

      setBeatRatio(0.9);
      setBpm(100);
      setDefaultVelocity(50);
   }

   public int getDefaultOctave() {
      return defaultOctave;
   }

   public void setDefaultOctave(int defaultOctave) {
      this.defaultOctave = defaultOctave;
   }

   public int getDefaultVelocity() {
      return defaultVelocity;
   }

   public void setDefaultVelocity(int defaultVelocity) {
        this.defaultVelocity = defaultVelocity;
    }

    public int getBpm() {
        return bpm;
    }

    public void setBpm(int bpm) {
        this.bpm = bpm;
        bps = bpm / 60.0;
        beatPeriod = 1000.0 / bps;
        beatSoundDelay = (int) (beatRatio * beatPeriod);
        beatSilenceDelay = (int) ((1.0 - beatRatio) * beatPeriod);
    }

    public void setBeatRatio(double beatRatio) {
        this.beatRatio = beatRatio;
        beatSoundDelay = (int) (beatRatio * beatPeriod);
        beatSilenceDelay = (int) ((1.0 - beatRatio) * beatPeriod);
    }

    private int noteNameToMidiKey(String n, int octave) {
  // @formatter:off
  int key = octave * 12 +
              ( n.equals("C") ? 0
              : n.equals("C#") ? 1
              : n.equals("Db") ? 1
              : n.equals("D") ? 2
              : n.equals("D#") ? 3
              : n.equals("Eb") ? 3
              : n.equals("E") ? 4
              : n.equals("F") ? 5
              : n.equals("F#") ? 6
              : n.equals("Gb") ? 6
              : n.equals("G") ? 7
              : n.equals("G#") ? 8
              : n.equals("Ab") ? 8
              : n.equals("A") ? 9
              : n.equals("A#") ? 10
              : n.equals("Bb") ? 10
              : n.equals("B") ? 11
              : -1);
  // @formatter:on

        if (key < 0 || key > 127) {
            System.err.println("miniMusicPlayer exception: attempt to access out of range MIDI key " + n + octave);
            System.exit(1);
        }
        return key;
    }

    // Silence
    public void rest(int beats) {
        try {
```

```java
         Thread.sleep((long) (beats * beatPeriod));
      } catch (InterruptedException e) {
         /* ignore interruptedException */ }
   }

   // Single notes
   public void play(int k) {
      try {
         channels[0].noteOn(k, defaultVelocity);
         Thread.sleep(beatSoundDelay);
         channels[0].noteOn(k, 0);
         Thread.sleep(beatSilenceDelay);
      } catch (InterruptedException e) {
         /* ignore interruptedException */ }
   }

   public void play(String n) {
      play(noteNameToMidiKey(n, defaultOctave));
   }

   public void play(String n, int octave) {
      play(noteNameToMidiKey(n, octave));
   }

   // Arrays of notes
   public void play(int[] k) {
      try {
         for (int i = 0; i < k.length; i++)
            channels[1].noteOn(k[i], defaultVelocity);
         Thread.sleep(beatSoundDelay);
         for (int i = 0; i < k.length; i++)
            channels[1].noteOn(k[i], 0);
         Thread.sleep(beatSilenceDelay);
      } catch (InterruptedException e) {
         /* ignore interruptedException */ }
   }

   public void playSequentially(int[] k) {
      try {
         for (int i = 0; i < k.length; i++) {
            channels[i].noteOn(k[i], defaultVelocity);
            Thread.sleep(beatSoundDelay);
            channels[i].noteOn(k[i], 0);
            Thread.sleep(beatSilenceDelay);
         }
      } catch (InterruptedException e) {
         /* ignore interruptedException */ }
   }

   // Scales
   public void playScale(String n, ARTScale s) {
      playScale(noteNameToMidiKey(n, defaultOctave), s);
   }

```

