VŠB — Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Generování hudebních skladeb počítačem
Generating Music Compositions by a Computer

2011 Ondřej Garncarz

Declaration

I declare this thesis was written solely by myself. All used sources and publications are listed.

Date: May 6, 2011 Ondřej Garncarz

Abstract and keywords

Abstract

The thesis is focused on a problem of generating music by a computer, music being random but not very different from one created by a human, following particular harmony rules and not overgoing into disharmony. Basic terms from music theory are introduced, then the developed algorithm is described, both generally and in a relation to the Haskell implementation. The program is compared to similar software and a possible future development is suggested.

Abstrakt

Práce se zabývá problémem generování hudby počítačem, která je náhodná, ale zároveň ne příliš odlišná od hudby složené člověkem, dodržující určitá pravidla harmonie a nevykračující do disharmonie. Jsou objasněny některé základní pojmy hudební teorie a následně je popsán vyvinutý algoritmus, jednak obecně a poté i v souvislosti k implementaci v jazyku Haskell. Program je srovnán s jemu podobnými, již existujícími, a je navržen jeho další možný vývoj.

Keywords

music, composition, music generated by a computer, randomness, harmony, MIDI

Klíčová slova

hudba, skladba, hudba generovaná počítačem, náhoda, harmonie, MIDI

Contents

List of Figures

Chapter 1

Preamble

The creation of music is a wonderful process known to humankind for a very long time, interesting not only because of output. During history, people have become able to write music down and analyze it. Sheet music turns out to be made just of few primitives satisfying rules of music which are easy to be interpreted mathematically. Also, it can be transferred back to music by letting musicians interpret it or, in the last few decades, by letting even computers do it which is very easy to achieve.

The goal of this thesis is to algorithmically analyze one of possible processes of the creation of music and to show it implemented. Thus, we start with Chapter 2 introducing basic definitions from music theory, using them in Chapter 3 to establish an algorithm, whose implementation is described in Chapter 4 and subsequently compared with similar programs in Chapter 5.

Have a good read.

Chapter 2

Introduction to music theory

This chapter is willing to introduce some basic terms which will be useful later in the process of creation. Understanding of relations is connected with insight into algorithms and names are the ones this thesis will reuse. Presented terms and relations are — in a very more complex way — taught by the book [1] which is also a great inspiration and source for this thesis.

2.1 Song

Although a little simplifying, let's call a musical work a song.

2.2 Tone

Tones are musical sounds of which the song is composed. They can be produced in many ways — singed, played on a instrument or created synthetically. They don't even have to be produced, they can exist in an abstract form as sheet music notes. They can have diverse qualities depending on their essence:

Pitch is a frequency of the sound.[1]

Duration is a time of how long the sound is lasting.

Dynamics is a strength of the expression, simplified a loudness.

Color (timbre) is an expression itself. Because the tone isn't just a sound wave of the fundamental frequency but a mixture of sound waves rooted from the loudest — fundamental — one, the expression varies on ratios of their strengths. This is what makes different instruments sound differently.

It's important to remark that the qualities can be approached as relative or absolute — for the musical use the relative approach is much more important — we want to know how tones are related to each other, we don't pay too much attention to a physical point of view.

[1] Precisely said: the fundamental frequency.

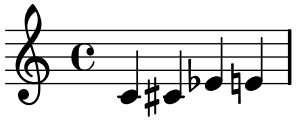


Figure 2.1: Example of tones of different pitches, in ascending order

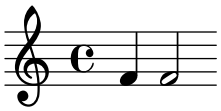


Figure 2.2: Example of tones of different durations

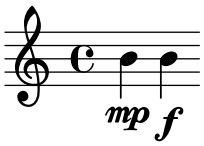


Figure 2.3: Example of tones of different dynamics

In this thesis, the essential qualities are pitch and duration.

The distance between two tones — their pitches — is called an interval. The distance of 12 semitones[2] is an important one — an octave — defining a pitch twice as high (or low). For the simplicity in a lot of practices, distances bigger than the octave can be recalculated as the value modulo 12 and those under zero — an unison — can be added by 12 resulting in all intervals being from the unison to the octave.

2.3 Melody

A melody is a sequence of tones.

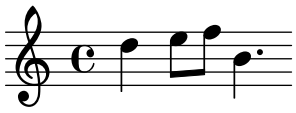


Figure 2.4: Example of a melody

2.4 Chord

A chord is a set of tones sounding at a moment. The fundamental tone — not necessarily the lowest one — on which the chord is built is called a root. Sometimes a subset of a chord can be viewed as of the same nature as the chord.

[2] A semitone is the smallest distance from the classical point of view.

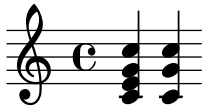


Figure 2.5: Example of a chord
followed by a chord of the same nature but with one tone suppressed

2.4.1 Triad

A triad is a commonly used chord, consisting of three tones between which these intervals can occur:

Major consisting of distances of 4 and 7 semitones to the root.

Minor consisting of distances of 3 and 7 semitones to the root.

Diminished consisting of distances of 3 and 6 semitones to the root.

Augmented consisting of distances of 4 and 8 semitones to the root.

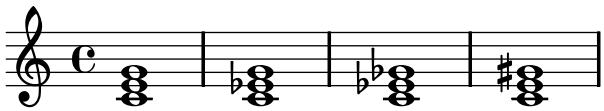


Figure 2.6: Examples of triads: major, minor, diminished and augmented built on C

2.5 Harmony

Harmony is a relation between notes in a chord. The chord can be viewed as an union of a root tone and other tones which have some distance from the root tone. Moreover, a progression of harmony sets "a feeling" of a song whereas a melody can be viewed as "a particular message".

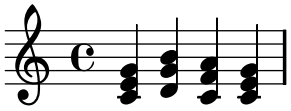


Figure 2.7: Example of a progression of harmony

2.5.1 Voices

A voice can be viewed as a vertical part of a staff. Four-part harmony has these four voices:

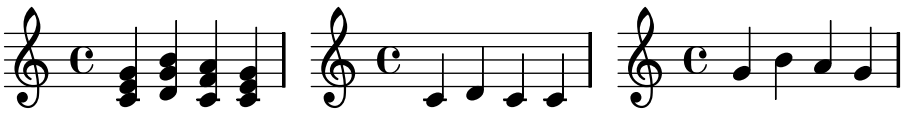
Bass is the lowest part.

Tenor is the second lowest part.

Alto is the second highest part.

Soprano is the highest part, holding a melody.

This thesis focuses mainly on bass and soprano.



(a) Full harmony (b) Bass voice (c) Soprano voice

Figure 2.8: Example of voices

2.5.2 Counterpoint

Counterpoint is a harmony progress where the bass voice and the soprano one evolve contrary.

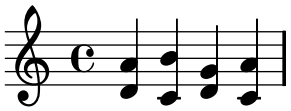
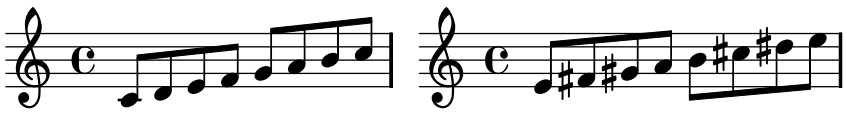


Figure 2.9: Example of counterpoint

2.6 Scale

Songs (or their parts) are usually based on a set of tones called a scale.

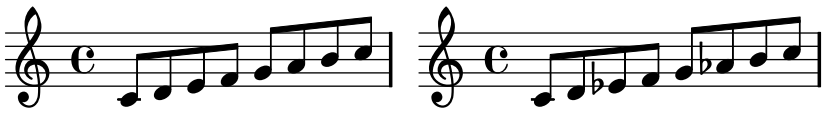


(a) C major (b) E major

Figure 2.10: Example of two scales

The scale is based on its key tone — that's the one the scale is named after — and intervals of included tones in relation to the key tone — that's what the second part of the scale's name stands for. Scales this thesis aims at and which are also the most frequently used in Western music are called the major scale and the minor scale[3].

[3]There are more minor scales, this thesis aims at the natural one.



(a) C major (b) C minor

Figure 2.11: Scales based on C but differing in intervals

2.6.1 Consonance and dissonance

A consonant chord is a chord coming from a scale and being of proper intervals, thus sounding pleasantly, while a dissonant one isn't and is tending to evolve into a consonant one to sustain the scale. This brings richer harmony and more possibilities for melodizing.

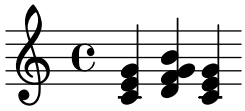


Figure 2.12: Scale C: consonant chord C evolves to dissonant G^7 and then back to C

2.7 Diatonic function

A diatonic function (or a harmonic function) defines a relation between a chord and a scale. Let's consider a C major scale as a referential scale in examples. The main functions presented here come from [1, p. 48].

2.7.1 Tonic chord

The tonic chord commonly begins and ends a song and defines its scale. It's a chord perceived as a tranquillity. The root tone is the key tone of the scale (C), other tones are the third and the fifth of the scale (E and G).

Figure 2.13: Tonic chord

2.7.2 Dominant chord

The dominant chord is a thrilling chord which is usually resolved (followed) by the tonic one. The root tone is the fifth tone of the scale (G), other tones are the last and the second of the scale (B and D).

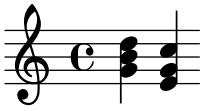


Figure 2.14: Dominant chord, resolved in a tonic one

A tone of particular interest is the **leading tone**, the last tone of the scale (H) — according to the strict harmony theory ([1, p. 38, 47]) it must be unique (not being in more octaves) and resolved and the right resolution is into the first higher tone of the scale — thus the first tone of the scale (C).

2.7.3 Subdominant chord

The subdominant chord is somewhere between the tonic and the dominant. It can be viewed as a preparation for a dominant chord with a medium amount of thrill. The root tone is the fourth tone of the scale (F), other tones are the sixth and the first of the scale (A and C).

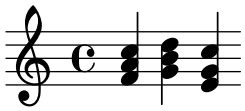


Figure 2.15: Subdominant chord, followed by a dominant one resolved in a tonic one

2.8 Rhythm

Rhythm is a timing of single tones based on a more or less regular patterns. The fundamental repeating element of rhythm is a **beat**. Beats are arranged into regular groups of few called **measures** (also bars). The first beat of the measure is usually stressed, called a downbeat. **Tempo** is a speed of a song, usually denoted as a count of beats per minute.

