# Learning Elementary Musical Programming with Extempore: Translating Arvo Pärt's *Fratres* into Live Code Snippets

Fabia F. A. E. Bertram
University of Cologne, Germany
fabia.bertram@yahoo.de www.soundcloud.com/fabiabertram

In: Jakubowski, K., Farrugia, N., Floridou, G.A., & Gagen, J. (Eds.)
Proceedings of the 7th International Conference of Students of Systematic Musicology (SysMus14)
London, UK, 18-20 September 2014, http://www.musicmindbrain.com/#!sysmus-2014/cfmp


Live coding is a programming practice that works through the real-time implementation of code (Wang and Cook 2003; McLean 2004) and thus vastly changes the perception of conventional computer programming in terms of melding design, coding and debugging phases (Fay-Wolfe 2003) into one single process. This means that immediately putting code into action and turning out results enables a direct verification of whether a program works at all, and if the results make sense as expected or need to be modified. Musical live coding may therefore motivate people through apprehending the skill of learning how to code their own improvisations or compositions. A wider margin of people, such as musicians or people with augmented musical interest in general, is reached, which is interesting from an educational perspective. But live coding contains many more interesting properties than just these pedagogical ones (Blackwell 2013). The Dagstuhl Report released at the end of 2013 offers a great overview of the different topics of interest at the moment. Further investigative questions relevant to the live coding sector will further be explored in the adjacent Master's thesis by he author.
The topic of this paper concerns the implementation of a given musical composition that is suitable to be coded. *Fratres* by Arvo Pärt is based on a simple mathematical structure (Åkesson 2007, Kautny 2005) that provides a live coder with a variety of tasks for creating musical structures (melodies, harmonies, rhythms). In addition to an overview about the question of why it may be interesting to work with pre-existing compositions in addition to tutorials, musical programming solutions will be provided succeeded by musical analysis of the fundamental composition.
This paper is a portion of the second part of the author's Master's thesis that documents an auto-immersion into the study of a live coding language – here Extempore – and aims to illuminate elementary musical programming.

This is a derivation of the practical research on musical live coding with Extempore performed in the context of a Master's thesis from the perspective of a programming beginner.[1] The thesis will consist of two parts, a historical disquisition of the live coding movement as well as a practical part documenting the author's own endeavors coding with Sorensen's programming environment and language, Extempore.[2]

Reading and studying the tutorials, both written and recorded, enhanced the motivation to somehow apply the accumulated knowledge. As previous, small improvisations were not that satisfying, the search for an adaptable composition ensued. The minimalistic *Fratres* by Arvo Pärt seems to fit into that category.

In the following, after emphasizing the pedagogical aspects of live coding and quickly introducing Extempore, the composition *Fratres* will be musically analyzed. Then, possible reasons as to why the translation of existing musical material during the practical learning phase may make sense in addition to tutorial-based studies will be reiterated. Finally and before concluding the paper, programming solutions for the particular piece are offered in Extempore.

## Educational live coding and Extempore

### Live coding: Dagstuhl Report 2013

The 2013 Dagstuhl Seminar concerning the *Collaboration and learning through live coding* investigated the characteristics, purposes, current situation and possible future directions of live coding based on the three perspectives - humanities, computer education and software engineering (Blackwell 2013). Next to the obvious notion of live coding as art, the implementation of live code may enhance computer-programming processes or provide new computational solutions in a computerized society such as the present (Ibid.).

As the process of computer software engineering in recent years has shifted towards more inclusive cooperation of developers, designers and customers, live coding offers an effective and simple way of understanding this evolution, as it enables collaboration between different artists as well as involving their audiences through the presentation of minimalistic tools that entrain interest while entertaining (Ibid.)

Live coding offers great tools for computer science education providing direct sound output according to the functions programmed. Dagstuhl confirmed it as an affective teaching strategy. Furthermore, students are motivated to investigate and understand the processes behind computer programming on deeper levels than the oftentimes focus on only a trivial "occupation with the operation of end-user application software" (Ibid.).