```
public void playScale(String n, int octave, ARTScale s) {
   playScale(noteNameToMidiKey(n, octave), s);
}

public void playScale(int k, ARTScale s) {
   int[] keys;
   switch (s) {
   case CHROMATIC:
      keys = new int[] { k, k + 1, k + 2, k + 3, k + 4, k + 5, k + 6, k + 7, k + 8, k + 9, k + 10, k + 11, k + 12 }
      break;

   case MAJOR: // TTSTTTS
      keys = new int[] { k, k + 2, k + 4, k + 5, k + 7, k + 9, k + 11, k + 12 };
      break;

   case MINOR_NATURAL: // TSTTSTT
      keys = new int[] { k, k + 2, k + 3, k + 5, k + 7, k + 8, k + 10, k + 12 };
      break;
   case MINOR_HARMONIC: // TSTTS3S
      keys = new int[] { k, k + 2, k + 3, k + 5, k + 7, k + 8, k + 11, k + 12 };
      break;
   case MINOR_MELODIC_ASCENDING: // TSTTS3S - harmonic with with sixth sharpened
      keys = new int[] { k, k + 2, k + 3, k + 5, k + 7, k + 9, k + 11, k + 12 };
      break;
   case MINOR_MELODIC_DESCENDING: // TSTTS3S - harmonic with seventh flattened making it the same as th
      keys = new int[] { k + 12, k + 10, k + 8, k + 7, k + 5, k + 3, k + 2, k };
      break;

   default:
      keys = new int[] { 0 };
      break;
   }
   playSequentially(keys);
}

// Programmed chords
public void playChord(String n, ARTChord type) {
   playChord(noteNameToMidiKey(n, defaultOctave), type);
}

public void playChord(String n, int octave, ARTChord type) {
   playChord(noteNameToMidiKey(n, octave), type);
}

public void playChord(int k, ARTChord type) {
   int[] keys;
   switch (type) {
   case NONE:
      keys = new int[] { k };
      break;
   case MAJOR:
      keys = new int[] { k, k + 4, k + 7 };
      break;
   case MAJOR7:
```

```
            keys = new int[] { k, k + 4, k + 7, k + 11 };
            break;
        case MINOR:
            keys = new int[] { k, k + 3, k + 7 };
            break;
        case MINOR7:
            keys = new int[] { k, k + 4, k + 7 };
            break;
        default:
            keys = new int[] { 0 };
            break;
        }
        play(keys);
    }

    private void tune() {
        int base = 47;
        play(base + 14);
        play(base + 12);
        play(base + 11);
        play(base + 7);
        play(base + 5);
        play(base + 7);
        play(base + 2);
        rest(2);
    }

    private void tuneChordMajor() {
        int base = noteNameToMidiKey("C", 5);
        playChord(base + 14, ARTChord.MAJOR);
        playChord(base + 12, ARTChord.MAJOR);
        playChord(base + 11, ARTChord.MAJOR);
        playChord(base + 7, ARTChord.MAJOR);
        playChord(base + 5, ARTChord.MAJOR);
        playChord(base + 7, ARTChord.MAJOR);
        playChord(base + 2, ARTChord.MAJOR);
    }

    private void tuneChordMinor() {
        int base = noteNameToMidiKey("C", 5);
        playChord(base + 14, ARTChord.MINOR);
        playChord(base + 12, ARTChord.MINOR);
        playChord(base + 11, ARTChord.MINOR);
        playChord(base + 7, ARTChord.MINOR);
        playChord(base + 5, ARTChord.MINOR);
        playChord(base + 7, ARTChord.MINOR);
        playChord(base + 2, ARTChord.MINOR);
    }

    public void close() {
        synthesizer.close();
    }

    public static void main(String[] args) {
      System.err.println("miniMusicPlayer test routine");
      ARTMiniMusicPlayer mp = new ARTMiniMusicPlayer();

        mp.playScale("C", ARTScale.CHROMATIC);
        mp.rest(2);
        String note = "C";
        int octave = 6;
        mp.play(note, octave);
        mp.rest(2);
        mp.playScale("C", ARTScale.MAJOR);
        mp.rest(2);
        mp.playScale("C", ARTScale.MINOR_NATURAL);
        mp.rest(2);
        mp.playScale("C", ARTScale.MINOR_HARMONIC);
        mp.rest(2);
        mp.playScale("C", ARTScale.MINOR_MELODIC_ASCENDING);
        mp.playScale("C", ARTScale.MINOR_MELODIC_DESCENDING);
        mp.rest(2);
        mp.playChord("C", ARTChord.MAJOR);
        mp.rest(2);
        mp.playChord("C", ARTChord.MINOR);
        mp.rest(2);
        mp.tune();
        mp.rest(2);
        mp.tuneChordMajor();
        mp.rest(2);
        mp.tuneChordMinor();
        mp.rest(2);
// Tritone scale and scale
      mp.playSequentially(new int[] { 50, 56, 62 });
      mp.rest(2);
      mp.play(new int[] { 50, 56, 62 });
      mp.rest(2);

