A Nineteen Tone Scale Synthesizer

A N D R E A S   L I N D S T R Ö M
a n d   A L B E R T   W I F S T R A N D

KTH Computer Science
and Communication

Bachelor of Science Thesis
Stockholm, Sweden 2012

A Nineteen Tone Scale Synthesizer

ANDREAS LINDSTRÖM
and ALBERT WIFSTRAND

DD143X, Bachelor's Thesis in Computer Science (15 ECTS credits)
Degree Progr. in Computer Science and Engineering 300 credits
Royal Institute of Technology year 2012
Supervisor at CSC was Anders Askenfelt
Examiner was Mårten Björkman

URL: www.csc.kth.se/utbildning/kandidatexjobb/datateknik/2012/
lindstrom_andreas_OCH_wifstrand_albert_K12047.pdf

Kungliga tekniska högskolan
Skolan för datavetenskap och kommunikation

KTH CSC
100 44 Stockholm

URL: www.kth.se/csc

Abstract

According to theoretical considerations involving acoustics, mathematics and human perception of sound, the 19-tone equal temperament scale, or 19-TET in short, might be a fascinating alternative to the common 12 tone scale (12-TET). The mathematical properties of 19-TET are examined. We show that instruments and music notation designed for 12-TET can be made to favorably correlate with these items for 19-TET, and how well the intervals of 19-TET approximate just intervals, making 19-TET an eligible alternative to (or even substitute for) 12-TET. 19-TET is also applied in a computer program intended for sequencing of musical pieces, which is used to perform listener tests with musicians and non-musicians alike.

Our study suggests that listeners accustomed to 12-TET can enjoy 19-TET music to some extent, but not as much as 12-TET music. Without the cultural bias, however, it might be the other way around, considering the mathematical properties of 19-TET.

Sammanfattning (Swedish abstract)

Den liktempererade 19-tonsskalan kan enligt teoretiska överväganden som innefattar akustik, matematik och hur människor uppfattar ljud vara ett fascinerande alternativ till den vedertagna 12-tonsskalan. 19-tonsskalans matematiska egenskaper undersöks. Vi visar att instrument och notsystem avsedda för 12-tonsskalan med fördel går att anpassa för 19-tonsskalan, samt hur väl intervallen i 19-tonsskalan approximerar strikt matematiskt rena intervall, vilket gör 19-tonsskalan till ett lämpligt alternativ (eller till och med ersättare för) 12-tonsskalan. 19-tonsskalan appliceras också i ett datorprogram avsett för sekvensering av musikstycken, vilket används för att utföra lyssnartester med både musiker och icke-musiker.

Vår studie antyder att musiklyssnare som är vana vid 12-tonsmusik kan uppskatta 19-tonsmusik i någon utsträckning, men inte lika mycket som 12-tonsmusik. Sett till 19-tonsskalans matematiska egenskaper hade det dock kunnat vara tvärtom, om vi bortser från kulturella preferenser.

Contents

1 Introduction

In Western music, the octave is divided into 12 steps by means of the *12 tone equal temperament scale* (12-TET), and it has been the *de facto* scale for virtually every music genre since the beginning of the 19th century. However, there are many ways of dividing the octave, and 12-TET is merely one of them. According to theoretical considerations involving acoustics, mathematics and human perception of sound, 19-TET could be a fascinating alternative to the standard 12 tone scale.

We will explore the 19-TET scale in two ways:

- theoretical discussions regarding the mathematical properties of 19-TET and how it compares to other scales (in particular 12-TET)
- the construction of a software synthesizer using 12 and 19-TET, which we will use for auditory tests with musically educated subjects as well as casual music listeners

By doing this exploration, we want to find answers to a few questions concerning how the 19-TET scale is perceived and appreciated by people:

- Most music listeners are accustomed to the 12 tone scale to such an extent that most other scales will be percieved as dissonant or otherwise foul-sounding. Is it possible to *convert* music written for 12-TET to 19-TET, in a way that is satisfactory to the human ear? If so, for what kind of compositions?
- With 12-TET songs that has been adjusted (converted) for 19-TET, listeners will have expectations on how the music should sound if they have heard said songs in the past. How is 19-TET music percieved by listeners, considering that the frequencies are altered?

2 Statement of collaboration

We have both worked on all parts of the project interchangeably. However, Lindström focused on writing about the theoretical and mathematical properties of scales, the listener tests, the discussion in section 6 and the conclusions in section 7. Wifstrand focused on the programming of the software synthesizer (described in section 4.1) and its documentation (most of section 4), LaTeX formatting and the compilation of the online material (software, source code and audio files) in section B.

3 Theoretical background

3.1 Intervals and just tuning

A musical interval is a combination of two notes, and can be seen as a ratio $f_i{:}f_j$ between the two notes respective frequencies. An interval is called harmonic if the two notes are played simultaneously, melodic if the notes are played in succession. If f_i/f_j can be reduced to a fraction of small integers (more often than not between 1 and 7), the interval is called a justly tuned interval. Justly tuned intervals are generally considered consonant and pleasant to the human ear because of the absence of acoustic beats that can be heard in non-just intervals. Acoustic beats are perceived as periodic changes in volume and arises when two or more sounds of slightly different frequencies are played together [9].

Just intervals are given different names depending on the values of f_i and f_j. For instance, the justly tuned perfect fifth has a frequency ratio of 3:2, the fourth 4:3, the major third 5:4 and the minor third 6:5. The octave has a frequency ratio of 2:1 and notes with this frequency relation are said to belong to the same pitch class, even though their absolute pitch differs. This is related to the concept of octave equivalence. The human ear hears notes an octave apart as being “the same” due to the numerous common harmonics of the two tones [11]. Notes an octave apart are given the same name in Western music notation. Since a scale is considered to start over when the octave is reached, it is sufficient to divide the octave interval into different steps to fully define a scale.

There is a commonly used unit for measuring musical intervals, namely cent. The cent is a logarithmic unit and defined in such a way that the distance between 12-TET semitones (i.e. adjacent keys on a piano) is always 100 cents. It follows that the size of an octave is 1200 cents. Generally, with two tones of frequencies a and b, the distance n in cents is

$$n = 1200 \log_2(b/a)$$

This unit allows for easy comparison of intervals in different tuning systems.

If just intervals are considered the most pleasing to the human ear, why not just tune all instruments in just tuning? The problem is that if an instrument with fixed frequencies (i.e. a piano or a fretted guitar played without bends or other frequency-altering techniques) would be tuned justly, intervals would only be justly tuned within one key, and only relative to one note. When playing in other keys than the justly tuned, intervals can be far from just and sound “off” to listeners. It is impossible to divide the octave into smaller just intervals. For example, four stacked just minor thirds almost, but not exactly, make up a just octave and no stack of just perfect fifths will fit exactly into a stack of octaves [12]. Since variety is an attribute generally expected in music [1], a tuning system is needed that can both offer the possibility to transpose music and to modulate between keys, while still approximating the justly tuned intervals frequency ratios to avoid beating intervals.

3.2 Equally tempered scales

An equally tempered scale is a scale where the ratio of frequencies is equal for all pairs of adjacent notes. To construct an equally tempered scale with N tones that preserves the octave, one must first choose a reference tone to derive all other frequencies from. This reference tone is by convention often set to 440 Hz (A4) in Western music. Since the octave is preserved, it is set to the just interval 2:1. The octave is then divided into N parts so that each interval has a frequency ratio of $2^{x/N}$, where x is the number of steps from the first tone in the scale. The reason why equally tempered scales are used in music is that they allow for musical pieces to be played in different keys without retuning the instrument. With an equally tempered scale, an interval's approximation of the corresponding justly tuned interval is equally good or bad in all keys. With equally tempered scales using a carefully selected value of N, the composer or musician also has many (relatively) consonant intervals to choose from for any given starting note [1]. The drawback with equal temperament is that no interval is justly tuned except for the unison and the octave, since all other frequency ratios are irrational.