### Extempore

The live coding environment and programming language used in the context of this master's thesis is Andrew Sorensen's[3] Extempore. Extempore is Sorensen's second live coding environment after Impromptu.[4] It works through the hybridization of Scheme[5] and xtlang (Swift 2011-2014). xtlang combines "the high-level expressiveness of Lisp with the low-level expressiveness of C" (Sorensen 2013). Scheme being the fundamental environment, it is of great importance to individually study its structural makeup and to learn its necessary building blocks – functions, etc. One example is the Scheme-innate *lambda* keyword that introduces a procedure all the while reminiscent of Church's *lambda calculus* (Dybvig 2009).[6]

## Why translate compositions into code?

Taking each musical element from a composition singularly and coding corresponding algorithms forms a guided collection of solutions to specific problems while showcasing the programmer's current level of ability. Different coding structures have to be applied to resolve each musical task. The memorization of previously learned algorithms is practiced which leads to both a deeper knowledge and a better overall understanding of the learned materials. It also brings them back to the surface to become part of the programmer's active knowledge. The "tested" materials will be more readily usable in the future. A piece may harbor musical elements that require a combination of different programming structures, thus improving programming abilities by adding new knowledge. Modified, these tools can later always be reapplied to other pieces or emerging improvisational ideas. Translating a score's musical elements can be compared to solving a cumulative math problem in a test situation: previously learned strategies are applied or recombined to solve a problem at hand.

The strategy of using existing music does not stray far from the obvious studying of tutorials. It tests or motivates the coding of different structures and, at the same time, pro-

vides deeper and step-by-step insight into the building blocks of a score alongside enriching one's own compositions or improvisations. Being able to apply the tutorial materials in such a new (but guided) surrounding can be compared to playing and practicing a musical work that contains specific technical difficulties. Instead of concentrating on purely technical exercises, the technical parts are embedded into a more entertaining situation.[7]

Transforming a musical score into coded segments with the aim towards a fully coded version is an extra credit exercise which fulfills two conditions that may be neglected when coding strictly tutorial-based or solely jumping from tutorials to improvisations: On one hand, the student will finally have a coding performance that can stand on its own while having practiced or improved, or even acquired new programming skills. This may motivate further study of the environment and programming language. On the other hand, students coming from more traditional music backgrounds with little expertise in musical improvisation skills will not have to deal with losing time over finding their own musical ideas on top of dealing with the programming aspect. They can focus their attention on the acquisition of programming skills.

In the end, at best one will have a coded version of the inspirational composition. But even if it is not possible to fully translate every part of the piece, be it due to the student's level of expertise or other factors, at worst, a high amount of learning and recollection will have taken place with more detailed information of what procedures to revise.

## *Fratres* – musical analysis

*Fratres* by Arvo Pärt is considered an important example of Pärt's style of composition, *tintinnabuli* (Åkesson 2007, Kautny 2005).[8] Although minimalist, the repetitive juxtaposition of choral and percussive moments amount to a total duration ca. 11 minutes. Since the first version's appearance in 1977, more have emerged in different arrangements and instrumentations such as duets (e.g. violin/piano) and other ensembles.

The piece essentially consists of two musical components, a choral segment surrounded by a percussive motif, that are repeated throughout the piece. The structure of the chorale's chord progression does not change structurally, essentially giving *Fratres* its repetitive character. But change occurs nonetheless; the choral is gradually transposed down a third for each segment's repetition following the harmonic D minor scale's key structure.[9] As a clarifying example, the top voice starts on E in the first choral, on C# in the second, on A in the third and so on.

**The three-voice choral** consists of two parts with each three measures of four, six and eight chords set in a changing time signature of 7/4, 9/4 and 11/4. The chords cannot be analyzed following traditional western harmonic rules.

An eight-chord-progression is formed with top and bottom voices based on the D minor scale and the middle voice filling in notes of the A minor triad. The first four notes of the scale-driven voices directly move down, followed by a cut in which the rest of the scale is moved up one octave to complete the scale in the same downward motion. Due to the diatonic scale character in which every eighth step of a scale has identical note character

(D, E, F, G, A, B♭, C#, then again D, E, F...), the chorale's chord progression finishes on the same note it started on.

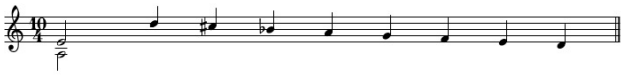

**Figure 1.** The A-triad (without C) and the D minor scale. Both are the fundament of Pärt's composition. While drone sounds and the middle voice of the chorale stay on sounds of the A minor triad (middle voice includes C), the top and bottom voices of the chorale move step-wise through the downward harmonic D minor scale. Both voices, however, do not start on D itself. The top voice starts on E, the bottom voice on C#.