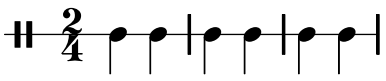


Figure 2.16: Rhythm: three measures, inside each two beats

Chapter 3

One of possible processes of the creation of music

Taking harmony and its progression as a basis in our process can lead us to a very straightforward approach. This chapter deals with an algorithmic concept I've designed, sections are describing details of it.

Let's see a song as an at least two dimensional playground, space of possibilities, where one dimension stands for time and the second one stands for tones. Some of games played there may be pleasant to watch, joyful or making patterns and some may be not. If we were told what moves are the only allowed, the game couldn't be rich in the end, independently of the quantity of the allowed moves. Instead, we can list just moves we wouldn't be happy to see and let the rest for players which brings freedom and a plenty of unexpected situations, hopefully pleasant.

In the beginning it's reasonable to think over an expected output. We may want a finished song or a real-time music stream, possibly never-ending. The first demand is leading to some calculations with the desired output in a finite time, the second one is more complex — calculations have to be done faster than playing. For simplicity let's focus on finished songs — even so it's not too far from a streaming idea assuming computing is fast enough.

Another concern is about output characteristics — for instance from theory we know songs or their parts are based on scales, thus we may want a song to be in a random scale, a certain scale or to change the scale randomly. According to [1, p. 110] the last demand brings some new rules, for the simplicity we will work with one scale songs. Songs are also of some tempo and rhythm, we will keep these constant as well.

The constants can be set randomly but they may be considered as inputs of the algorithm, allowing an user to demand his favorite style.

The first thing to start a song with is a tonic chord so a listener accustoms to the used scale. Then chords progress freely, avoiding non-allowed moves, till the ending tonic chord again closes the song. This is a very general concept so it limits output as little as possible.

Supplying the process with new chords is a job of a random chords generator. It takes a general random number generator and gives a couple — tones of a chord (which are of some

count) and a duration of the chord. The result should be random as possible, but a little adjustment can be made to accelerate the following procedures — the probability distribution of a tone can be the normal (Gaussian) distribution, thus focusing more on tones around the center, not going into extremes too often. Another adjustment could be made by telling the chords generator about the used scale, thus allowing it to give a scale chord more probably than an out of scale chord.

So we will ask for random chords until we receive a tonic chord. The truth is we could generate the chord in a fixed way, as a function of the scale, but this would fix the chord inversion — making the chord structure fusty. The received tonic chord is to become the song's beginning.

Most of following supplied chords will probably sound weird because of their fortuity. Thus we want to relate them to past chords and analyze how much a possibly succeeding chord fits with the past. Rating can be split into two phases: rating of harmony relations and rating of rhythm relations. This split allows us to find a harmonically fitting tones at first and let the finding of a right duration as an independent next step. The less combinations we have to choose from, the more fast we will find the right value(s).

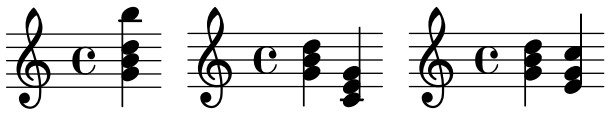
Both ratings can be of many models. The used one is making the final rating by composing (multiplying) elementary ratings, each of them, moreover, being of some defined importance. Sections 3.1 and 3.2 are describing the elementary rules–ratings.

Once having created a latent harmony of chords — a harmonic principle of a song, "a thought" — we need to convert it to few lines, each representing different instrument with its own approach, together creating the final output. This thesis thinks of few lines inspired by real groups, principles are described in Section 3.3.

3.1 Harmony rules

This section lists elementary harmony rules that can be used for rating a possible succeeding chord in a relation with past chords. Every rule is independent of others, thus the final harmony rating must be a compound of them. Examples are based on the C major scale. The rules are extracted from or inspired by [1], a page is often specified.

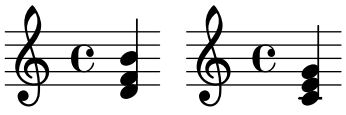
- Is a leading tone treated the right way? According to [1, p. 38, 47] we can't double the leading tone into more octaves and the leading tone has to be led one semitone up to the tonic tone.



(a) Bad, doubled (b) Badly resolved (c) Good

Figure 3.1: Treating of a leading tone rule

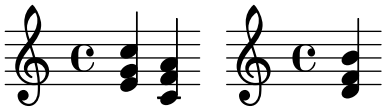
- If this is a beginning of the song, is it a tonic triad chord? The tonic triad is important for introducing the scale and all the examples in [1] begin with the tonic chord.



(a) Bad, not tonic triad (b) Good

Figure 3.2: Beginning with a tonic triad rule

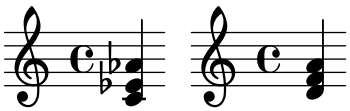
- Isn't a subdominant chord succeeding a dominant chord? According to [1, p. 44, 48] we avoid appending a subdominant chord to a dominant one, as it would weaken harmony.



(a) Bad (b) Anything else, good

Figure 3.3: Subdominant not succeeding a dominant rule

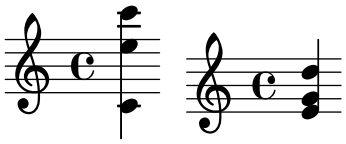
- Is it a scale chord? According to [1, p. 48–50, 110] scale chords are the most important ones and fully sufficient for not long compositions.



(a) Bad (b) Good

Figure 3.4: Scale chord rule

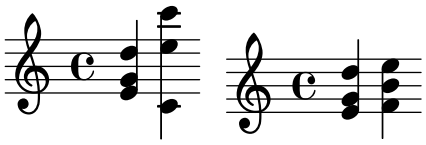
- Is it a “thick” chord? This rule acts against too loose chords (having too big range or being too far from the center of the scale).



(a) Bad (b) Good

Figure 3.5: Thick chord rule

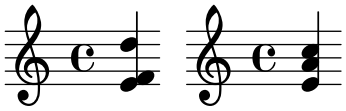
- Isn’t there a too much “jumping”? According to [1, p. 39] it seems that melody shouldn’t consist of big distances between tones, we apply this rule for all the voices as being melodies.



(a) Bad (b) Good

Figure 3.6: Antijumping rule

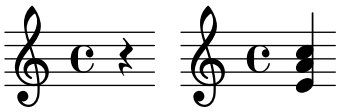
- Is it a triad? According to [1, p. 49–50] it is quite efficient and sufficient for harmony to consist of triads.



(a) Bad (b) Good

Figure 3.7: Triad rule

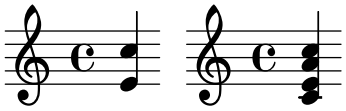
- Isn’t it silence? (In this thesis we avoid silence.)



(a) Bad (b) Good

Figure 3.8: Antisilence rule

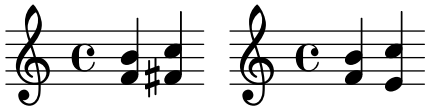
- Is the count of chord tones right? We try to hold four voices harmony.



(a) Bad, just two voices (b) Good

Figure 3.9: Tones count rule

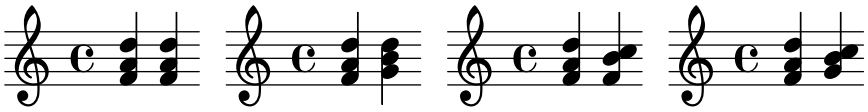
- Is a counterpoint move applied? According to [1, p. 42, 44] it is efficient to use counterpoint move in bass and soprano.



(a) Bad (b) Good

Figure 3.10: Counterpoint rule

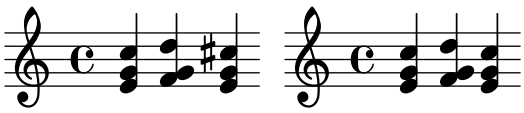
- Is it moving? We try to avoid voices to be stuck.



(a) Bad (b) Bad, soprano stuck (c) Not good, bass stuck (d) Good

Figure 3.11: Moving rule

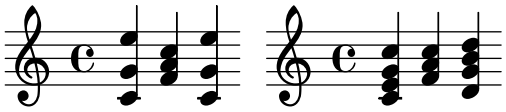
- Is it consonant? We allow dissonant chords to interlace consonant ones, vividing the composition.



(a) Bad, two dissonant chords inline (b) Good

Figure 3.12: Consonant rule

- Isn't it repeating? We try to avoid repetition of recent chords for vividness.

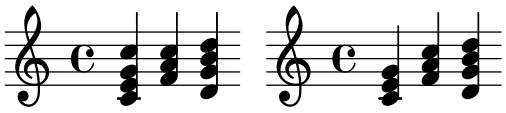


(a) Bad, the last chord is the same as the first one

(b) Good

Figure 3.13: Antirepeating rule

- Isn't soprano repeating? We try to make soprano voice more vivid by not repeating recent soprano tones.



(a) Bad

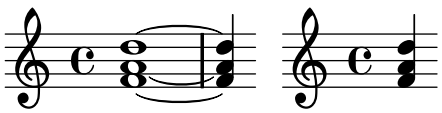
(b) Good

Figure 3.14: Antirepeating soprano rule

3.2 Rhythm rules (of harmony)

Since rhythm is not the main concern of this thesis, just a few simple rhythmic rules were dealt with to make songs more vivid.

- Isn't it overlapping the end of a measure? According to [1, p. 50] it's not so efficient to end the measure with the same chord that begins the following one. Avoiding overlapping also makes harmony more dense, which is quite wanted as this thesis takes harmony as the basis.

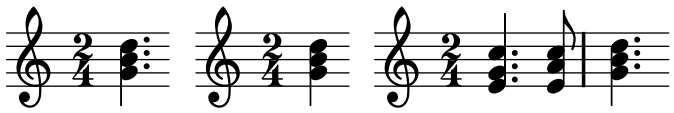


(a) Bad

(b) Good

Figure 3.15: Antioverlapping rule

- Is it ending on a beat, or least copying rhythm from the last measure? We try to tighten the feeling of measure beats by making chords end on beats but also a repetition of the last irregularity is allowed not to have a still robotic rhythm.



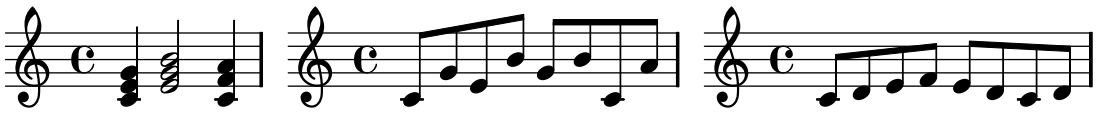
(a) Not so good (b) Good, ending on a beat (c) Good, copying rhythm

Figure 3.16: Beat-copying rule

3.3 Interpretation

This thesis deals with these interpretation lines: a bass line, a melody line and a rhythmic line.