3.3 Why 19-TET?

Why is 19-TET of special interest? Why have we chosen that particular scale for comparison with the standard 12-TET, and not some other N-tone equal temperament scale? One could also ask why 12-TET has been established as the standard scale for Western music. The answer to these questions lies mainly in the intervals that the scales generate and how close these intervals approximate just intervals. The most important just intervals and the corresponding approximations in 12 and 19-TET are written out in Table 1.

The pairs of intervals that make up an octave when stacked are called *inverses*. To invert an interval, the root note is raised an octave. For exam-

Interval	**Just, ratio**	**Just, cents**	**12-TET, cents**	**12-TET, error**	**19-TET, cents**	**19-TET, error**	**12-TET, steps**	**19-TET, steps**
19-TET, 18 steps	N/A	N/A	N/A	N/A	1136.84	N/A	N/A	18
Major seventh	15:8	1088.27	1100	11.73	1073.68	-14.58	11	17
Minor seventh	7:4	968.83	1000	31.17	1010.53	41.70	10	16
19-TET, 15 steps	N/A	N/A	N/A	N/A	947.37	N/A	N/A	15
Major sixth	5:3	884.36	900	15.64	884.21	-0.15	9	14
Minor sixth	8:5	813.69	800	-13.69	821.05	7.37	8	13
19-TET, 12 steps	N/A	N/A	N/A	N/A	757.89	N/A	N/A	12
Perfect fifth	3:2	701.96	700	-1.96	694.74	-7.22	7	11
Augmented fourth	10:7	617.49	600	-17.49	631.58	14.09	6	10
Diminished fifth	7:5	582.51	600	17.49	568.42	-14.09	6	9
Perfect fourth	4:3	498.04	500	1.96	505.26	7.22	5	8
19-TET, seven steps	N/A	N/A	N/A	N/A	442.11	N/A	N/A	7
Major third	5:4	386.31	400	13.69	378.95	-7.37	4	6
Minor third	6:5	315.64	300	-15.64	315.79	0.15	3	5
19-TET, three steps	N/A	N/A	N/A	N/A	252.63	N/A	N/A	4
Major second	9:8	203.91	200	-3.91	189.47	-14.44	2	3
Minor second	16:15	111.73	100	-11.73	126.32	14.58	1	2
19-TET, one step	N/A	N/A	N/A	N/A	63,16	N/A	N/A	1

Table 1: Cent values for intervals in just tuning, 12 and 19-TET, along with deviations (error) from just intervals for 12 and 19-TET. The *steps* value represents the interval's size in scale steps from the root note.

ple: C to G makes a perfect fifth, and G to C makes a perfect fourth - the perfect fourth is the inverse of the perfect fifth. An interval has the same absolute value error (but inverted sign) as its inverse in the given scale.

As can be seen in the table, 12-TET closely (an error of 1.96 cents) approximates the just perfect fifth and its inverse the perfect fourth. 19-TET does not approximate the fifth as accurately (the fifth has an error of 7.22 cents) but approximates the just minor third and its inverse the major sixth extremely accurately (error of 0.15 cents). The just major third and its inverse the minor sixth are also closer approximated in 19-TET than in 12-TET (errors with absolute values of 7.37 and 13.69 cents, respectively).

How much these errors affect human perception of music played in the both scales will be evaluated in the listener tests, but an intelligent guess about the qualities of the both scales can be made already at this stage; 12-TET might be better suited for music which relies heavily on the consonance of fifths and fourths, while 19-TET might be better suited for music which relies on consonant thirds and sixths.

Further analytical comparisons of the two scales have been made in the past. These studies are summarized and discussed in section 3.4.

One fascinating aspect of 19-TET is the possibility to map the frequencies of 19-TET to a keyboard similar to the keyboard of a standard 12-TET piano. This was first suggested by composer Joseph Yasser, who made a design for a 19-TET keyboard of which we have drawn a sketch that you can see in Figure 1 [5]. The frequencies of a diatonic C major / A minor scale can be mapped onto the white keys, just like on a 12-TET keyboard. Each black key of the piano is divided into two black keys, and to name the notes of these split keys it is convienient to use the already standardized suffixes $\flat$ and $\sharp$. This means that, for example, the black key on the right hand side of C will be named $C\sharp$, and the black key right next to $C\sharp$ will be named $D\flat$. Since this splitting of the black keys in addition to the white keys has given us 17 keys, there are two more notes to name and map to the keyboard. Since there are two pairs of white keys which have no black key in between them on a standard 12-TET keyboard, namely between E and F and between B and C, it is convienient to add one black key between each of these pairs (these are colored gray in the figure). Each of these notes will have double names since they are not split in two: both names $E\sharp$ and $F\flat$ corresponds to the key between E and F, and names $B\sharp$ and $C\flat$ corresponds to the key between B and C.

This new keyboard layout with updated note names allows (quite remarkably!) for easy "translation" of notated 12-TET music to 19-TET. The white keys and their names, like previously stated, already maps to a diatonic C major / A minor scale. But how does a musician trying to play this new instrument know which black key to play of the "split" ones? For example, if a minor third interval from C is notated, should $D\sharp$ or $E\flat$ be played (these notes are in 12-TET mapped to the same key)? Luckily, since a minor third consists of five steps in 19-TET, we should play $E\flat$. This is desirable since the minor third in C minor has the name $E\flat$ in standard notation, and not $D\sharp$ (there are no sharps in the key of C minor). Even though we have not implemented an interactive keyboard, this is still very helpful in our software since we can, at least with diatonic music, parse notation written for 12-TET and play the 19-TET version of the musical piece by mapping the note names to the corresponding frequencies in 19-TET.

For atonal music, the translation is not as simple and more intricate

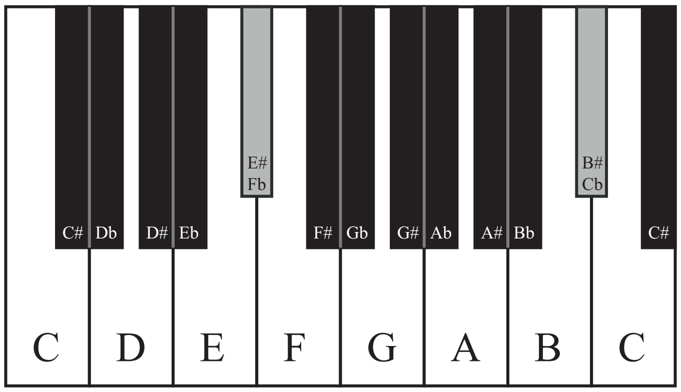

Figure 1: Yasser's design for a 19-TET claviature. Note that $C\sharp$ and $D\flat$ and the rest of the black keys in this figure are separate notes, whereas note pairs $E\sharp$ / $F\flat$ and $B\sharp$ / $C\flat$ are not.

rules of which notes to play when translating from 12-TET to 19-TET would have to be worked out. These approaches of translation need not necessarily be as consistent as when translating diatonic music; one could just as well choose the notes that seem to fit best given the musical context, style and surrounding notes.