The combination of voices that freely move in melody lines (top and bottom voices) with others that only cover one triad's sounds (middle voice) is the fundament of Pärt's tintinnabuli style of composition (see also endnote 8). Due to the A minor triad's C's friction with the D minor scale's C#, a floatation of the tonal center between A minor and A major occurs.

The chorale's first part is built gradually coming in from the eight-chord-progression's outer sides. For the first measure with four chords, chords 1,2,7 and 8 are taken (see numbered chords in Fig. 2) while chords 3, 4, 5 and 6 are left out. For the second measure with its six chords, 1,2,3,6,7 and 8 are taken while 4 and 5 are still not used. All eight chords are finally used in succession in the final measure (Ibid.).

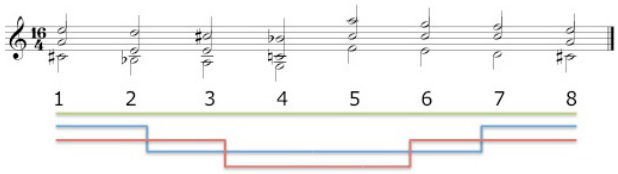


**Figure 2.** The un-rhythmized eight-chord progression of the chords used for each of the three segments of the chorale parts 1 and 2. Top and bottom voice clearly show a scale-like movement in harmonic D minor while the middle voice is restricted to the notes of the A minor triad (A, C, E). The color-coding marks the chords used in the different measures of the chorale: Blue for the chords (1,2,7,8) of the first and fourth measures, red for the chords (1,2,3,6,7,8) of the second and fifth measures, and green (1,2,3,4,5,6,7,8) for the third and sixth measures (retrogradation clarified in fig. 3).

The order of the chords in the chorale's second part is a retrograde of the chords in the first part, although the rhythm is not reversed but stays the same (Fig.3).

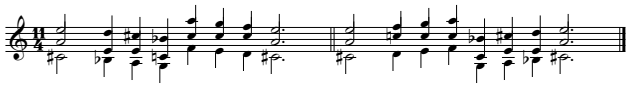

**Figure 3.** The retrograde character of the chorale's second part to the first is shown through measure 3 (first part, choral) and measure 6 (second part, choral). The rhythmical structure is not inverted. It stays the same and is only repeated.

**The percussive motif** is a simple twice-repeated percussive rhythm of a half note followed by both a quarter and a dotted half note set in 6/4-time signature (Fig. 4) set between each choral segment. It is accompanied by drone sounds of A minor that sound throughout the entire piece.

**Figure 4.** The percussive motif appearing before and after each choral segment.

### Connecting score and coding

Due to its simple mathematical formula[10], a translation of Pärt's piece into algorithms

seems possible. Listening to recordings of *Fratres,* an array of musical elements can be deduced – from the use of instruments to the building of melodic, harmonic and rhythmical structures. It is possible to code close-to-score, but also to venture out and create suitable musical structures that may help an overall understanding of the score, such as a simple D minor scale or an A minor chord. Figure 5 provides an overview of the two versions for comparison.

In its totality, the musical elements of Arvo Pärt's *Fratres* useful for programming can be concluded to be

- the creation of different external or internal instruments as well as their connection to the audio output
- drone sounds of the A minor triad
- a percussive loop that plays throughout the piece
- a rhythmized three-voice choral built on the juxtaposition of D and A minor that can be modified dynamically to ensure a dramaturgical arch during the performance

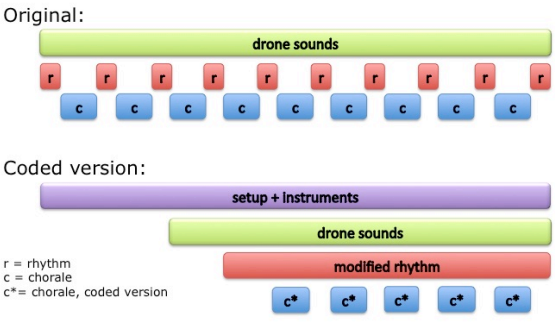


**Figure 5.** A juxtaposition of the two *Fratres* versions, the upper being the one for strings and percussion (1977), the lower a possible coded version as discussed in this paper.