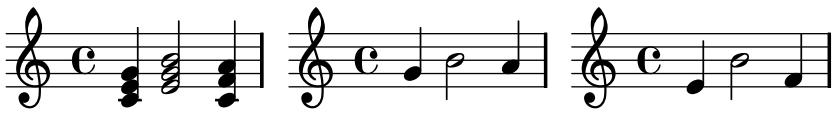
The bass line is fundamentally a line made of the lowest tones of harmony flow — the bass voice. But not in a strict way: the fundamental tones have to sound on a stressed beat, otherwise we can use other tones of the chord (called the fingered way) or we can use tones between the fundamental ones (called the walking bass)[1].



(a) Harmony flow (b) Fingered bass (c) Walking bass

Figure 3.17: Interpretation of bass

The melody line is fundamentally a line made of the highest tones of harmony flow — the soprano voice. Again, it doesn't have to be in a strict way. An analogue of the walking bass was experimented with, but results weren't too pleasing, so just soprano is used — this is something to be extended. Another melody line is a random melody line made of single random tones of harmony flow, it's led the strict way as well.



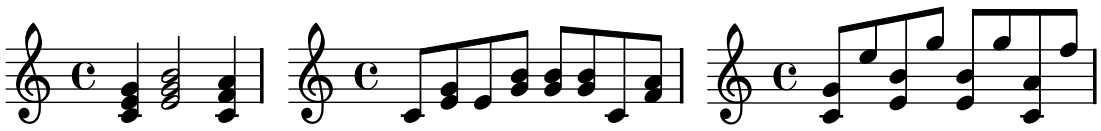
(a) Harmony flow (b) Soprano melody (c) Random melody

Figure 3.18: Interpretation of melody

The rhythmic line which is to boost the feeling of a rhythm of a song is made of chords of harmony flow arranged to meet song's rhythm (but allowed being faster). Chords can be split when repeated, for example the lowest tone the first time of the repeating, then the rest tones, or by a principle of the broken chord 1–5–10[2] where the first and the third tone of a chord is

[1] Inspired by [2].
[2] Inspired by [2].

alternating with the second tone shifted up by an octave. Also percussions could be used to boost the rhythm, but this is not treated in this thesis.



(a) Harmony flow (b) Rhythm chords (c) Broken chords 1–5–10

Figure 3.19: Interpretation of rhythm part

Also the very harmonic flow can be a line in the interpretation, just for the case of making the total sound more dense or to boost the harmony feeling.

Chapter 4

Implementation

This chapter is a bridge between the described algorithm and the implementation itself — named MusGen — done in the language Haskell[1]. Mathematically straightforward and self-describing that Haskell source lines are, they can also be supplemented by a text in a natural language for an easier insight.

4.1 Input

The generator program takes as input these parameters which the generated song should fulfill:

- Key of harmony, given as tone's MIDI number.
- Scale's intervals of harmony. There are two options: major and minor[2].
- Beats per measure, given as an integer.
- Tempo of the song, given as a count of quarter notes per minute.
- Minimal duration of the song, given as a minimal count of measures.
- Interpretation style of the song. Possible options are: church, pop and rock.

User can also specify the song's name and make the program generate a new flow, otherwise an old one will be used if exists. Given a parameter `-?` the program prints help, as captured in Appendix A.

4.2 Output

The program execution can be seen as having two phases where each produces output. The first one's product is a file reflecting generated harmony flow. Its name is the song's name appended

[1] For the reason for choosing Haskell see http://i.imgur.com/hF6mS.jpg.
[2] Meant the natural minor.

by the suffix `.flow`. An example of the file containing harmony flow of a short song is depicted in Figure 4.1. The second phase's product is a MIDI file containing a particular interpretation of harmony flow. The first file can be reused in next run of program, resulting in a MIDI output with other interpretation.

There is also a script named MIDI2PDF.SH which given the generated song's filename produces a PDF file containing sheet music based on the input file, as depicted in Figure 4.2.

```
Chord {tones = [66,69,74,78], key = 62, intervals = [0,2,4,5,7,9,11],
    begin = 0, dur = 2, measure = 12, beats = 3}
Chord {tones = [67,71,74], key = 62, intervals = [0,2,4,5,7,9,11],
   begin = 2, dur = 2, measure = 12, beats = 3}
Chord {tones = [67,71,76], key = 62, intervals = [0,2,4,5,7,9,11],
   begin = 4, dur = 6, measure = 12, beats = 3}
Chord {tones = [66,71,74,78], key = 62, intervals = [0,2,4,5,7,9,11],
    begin = 10, dur = 2, measure = 12, beats = 3}
Chord {tones = [66,69,73], key = 62, intervals = [0,2,4,5,7,9,11],
   begin = 0, dur = 4, measure = 12, beats = 3}
Chord {tones = [62,67,71,74], key = 62, intervals = [0,2,4,5,7,9,11],
    begin = 4, dur = 2, measure = 12, beats = 3}
Chord {tones = [64,67,71], key = 62, intervals = [0,2,4,5,7,9,11],
   begin = 6, dur = 6, measure = 12, beats = 3}
Chord {tones = [62,66,71,74], key = 62, intervals = [0,2,4,5,7,9,11],
    begin = 0, dur = 8, measure = 12, beats = 3}
Chord {tones = [62,66,69], key = 62, intervals = [0,2,4,5,7,9,11],
   begin = 8, dur = 4, measure = 12, beats = 3}
Chord {tones = [59,64,67,71], key = 62, intervals = [0,2,4,5,7,9,11],
    begin = 0, dur = 2, measure = 12, beats = 3}
Chord {tones = [59,62,66], key = 62, intervals = [0,2,4,5,7,9,11],
   begin = 2, dur = 4, measure = 12, beats = 3}
Chord {tones = [57,62,66,69], key = 62, intervals = [0,2,4,5,7,9,11],
    begin = 6, dur = 6, measure = 12, beats = 3}
```

Figure 4.1: Example of saved harmony flow

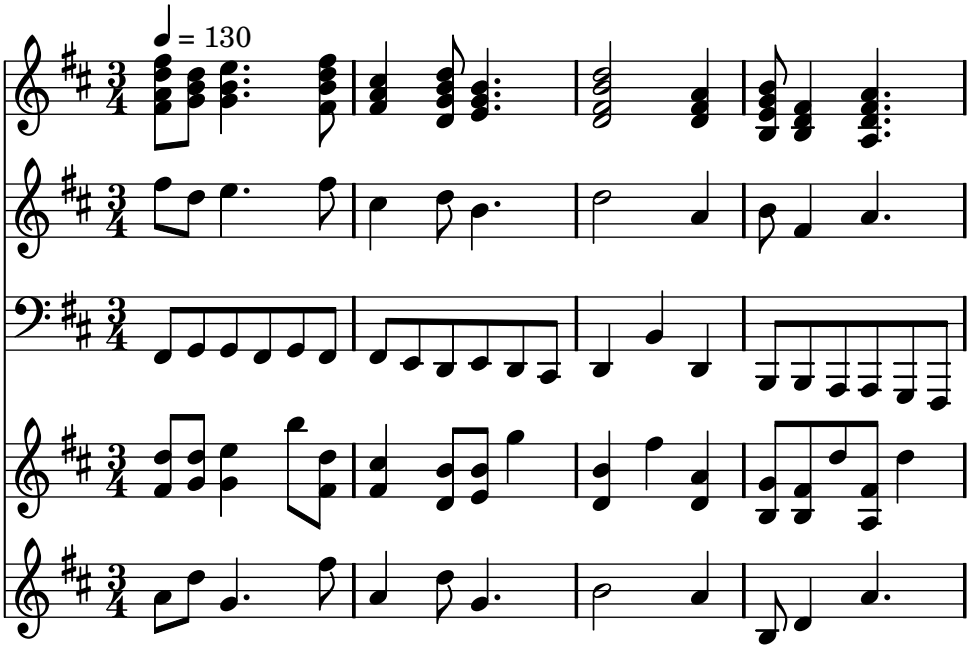


Figure 4.2: Example of generated song in sheet music

4.3 Environment

The generator program is written in Haskell using two extra packages for processing MIDI data and command line arguments, named HCodecs[3] and CmdArgs[4]. The program can be run with GHC[5], the packages can be installed with Cabal[6].

The process of building source codes into a runnable binary can be initiated by calling program make[7] with no arguments.

Script midi2pdf.sh is written in BASH[8] and uses LilyPond[9].

See Appendix A for an exemplary use of the program.

Diagrams used in this thesis are made using SourceGraph[10], which uses Graphviz[11], and svg2pdf[12].

[3] http://hackage.haskell.org/package/HCodecs
[4] http://hackage.haskell.org/package/cmdargs
[5] http://www.haskell.org/ghc/
[6] http://www.haskell.org/cabal/
[7] http://www.gnu.org/software/make/
[8] http://www.gnu.org/software/bash/
[9] http://lilypond.org/
[10] http://hackage.haskell.org/package/SourceGraph
[11] http://www.graphviz.org/
[12] http://wiki.inkscape.org/wiki/index.php/Tools#svg2pdf

4.4 Main program and helpers

The executive core of the program is contained in the file Main.hs. See Figure 4.3 for functions references and Figure 4.4 for module imports. Root elements of the file are:

Input which is a data type determining command line arguments which specify requirements on an output song.

use which is data based on Input giving more information for arguments processing, including help descriptions.

checkArg which is a monad which given a (non evaluated) value and its name as a string tries to evaluate it and in case of error exits the program printing an explanatory information, meant for checking right types of the arguments.

main which is a monad executing this sequence:

1. Processing command line arguments. In case of demand of help or the program's version, the demanded is printed and the program exits.
2. Generating a new random generator.
3. Checking processed arguments for being of right types.
4. In case a harmony flow file doesn't exist or generating of a new one is demanded, a new harmony flow is generated and saved into the file. Otherwise it's loaded from the file.
5. A MIDI file with an interpreted harmony flow is generated.

4.4.1 Types

Haskell comes with a set of basic types which aren't of much difference to types of other typed programming languages, see [3]. New types can be defined from existing ones or compounded, which is used in the program for the sake of better readability. New types and some functions of them are defined in the file Types.hs like this:

Tone is a type meant as a MIDI[13] tone, defined as an integer.

Interval is a type meant as an interval between two tones, defined as an integer.

Intervals is a type meant as a list of intervals — mainly for representing scale intervals and chord intervals, allowing the program to work with any scale or any chord user would define.