3.4 Previous research

As stated in the introduction of this report, the 12-tone equal temperament has been the *de facto* scale in Western music for quite some time. Reasons for using an equally tempered N-tone scale have already been stated, but some discussion on the choice of N is necessary to further deepen the understanding of why certain tuning systems are more suitable than others. Yunik and Swift try to "determine, as objectively as possible, the most 'pleasing' and useful division of a musical octave into k equally spaced tones" [3]. They first define degree of consonance (c) of two tones. It is mathematically defined as

$$c = \frac{1}{\sqrt{mn}}$$

where m and n are integers and m/n corresponds to the frequency ratio of the two tones. The value of c depends on the distance between the lowest common harmonic of the two tones and the geometric mean of the two tones themselves. A long distance results in a low value of c and vice versa. It is shown, not surprisingly, that the degree of consonance is greatest for intervals with small integer ratios, such as 2:1 (octave), 3:2 (perfect fifth), 4:3 etc. The unison interval (1:1) has a c-value of 1, which is the highest possible value. In order to be able to objectively compare equal tempered scales, the authors calculate the *merit* values for different scales. This is done by first dividing the octave into an arbitrary number of N steps (in the article it is done for $1 \leq N \leq 60$), and then comparing each tone of the scale with 50 intervals that correspond to the 50 highest values of c. The comparison is made by checking if the tones of a scale are within 0.5% of any consonant ratio. The value 0.5% is derived from the mean frequency "errors" of the 12-TET approximations of the most important intervals (3:2, 4:3, 5:3 and 5:4). This deviation is considered what "musicians have endured all these years", and could be considered an acceptable frequency error which could be tolerated by most listeners accustomed to Western music. Thus, a "hit" is registered if a tone of a compared scale lies within 0.5% of any top-50 ratio frequency. The hits are weighted depending on the c-value of the approximated just interval. The sum of the hits are then divided by the number of tones in the scale, in order to "penalize" the scale for intervals in the scale that do not approximate any just interval. The result is defined as the merit of the scale. A function is plotted with the merit on the y-axis and the N-value on the x-axis (ranging from 0 to 60). A number of peaks are apparent, and the highest peak is that of $N = 19$ with a merit value of approximately 0.13. $N = 12$ is also clearly visible as a peak, although with a lower merit value of approximately 0.092. It is however the first value of N that gives a reasonably high merit value. The conclusion is drawn that both the 12-TET and 19-TET scales are reasonable choices; the 12-tone scale mainly because it is practical (containing a relatively small number of tones fitting the dimensions of the human hand, which is an important factor in the design of musical instruments), the 19-tone scale because it has a significantly better merit value that any other N between 2 and 60. This conclusion justifies our decision to further investigate the 19-TET scale and compare it to the 12-TET scale.

In another paper highly relevant to this report, Hartmann compares different equal temperament scales by examining how well they can represent the just intervals 3:2, 4:3, 5:4, 5:3 and different combinations of these [2]. The comparison is made by defining an error function $E(N)$, which depends on how close an N-tone equal temperament scale can approximate the listed just intervals. Even though this paper takes a more strictly mathematical approach than that of Yunik and Swift [3] (no assumptions of human tolerance levels for frequency errors are made), the results and conclusions are remarkably similar. The error function has clearly defined local minimums at both $N = 19$ and $N = 12$, further implying that these values for N are of special interest when constructing N-tone equal temperament scales.

4 Method

4.1 ChucKpond: A 12 and 19-TET software synthesizer

We have written a software synthesizer in the audio programming language ChucK that essentially renders textual descriptions of notes to music, using a notation for music similar to that of the music engraving program Lilypond, but our notation is much simpler than Lilypond's[1]. We call our program *ChucKpond* and the Lilypond-like syntax it utilizes was defined by us with the intention of using it solely for version 0.1.0 of the program. Each note is rendered into an audible tone using the sawtooth *unit generator* (ChucK terminology) *SawOsc*, a digital function generator which is programmed with a pre-calculated frequency corresponding to the note name, along with a generic ADSR envelope. This way, Lilypond-like text is effectively rendered into WAVE files of music.

We chose to build a sequencer of this variety instead of a synthesizer program that responds to, say, MIDI events from a USB-connected MIDI claviature. Granted, it would have been exciting to build an actual, physical Yasser-like 19 tone keyboard, but we have neither the time nor the resources. In addition, there are usability issues of a 19 tone keyboard that you would have to be a highly skilled piano player to be able to deal with. These factors makes it more appealing to create a sequencing mechanism that allows us to hear what 19-TET would sound like applied on well-known musical pieces, rather than creating a program intended for playing with in real-time.

4.2 ChucK as our choice of audio programming language

We investigated the audio programming languages ChucK, Csound, Nsound, Pure Data, sfront and SuperCollider. We established that we needed a language that allowed for low-level modelling of sounds, a means of effectively parsing text for processing and an option to output sound to WAVE or AIFF files for convenient demonstration; that is, a way of listening to the output of our program on any system with an audio player. We also took into consideration our collective experience of programming and computing environments, and lastly, our personal preferences.

[1]The Lilypond syntax has, amongst other things, commands for how notes are rendered onto paper or PDF. These features are, naturally, unnessecary for us.

For our purposes, we believe ChucK has five advantages. It is *sample accurate*, meaning that its algorithms are constructed in such a manner that given a specific source file, sound events consistently occur at the same moments (measured in samples) upon each render, as opposed to most threaded audio system in which the rendering or timing of sound events can be interfered by other processes in the operating system or concurrent sound events in the audio system itself.

Not only is ChucK sample accurate—you can also program sound events at the sample level. This allows for full control of audio rendering that could prove helpful when modelling acoustic instruments[2].

It comes with a set of example source files that demonstrates separate components in a way that is easily understandable for novices, whereas in general, the example source files of the other programs are rather made to impress and as such are somewhat complex to use as learning material.

We are already familiar with the imperative programming paradigm of ChucK. ChucK is object oriented (albeit in a primitive way) and the syntax is nearly identical to that of Java, a language we have used extensively in our previous studies. What distinguishes ChucK from Java is the multi-purpose *ChucK operator* which we explain briefly in section 4.4.

Lastly, it has a convenient command line interface.

Our reasons for excluding the other languages were mainly that they generally required programming skills we could not gather within reasonable time, that they were poorly documented and that some of them were controlled through rather non-usable GUI:s.

4.3 Lilypond and MIDI

Without careful consideration, it would seem that we could use the MIDI file format for our exploration of 19-TET. The problem is that MIDI files, as most other music related file formats, are only designed for conventional 12-TET. There is a way to circumvent this limitation by using the pitch bend feature of MIDI: you make a mapping of 19-TET notes and their frequency adjacency to 12-TET notes and then adjust pitches relatively[3]. This technique is used in a Wikipedia article on 19-TET for demonstration [10].

[2]This, however, was not one of our top priorities. For our purposes, which we expand upon in section 4.5, a simple sawtooth wave generator (ChucK's *SawOsc*) was sufficient, but admittedly somewhat perfunctory compared to the sound of a more vivid, acoustic instrument.

[3]If a tone has the frequency f in 12-TET and $f - x$ in 19-TET, you adjust the pitch x steps downwards using the MIDI pitch bend.

While this will work logically, there is a risk of confusion since you define one system (19-TET) relative to another system (12-TET) and both of these systems are—from a computer science point of view—arbitrary methods of selecting discrete frequency points.

Lilypond even has a MIDI rendering feature that can, with appropriate configuration, utilize this technique. However, we would not have full control over the actual frequencies since MIDI files are rendered differently on different systems, or to be specific, different MIDI players. We are dealing with very fine variations of frequency and if a tone rendered out of any music file we use in our project is off by even two or three cents with some frequency inaccurate MIDI player, it will clearly disrupt one of the main topics of our work.

A solution would be to write our very own MIDI player, in which we would always know what frequencies are played. However, it is more difficult to parse and interpret MIDI files[4] compared to parsing our own simplified variant of the Lilypond format.

4.4 Implementation

4.4.1 A Lilypond-like notation for scores

As stated earlier, ChucKpond parses Lilypond-like text files. The text files contains brace bracket separated blocks of voices and in each of these blocks there are space separated note tokens defined as

<note name><octave selection><note duration>

A specification of the token definition follows[5].

Note name: in lower case letters. Sharp and flat notes are suffixed with -*is* and -*es*, respectively.
Octave selection: optional. Specified with one apostrophe for each increase of octave, *or* one comma for each decrease of octave. You cannot have both apostrophes and commas. The starting point for octave selection, which will also be the note's octave if you leave out this parameter altogether, is the fourth octave (in which notes usually are suffixed with the number 3) on a

[4]Mostly since it is a binary format, i.e. you cannot edit and view MIDI files with a common text editor—you need to use a domain specific program such as Sibelius.

[5]Note that as for version 0.1.0 of ChucKpond, you cannot have double dotted notes, neither can you have ties between notes. However, these features were not needed for the songs in our listener tests.

piano.
Note duration: optional. The format is <xy>, where $x = 2^n$ ($0 \leq n \leq 5, n \in \mathbb{Z}$) is the note duration's denominator and y is an optional dot for dotted note lengths. If you leave out this parameter, the duration of the note will be the same as that of the preceding note.