## *Fratres* – source code

**Setup.** The first steps concern the program's setup, which means the creation of all instruments needed as well as possibly equipping them with the associated samplers and consequently connecting all instruments to the DSP[11] output so that a sound can be heard.

An important detail is that Extempore differs between internal and external instruments. Different commands and libraries are used. Whereas a simple synth (internal instrument) is created via the "`define-instrument`" command, an external instrument is created through "`define-sampler`". Also the required libraries that need to be loaded into the program differ, which either results in

```
(sys:load "libs/core/instruments.xtm")
(define-instrument synth synth_note_c synth_fx)
```

or

```
(sys:load "libs/external/instruments_ext.xtm")
(define-sampler sampler sampler_note_hermince_c
sampler_fx)
```

with the loading of respective folders containing sampler files (not needed in internal instruments)

```
(load-sampler sampler "/Users/<name>/<path to
sampler folder>")
```

To be able to set up a sampler (external instrument), an additional parser-function needs to be included that organizes the different sampler files into an ordered a usable list (Swift 2011-2014).

To conclude the setup and to be able to hear the sound structures that will subsequently be programmed, the created instruments

need to be connected to the DSP output. This is done via the following algorithm:[12]

```
(bind-func dsp:DSP
(lambda (in time chan dat)
(cond ((< chan 2.0)
    (+ (synth in time chan dat)
       (drums in time chan dat)))
(else 0.0))))

(dsp:set! Dsp)
```

From now on, a trial of each algorithm by itself will be possible. The setup has provided the system with a valid connection to generate sound output.

**Drone sounds.** A basic solution to playing the long A minor triad sounds can be achieved through the "play-note" function. This function does exactly what its name says, with the corresponding parameters defining at what time the sound should start, which instrument in which pitch and with which volume should resound as well as for how long the sound is supposed to last. As the drone sound is an element that sounds throughout both elements of the composition, this solution may be adequate enough. To form a drone sound of more than one sound, as many "play-note" functions as necessary have to be instilled. Here, that would be two, A minor's A and its fifth, E. The friction between C# and C of A major and A minor is created through the co-sounding of the chorale's melody with the middle voice. As it is easily possible to use a few "play-note" functions at once, it might be best to still group them together in the program for easy structure and access. The drone sounds' volume should be chosen in a range that will not overpower the choral segment that is supposed to appear later on.

```
;; play drones A2 and E3
(play-note (now) synth 45 50 (* 10.0 *minute*))
(play-note (now) synth 52 50 (* 10.0 *minute*))
```

**Beat pattern.** To program the rhythmical structure, the creation of a sampler-based drum instrument is necessary (external instrument).[13] A pattern consisting of a half, a quarter and a dotted half note should be programmed into a loop. This will not only result in its two-time repetition between the chorales as in the original score, but in a continuous beat that plays throughout the piece.[14] Although easily programmable, the original rhythmic pattern might sound quite harsh when combined with the chorale.

Therefore, a modified beat may blend in more easily.[15] This structure is a pattern of four consecutive rhythmized notes, two halves each followed by one quarter note (see fig.6).

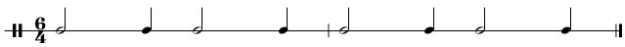

**Figure 6.** The modified beat pattern used instead of the original one (fig.4).

```
;; needed tempo (metronome)
(define *metro* (make-metro 98))

;; list with modified rhythmic pattern
(define *metre1* (make-metre '(4 2 4 2) 0.5))

;; beat pattern
(define metre-test3
  (lambda (time)
    (play-note (*metro* time) drums
               (random (cons .8 *gm-kick*) (cons
.2 *gm-kick*))
               (+ 40 (* 20 (cos (* 2 3.441592
time))))
               (random (cons .8 500)  (cons .2
2000)))
    (if (*metre1* time 1.0)
        (begin (play-note (*metro* time) drums
*gm-kick* 80 10000)
               (play-note (*metro* time) drums
*gm-kick* 80 100000)))
    (callback (*metro* (+ time 0.2)) 'metre-
test3 (+ time 0.25))))

(metre-test3 (*metro* 'get-beat 1.0))
```

The following structure will play an underlying metronome with above the pattern of half, quarter, half and again quarter notes.