      mp.close();
    }
}
```

10 `minimusic` – a DSL to access `MiniMusicPlayer`

```
melody sanctuary {

D+M C+M B+ G F G m D m7
}

x = 3;
while x > 0 do { print("x is ", x, "\n"); x = x -1; }

play sanctuary;
```

```
(******************************************************************************
*
* miniMusic.art - Adrian Johnstone 18 Februrary 2017
```

```
*
******************************************************************************)
prelude { import java.util.HashMap; import uk.ac.rhul.cs.csle.artmusic.*; }

support {
HashMap<String, Integer> variables = new HashMap<String, Integer>();
HashMap<String, ARTGLLRDTHandle> melodies = new HashMap<String, ARTGLLRDTHandle>();
ARTMiniMusicPlayer mp = new ARTMiniMusicPlayer();
}

whitespace &WHITESPACE
whitespace &COMMENT_NEST_ART
whitespace &COMMENT_LINE_C

statements ::= statement | statement statements

statement ::= ID '=' e0 ';' { variables.put(ID1.v, e01.v); } | (* assignment *)

              'if' e0 'then' statement< elseOpt< (* if statement *)
              { if (e01.v != 0)
                  artEvaluate(statement.statement1, statement1);
                else
                  artEvaluate(statement.elseOpt1, elseOpt1);
              } |

              'while' e0< 'do' statement< (* while statement *)
              { artEvaluate(statement.e01, e01);
                while (e01.v != 0) {
                  artEvaluate(statement.statement1, statement1);
                  artEvaluate(statement.e01, e01);
                }
              } |

              'print' '(' printElements ')' ';' | (* print statement *)

              'melody' ID statement< { melodies.put(ID1.v, statement.statement1); } |
              'play' ID ';'
                { if (!melodies.containsKey(ID1.v))
                      artText.println(ARTTextLevel.WARNING, "ignoring request to play undefined melody: " + ID1.v
                  else
                      artEvaluate(melodies.get(ID1.v), null);
                } |

              '{' statements '}' | (* compound statement *)

              bpm | defaultOctave | note | chord | rest

elseOpt ::= 'else' statement | #

bpm ::= 'bpm' INTEGER { mp.setBpm(INTEGER1.v); }

beatRatio ::= 'beatRatio' REAL { mp.setBeatRatio(REAL1.v); }

defaultOctave ::= 'defaultOctave' INTEGER
```

```
 { if (INTEGER1.v < 0 || INTEGER1.v > 10)
      artText.println(ARTTextLevel.WARNING, "ignoring illegal MIDI octave number " + INTEGER1.v);
    else
       mp.setDefaultOctave(INTEGER1.v);
 }

note ::= simpleNote chordMode { mp.playChord(simpleNote1.v.trim(), chordMode1.v ); } |
         simpleNote shifters chordMode { mp.playChord(simpleNote1.v.trim(),
           mp.getDefaultOctave() + shifters1.v, chordMode1.v); } |
         simpleNote INTEGER chordMode { mp.playChord(simpleNote1.v.trim(), INTEGER1.v, chordMode1.v); }

chordMode <v:ARTChord> ::= # { chordMode.v = ARTChord.NONE; } |
                           'm' { chordMode.v = ARTChord.MINOR; } | 'm7' { chordMode.v = ARTChord.MINOF
                           'M' { chordMode.v = ARTChord.MAJOR; } | 'M7' { chordMode.v = ARTChord.MAJO

simpleNote<leftExtent:int rightExtent:int v:String> ::=
  simpleNoteLexeme { simpleNote.v = artLexeme(simpleNote.leftExtent, simpleNote.rightExtent).trim(); }

simpleNoteLexeme ::= 'A' | 'A#' | 'Bb' | 'B' | 'C' | 'C#' | 'Db' | 'D' | 'D#' | 'Eb' | 'E' | 'F' | 'F#' | 'Gb' | 'G' | 'G#'