Volume is a type meant as a MIDI volume, defined as an integer.

[13] More about MIDI at 4.7.

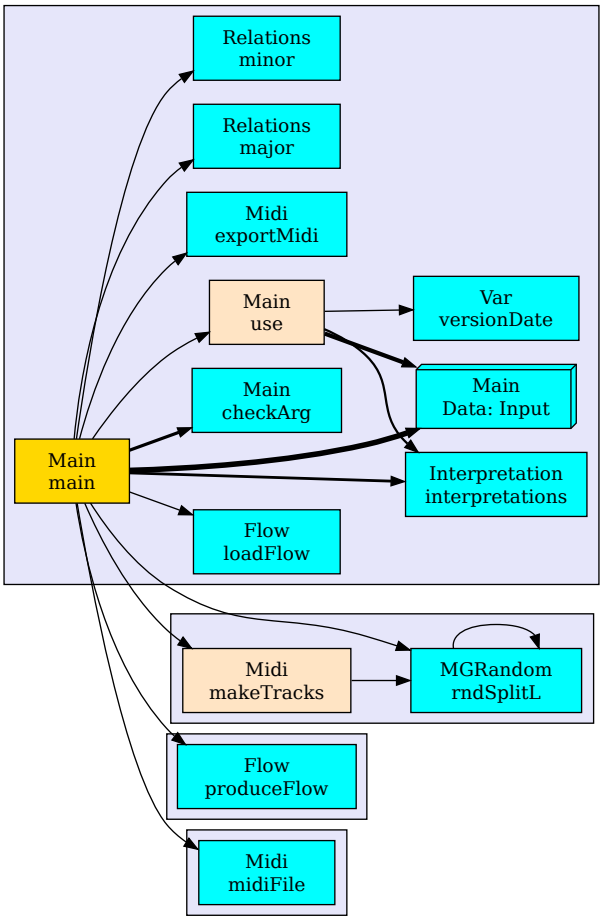


Figure 4.3: Functions references for the file Main.hs.

Duration is a type meant as a duration of a tone or a chord, defined as an integer.

Chord is a data type meant as a chord description, compounded of the following data parts:

tones of the chord,

key of the scale to which the chord relates,

intervals of the scale to which the chord relates,

begin which is a duration since the beginning of the measure,

dur which is a duration of the chord,

measure which is a duration of the whole measure and

beats meant as a count of beats per measure.

Thus, all the chords contain the information about the used scale although this thesis works with a constant scale in the song, but it's a framework allowing to easily extend the concept with switching scales.

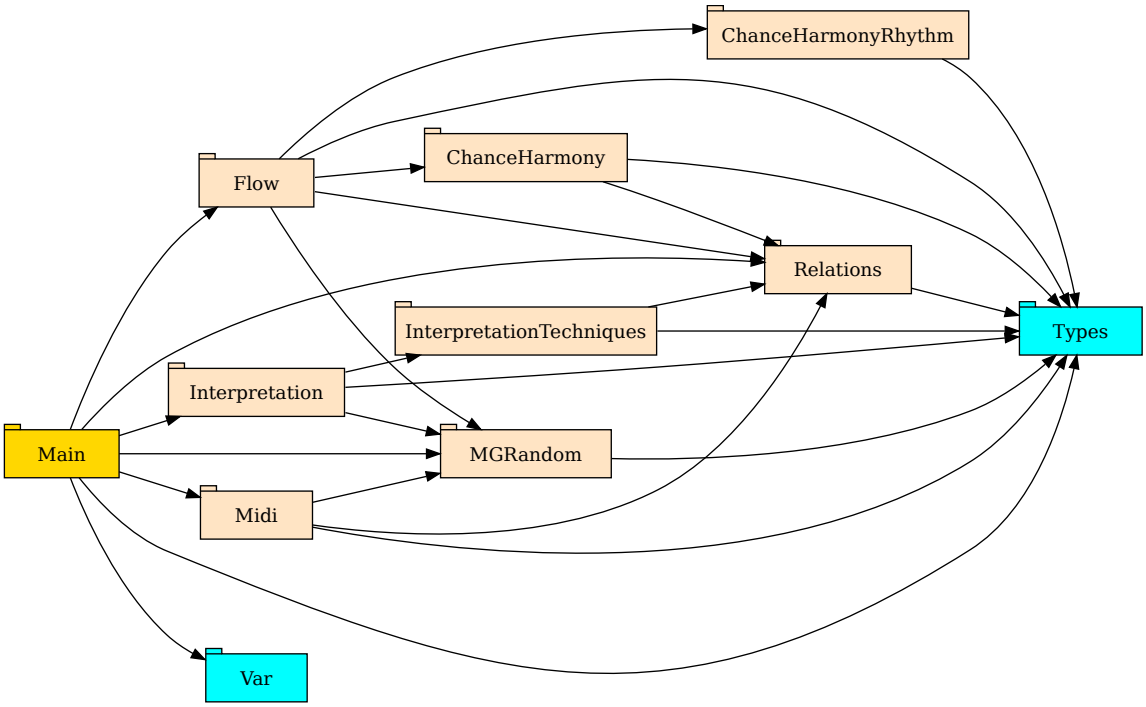


Figure 4.4: Module imports.

For a better understanding of the Chord type, Figure 4.5 is introduced. Its chords are of these values:

1. tones = “C, E, G” (acutally 60, 64 and 67 in MIDI), key = “C” (actually 60 in MIDI), intervals = “major scale“ (actually 0, 2, 4, 5, 7, 9, 11 in semitone distances), begin = 0, dur = 2, measure = 8, beats = 2 (meaning the upper number in the figure)
2. differing values: tones = “E, G, B”, begin = 2, dur = 4
3. differing values: tones = “C, F, A”, begin = 6, dur = 2

This means a beat is of the duration of measure divided by beats. The smallest value is 1 so depending on the two values we determine the fastest sound we can produce.

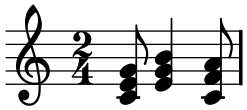


Figure 4.5: Chord type help illustration

remain is a function of a chord, defining a duration which the chord is missing to complete the measure.

showBrief is a function of a chord, defining a string which captures the chord’s information in a brief printable format.

Flow is a type meant as a list of chords.

ChanceType is a type meant as a function mapping a chord and a flow into a float.

MidiEvent is a type meant as a MIDI event, defined as a pair of MIDI ticks and a MIDI message.

MidiTrack is a type meant as a MIDI track, defined as a list of MIDI events.

TracksDefs is a type meant as a list of definitions of how to create and set MIDI tracks.

RndGen is a type meant as a reference to the used random generator[14], StdGen is used.

floatMin is a function defining the minimal suitable float value used in chance functions[15], the value 0.1 is used.

floatZero is a function defining the "zero" float value used in chance functions, the value 0.001 is used.

floatHalf is a function defining the "half" float value used in chance functions, the value $0.5 + floatMin$ is used.

4.4.2 Random generator

For the sake of randomization we'll utilize a set of frequently used randomization functions which are the content of the file MGRandom.hs. They use Haskell package System.Random[16] and its class StdGen, taking its random numbers as input for generating values more desirable like a duration or a chord's tones.

When working with random numbers in Haskell, there can be chosen one of two main ways:

- Getting random numbers from monads, but that leads to forcing functions using those numbers and all functions above[17] to be monads too, which deforms a functional approach a bit.
- Getting random numbers from functions whose one argument must be a random generator. This approach, used in the program, is pure functional but every used random generator needs to be unique, otherwise functions will return always the same results because of the definition of a function as unambiguous. But this is no problem — functions above can split their input random generators and then pass new and unique ones.

For more information about randomness in Haskell see [4].

[14] More about random values generating at 4.4.2.
[15] See Subsections 4.5.2 and 4.5.3.
[16] http://hackage.haskell.org/packages/archive/random/1.0.0.2/doc/html/System-Random.html
[17] In a definition chain.

rndDuration is a function of a random generator, defining a duration randomly picked from even numbers from 2 to 16.

rndTonesCount is a function of a random generator, defining an integer meant as a count of tones. For simplifying things, the result is always 4.

rndIntervals is a function of a random generator, defining an infinite list of (musical) intervals where each is a product of a random sign and an unsigned interval which is picked like this:

Probability	**Interval**
25%	0
15%	4
15%	3
10%	2
10%	5
10%	6
5%	1
5%	7
5%	randomly picked from 0 to 12

rndNormal is a function of an integer meaning an minimum, an integer meaning an maximum and a random generator, defining an integer randomly picked from the normal distribution[18] of the range.

rndTones is a function of a random generator, defining an infinite list of random tones generated with the use of rndNormal for the possible MIDI tones from 0 to 127.

rndChordTones is a function of a random generator, defining a sorted list of tones meant as a chord. It is using rndTonesCount and rndIntervals. The tones are trying to reach the number of rndTonesCount and are made sequentially by adding the corresponding value of rndIntervals to the preceding tone, starting with the MIDI center value 64 plus the first interval. This process is meant to ensure chords are balanced to the center, not exaggerating.

rndSplitL is a function of a random generator, defining an infinite list of random generators made by splitting the input one.

testRndNormal is a testing monad which generates 10000 tones using rndTones and prints counts of occurrences of possible tone values to verify normality of the used randomization.

See Figure 4.6 for functions references.

[18] Precisely said not the normal distribution, but a slightly modified similar one — the Irwin–Hall distribution which is easy to implement: wanting a random number from 0 to max, we sum max random values from the interval $< 0, 1 >$, see [5].

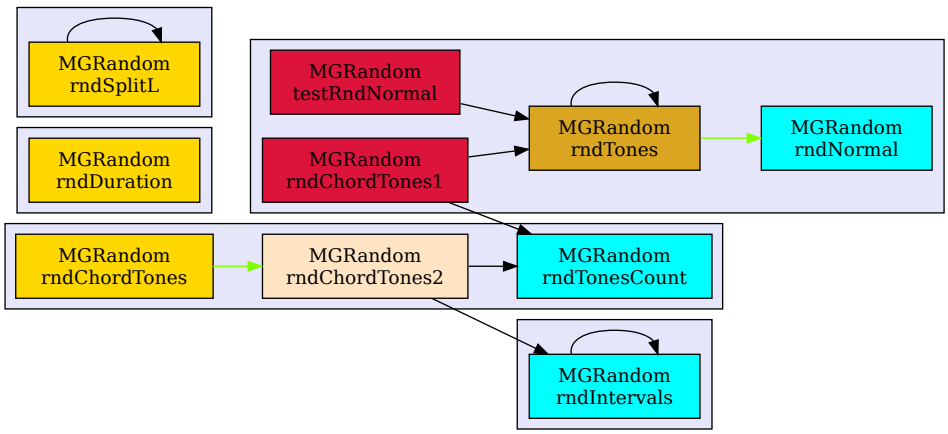


Figure 4.6: Functions references for the file MGRandom.hs.