Some examples of note tokens are `dis` which is a $D\sharp 3$ with the duration of the preceding note, `e'` which is an $E4$, also with the duration of the preceding note, `f,16` which is an $F2$ sixteenth note, and `ges''4.` which is a $G\flat 5$ dotted quarter note.

4.4.2 Distinguishing features of ChucK and how they are utilized

ChucK is an interpreted language for programming audio and graphics [6]. In ChucK, there are two major concurrent processes: the conventional program logic flow consisting of common control structures such as *if / else*, *while* and *for*, and the audio flow controlled by the ChucK operator which is denoted as => or @=>, depending on the context [7]. As such, the language lends itself to programming audio. The semantic is that you alternate between computing variables and "allowing time to pass" using the keyword *now* [8]. We have used this paradigm in ChucKpond according to the following pseudo-code:

```
while (voice.hasTokens())
  token = parseToken()

  oscillator.setFrequency(token.getFrequency())
  envelope.setRelease(token.getDuration())

  /* prepare for triggering the envelope as if pressing down
      a key on a claviature */
  envelope.keyDown()

  /* let time pass for one sample. this will trigger the
      envelope */
  1::samp => now

  /* release the key */
  envelope.keyUp()

  /* let the duration of the note pass, minus the one sample
      we needed to trigger the envelope */
  token.getDuration() - 1::samp => now
```

It turned out during development that a *substring* function for strings was absent from the API of ChucK, which we needed for the parsing functions. We implemented this tool in the source code of ChucK 1.2.1.3 itself, which is written in C++, by writing a "wrapper" for the underlying `substr` method of C++.

The code is executed in separate *shreds* by using the keyword *spork* (these are analogous to *thread* and *fork*, respectively), like so:

```
for each voice
  spork ~ playVoice()
```

A *shred* is essentially a thread that is always computed in the same manner (inside ChucK's virtual machine) for each time you run the ChucK source file it belongs to. This means that shreds cannot, by priority schemes or otherwise, be interfered by other shreds or processes in the underlying operating system, thus ensuring the aforementioned sample accuracy (see section 4.2).

4.4.3 ChucKpond prerequisites and usage

To run ChucKpond 0.1.0 which is supplied in section B.2, you need[6]

- Mac OS X 10.6.8 or later, or
- Windows 7 with
  - Cygwin 1.7.11-1 or later, or
  - complementary Cygwin DLL files supplied in section B.1
- our extended version of ChucK (*chuck-aw*), also supplied in section B.1

ChucKpond is then run from *cmd* or a Unix terminal. Assuming that the binary of the extended ChucK (*chuck-aw* or *chuck-aw.exe*) is in your environment path, and that you are located in the folder with the source code files (suffixed *.ck*) of ChucKpond, it is invoked like so:

computer:∼/chuckpond-0.1.0$ chuck-aw main:<tuning of choice>:<file name>:<WAVE file name>

[6]You can also run ChucKpond on GNU/Linux. However, you will have to build our extended version of ChucK yourself from source, as we have not pre-built any binaries for GNU/Linux. See section B.1 for instructions.

where <*tuning of choice*> is 12 or 19, <*file name*> is a file with Lilypond-like text and <*WAVE file name*> (optional) is used if you want to output a WAVE file as you listen to the score.

An example of playing to "Ode to Joy" in 19-TET and simultaneously creating a WAVE file could look like so: chuck-aw main:19:ode-to-joy:"ode-19.wav". Section B.3 consists of the source file *ode-to-joy* and the corresponding WAVE file *ode-19.wav*, which was also used in the listener tests in section 4.5.

4.5 Listener tests

In order to compare the 19-TET scale to the 12-TET scale and how they are perceived by humans, listener tests were performed. The purpose of the tests was to, as objectively as possible, see how humans perceive intervals and musical pieces in 19-TET and 12-TET—to what extent do they differ, and what intervals and musical pieces played in the tuning systems are pleasant sounding to the human ear? Both musically trained subjects and casual music listeners (non-musicians) were used as test subjects. A listening test session with a test subject consisted of eleven different test cases. Each test case consisted of two parts *a* and *b*, which corresponded to playing an interval or a musical piece with frequencies derived from 19-TET or 12-TET. Which scale was used as the *a* part varied throughout the test session to prevent the effect of "adjusting" the listener's ears to the same scale in every test case. A random binary number generator was used to decide which scale should be used as *a* part in each test case.

The intervals and musical pieces had been rendered with ChucKpond beforehand into WAVE files. This means that there were 22 sound clips played in one test session. After each sound clip had been played, the listener was asked to answer how the sound clip had been perceived—was the interval or musical piece pleasant to listen to? This was done by marking a point on a horizontal line. A marking on the right part of the line corresponded to "pleasant sounding", while a marking on the left part corresponded to "unpleasant sounding". After the listener had heard both the 19-TET version and the 12-TET version of a test case, the listener was asked to mark the perceived difference between the two sound clips in a similar manner. No information about the scales used was given to the listener beforehand. Of note is that we did not give the listeners any more information than this prior to playing the sound clips, as the test was designed for learning how listeners subjectively perceived and experienced 12 and 19-TET—we did not want to imply that there ever was a "scientifically correct" scale (since there is no such scale), or in other words, that one scale is better than the other, let alone that they should hear a difference in the first place. The test cases were carried out in the following order: first, simple intervals were played harmonically. The intervals chosen were

- Perfect fifth
- Perfect fourth
- Major third
- Minor third

The reason why we chose these particular intervals is that they are considered important intervals in music and they all have a high degree of consonance in just tuning (see section 3.1).

Second, two simple tunes were played in one voice (no harmonies). The tunes chosen were

- Twinkle, Twinkle, Little Star in C major
- "Ode to Joy" (Beethoven's Ninth Symphony, final movement) in G major [4]

Third, the four intervals were played again but this time melodically. Last, "Ode to Joy" was played in a four-part harmony arrangement, which is also available for download in section B.3.

All tests were recorded using A0 (the lowest key on the piano with the frequency 440 / 16 = 27.5 Hz) as the reference tone, for both 12-TET and 19-TET. For the harmonic and melodic intervals we used A4 (440 Hz) as the lower tone frequency since we wanted the intervals to start at the same frequency for both 12-TET and 19-TET; reason being that it makes for easier comparison of the scales when one of the interval notes is in a fixed frequency position.

5 Results

Table 2 gives the results from the listener tests in section 4.5. The values are given as a percentage of where on each horizontal line the test subjects drew a point, and as such are degrees of pleasantness for the 12 and 19-TET versions of each test item (0% being "unpleasant sounding" and 100% being "pleasant sounding"), and perceived difference between the two versions (0% being *equal* and 100% being *fully separate*). There were seven test subjects, and the *Subject #* column contains indices of these, i.e. each row of values were given by a particular person.

