A Procedural Approach to Representing Scales

This document is intended to describe a procedural approach to representing scales, and to present a computational manifestation of the approach. This computational manisfestation is what my fingers typed as a first approximation to the scalar infrastructure to an implementation of MxM in Euterpea (Haskell).

More or less for the fun of it, some conceptual material pertaining to cognitive musicology is presented as a "lead-in" to the computational material.

Music? Melodies? Notes?

A Multidimensional Definition of Music

Music is the creation, reification, and appreciation of sonic semiotic streams. Deconstructing the definition ...

- **Creation** is the action or process of bringing something into existence. (composition)
- **Reification** is making something real, bringing something into being, or making something concrete. (performance)
- **Appreciation** is the recognition and enjoyment of the good qualities of something. (listening)
- **Sonic** refers to the fact that music is communicated through sound, and the term is intended to suggest a nonverbal connotation.
- **Semiotic** has to do with meaning. Unless the sound is meaningful in some way (e.g., aesthetically meaningful, emotionally meaningful, intellectually meaningful), it doesn't rise to the level of music.
- **Streams** are temporality ordered sequences of events - music is diachronic in nature.

A Working Definition of Music

Music is the set of all phenomena surrounding musical notes in sequence or combination.

Simple Conception of a Melody

A **melody** is a sequence of notes. If that is too broad a definition for certain purposes, it is easy to restrict the definition by placing constraints on the sequence of notes. For example:

- A random melody is a sequence of random notes.
- A tonal melody is a sequence of notes consistent with the principles of tonality.
- An atonal melody is a sequence of notes consistent with the principles of atonality.
- A modal melody is a sequence of notes consistent with modal music.

Simple Conception of a Note

A **note** is a set of salient sonic properties. **We will generally consider a note to simply be a pitch/duration pair.** This leaves a great deal unstated! For example, the pitch will generally be contextualized within some scale, and the duration will generally be understood with respect to the beat (unit of time). Furthermore, any number of other unstated properties might be incorporated into our conception of note, depending on interests or needs or inclinations. For example:

- the note will be sounded with some particular volume
- the note will be heard with in terms of some timbre (instrument)

Cognitive Musicology and Distributed Cognition

Cognitive Musicology?

Cognitive Musicology is the set of all phenomena surrounding the computational modeling of musical thought and action. The focus on computational modeling suggests an emphasis on knowledge representation, which implies that the roots of cognitive musicology lie in artificial intelligence. The thought and action tag reminds us that the field also has historical ties to the psychology of music. Since the field investigates parallels between music and language, and incorporates biologically inspired models of computation in the form of neural networks and evolutionary programs, it is clear that cognitive musicology is a highly interdisciplinary field which lies within the bounds of cognitive science. According to Otto Laske, one of the foremost champions of the field, the goal of cognitive musicology is to computationally model musical knowledge with the aim of advancing understanding of music from the perspective of engagement in musical activities (e.g., composing, performing, listening, improvising). The computer is central to the enterprise since computer modeling invites comparison to cognitive processing, provides an exacting medium in which to formulate and test executable theories, and facilitates the collection of data for analysis.

The field of cognitive musicology (AI and music) overlaps considerably with the field of music cognition (psychology of music), its better known relation. It is a difference in methodological emphasis that distinguishes the two fields far more than the problems considered.

One potentially productive way to frame the field of cognitive musicology is to consider it through the lens of distributed cognition as the art and science of inventing cognitive artifacts that afford enhanced musical experiences, particularly with respect to composition, performance, and listening.

Distributed Cognition and Cognitive Artifacts in Cognitive Musicology

It is interesting to frame cognitive musicology within the field of distributed cognition.

Distributed Cognition?

According to Roy Pea, **distributed cognition** is the conception of cognition as something accomplished through collaborative interactions involving people and artifacts, as opposed to something possessed by individuals in isolation.

Cognitive Artifacts?

According to Don Norman, a **cognitive artifact** is an artificial device that maintains, displays, or operates upon information in order to serve a representational function and affect human cognitive performance. Put more simply, a cognitive artifact is a man-made object that helps one to think.

A Perspective on DC and CAs in CM

Carefully designed, well-crafted cognitive artifacts for musical activity may be essentially what Todd Machover (reference is from Tim Rowe) was referring to when talking about the possibility and consequence of placing focus on the mental and emotional activities of music, rather than the physical skills required to play a musical instrument:

> Traditional instruments are hard to play. It takes a long time to [acquire] physical skills which aren't necessarily the essential qualities of making music. It takes years just to get good tone quality on a violin or to play in tune. If we could find a way to allow people to spend the same amount of concentration and effort on listening and thinking and evaluating the difference between things and thinking about how to communicate musical ideas to somebody else, how to make music with somebody else, it would be a great advantage. Not only would the general level of musical creativity go up, but you'd have a much more aware, educated, sensitive, listening, and participatory public.