4.5 Harmony flow

A product of the first stage is a harmony flow. Its creation and file handling is taken care of by monads and functions from the file Flow.hs:

loadFlow is a monad which given a filename reads a content of the file and returns a harmony flow contained in the content.

produceFlow is a monad which given a chord meant as a determination of flow's scale, an integer meant as flow's minimal required measure count and a filename produces a new harmony flow based on the input parameters using a new random generator. The flow is briefly printed, saved in the file and also returned.

nextFlow is a function of a past harmony flow, an integer meaning a measures count, an integer meaning a minimal required measures count and a random generator, defining a harmony flow which is a sequel of the input flow.

nextTonesChord is a function of a past harmony flow and a random generator, defining a chord harmonically following the input flow, having a zero duration. Until a suitable chord is found, new ones are created invoking rndChordTones and rated by harmonyChance — the quality of the suitable chord must beat a minimal required quality which is a random float between 0.5 and 0.7.

nextDurChord is a function of a chord, a past harmony flow and a random generator, defining a chord harmonically equivalent with the input chord, having a duration suitable for a sequel of the input flow. Until a suitable duration is found, new ones are created invoking rndDuration and rated by harmonyRhythmChance — the quality of the suitable duration must beat a minimal required quality which is 0.5.

canBeEnd is a function of a chord and a past harmony flow, defining a boolean value whether the chord can end the flow. Harmonic rule checking function isTonicTriadIn is used to determine the possibility of ending, the chord also can't be shorter in duration than half the measure duration.

realPast is a helper function of a harmony flow, defining a harmony flow which is equivalent to the input flow if it's a regular flow, otherwise to an empty flow (that's in the case the input flow starts with an empty chord, meaning it's a helper flow determining the flow's scale and beat).

See Figure 4.7 for functions references.

4.5.1 Relations

Functions evaluating musical relations are located in the file Relations.hs.

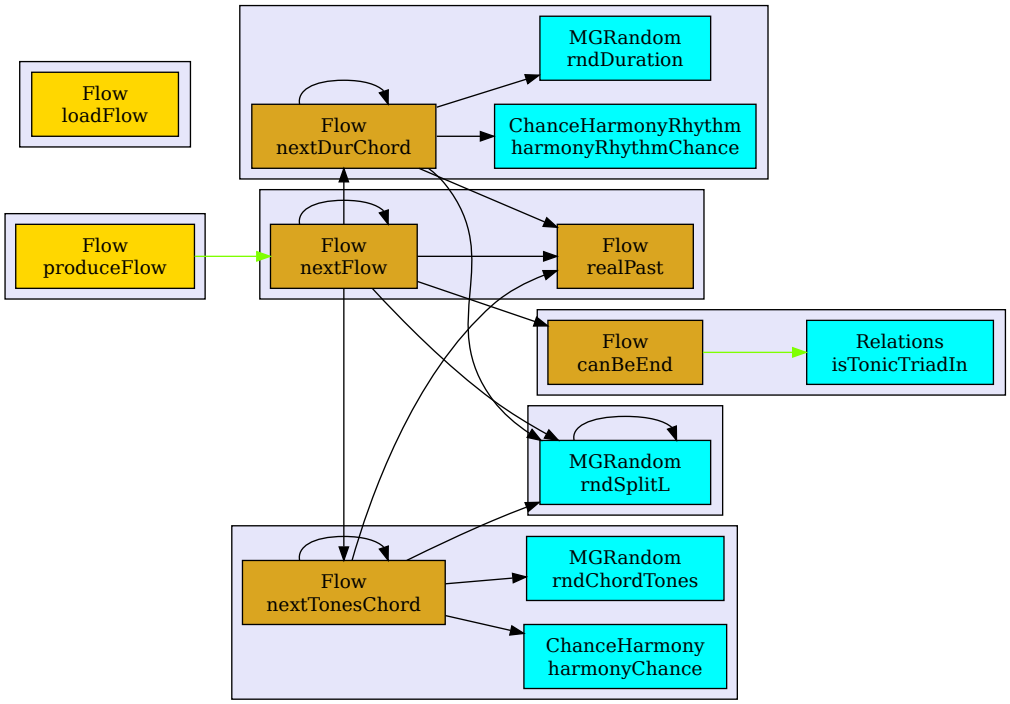


Figure 4.7: Functions references for the file Flow.hs.

scaleSize is a function defining the size of octave in semitones, the value is 12.

major is a function defining intervals of major scale in a relation to key tone.

minor is a function defining intervals of minor scale in a relation to key tone.

chordIntervals is a function defining a list of intervals that a chord can be made of. They are the following:

majorTriad which is a list of intervals making a major triad,

minorTriad which is a list of intervals making a minor triad,

diminishedTriad which is a list of intervals making a diminished triad and

augmentedTriad which is a list of intervals making a augmented triad.

toneJumpFrom is a function of a list of tones and a tone, defining a minimal interval between the tone and the tones from the list.

intervalFromTo is a function of two tones, defining an interval between them in a range from 0 to $scaleSize - 1$, ignoring octaves.

intervalAt is a function of intervals and an integer, defining an interval at the position of the integer from the intervals, cycling in values.

succToneIn is a function of a tone meant as a scale's key, the scale's intervals and a tone, defining a tone succeeding the input tone in the scale.

predToneIn is a function of a tone meant as a scale's key, the scale's intervals and a tone, defining a tone preceding the input tone in the scale.

isFromScale is a function of a tone meant as a scale's key, the scale's intervals and a tone, defining a truth value whether the tone comes from the scale.

areFromScale is a function of a tone meant as a scale's key, the scale's intervals and a list of tones, defining a truth value whether the tones come from the scale.

fitsIntervalsFrom is a function of intervals, a tone meant as a chord's root and a list of the chord's tones, defining a truth value whether the chord is of the intervals.

fitsIntervals is a function of intervals and a list of a chord's tones, defining a truth value whether the chord is of the intervals, by checking all the possible chord's roots.

hasRoot is a function of a tone meant as a chord's root and a list of the chord's tones, defining a truth value whether the chord is of some intervals defined by the function chordIntervals.

isTriad is a function of a list of a chord's tones, defining a truth value whether the chord is a triad.

isFullTriad is a function of a list of a chord's tones, defining a truth value whether the chord is a triad with no tones omitted.

isTonicTriadIn is a function of a tone meant as a scale's key, the scale's intervals and a list of a chord's tones, defining a truth value whether the chord is a tonic triad within the scale.

isSubdominantIn is a function of a tone meant as a scale's key, the scale's intervals and a list of a chord's tones, defining a truth value whether the chord is a subdominant within the scale.

isDominantIn is a function of a tone meant as a scale's key, the scale's intervals and a list of a chord's tones, defining a truth value whether the chord is a dominant within the scale.

isLeadingToneIn is a function of a tone meant as a scale's key, the scale's intervals and a tone, defining a truth value whether the tone is the leading-tone of the scale.

isLeadingToneOkIn is a function of a tone meant as a scale's key, the scale's intervals, a list of the first chord's tones and a list of the second chord's tones, defining a truth value whether, within the scale, the first chord's possible leading-tone is resolved or continued correctly into the second chord, plus there is no more than one leading-tone at a time.

isCounterpoint is a function of a list of the first chord's tones and a list of the second chord's tones, defining a truth value whether the chords progress in counterpoint.

isSopranoMoving is a function of a list of the first chord's tones and a list of the second chord's tones, defining a truth value whether the soprano part of the chords moves.

isBassMoving is a function of a list of the first chord's tones and a list of the second chord's tones, defining a truth value whether the bass part of the chords moves.

percentageMoving is a function of a list of the first chord's tones and a list of the second chord's tones, defining a float meant as a percentage of the second chord's tones not being part of the first one.

isConsonantIn is a function of a tone meant as a scale's key, the scale's intervals and a list of tones, defining a truth value whether the tones are consonant within the scale.

See Figure 4.8 for functions references.

4.5.2 Chances for harmony

Deciding whether a chord can be an appropriate successor to past chords harmonically is is done using functions from the file ChanceHarmony.hs.

harmonyChance is a function of a possible successor chord and past chords ordered backwards, defining a float value — reflecting how the chord fits the past harmonically — as a product of elementary deciding functions, each powered by its significance, as defined by the list **chances**, each function having the same type signature with the expected range of the float being from 0 meaning not fulfilling at all to 1 meaning totally fulfilling:

chanceThick defines whether the chord is thick — as not of a big distance between bass and soprano — and not too far from the key.

chanceJumps defines whether there's not too much "jumping" from the last past chord — meant as too big moves in voices.

chanceInScale defines whether the chord comes from the scale.

chanceTriad defines whether the chord is of a triad nature.

chanceTonicStart defines whether the chord is a tonic triad and thus can begin a song.

chanceNotDomThenSub defines whether the last past chord isn't dominant and at the same time the actual chord isn't subdominant, which would be an illegal move.

chanceLeadingTone defines whether the rule of leading tone isn't broken.

chanceNotEmpty defines whether the chord isn't empty (just a pause).

chance4Tones defines whether the chord has 4 unique tones (octaves differ).