shifters<v:int> ::= '+' {shifters.v = 1;} | '−' {shifters.v = −1;} |
                    '+' shifters {shifters.v = shifters1.v + 1; } |
                    '−' shifters {shifters.v = shifters1.v − 1; }

chord ::= '[' notes ']'

notes ::= note | note notes

rest ::= '.' { mp.rest(1); } | '..' { mp.rest(2); } | '...' { mp.rest(3); } | '....' { mp.rest(4); }

printElements ::= STRING_DQ { artText.printf("%s", STRING_DQ1.v); } |
                  STRING_DQ { artText.printf("%s", STRING_DQ1.v); } ',' printElements |
                  e0 { artText.printf("%d", e01.v); } | e0 { artText.printf("%d", e01.v); } ',' printElements

e0 <v:int> ::= e1 { e0.v = e11.v; } |
               e1 '>' e1 { e0.v = e11.v > e12.v ? 1 : 0; } | (∗ Greater than ∗)
               e1 '<' e1 { e0.v = e11.v < e12.v ? 1 : 0; } | (∗ Less than ∗)
               e1 '>=' e1 { e0.v = e11.v >= e12.v ? 1 : 0; } | (∗ Greater than or equals∗)
               e1 '<=' e1 { e0.v = e11.v <= e12.v ? 1 : 0; } | (∗ Less than or equals ∗)
               e1 '==' e1 { e0.v = e11.v == e12.v ? 1 : 0; } | (∗ Equal to ∗)
               e1 '!=' e1 { e0.v = e11.v != e12.v ? 1 : 0; } (∗ Not equal to ∗)

e1 <v:int> ::= e2 { e1.v = e21.v; } |
                e1 '+' e2 { e1.v = e11.v + e21.v; } | (∗ Add ∗)
                e1 '−' e2 { e1.v = e11.v − e21.v; } (∗ Subtract ∗)

e2 <v:int> ::= e3 { e2.v= e31.v; } |
                e2 '∗' e3 { e2.v = e21.v ∗ e31.v; } | (∗ Multiply ∗)
                e2 '/' e3 { e2.v = e21.v / e31.v; } | (∗ Divide ∗)
                e2 '%' e3 { e2.v = e21.v % e31.v; } (∗ Mod ∗)

e3 <v:int> ::= e4 {e3.v = e41.v; } |
                '+' e3 {e3.v = e41.v; } | (∗ Posite ∗)
                '−' e3 {e3.v = −e41.v; } (∗ Negate ∗)
```

```

e4 <v:int> ::= e5 { e4.v = e51.v; } |
                        e5 '**' e4 {e4.v = (int) Math.pow(e51.v, e41.v); } (* exponentiate *)

e5 <v:int> ::= INTEGER {e5.v = INTEGER1.v; } | (* Integer literal *)
                        ID { e5.v = variables.get(ID1.v); } | (* Variable access *)
                        '(' e1 { e5.v = e11.v; } ')' (* Parenthesised expression *)

ID <leftExtent:int rightExtent:int lexeme:String v:String> ::=
   &ID {ID.lexeme = artLexeme(ID.leftExtent, ID.rightExtent); ID.v = artLexemeAsID(ID.leftExtent, ID.rightExtent); }

INTEGER <leftExtent:int rightExtent:int lexeme:String v:int> ::=
   &INTEGER {INTEGER.lexeme = artLexeme(INTEGER.leftExtent, INTEGER.rightExtent);
      INTEGER.v = artLexemeAsInteger(INTEGER.leftExtent, INTEGER.rightExtent); }

REAL <leftExtent:int rightExtent:int lexeme:String v:double> ::=
   &REAL {REAL.lexeme = artLexeme(REAL.leftExtent, REAL.rightExtent);
      REAL.v = artLexemeAsInteger(REAL.leftExtent, REAL.rightExtent); }

STRING_DQ <leftExtent:int rightExtent:int lexeme:String v:String> ::=
   &STRING_DQ {STRING_DQ.lexeme = artLexeme(STRING_DQ.leftExtent, STRING_DQ.rightExtent);
      STRING_DQ.v = artLexemeAsString(STRING_DQ.leftExtent, STRING_DQ.rightExtent); }
```