Example Musical Cognitive Artifacts

1. The Virtual Piano (`https://www.onlinepianist.com/virtual-piano`)
2. Chrome Music Lab (`https://musiclab.chromeexperiments.com/`)
3. GarageBand
4. EasyABC
5. MxM (Music Exploration Machine)

Music Knowledge Representation

What is a Knowledge Representation (KR)?

A **knowledge representation (KR)** is a set of conventions for representing knowledge. Some examples of knowledge representations:

- Natural language (English, Russian, French)
- Computer programming languages (Lisp, Prolog, Java)
- Classic AI knowledge representations (Frames, Scripts, Productions)

Knowledge representations vary in terms of their scope of expression, their naturalness of expression, their precision of expression, and many other things.

It is worth emphasizing the fact that knowledge representation is a cornerstone (inherited from its traditional AI roots) of cognitive science, and hence cognitive musicology. If you are not focussed on knowledge representation, you are not engaged in these fields!

Music Knowledge Representation Languages (MKRLs)

Domain specific knowledge calls for domain specific knowledge representations. Some examples of music knowledge representations:

- common music notation (sheet music)
- MIDI
- JFugue
- ABC notation
- General purpose programming languages, like Lisp
- Domain (music) specific programming languages, like Euterpea and Musical Clay in MxM

These musical knowledge representations, like all knowledge representations, vary in terms of their scope of expression, their naturalness of expression, their precision of expression, and many other things. Among the desirable characteristics of a music knowledge representation language are theses:

- Executability
- Comprehesibility
- Expressivity
- Structural integrity

What do these words mean?

- **Executability** refers to whether or not a KR can be run on a computing machine when it is operating in mindless mode.

- **Comprehesibility** refers to the ease with which a typical human can understand the knowledge is being represented (in various ways at various levels of abstraction.
- **Expressivity** in a representation refers to the degree to which concepts (musical concepts) can be represented with precision, clarity, and flexibility.
- **Structural integrity** in a representation refers to the degree to which hierarchy (structure) is captured by the representation.

How do each of the following music knowledge representation languages rate in terms of the four desired characteristics?

- common music notation (sheet music)
- MIDI
- JFugue
- ABC notation
- Lisp
- Euterpea
- Musical Clay in MxM

Haskell Representation of MxM State

Musical **infrastructure** for the representation of melodies is introduced conceptually and rendered computatioonally in this section, a little bit at a time.

Modes

A **mode** is simply a sequence of numbers, in which a number represents the distance in semitones between one note and the next in a scale. Given a pitch, the sequence can be used to generate a scale. Note that a mode by itself does not contain enough information to suggest the play of any particular notes.

Here is Haskell code for a dozen representative modes:

```
major_mode = [2,2,1,2,2,2,1] :: [Int]
minor_mode = [2,1,2,2,1,2,2] :: [Int]

wholetone_mode = [2,2,2,2,2,2] :: [Int]
chromatic_mode = [1,1,1,1,1,1,1,1,1,1,1,1] :: [Int]

mymajorblues_mode = [2,1,1,3,2,3] :: [Int]
myminorblues_mode = [3,2,1,1,3,2] :: [Int]

major7th_mode = [4,3,4,1] :: [Int]
minor7th_mode = [3,4,3,2] :: [Int]
dom7th_mode = [4,3,3,2] :: [Int]
dim7th_mode = [3,3,3,3] :: [Int]

x1_mode = [2,1,2,1,2,1,2,1] :: [Int]
x2_mode = [3,3,2,2,1,1] :: [Int]
```

Keys

A **key** is a pitch class together with a mode. A key defines a scale in terms of a sequence of pitch classes. Note that a key by itself does not contain enough information to suggest the play of any particular notes. (You need pitches, not merely pitch classes, in order to sonically render notes.)

Here is Haskell code for two dozen representative keys:

```
c_major_key = ("C",major_mode)
d_major_key = ("D",major_mode)

c_minor_key = ("C",minor_mode)
d_minor_key = ("D",minor_mode)

c_wholetone_key = ("C",wholetone_mode)
d_wholetone_key = ("D",wholetone_mode)
```

```
c_chromatic_key = ("C",chromatic_mode)
d_chromatic_key = ("D",chromatic_mode)

c_mymajorblues_key = ("C",mymajorblues_mode)
d_mymajorblues_key = ("D",mymajorblues_mode)

c_myminorblues_key = ("C",myminorblues_mode)
d_myminorblues_key = ("D",myminorblues_mode)

c_major7th_key = ("C",major7th_mode)
d_major7th_key = ("D",major7th_mode)

c_minor7th_key = ("C",minor7th_mode)
d_minor7th_key = ("D",minor7th_mode)

c_dom7th_key = ("C",dom7th_mode)
d_dom7th_key = ("D",dom7th_mode)

c_dim7th_key = ("C",dim7th_mode)
d_dim7th_key = ("D",dim7th_mode)

c_x1_key = ("C",x1_mode)
d_x1_key = ("D",x1_mode)

c_x2_key = ("C",x2_mode)
d_x2_key = ("D",x2_mode)
```