	Subject #	**Harmonic intervals**				**Melodic intervals**				**Songs**		
		Minor third	**Major third**	**Perfect fourth**	**Perfect fifth**	**Minor third**	**Major third**	**Perfect fourth**	**Perfect fifth**	**Twinkle, ...**	**Ode..., one voice**	**Ode..., four voices**
12-TET	1	69	60	49	52	48	52	46	48	54	69	81
	2	30	84	75	99	27	80	76	76	14	80	98
	3	24	85	73	94	27	99	59	3	94	92	95
	4	28	63	58	55	95	92	93	95	84	89	96
	5	32	64	46	48	49	49	46	49	64	50	69
	6	64	84	70	75	59	59	63	81	69	85	80
	7	64	85	66	79	52	47	36	29	31	36	77
Average		44	75	62	72	51	68	60	54	59	71	85
Median		32	84	66	75	49	59	59	49	64	80	81
19-TET	1	67	62	50	50	47	53	49	47	36	47	69
	2	35	21	53	93	52	48	79	75	7	83	48
	3	87	23	19	57	70	3	43	98	30	1	61
	4	28	55	55	55	95	71	93	95	84	89	96
	5	37	59	57	47	42	45	57	31	69	39	54
	6	60	79	71	75	39	40	74	59	63	68	65
	7	34	85	65	78	53	47	38	31	32	68	96
Average		50	55	53	65	57	44	62	62	46	56	70
Median		37	59	55	57	52	47	57	59	36	68	65
Difference	1	5	19	11	6	3	18	14	7	11	47	24
	2	62	97	99	84	35	25	3	3	8	5	60
	3	95	95	72	64	96	92	60	60	71	97	99
	4	3	9	9	7	1	18	1	1	1	3	1
	5	18	19	24	3	47	27	56	53	12	18	53
	6	32	49	20	15	74	50	43	42	26	60	38
	7	55	1	4	1	2	2	2	2	2	33	21
Average		38	41	34	26	37	33	26	24	19	37	42
Median		32	19	20	7	35	25	14	7	11	33	38

Table 2: Results from the listener tests.

General averages and medians of all interval and song values for 12-TET, 19-TET and difference separately follows. For instance, the first number in the following list is an average of all percentage values spanning from *12-TET, subject #1, minor third* to *12-TET, subject #7, Ode..., four voices.*

Average of 12-TET: 64%.

Median of 12-TET: 64%.

Average of 19-TET: 56%.

Median of 19-TET: 55%.

Average of difference: 33%.

Median of difference: 20%.

6 Discussion

Both the harmonic and melodic version of the minor third were preferred in 19-TET. This complies with our initial guess based on the fact that the 19-TET approximation of a minor third has a much smaller deviation from the just ratio of 6:5 compared to the 12-TET approximation. The test subjects also generally heard a relatively big difference between the 12-TET and 19-TET versions of the interval. The major third, on the other hand, was preferred in 12-TET even though the 12-TET approximation of the just major third has a larger error than 19-TET. The reason for this could be that the 12-TET major third is slightly sharp and the 19-TET major third is slightly flat with respect to the just major third. Since our test subjects were used to Western music and the 12-TET scale, they might have found the 19-TET major third odd sounding because of its flatness, even though the absolute value of the error is smaller than that of 12-TET.

The harmonic perfect fourths and fifths were generally preferred in 12-TET, which was to be expected since they do not deviate as much from the corresponding just intervals in this tuning as in 19-TET. It is worth noting, however, that the median value of the perceived difference between a 12-TET and a 19-TET harmonic perfect fifth was only 7%. The results from the melodic fourths and fifths are harder to interpret. The melodic perfect fourth seems to have been perceived as almost equally pleasant in both tunings, while the perfect fifth seems to have been preferred in 19-TET by a small margin. The median value of the perceived difference of a melodic perfect fifth in 19-TET and 12-TET is only 7% (just like the harmonic version of the interval) which suggests that the preference should not be weighted too heavily.

The songs were all generally preferred in 12-TET, and the greatest difference was perceived in the song with four voices ("Ode to Joy"). It was probably easier for the test subjects to differentiate between the different tunings when hearing full chords and four-part harmonies, compared to when hearing simple two-note intervals.

One important factor to consider when trying to interpret the results from the listener tests is of course the fact that all test subjects were used to the 12-TET scale and very unfamiliar with the 19-TET scale. Intervals and musical pieces written for 12-TET might sound odd and even out of tune in 19-TET simply because 12-TET intervals are expected, even though certain intervals in 19-TET are less dissonant from a physical-acoustic point of view. These expectations could very well be the reason why 12-TET got a higher overall average "score" than 19-TET.

Another factor that might have biased the test subjects is the order in which the sound clips were played. One test subject stated that he might have found a musical piece played right after a series of simple intervals pleasant, only because it provided a change from the dull and rather non-musical intervals. It is also hard to say to what degree the tests with simple intervals represent the musical qualities of said intervals; it is quite a big difference between hearing a simple interval played alone for a short period of time and hearing the same interval in a more complex musical context with several instruments of different timbres playing together.

The test subjects might also have been affected by the synthesizer's sound characteristics when judging the pleasantness of a song or interval. As stated in section 4, our program used a sawtooth wave generator to generate sound. Since that shape of sound is generally associated with certain genres of electronic music, the test subjects' musical genre preferences might have affected their judgement in addition to their perception of intervals and pitch.

On a related note, our selection of songs might have been unappealing to some of the listeners, in which case they will not consider any subtle differences in the two versions of each song: the result is that the listener perceives both versions as equally unpleasant, and that there is no difference between them. The opposite case is that if a listener does appreciate a certain song, they are more alert to these differences. It follows that a wider variety of genres for the songs in the listener tests might have allowed us to even more accurately understand the listeners' degree of appreciation for the 19-TET scale.

We would definitely have benefitted from having more test subjects. It is difficult to say, and also outside of the scope of the project, how many test subjects we would have needed for the test results to be statistically relevant, and to whom—in the largest sense, our subject is relevant to everyone in general, and the Western world in particular. The test results are also, naturally, affected by the subjects' own musical abilities and sense of pitch, which is yet another reason for having more test subjects.

7 Conclusion

The results from the listener tests suggest that the test subjects generally preferred music and intervals from the 12-TET scale. Both the general average and median values in section 5 support this conclusion. There were however some interesting exceptions from this rule. The minor third was generally regarded as more pleasant sounding in 19-TET than in 12-TET, even though it was by a small margin. A bigger difference between the scales was perceived by listeners when they listened to songs and not simple intervals.

With our software, translation of musical pieces from 12-TET to 19-TET is straightforward and notation originally intended for 12-TET instruments may be used to play the same notated music with frequencies derived from the 19-TET scale.

The translated pieces of music turned out to be satisfactory for the listeners, but not quite as satisfactory as the original 12-TET versions—it goes to show that we are creatures of habit, in the sense that in our formative years we (as Westeners) are exposed to mostly 12-TET when listening to music and become accustomed to it. In addition, we are conditioned to perceive certain kinds of arts and music (and within that, scales) as beautiful; you could say that we are "taught" what is pleasant.

8 References

8.1 Printed material

[1] **Douthett, J., & Krantz, R. J.** *Construction and interpretation of equal-tempered scales using frequency ratios, maximally even sets, and P-cycles.* 2000.

[2] **Hartmann, G. C.** *A numerical exercise in musical scales.* 1986.

[3] **Swift, G. W., & Yunik, M.** *Tempered Music Scales for Sound Synthesis.* 1980.

8.2 WWW

[4] **Beethoven, L. V.** *Ode to Joy.* http://www.mutopiaproject.org/ftp/BeethovenLv/ode/ode.ly, c. 1800. Typeset by Peter Chubb (Viewed on 2012-04-11).

[5] **Keislar D.** *History and principles of microtonal keyboard design.* https://ccrma.stanford.edu/files/papers/stanm45.pdf, 1988. (Viewed on 2012-04-11).

[6] **The ChucK team**. *ChucK - [Developer's Guide].* http://chuck.cs.princeton.edu/doc/develop/. (Viewed on 2012-04-04).

[7] **The ChucK team**. *ChucK - [Language Specification : Operators & Operations].* http://chuck.cs.princeton.edu/doc/language/oper.html. (Viewed on 2012-04-03).

[8] **The ChucK team**. *The ChucK tutorial.* http://chuck.cs.princeton.edu/doc/learn/tutorial.html. (Viewed on 2012-03-19).

[9] **Wikipedia**. *Beat (acoustics).* http://en.wikipedia.org/wiki/Beat_%28acoustics%29. (Viewed on 2012-04-11).

[10] **Wikipedia**. *File:19-et diatonic scale on C.mid.* http://en.wikipedia.org/wiki/File:19-et_diatonic_scale_on_C.mid. (Viewed on 2012-04-03).

[11] **Wikipedia**. *Octave.* http://en.wikipedia.org/wiki/Octave. (Viewed on 2012-04-10).