The 6/4-time signature of the beat pattern should not collide with the twice repeated (7/4 + 9/4 + 11/4)-structure of the chorale segment as the latter unfolds on 54 quarter beats in total. The beat loop's still runs on 6/4, it should be able to play nine times through without mathematically disturbing a single run of the chorale. Problems with the friction between the different time signatures at hand should be solved via adjustment of the drums' volume to lower-to-medium volume, so that the percussive motif disappears into the background when the choral segment appears.[16] Pärt's tempo is "98". Extempore's internal metronome runs on 60bpm. To be able to play the beat pattern (and later also the chorale structure) in the right tempo, a new metronome needs to be imposed onto the program. This will be accomplished by the "`make-metro`"-function (see coding section above).

**Chorale.** Here, it is necessary to program three individually pitched voices that have the same rhythmical structure and need to be played simultaneously. All voices are melodies. According to the tutorials, there are different possibilities to how melodies as well as chords can be programmed (Swift 2011-2014). For the chord progression, one may attempt to create a chord sequence using the "`play-note`" function with different time orders. The procedure reads all pitches and plays them according to their distance to "`(now)`". This procedure would, however, be very tedious.

Another way would be to include the "`play-note`" command into a loop function similar to the percussive motif's structure, this time playing a pitched melody with rhythmical annotations (Ibid.). To do so, two ordered lists are needed (pitch list; rhythm list). The loop will read the first pitch of the sequence and output it with its corresponding time, then proceed to the next one until all pitches from the pitch list are used up. After this, the loop stops.[17] Volume and duration of the sounds can stay the same throughout the sequence.

```
(define chorale
  (lambda (time plst rlst)
    (play-note (*metro* time) synth (car plst)
120 33150)
    (if (not (null? (cdr plst)))
    (callback (+ (*metro* time) (* .5 (car
rlst))) 'chorale (+ time (car rlst))
                  (if (null? (cdr plst))
                      '(car plst)
                      (cdr plst))
                  (if (null? (cdr rlst))
                      '(44100)
                      (cdr rlst))))))
```

All three voices use the same procedure, which can be effectuated more than once simultaneously. Therefore, only the pitch list will be modified when calling the function "`chorale`" according to the voice it is supposed to output (top; middle; bottom). The rhythm list stays the same for all three voices and thus

```
;; rhythm list for chorale, parts 1 and 2

(2 1 1 3 2 1 1 1 1 3 2 1 1 1 1 1 1 3
2 1 1 3 2 1 1 1 1 3 2 1 1 1 1 1 1 3)
```

is to be copied into each rhythm list. The called function for the top voice would look like this:

```
(chorale (*metro* 'get-beat 1.0)
'(76 74 77 76
76 74 73 79 77 76
76 74 73 70 81 79 77 76
76 77 74 76
76 77 79 73 74 76
76 77 79 81 70 73 74 76)
'(2 1 1 3 2 1 1 1 1 3 2 1 1 1 1 1 1 3
2 1 1 3 2 1 1 1 1 3 2 1 1 1 1 1 1 3))
```

The "`chorale`"-function also refers to the "`*metro*`" of 98 that had already been implemented during the building of the beat pattern. It is important to include it in this

function as well to rightly align both functions.[18]

**Summary.** The result is three different musical structures (drone sounds, beat pattern, choral) executed by different instruments (drums – sampler; synth – internal instrument) that are correctly connected to the audio output. A first trial of all coded elements of *Fratres* is now possible.

### Further research and questions

There are a few more musical elements that are derivatives of the *Fratres*' score:

- the modification of the chorale's code to enable a transposition down a third for each choral segment (mentioned above in the musical analysis)
- heard in the ensemble versions with solo instrument: differently paced arpeggiated figures connected to the choral and playing simultaneously to it)

It might be useful to figure out how to compress the chorale so that it fits into just one single structure instead of the three single melodies mentioned above. For this purpose, it should also be taken into account that the bottom voice mimics the top one so that a derivation of the bottom from the top voice falls into the same problem situation. The single structure should, furthermore, be able to transpose the pitches of the top and bottom voices down a third according to the D minor scale and rearrange the A minor triad pitches of the middle voice to fit each new transposed segment.[19] The most probable strategy seems to be to convert the chorale's pitch list of the top voice into a pitch class structure that is generally scaled to the harmonic D minor scale. The bottom voice should do the same motion as the top, but transposed down three half steps, then an octave. Then, also the transposition of the entire chorale sections needs to be addressed.[20]

When hearing other versions of *Fratres*, for example the violin and piano version, the violin's introductory variation on the choral sequence stands out. It consists of the choral as arpeggiated chords. It may be interesting to build an arpeggio sequence that plays the sum of the three voices as similar arpeggiated chords without changing the overall rhythmical proportions of the choral.