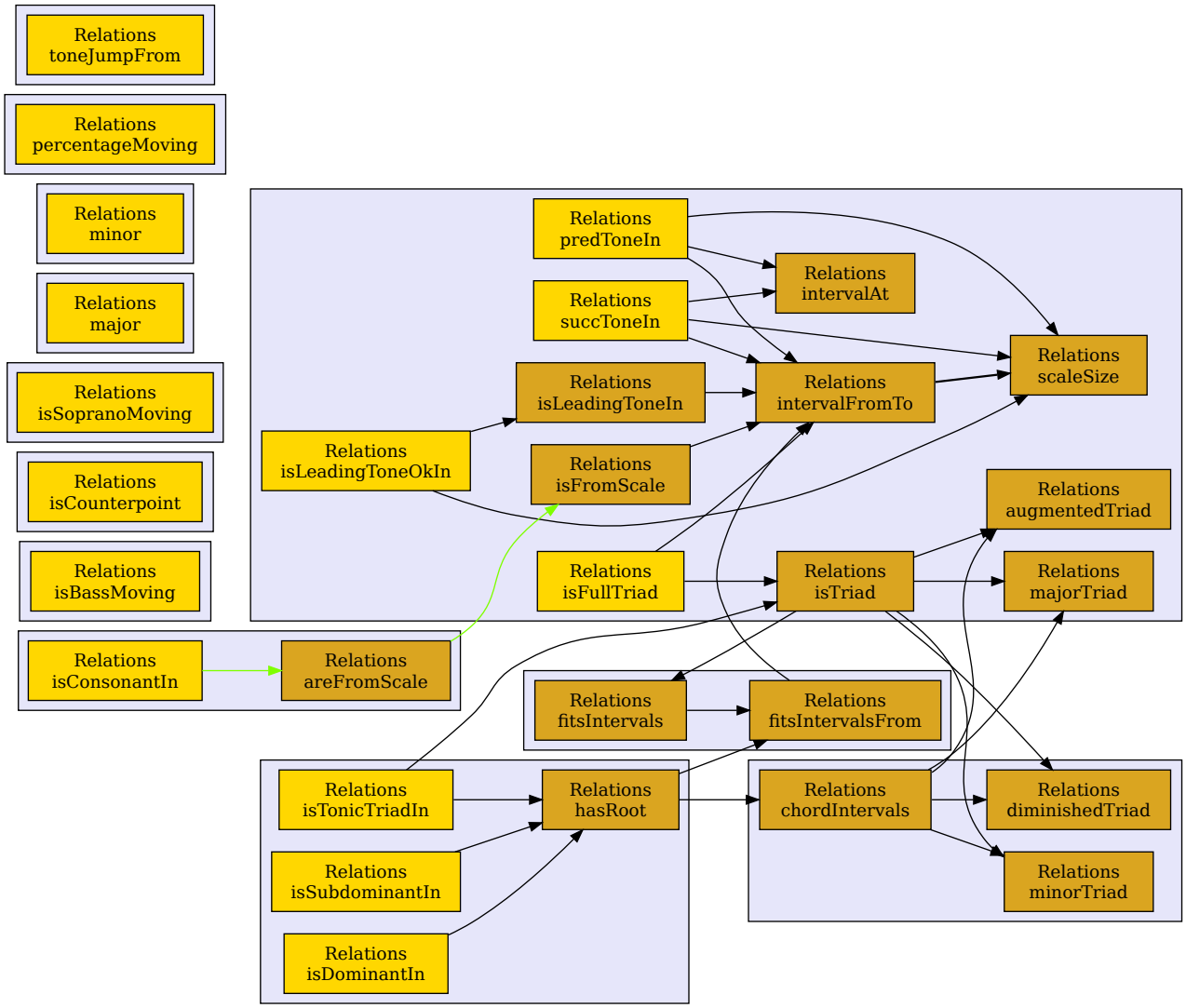


Figure 4.8: Functions references for the file Relations.hs.

chanceCounterpoint defines whether bass and soprano are moving in a counterpoint manner.

chanceMove defines how much voices are moving, the most important are moves in bass and soprano.

chanceConsonance defines whether the progress is of a consonant nature by beginning a flow with a consonant chord and not ensuing a dissonant one by another dissonant.

chanceAntiRepetition defines whether the chord isn't a repetition of one of the last three chords.

chanceAntiRepetitionForSoprano defines whether the soprano doesn't repeat one of the last three tones.

See Figure 4.9 for functions references.

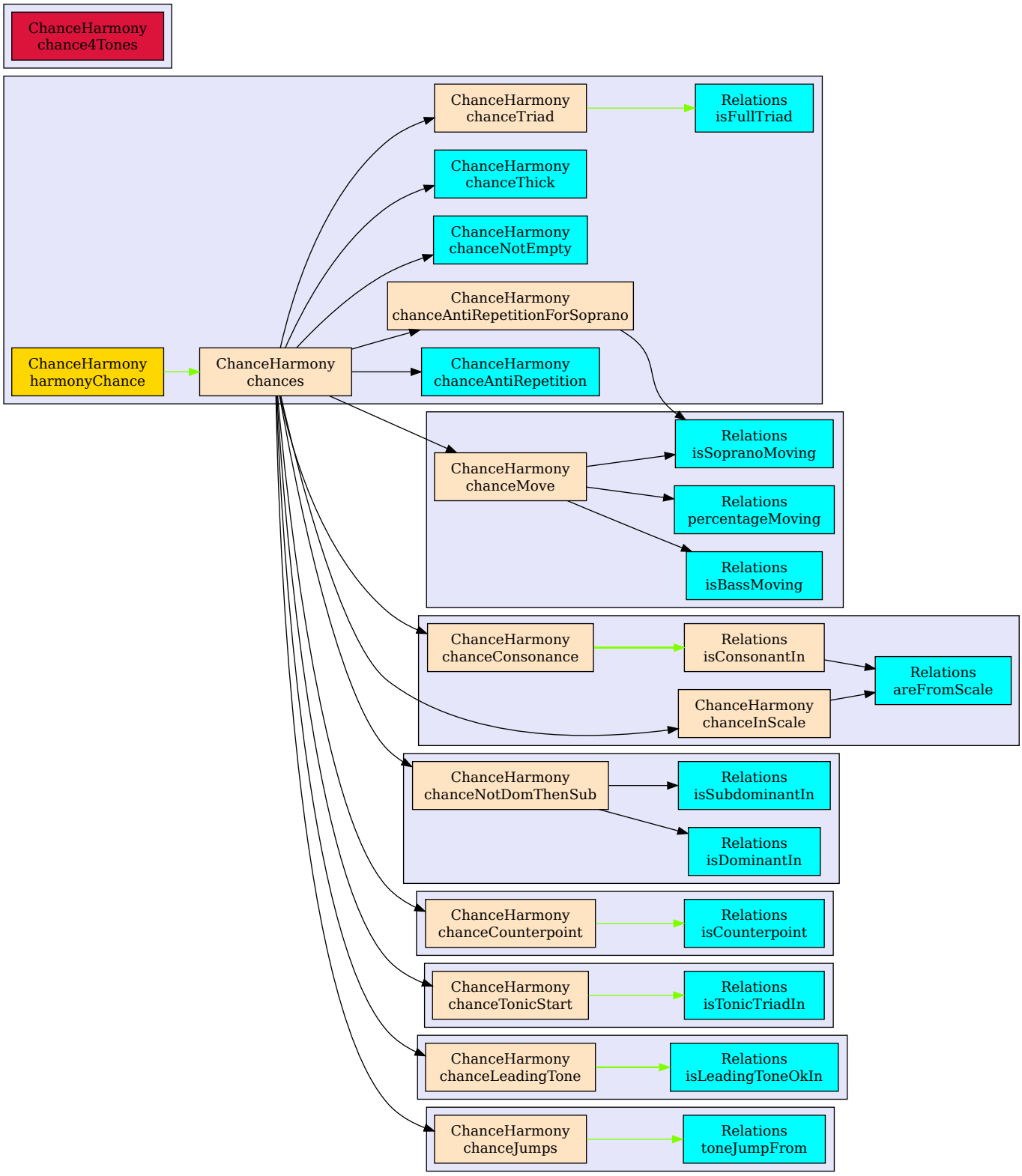


Figure 4.9: Functions references for the file ChanceHarmony.hs.

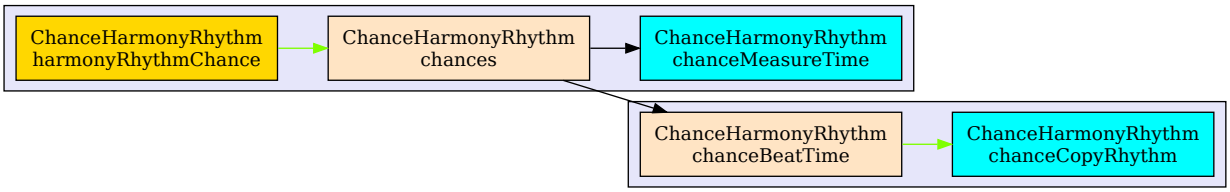


Figure 4.10: Functions references for the file ChanceHarmonyRhythm.hs.

4.5.3 Chances for rhythm

Deciding whether a chord can be an appropriate successor to past chords rhythmically is is done using functions from the file ChanceHarmonyRhythm.hs.

harmonyRhythmChance is a function of a possible successor chord and past chords ordered backwards, defining a float value — reflecting how the chord fits the past rhythmically — as a product of elementary deciding functions, each powered by its significance, as defined by the list **chances**, each function having the same type signature with the expected range of the float being from 0 meaning not fulfilling at all to 1 meaning totally fulfilling:

chanceMeasureTime defines whether the chord ends before or just at ending of measure, meant to prevent overlapping measures.

chanceBeatTime defines whether the chord ends at beat or copies a rhythmical pattern used in the last measure, which is defined by the function **chanceCopyRhythm**.

See Figure 4.10 for functions references.

4.6 Interpretation

The core of the second phase of the program — interpretation — is located in the file INTERPRETATION.HS.

The list **interpretations** is a list of all possible interpretation styles, including:

churchTracks interpreting flow by 5 instruments (organ, piano, acoustic bass, acoustic steel guitar and harpsichord) in a church music resembling style,

popTracks interpreting flow by 5 instruments (synthetic strings, square lead, synthetic bass, acoustic steel guitar and sawtooth lead) in a pop music resembling style and

rockTracks interpreting flow by 5 instruments (distorted guitar, overdriven guitar twice, electric bass and harpsichord) in a rock music resembling style.

Following functions are functions of a harmony flow and a random generator, defining a harmony flow interpreting the input flow in a particular way:

harmonyTrack defines the same flow, copying the harmony.

sopranoTrack defines a soprano melody flow.

harmonyRhythmTrack defines a rhythmical chords flow.

harmonyRhythmTrackRock defines a rhythmical chords flow, transposed one octave down.

additionTrack defines a random melody flow.

bassTrack defines a bass melody flow, randomly switching fingered bass or walking bass styles.

See Figure 4.11 for functions references.

4.6.1 Techniques used in interpretation

The interpretation techniques are implemented by functions contained in the file INTERPRETATIONTECHNIQUES.HS.

There's an auxiliary function named **octaveShift** which is a function of an integer and a musical flow, defining a flow made of the input flow transposed by the integer count of octaves.

Other functions are standardized to the form of a function of an random generator and a musical flow, defining a musical flow. Sometimes the random generator can be unused if not needed. These are the functions and how they define the output flow:

sopranoFlow defines the flow made of the soprano part[19] of the input flow.

randomMelodyFlow defines the flow made of the input flow where just one tone is randomly picked from each chord.

[19] The highest tones.