[12] **Wikipedia**. *Pythagorean tuning.* http://en.wikipedia.org/wiki/Pythagorean_tuning. (Viewed on 2012-04-10).

A Glossary

ADSR Attack, Decay, Sustain, Release. A common paradigm for consecutive increasing and decreasing of volume (also known as *envelope*) in synthesizer sounds. Commonly used in the design of electronic models of instruments.

atonal Lacking key and/or tonal center.

beat An interference between two sounds of slightly different frequencies, perceived as periodic variations in volume whose rate is the difference between the two frequencies [9].

Cygwin Linux-like environment for Windows.

diatonic Derived from the major scale, belonging to one key.

DLL file Dynamic Link Library: a system file containing functions that can be shared between programs.

dotted (note) Musical notation indicating an increase of note length by 50%.

duration (note) Time length of a played note.

interval A combination of two notes, or a ratio of their frequencies.

key (music) A set of notes with a root note as tonal center.

Lilypond A computer program for engraving of sheet music.

MIDI Musical Instrument Digital Interface: electronic musical instrument industry specification for digital communication between audio devices.

note (music) A mapping of a name or symbol to a tone.

octave Interval with frequency ratio 2:1. The human ear hears notes an octave apart as being "the same" because of the numerous common harmonics of the two notes.

sample (audio) A discretized point (of a signal) derived from, or simulating, a continuous audio signal. The smallest unit of sound data in an audio file.

Sibelius A music notation software.

thread (computing) One of many parallell instruction sequences within a computer program's process.

tie (music) Musical notation indicating that the duration of one note should be prolonged with the duration of the note it is tied to.

tone (music) A frequency in a musical context.

B Online material

B.1 Our extended version of ChucK

Executable for Mac OS X 10.6.8 or later.
URL: `http://www.wifstrand.se/albert/dkand12/chuck-aw-120328.gz`

Archive with executable for Windows 7 along with complementary Cygwin DLL files. You can also run the executable (*chuck-aw.exe*) from inside the Cygwin terminal, in which case you will not need the DLL files supplied in this archive.
URL: `http://www.wifstrand.se/albert/dkand12/chuck-aw-with-cygwin-dll-files-120328.zip`

C++ source code for our modifications in ChucK 1.2.1.3. The following four source code files (which were already in the ChucK source tree) were modified; our modifications can be traced by searching for C++ comments containing the string `awifstrand`. For simplicity's sake, we supply this set of source code files in its entirety. In order to get a working copy of our ChucK branch, replace the source files in the official release of ChucK 1.2.1.3 with these.
`http://www.wifstrand.se/albert/dkand12/chuck_lang.cpp`
`http://www.wifstrand.se/albert/dkand12/chuck_lang.h`
`http://www.wifstrand.se/albert/dkand12/util_string.cpp`
`http://www.wifstrand.se/albert/dkand12/util_string.h`

B.2 ChucKpond 0.1.0, archived

`http://www.wifstrand.se/albert/dkand12/chuckpond-0.1.0.tgz`

B.3 A four-part harmony arrangement of "Ode to Joy" generated by ChucKpond

`http://www.wifstrand.se/albert/dkand12/ode-to-joy`

`http://www.wifstrand.se/albert/dkand12/ode-12.wav` http://www.wifstrand.se/albert/dkand12/ode-19.wav

C Source code for ChucKpond 0.1.0

C.1 chuckpond_main.ck

```
// arg 0: scale
// arg 1: name of .ly file
// arg 2: wave file name
// arg 3: if set to 1, use debug prints

Std.atoi(me.arg(3)) => int LOCAL_DEBUG;   // atoi: string to
    int

// render to wav
me.arg(2) @=> string wav_filename;
Gain g;
WvOut w;
if (wav_filename.length() > 0) {
  dac => g => w => blackhole;
  wav_filename => w.wavFilename;
  .97 => g.gain;
}

// event for voices
Event finish;

// associative array of voice objects
voice voices[0];

// can we use FileIO w. StringTokenizer?? YES
FileIO file_io;

// tuning object
tuning main_tuning;

/* INITIALIZE THINGS BEFORE READING FROM FILE */

// open file
file_io.open(me.arg(1), FileIO.READ);
if(!file_io.good()) {
  cherr <= "can't␣open␣file" <= IO.newline( );    // IO ??
  me.exit();
}

// choose scale (i.e., fill freqs array with freqs).
// C0 is the reference freq., dervied from the lowest key on a piano (A0, 27.5 Hz)
<<< "using", me.arg(0), "TET" >>>;
main_tuning.init(me.arg(0));

// init. parser/player
parse_and_play main_parser_player;
if (LOCAL_DEBUG) <<< "init.␣main_parser_player" >>>;
main_parser_player.set_event(finish);
main_parser_player.set_freqs(main_tuning.get_freqs());  // TODO : objects passed around haphazardly. fix

if (LOCAL_DEBUG) {  // can we look at these now?
  <<< "main_parser_player,␣freqs:", main_parser_player.get_freqs() >>>;
  //<<< "main_parser_player, freqs without the get function:", main_parser_player.freqs >>>;
  <<< "main_parser_player,␣event:", main_parser_player.get_event() >>>;
}

/* FILE READING LOOP */

0 => int do_parse;
0 => int voice_index; // counter for the ass. array of voices

while(file_io.more()) {
  file_io.readLine() @=> string line;

  if (line.length() == 0) { // skip empty lines
    continue;
  }

  if (do_parse) {   // manage parsable line
    if (main_parser_player.index_of(line, "}") != -1) { // end of "bracket block"
      0 => do_parse;
      voice_index++;
      continue; // go immediately to the next line
    }

    //if (LOCAL_DEBUG) <<< "parsable line:", line >>>;

    // found a line
    //voices["voice" + voice_index].get_fifo().push(line); // obsolete

    // add the tokens from this line
    StringTokenizer tok;  // TODO: split not only on whitespace but also bar signs etc.?
    tok.set(line);

    while(tok.more()) {
      tok.next() @=> string token;

      /* FORMAT: at least 1 char of letters a..z    (name)
       * possibly 1 or more chars of ' or ,      (octave)
       * possibly duration, that is
       *    possibly integer
       *    possibly 1 dot  (TODO: 2 dots)
       *    possibly ~
       */

      if (main_parser_player.is_parsable(token)) {  // skip sheet formatting things
        voices["voice" + voice_index].get_fifo().push(token);
      }
    }
  } else {  // outside of "bracket blocks"
    // look for { at end of line
    if (line.length() > 0 && line.substring(line.length() - 1, 1) == "{") {   // TODO: remove line.length() > 0 check
      if (LOCAL_DEBUG) <<< "found␣{" >>>;

      1 => do_parse;
    }

    /* PUT VOICE IN ASSOCIATIVE ARRAY AND SET META-DATA */
    new voice @=> voices["voice" + voice_index];

    // look for transpose command in that line and set the field in the voice object
    // TODO: proper transpose. for now, always use 2
    if (main_parser_player.index_of(line, "\\transpose") != -1) {
      if (LOCAL_DEBUG) <<< "found␣\\transpose" >>>;
      voices["voice" + voice_index].set_transpose(2);
    }
  }
}

/* SPORK */

// put each of the voices in a separate thread ("spork" in ChucK terminology)
for (0 => int i; i < voice_index; i++) {
  <<< "playing/sporking␣voice␣#", i>>>;
  spork ~ main_parser_player.play_voice(voices["voice" + i]);
}

// allow time to pass
finish => now;
```