It may also be plausible to further divide the arrangement by using an organ[21] for the choral, or playing the arpeggios with a piano or string instrument sampler. I may come back to these tasks later when I will have gained more insight into the Scheme programming language as well as more practical expertise in Extempore.

## Summary and conclusion

As shown, *Fratres* by Arvo Pärt is a good example of a composition that can be turned into code. Its simple minimalist structure provides an array of different tasks that allow practice of the materials learned in tutorials and result in an achievable performance early on.

Extempore offers a great fundament to building solutions for different musical ideas or necessities. The tutorials provided by Swift explain various ways of implementing structures.[22] Decisions have to be made concerning the translation of the score into code based on what is thought to best suit each musical element and gradually learn how to

program more elegantly. This emphasizes live coding's ability to further computer science education.

Translating an existing score might move away from the current notion of a live coder creating improvised material from scratch in front of an audience. Nonetheless, it does offer a lot of benefits for learning purposes. This starts by analyzing the piece that is supposed to be transferred. The musical dissection may provide better insight into musical elements that make up a successful composition. Each piece of music that is used creates its own array of musical elements that have to be implemented. These elements present problems that may go beyond what an individual would think of when improvising. They provide time to figure out solutions while trying out different strategies. Knowledge is strengthened and adapted. There is no need to expend or divide energies to create improvisation ideas; all focus goes towards the implementation into code. Finally, the consolidation of all structures derived from the score[23] may lead to a successful early performance that motivates future endeavors and provides a cementation of acquired knowledge.

**Acknowledgements.** I want to thank all people that have accompanied my thesis process so far: Prof. Dr. Uwe Seifert for letting me do the very new and exciting project of researching live coding under his guidance, Andrew Sorensen for providing advice concerning my chosen environment and language as well as Ben Swift for the well-explained tutorials that are available online and for the readily given advice to a "newb" such as myself. I also want to thank the Extempore Google-group for its animated discussions of different subject matters concerning language and community and my family and friends for providing loads of emotional support as well as critical comments. An honourable mention goes to the series *Rectify*[24] whose final episode provided inspiration to this project. Last but not least, I want to thank the SysMus-conference for providing a platform to my first live coding steps.

## References

Åkesson, L. (2007). Fratres. Personal Homepage. Retrieved from http://linusakesson.net/music/fratres/index.php

Blackwell, McLean, A., Noble, J., & Rohrhuber, J. (2013). Collaboration and learning through live coding. *Dagstuhl Reports*, 3(9), 130-168. Leibniz Center for Computer Science. Dagstuhl Publishing. Germany.

Dybvig, R. (2009). Procedures and variable bindings – Lambda. *The Scheme Programming Language*. 4th Ed. MIT Press. Cambridge, Mass., USA.

Kautny, O. (2005). Arvo Pärt. *MGG Die Musik in Geschichte und Gegenwart: allgemeine Enzyklopädie der Musik; Personenteil*. 2nd Ed, 146-151. Published by Ludwig Finscher. Bärenreiter. Kassel.

McLean, A. (2004). *Hacking Perl in Night clubs*. Retrieved from http://www.perl.com/pub/a/2004/08/31/livecode.html

Smith, S. (1997). Digital Signal Processors. *The Scientist and Engineer's Guide to Digital Signal Processing*, 503-534. California Technical Pub.

Sorensen, A. (2010). Impromptu Homepage. Retrieved from http://impromptu.moso.com.au/

Sorensen, A. (2011). Extempore Homepage. Retrieved from http://extempore.moso.com.au/

Sorensen, A., Swift, B., & et al. (2011-2014). Extempore Google group. Retrieved from

https://groups.google.com/forum/#!forum/extemporelang

Swift, B. (2011-2014). Extempore tutorials. Retrieved from http://benswift.me/extempore-docs/
- DSP basics in Extempore
- Loading and using a sampler
- Playing an instrument (part I)
- Playing an instrument (part II)

Wang, G., Cook, P. et al. (2003). Chuck: A concurrent, on-the-fly audio programming language. Proceedings of International Computer Music Conference, 219–226. Citeseer.