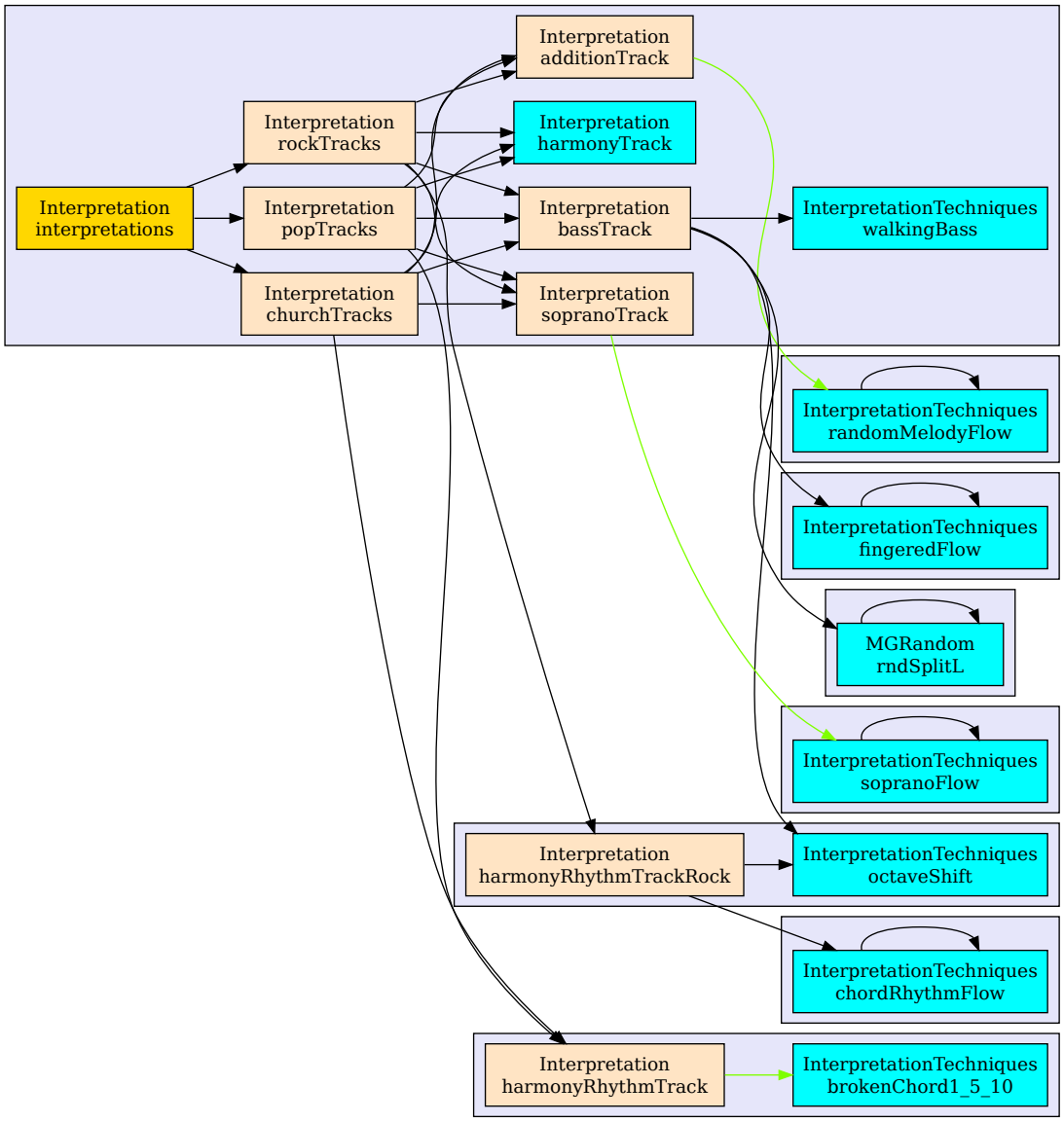


Figure 4.11: Functions references for the file Interpretation.hs.

fingeredFlow defines the flow made of the input flow where each chord is arranged to a sequence of its tones in the form: the first, the third, the second, the third, the second, ..., the sequence meeting the beat of the flow.

chordRhythmFlow defines the flow made of the input flow where each chord is repeated to meet the beat of the flow.

brokenChord1_5_10 defines the flow made of the input flow where each chord is repeated in alternating forms, the first form being the first and the third tone of the chord, the second form being the second tone transposed up by an octave; the repetition meeting the beat of the flow.

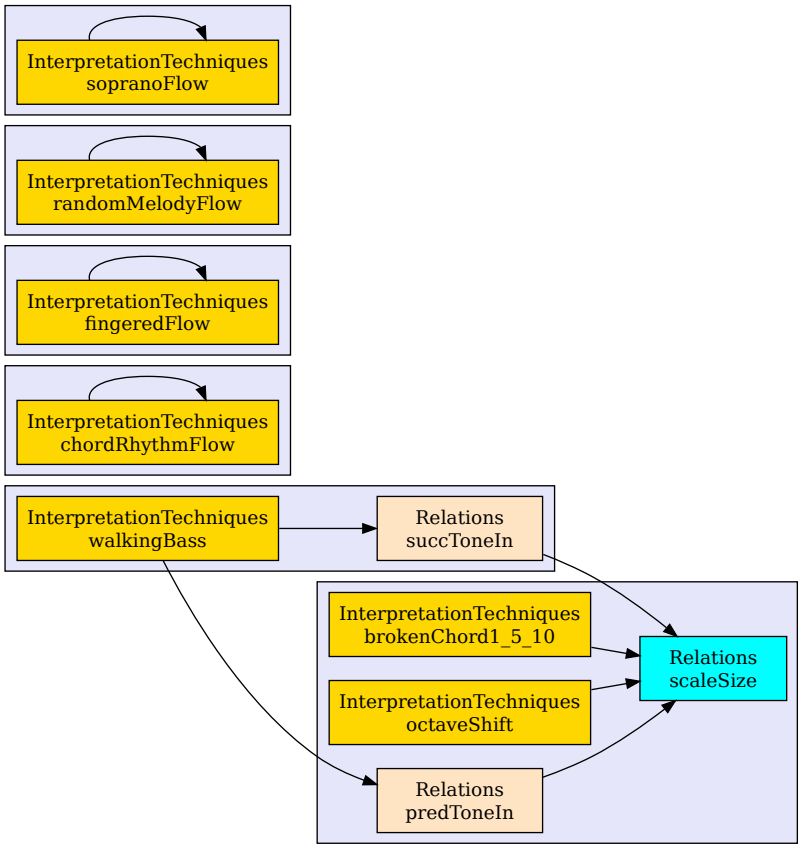


Figure 4.12: Functions references for the file InterpretationTechniques.hs.

walkingBass defines the flow made of the input flow where each chord is arranged to a sequence of tones with the first tone being the first tone of the chord, others going up and down in the scale of the flow to approach the first tone of the succeeding chord or the first tone of the actual chord if it's the last one; the sequence meeting the beat of the flow but twice the speed.

See Figure 4.12 for functions references.

4.7 MIDI

According to [6], MIDI, standing for Musical Instrument Digital Interface, is a protocol of communication enabling different electronic musical instruments and computers to interchange digital messages about music being played. Moreover, there's a MIDI file format which allows us to save these data. Having the well-defined file format we can operate with it in many ways, we'll use it as a final stage of our generated songs. Then we'll be able to play it, make sheet music of it and there's also a possibility of generated songs being just the first step in a chain of creation of music.

We'll use a MIDI file format which is composed of MIDI tracks where each represents a specific interpretation of harmony flow for a specific instrument. A track is a list of MIDI events where each is defined by a delay since the preceding event and a MIDI message.

4.7.1 MIDI messages

Let's have a look at some basic MIDI messages — as defined in the Haskell package HCodecs, see [7] — the generator program uses:

NoteOn tells what tone on what channel should be played with what volume, zero volume means muting the tone.

KeySignature tells how many sharps or flats a musical staff has and whether it's a major scale.

TimeSignature tells two numbers which are written at the beginning of a musical staff, indicating a time signature.

ChannelPrefix tells what channel is meant by following messages.

InstrumentName tells a name of an instrument, just informational message.

ProgramChange tells what channel has what MIDI instrument.

TempoChange tells a new tempo in an uncommon format of the value of microseconds per minute divided by beats per minute, see [8].

TrackEnd tells a MIDI track ends, all tracks should end with this message.

4.7.2 Implemented functions and monad in relation to MIDI output

These are the functions and the monad contained in the file Midi.hs:

toneMidi is a function of a tone and a volume, defining a MIDI event that starts playing the tone with the volume.

pauseMidi is a function of a duration, defining a MIDI event that moves the MIDI play forward by the duration.

flow2Midi is a function of a musical flow, defining a list of MIDI events playing this flow. Note that each part of flow is in fact an isolated chord, so there's no way to create pervading tones in one flow. Another note: volume set for the events is constant, set to the MIDI maximum 127.

keySignature is a function of a tone and a list of intervals, defining a message part of a MIDI event that represents a MIDI key signature of a scale made of the tone as the key and the intervals.

timeSignature is a function of an integer, defining a MIDI event that sets the time signature with the integer as the upper number, the lower number is 4.

eventsParam is a function of a channel number, a float and a list of MIDI events, defining an altered list where every tone playing event is set to be played on the given channel with volume being multiplied by the float.

midiTrack is a function of a channel number, an instrument name, an instrument MIDI number, a volume, a list of MIDI events, a beginning chord stating the scale of a track, and a tempo, defining a list of MIDI events fully covering the track made of the input list and adjusted to the rest of arguments.

makeTracks is a function of a list of definitions of MIDI tracks, a musical flow, a random generator and a tempo, defining a list of MIDI tracks where each is created by an interpretation of the flow as defined for the track with the use of a new random generator split from the input one, processed by the function midiTrack to meet the other arguments.

midiFile is a function of a list of MIDI tracks, defining a content of a MIDI file made of the tracks.

exportMidi is a monad which given a filename and a content of a MIDI file exports the content into a file with the name.

See Figure 4.13 for functions references.