C.2 main.ck

```
/*
 * ChucKpond 0.1.0
 *
 * by Albert Wifstrand (albert@wifstrand.se) and Andreas
     Lindström (andlinds@kth.se), 2012
 *
 * Usage:
 *    chuck-aw main:<tuning of choice>:<file name>:<WAVE file
      name>
 *
 *    <tuning of choice> (arg 0): 12 or 19
 *    <file name> (arg 1): sheet music file with Lilypond-like
      syntax
 *    <WAVE file name> (arg 2): if this arg is used, a WAVE
      file with the chosen name will be created
 */

// TODO: rests
// note bindings
// code clean-up (comments)
// error handling for substring startpos > str.length (if
    necessary)
// proper definition of args in all files
// refactoring (to Java-style), i.e. for
//    (almost) every variable from the this_is_a_variable style
     to the thisIsAVariable style
//    classes: ThisIsAClass (capital letter in beginning of
    class name), instances of classes: thisIsAnInstance
// render straight to hdd (instead of "chucking" with rec.ck)
// do away with (strictly "return or assignment" mannered)
    setters and getters since everything is public anyway
// render to wave "directly", without passing through dac

0 => int DEBUG;

// add .ck files in the right order
Machine.add("string_fifo.ck");
Machine.add("voice.ck");
Machine.add("tuning.ck:" + DEBUG);
Machine.add("parse_and_play.ck:" + DEBUG);
Machine.add("chuckpond_main.ck:" + me.arg(0) + ":" + me.arg(1)
    + ":" + me.arg(2) + ":" + DEBUG);
```

C.3 parse_and_play.ck

```
// TODO: program crashes if the first note of an .ly-like file
    doesn't have duration value. fix
// TODO (possibly!): separation of parsing and playing

// arg 0 is debug
Std.atoi(me.arg(0)) => int LOCAL_DEBUG;   // atoi: string to
    int

public class parse_and_play {
  Event event;
  float freqs[];

  fun void set_event(Event e) { e @=> event; }
  fun void set_freqs(float f[]) { f @=> freqs; }

  fun Event get_event() { return event; }
  fun float[] get_freqs() { return freqs; }

  // return pos. if found, -1 if not found
  fun int index_of(string in, string search_str) {
    for (0 => int i; i <= (in.length() - search_str.length());
        i++) {
      int j;

      for(0 => j; j < search_str.length(); j++) {
        if(search_str.substring(j, 1) != in.substring(i + j, 1)
            ) {
          break;
        }
      }

      if(j == search_str.length()) {
        return i;
      }
    }

    return -1;
  }

  // somewhat ad hoc; checks if the first char of note is a
      letter in [a..z]
  // if it is, the token SHOULD be a note
  // TODO: fix.
  fun int is_parsable(string note) {
    note.substring(0, 1) => string c;
    return c >= "a" && c <= "z";
  }

  fun void play_voice(voice in_voice) {
    dur player_dur;

    if (LOCAL_DEBUG) <<< "in_voice:", in_voice, ",␣transpose:", in_voice.get_transpose() >>>;

    in_voice.get_envelope() @=> ADSR envelope;
    in_voice.get_fifo() @=> string_fifo voice_fifo;
    in_voice.get_oscillator() @=> SawOsc oscillator;

    if (LOCAL_DEBUG) {
      <<< "envelope:", envelope, ",␣voice_fifo:", voice_fifo, ",␣oscillator:", oscillator >>>;
      <<< "q␣empty?␣", voice_fifo.is_empty() >>>;
    }

    while(!voice_fifo.is_empty()) {
      // TODO: wrappers for parse functions (with or without var. i)
      // > 0 means that there's an index
      // -1 means that the function needs to find it's chars on its own

      int i;  // parser index for note string
      voice_fifo.pop() @=> string note;
      parsed_tone current_tone;
      current_tone.init(freqs);

      if (LOCAL_DEBUG) <<< "play_voice,␣before␣parsing.␣note␣=", note >>>;

      /* PARSE START. uses parse_note_name, parse_octave, parse_duration */

      // I. note name (always present)
      current_tone.parse_note_name(note) => i;

      // II. octave MULTIPLIER (possible chars: ' , nothing)
      current_tone.parse_octave(note, i) => i;

      // TODO: some kind of loop here for managing ties
      // III. duration (always comes after octave)
      current_tone.parse_duration(voice_fifo, player_dur, note, i);

      // if the current duration differs from the previous duration, change the actual duration var. that the ChucK-specific code uses
      if (player_dur != current_tone.duration) {
        current_tone.duration => player_dur;
      }

      // calculate actual freq.
      current_tone.set_freq(in_voice.transpose);

      // TODO: check if there's anything left to parse (??)

      // note info print
      <<< "---\nvoice", in_voice >>>;   // TODO: print voice index
      <<< "note string:", note >>>;
      <<< "note duration:", player_dur, "samples" >>>;
      <<< "note freq:", current_tone.freq, "Hz" >>>;

      /* PARSE END */

      /* PLAY */

      current_tone.freq => oscillator.freq;

      envelope.set(0::ms, 0::ms, .5, 2 * player_dur); // why .5 ?
      envelope.keyOn();
      1::samp => now;
      envelope.keyOff();
      player_dur - 1::samp => now;
    }

    event.broadcast();    // broadcast when the entire voice has been parsed
  }
}

class parsed_tone {
  float freqs[];  // from parse_and_play

  100 => int tempo;     // TODO: tempo shouldn't be here

  string name;    // fetched from note_names in tuning.ck
  int octave;
  float freq;
  dur duration;   // if we find ties, they are compounded into this field (i.e. tokens c4~ c will end up in the same parsed_tone object)
```

```

    /* CONSTRUCTOR START */

    dur durations[0]; // associative array of possible duration values
    // init. durations
    240000::ms / ( 1 * tempo )  => durations["1"];          // whole
    240000::ms / ( 2 * tempo )  => durations["2"];          // half
    240000::ms / ( 4 * tempo )  => durations["4"];          // quarter
    240000::ms / ( 8 * tempo )  => durations["8"];          // eighth
    240000::ms / ( 16 * tempo )  => durations["16"];          // sixteenth
    240000::ms / ( 32 * tempo )  => durations["32"];          // 32 nd

    (durations["2"] + durations["4"]) => durations["2."];      // dotted half
    (durations["4"] + durations["8"]) => durations["4."];      // dotted quarter
    (durations["8"] + durations["16"]) => durations["8."];      // dotted eight
    (durations["16"] + durations["32"]) => durations["16."];      // dotted sixteenth

    /*
    240000::ms / ( 3 * tempo )  => durations["half_trip"];      // half triplet

    240000::ms / ( 5 * tempo )  => durations["quarter_quint"];      // quarter quintuplet
    240000::ms / ( 6 * tempo )  => durations["quarter_trip"];      // quarter triplet
    240000::ms / ( 7 * tempo )  => durations["quarter_sept"];      // quarter septuplet

    240000::ms / ( 10 * tempo )  => durations["eighth_quint"];      // eighth quintuplet
    240000::ms / ( 12 * tempo )  => durations["eigth_trip"];      // eighth triplet
    240000::ms / ( 16 * tempo )  => durations["sixteenth"];      // sixteenth
    */

    /* CONSTRUCTOR END */

    fun void init(float f[]) {
```

```
    f @=> freqs;
  }

  fun void set_freq(int transpose) {
    transpose * octave * freqs[name] => freq;
  }

  fun int parse_note_name(string note) {
    0 => int i;
    string c;

    if (LOCAL_DEBUG) {
      <<< "parse_note_name,␣note␣string:", note, "note␣string␣length:", note.length() >>>;
    }

    // only allow (english) chars a - z, small letters
    do {
      // substring usage: the_string.substring(start_pos_of_wanted_substring, length_from_start_pos)
      note.substring(i, 1) @=> c;   // pick a single char
      i++;
    } while (c >= "a" && c <= "z"); // parse single chars until we arrive at something that isn't in [a..z]

    if (LOCAL_DEBUG) {
      <<< "parse_note_name:", note.substring(0, i - 1) >>>;
    }

    note.substring(0, i - 1) @=> name;

    return name.length();
  }

  fun int parse_octave(string note, int parser_index) {
    if (LOCAL_DEBUG) <<< "parse_octave.␣note␣string:", note >>>;

    4 => int ret_octave;  // default octave multiplier
    parser_index => int i;

    string pre_c, c;
    while (true) {
      note.substring(i, 1) @=> c;

      if (c == "'") {      // increase
        ret_octave << 1 => ret_octave;
      } else if (c == ",") {  // decrease
        ret_octave >> 1 => ret_octave;
      } else {
        if (LOCAL_DEBUG) <<< "parse_octave.␣break" >>>;
        break;
      }

      if (i > parser_index && c != pre_c) {
        <<< "ly␣syntax␣error␣(octaves)" >>>;
        me.exit();
      }

      i++;
      c => pre_c;
    }

    ret_octave => octave;

    if (LOCAL_DEBUG) {
      <<< "parse_octave.␣octave␣=", octave, ",␣name␣=", name >>>;
    }

    return parser_index + Math.abs(Math.log2(octave) $ int - 2);
  }

  // call without any start pos.
  fun void parse_duration(string_fifo voice_fifo, dur player_dur, string note) {
    parse_duration(voice_fifo, player_dur, note, -1);
  }

  // TODO: entire function is logically correct but messed up; fix
  // TODO: allow for dotted notes without putting an integer value in front of the dot
  // TODO: double dotted. we can only parse single dots atm
  fun void parse_duration(string_fifo voice_fifo, dur player_dur, string note, int parser_index) {
    if (parser_index < 0) {   // parser_index is -1, find the first integer
      if (LOCAL_DEBUG) <<< "␣␣␣␣parser_index␣<␣0" >>>;

      string c_init;

      do {
        parser_index++;
        note.substring(parser_index, 1) @=> c_init;   // pick a single char
      } while (Std.atoi(c_init) == 0 && parser_index < note.length());
    }

    parser_index => int i;
    if (LOCAL_DEBUG) <<< "␣␣␣␣parse_duration.␣i␣=", i >>>;

    string c;

    if (LOCAL_DEBUG) <<< "parse_duration,␣note␣string:", note, "note␣string␣length:", note.length() >>>;

    // TODO: inappropriate to evaluate this in here
    if (i == note.length()) { // previously: note.length() == 0
      player_dur => duration;
      return;
    } else if (i > note.length()) {
      cherr <= "parse_duration,␣i␣out␣of␣bounds" <= IO.newline();
      me.exit();
    }

    do {
      note.substring(i, 1) @=> c;   // pick a single char
      i++;
    } while (c >= "0" && c <= "9");

    // is there a dot at pos. i - 1 ?
    if (note.substring(i - 1, 1) != ".") {
      i--;
    }

    // tie
    /*
    // is there a tilde at position i?
    if (note.substring(i,1) == "~"){
      durations[note.substring(0, i)] + durations[tok.next]
    }
    */

    if (LOCAL_DEBUG) <<< "␣␣␣␣at␣end␣of␣parse_duration,␣parser_index:", parser_index, ",␣i:", i >>>;

    durations[note.substring(parser_index, i)] => duration;
  }
}
```