Fay-Wolfe, V. (2003). *Computer Programming*. Retrieved from http://homepage.cs.uri.edu/faculty/wolfe/book/Readings/Reading13.htm

[1] The first programming with Extempore took place around the 15th of July 2014.
[2] The title of the thesis is *Live Coding: An Artist Movement and Elementary Musical Programming with Extempore*.
[3] Andrew Sorensen is a computer scientist and researcher, computer artist and composer of electronic music. He currently works at the Queensland University of Technology in Brisbane, Australia.
[4] Sorensen's first live coding environment which is strictly Scheme-based and no longer maintained. *A Study in Keith* and other coding examples are available online (Sorensen 2010).
[5] Scheme is a dialect of the Lisp family. It appeared in 1975 and has since had a long tradition in computer science education (MIT, Yale, Stanford). Scheme is a high-level functional programming language in polish/ prefix notation. Brackets, in which the operands follow the operator, surround each function as well as its sub-functions (Dybvig 2009).
[6] In Scheme, *lambda* is a syntactic form creating any procedure and always antecedes the following formal parameters of the procedure. It can thus be understood as a signaling symbol (Dybvig 2009).
[7] Liszt's *Orage* from *Années de Pèlerinage – Première Année: Suisse* technically focuses on quick and strong octave scales. Compared to practicing these same scales in the context of an etude (e.g. Czerny), a pianist's experience is greatly enhanced and motivated.
[8] The tintinnabuli style of composition has been created by Arvo Pärt and consists of two different types of voices. One roams around the notes of a specific root triad (tintinnabular voice; here the middle voice in A minor) while the other follows a diatonic scale-step motion (here top and bottom voices in harmonic D minor). The name tintinnabuli comes from tintinnabulum, Latin for "(little) bell". Pärt was inspired by the old technique of Gregorian chants; his compositions are generally minimalistic and of slow to moderate tempo (MGG 2005).
[9] Because of this downward harmonic D minor scale, the intervals of transposition may vary between either a minor or a major third. The sequence of each chorale's starting point is E, C#, A, F, D, B♭, G etc. (taking the top voice of the chorale as orientation).
[10] The mathematical formula herein is based on the vertical structure of the A minor triad standing as an axis, while the choral structure (in tintinnabuli style) performs a horizontal figure primarily based on the harmonic D minor scale. The chorale is itself cut in two, the second part mirroring the first's notes as a retrogradation (MGG 2005).
[11] Here, DSP is the abbreviation for Digital Signal Processing as used by algorithm developers (Smith 1997).
[12] The two instruments bound to the DSP output are drums and synth, the only ones used for this programming. The outer sub-fragment ( "(+ (…))" ) presents the location in the function where added instruments are enumerated. If only one instrument is needed, the outer sub-fragment is not necessary. If more instruments are used, their information needs to be included in the same fashion as the synth's and drums'.
[13] The respective sampler needs to be loaded and bound to the DSP output (as already done, see setup section).
[14] This also helps to decrease the amount of structures that are in need of manual activation.
[15] It is still to be further investigated, whether a more authentic structure can be found.
[16] Although in low-to-medium volume, the beat loop will still prevail in the sections that revolve around it (in between chorales).
[17] This loop structure differs from the beat pattern's loop. The latter continuously plays a beat while activated (continuous loop), while the melody loop stops after all notes from the pitch list have been used up (loop with termination).
[18] Otherwise, a rift forms between the function that uses the „new" metronome and the one working with Extempore's internal one which is much slower (60 bpm).
[19] See endn.9
[20] Extempore provides a library that manages anything related to pitch class. Two pitch class sets (pc for the top and bottom voice, pc for the entire structure's downward progression (see endn.9) need to be intertwined.
[21] Swift explains how to build a simple drone organ in one of his tutorials (Swift 2011-2014).
[22] There is always more than just one way. Also see section about the chorale's programming.
[23] Here, exemplary: the drone sounds with the beat pattern and the chorale.
[24] *Rectify* (2013), SundanceTV, created by Ray McKinnon: Season 2, episode 16, *Unhinged*.