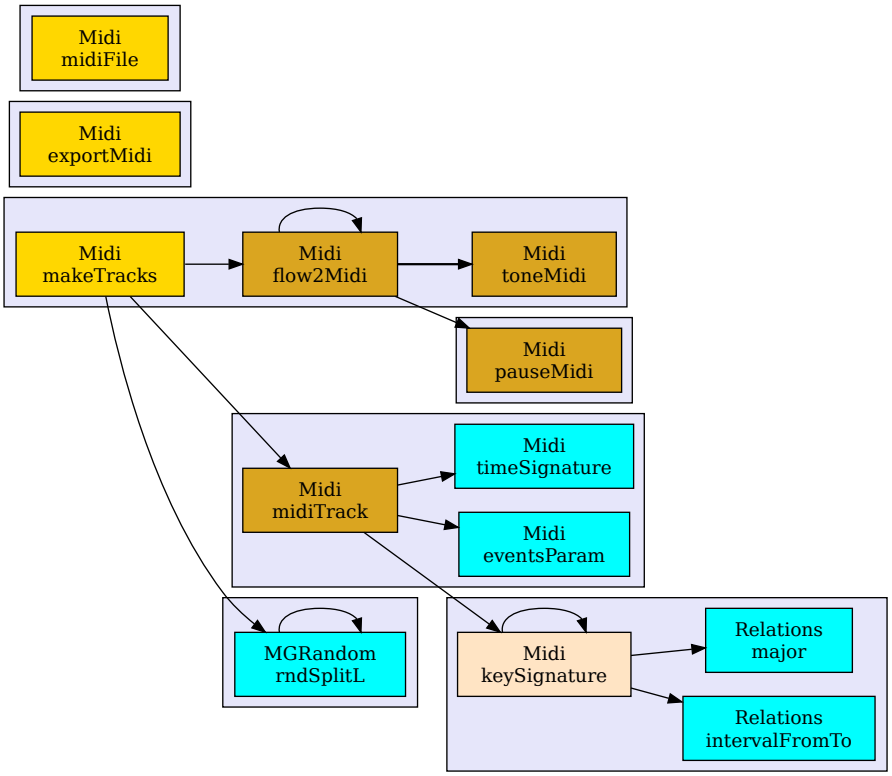


Figure 4.13: Functions references for the file Midi.hs.

Chapter 5

Comparison with similar software

MusGen is not the only software targeting computer music creation — according to [9] a lot of similar software exists and a lot of papers has been published. Let's take a look at some programs generating music in a relation to MusGen.

5.1 WolframTones

WolframTones[1] comes with an algorithm which, according to [10], works this way:

- There's a 2D matrix.
- Color (just black or white) of every cell in the matrix is defined in a relation to neighbour cells.
- A strip from the matrix is taken.
- One dimension from the strip is considered as a pitch, the other one as a time.
- Black cells from the strip are instrumentalized based on their position. Multiple instruments are supported, thus a melody or bass line can exist and the piece can be supplemented by percussion.

Compared to Musgen, it's more deterministic — just the rule of coloring is optional and bringing a space of possibilities — and it doesn't obey classical music rules, actually it produces output which is to be musical naturally but maybe not so much in a Western way.

5.2 FractMus

FractMus[2] is another program using mathematical formulas to create a song. Described as just a helping tool in the process of creation of music, it can be seen as an example of the group of

[1] Homepage http://tones.wolfram.com/.
[2] Homepage http://www.gustavodiazjerez.com/fractmus_overview.html.

programs which not tend to compose the final piece but just help the user use algorithms and computation in the process.

MusGen can be seen in this way as well — generated songs being in the MIDI format can be changed, taken as a basis for more elaborated songs.

5.3 C.P.U. Bach

C.P.U. Bach[3] has, according to [11], a lot of similarity with MusGen: it uses a so called "weighted exhaustive search" which seems to obey rules what can or cannot be produced, but it encompasses more things like "tendencies" — not so strict rules — and it also tries to create a song from parts which gives more musical sense. Actually, being more enhanced, MusGen would probably be very close to this.

5.4 Virtual music composer

Virtual music composer[4] is an example of programs which are similar in input and output to MusGen but there's no insight into their algorithms because they're not free software and there's no description of their principles released.

Virtual music composer takes few input parameters determining output's scale and style and just creates MIDI, like MusGen.

[3] No homepage, also works only on an uncommon platform called 3DO.

[4] Homepage http://www.virtualmusiccomposer.com/.

Chapter 6

Conclusion

After introducing necessary music theory terms in a brief way, the core of the thesis — the algorithm — was presented. Its development came from self-experimenting with both practical (MIDI) and theoretical (harmony theory, mainly taught by the great book [1]) stuff. Its principle is very easy and lightweight, thus not tending to create a complex masterpieces but mainly showing the possibility of creating a framework which can give algorithmic music and can be extended by more, especially harmonic, rules to get closer to classical Western music.

The implementation done in language Haskell proves the functionality of the algorithm and also describes it including used rules in a mathematical bright way, not seeing Haskell so much as a programming language but more as a definition language.

The comparison with similar software shows there's more people wanting to create and improve programs generating random music which leads to propositions of the next development: letting extending a count of harmony rules aside, melody and rhythm are two things where only cornerstones were put and thus can be developed almost any way. Also a too much of randomness may be more than sufferable making output too much chaotic — if the random seed repeats sometimes, we could be able to produce more accustomed songs with choruses and verses. Then, a lot of people would surely appreciate a possibility of engaging a percussion stuff.

Making MusGen was quite a fun and I'm looking forward to extend it and maybe cooperate with other people interested in music and computers.

Bibliography

[1] KOFROŇ, Jaroslav. *Učebnice harmonie*. 10th edition, 2006. 178 pages. ISBN 80-86385-14-0. 8, 12, 13, 14, 15, 16, 17, 18, 19, 46

[2] KALENDA, Václav. *Na klavír za několik týdnů* [online]. [cited 27th April 2011]. <http://hobby.idnes.cz/skola-hry-na-klavir-0o1-/hobby-domov.asp?o=0&klic=145025>. 20

[3] Types and Typeclasses. *Learn You a Haskell for Great Good!* [online]. [cited 25th March 2011]. <http://learnyouahaskell.com/types-and-typeclasses>. 25

[4] Randomness. *Learn You a Haskell for Great Good!* [online]. [cited 24th March 2011]. <http://learnyouahaskell.com/input-and-output#randomness>. 28

[5] Normal distribution — Generating values from normal distribution. *Wikipedia, the free encyclopedia* [online]. [cited 22nd March 2011]. <http://en.wikipedia.org/wiki/Normal_distribution#Generating_values_from_normal_distribution>. 29

[6] Musical Instrument Digital Interface. *Wikipedia, the free encyclopedia* [online]. [cited 18th March 2011]. <http://en.wikipedia.org/wiki/Musical_Instrument_Digital_Interface>. 41

[7] Codec.Midi. *HackageDB* [online]. [cited 18th March 2011]. <http://hackage.haskell.org/packages/archive/HCodecs/0.2/doc/html/Codec-Midi.html>. 41

[8] MIDI File Format. *The Sonic Spot* [online]. [cited 18th March 2011]. <http://www.sonicspot.com/guide/midifiles.html>. 41

[9] Algorithmic composition resources. *Algorithmic.net* [online]. [cited 29th April 2011]. <http://www.flexatone.net/algoNet/>. 44

[10] How WolframTones Works. *WolframTones* [online]. [cited 29th April 2011]. <http://tones.wolfram.com/about/how.html>. 44

[11] MEIER, Sidney K. — et al. System for real-time music composition and synthesis. *Google patents* [online]. [cited 29th April 2011]. <http://www.google.com/patents?vid=USPAT5496962>. 45

Appendix A

Exemplary use of the program

1. Installing needed packages:

```
$ cabal install hcodecs cmdargs
```

2. Compiling the program:

```
$ make
mkdir tmp
ghc --make Main -outputdir tmp -o musgen -O2
[ 1 of 11] Compiling Var              ( Var.hs, tmp/Var.o )
[ 2 of 11] Compiling Types            ( Types.hs, tmp/Types.o )
[ 3 of 11] Compiling ChanceHarmonyRhythm ( ChanceHarmonyRhythm.hs, tmp/ChanceHar
monyRhythm.o )
[ 4 of 11] Compiling Relations        ( Relations.hs, tmp/Relations.o )
[ 5 of 11] Compiling ChanceHarmony    ( ChanceHarmony.hs, tmp/ChanceHarmony.o )
[ 6 of 11] Compiling MGRandom         ( MGRandom.hs, tmp/MGRandom.o )
[ 7 of 11] Compiling InterpretationTechniques ( InterpretationTechniques.hs, tmp
/InterpretationTechniques.o )
[ 8 of 11] Compiling Midi             ( Midi.hs, tmp/Midi.o )
[ 9 of 11] Compiling Interpretation   ( Interpretation.hs, tmp/Interpretation.o
)
[10 of 11] Compiling Flow             ( Flow.hs, tmp/Flow.o )
[11 of 11] Compiling Main             ( Main.hs, tmp/Main.o )
Linking musgen ...
```

3. Asking for help:

```
$ ./musgen -?
MusGen, version date: 2011-03-16
```

```
musgen [OPTIONS] [SONG_NAME]

Common flags:
  -k --key[=MIDI_TONE]          Key of harmony
  -s --scale[=major|minor]      Scale of harmony
  -b --beats[=INT]              Beats per measure
  -t --tempo[=INT]              Quarter notes per minute
  -m --minmeasures[=INT]        Minimal number of measures
  -i --interpretation[=STYLE]   Style of interpretation
  -n --new                      Generate new flow
  -? --help                     Display help message
  -V --version                  Print version information

Generates a song fulfilling given parameters, song's name is "song" by default.
Available styles of interpretation are: church, pop, rock.
Output is SONG_NAME.midi with playable MIDI data and SONG_NAME.flow with
reusable information about harmony flow.
```

4. Generating a song:

```
$ ./musgen -k62 -smajor -b3 -t130 -m4 -ipop
([54,57,62,69],b0,d4,r12)
([57,61,64],b4,d6,r8)
([57,61,66],b10,d2,r2)
([55,59,62,67],b0,d2,r12)
([54,57,62,69],b2,d10,r10)
([59,62,66],b0,d2,r12)
([57,61,64,69],b2,d2,r10)
([61,64,67],b4,d2,r8)
([57,62,66,69],b6,d4,r6)
([61,64,67],b10,d2,r2)
([59,62,66,71],b0,d2,r12)
([61,66,69],b2,d2,r10)
([59,62,67,71],b4,d8,r8)
([59,62,67],b0,d10,r12)
([57,61,66,69],b10,d2,r2)
([61,64,67],b0,d10,r12)
([57,61,64,69],b10,d2,r2)
([59,62,66],b0,d12,r12)
([55,61,64,67],b0,d6,r12)
([57,62,66],b6,d6,r6)
```

```
Flow generated.
MIDI generated.
```

5. Generating a sheet music:

```
$ ./midi2pdf.sh song.midi
```