C.4 string_fifo.ck

```
// TODO: try to make a generic queue (Object q) (or perhaps an
    ArrayList kind of thing?) so that it becomes a fully
    abstract data structure (then you can insert whatever you
    want, whereas now it (obviously) only works for strings)

// TODO: re-arrange this class so that it uses an associative
    array for q (currently it's fixed size, with conventional
    integer-index lookup)

// TODO: warning/error when trying too insert elements beyond
    the capacity of q

// FIFO queue for strings
public class string_fifo {
  int num_elements, first, last;
  string q[500];  // arbitrary capacity. # of tokens in a voice

  // pre-constructor
  empty();

  fun int is_empty() {
    return num_elements == 0;
  }

  fun void empty() {
    0 => num_elements;
    -1 => first;
    -1 => last;
  }

  fun void push(string element) {
    if (is_empty()) {
      element @=> q[0];
      0 => first;
      0 => last;
    } else {
      if (last == (q.cap() - 1)) {
        0 => last;
      } else {
        last++;
      }

      element @=> q[last];
    }

    num_elements++;
  }

  fun string pop() {
    if (is_empty()) {
      cherr <= "list␣is␣empty";
        me.exit();
    }

    q[first] @=> string element;
    if (first == (q.cap() - 1)) {
      0 => first;
    } else {
      first++;
    }

    num_elements--;

    return element;
  }

  fun string peek() {
    return q[first];
  }

  fun int size() {
    return num_elements;
  }
}
```

C.5 tuning.ck

```
// TODO: the whole k_select thing is obsolete, fix

// arg 0 is debug
Std.atoi(me.arg(0)) => int LOCAL_DEBUG;   // atoi: string to
    int

public class tuning {
  // tuning selector
  int k_select;

  // global reference freq.
  float ref_freq;

  // frequency (associative) array mapped to names in
      note_names
  float freqs[];

  /*
  Lilypond syntax (can also be defined for actual .ly files)
      array. is: sharp, es: flat
```

```
for 12-tone: c|bis cis|des d dis|ees e|fes eis|f fis|ges g
    gis|aes a ais|bes b|ces
for 19-tone: c cis des d dis ees e eis|fes f fis ges g gis
    aes a ais bes b bis|ces
(pipe sign indicates that the tones have a common freq.)
*/

// definitions for TET
0 => int TET_12;
1 => int TET_19;

[
  [ // pos. 0: 12TET
    ["c", "bis"],
    ["cis", "des"],
    ["d"],
    ["dis", "ees"],
    ["e", "fes"],
    ["eis", "f"],
    ["fis", "ges"],
    ["g"],
    ["gis", "aes"],
    ["a"],
    ["ais", "bes"],
    ["b", "ces"]
  ],
  [ // pos. 1: 19TET
    ["c"],
    ["cis"],
    ["des"],
    ["d"],
    ["dis"],
    ["ees"],
    ["e"],
    ["eis", "fes"],
    ["f"],
    ["fis"],
    ["ges"],
    ["g"],
    ["gis"],
    ["aes"],
    ["a"],
    ["ais"],
    ["bes"],
    ["b"],
    ["bis", "ces"]
  ]
] @=> string note_names[][][];

```

```
  fun void set_freqs(float f[]) { f @=> freqs; }

  fun float[] get_freqs() { return freqs; }

  fun void init(string arg) {
    if (arg == "12") {
      calculate_freq(27.5, 3, 12) => ref_freq;  // 3 steps from A to C for 12TET
      TET_12 => k_select;
    } else {
      calculate_freq(27.5, 5, 19) => ref_freq;  // 5 steps from A to C for 19TET
      TET_19 => k_select;
    }

    calculate_freqs(k_select) @=> freqs;
  }

  fun float calculate_freq(float local_ref_freq, int i, int k) {
    i / k $ float => float f;
    return local_ref_freq * Math.pow(2, f);
  }

  // k_select: selector for # of tones in equal temp. scale
  fun float[] calculate_freqs(int k_select) {
    int k;

    // set k
    if (k_select == TET_12) {
      12 => k;
    } else {
      19 => k;
    }

    float ret_freqs[k];   // not necessary to declare capacity k ??

    for(0 => int i; i < k; i++) {
      calculate_freq(ref_freq, i, k) => float freq;

      // store freq in freqs
      for (0 => int j; j < note_names[k_select][i].size(); j++) {
        freq => ret_freqs[note_names[k_select][i][j]];
      }
    }

    return ret_freqs;
  }
}
```

C.6 voice.ck

```
// TODO: refactor queue names
// voice container class with ChucK-specific variables,
    Lilypond-like tokens (in the FIFO queue) and meta-data
public class voice {
  // ChucK
  SawOsc oscillator;    // TODO: oscillator change from command
       line arg
  ADSR envelope;

  // Lilypond-like tokens
  string_fifo q;

  // meta-data
  int transpose;

  // pre-constructor. default values
  1 => transpose;
  .5 => oscillator.gain;

  // chuck things
  oscillator => envelope => dac;

  fun void set_fifo(string_fifo sf) { sf @=> q; }
  fun void set_transpose(int t) { t => transpose; }

  fun ADSR get_envelope() { return envelope; }
  fun int get_transpose() { return transpose; }
  fun string_fifo get_fifo() { return q; }
  fun SawOsc get_oscillator() { return oscillator; }
}
```