MxM State

By adding a few more items of information to a key you will be in a position to play a note! Essentially you need to adda **degree** to select a pitch class from a key, a deregister value to select one particular pitch from the pitch class, and a duration for the note. Rather than explicitly incorporating a register value, the MxM state incorporates a midi value. This is a bit of overkill in some ways, but a midi value implicitly references a register value. The reason for this is largely one of convenience. The midi value can be useful for rendering notes on a machine.

I have chosen to represent a state as a 5 tuple, whose components represent:

1. A midi value
2. A key (2-tuple consisting of a pitch class and a mode)
3. A degree (integer between 1 and the size of the mode)
4. A duration (real number)
5. A hint regarding pitch spelling ("#" means default to sharp; "b" means default to flat)

All that said, here is Haskel code for a couple of dozen "initial states", states that can be used to commence a melodic composition:

```
c_major_state = (60,c_major_key,1,1.0,"#") :: (Int,([Char],[Int]),Int,Float,[Char])
d_major_state = (62,d_major_key,1,1.0,"#") :: (Int,([Char],[Int]),Int,Float,[Char])
```

```
c_minor_state = (60,c_minor_key,1,1.0,"b") :: (Int,([Char],[Int]),Int,Float,[Char])
d_minor_state = (62,d_minor_key,1,1.0,"b") :: (Int,([Char],[Int]),Int,Float,[Char])

c_wholetone_state = (60,c_wholetone_key,1,1.0,"#") :: (Int,([Char],[Int]),Int,Float,[Char])
d_wholetone_state = (62,d_wholetone_key,1,1.0,"#") :: (Int,([Char],[Int]),Int,Float,[Char])

c_chromatic_state = (60,c_chromatic_key,1,1.0,"#") :: (Int,([Char],[Int]),Int,Float,[Char])
d_chromatic_state = (62,d_chromatic_key,1,1.0,"#") :: (Int,([Char],[Int]),Int,Float,[Char])

c_mymajorblues_state = (60,c_mymajorblues_key,1,1.0,"b") :: (Int,([Char],[Int]),Int,Float,[Char])
d_mymajorblues_state = (62,d_mymajorblues_key,1,1.0,"b") :: (Int,([Char],[Int]),Int,Float,[Char])

c_myminorblues_state = (60,c_myminorblues_key,1,1.0,"b") :: (Int,([Char],[Int]),Int,Float,[Char])
d_myminorblues_state = (62,d_myminorblues_key,1,1.0,"b") :: (Int,([Char],[Int]),Int,Float,[Char])

c_major7th_state = (60,c_major7th_key,1,1.0,"#") :: (Int,([Char],[Int]),Int,Float,[Char])
d_major7th_state = (62,d_major7th_key,1,1.0,"#") :: (Int,([Char],[Int]),Int,Float,[Char])

c_minor7th_state = (60,c_minor7th_key,1,1.0,"b") :: (Int,([Char],[Int]),Int,Float,[Char])
d_minor7th_state = (62,d_minor7th_key,1,1.0,"b") :: (Int,([Char],[Int]),Int,Float,[Char])

c_dom7th_state = (60,c_dom7th_key,1,1.0,"b") :: (Int,([Char],[Int]),Int,Float,[Char])
d_dom7th_state = (62,d_dom7th_key,1,1.0,"b") :: (Int,([Char],[Int]),Int,Float,[Char])

c_dim7th_state = (60,c_dim7th_key,1,1.0,"b") :: (Int,([Char],[Int]),Int,Float,[Char])
d_dim7th_state = (62,d_dim7th_key,1,1.0,"b") :: (Int,([Char],[Int]),Int,Float,[Char])

c_x1_state = (60,c_x1_key,1,1.0,"#") :: (Int,([Char],[Int]),Int,Float,[Char])
d_x1_state = (62,d_x1_key,1,1.0,"#") :: (Int,([Char],[Int]),Int,Float,[Char])

c_x2_state = (60,c_x2_key,1,1.0,"#") :: (Int,([Char],[Int]),Int,Float,[Char])
d_x2_state = (62,d_x2_key,1,1.0,"#") :: (Int,([Char],[Int]),Int,Float,[Char])
```

Some Musical Materails

Some "Static" Materials

Here are two little Clay programs, each of which represents the essence of a simple melody:

```
demo = "p rp rp p rp rp x2 p s2 s2 p lp p lp p lp p lp x2 x2 p s2"

lt1 = "rp rp p lp p rp p lp lp p rp p lp p rp x2 p s2 lp"
lt2 = "rp rp p lp p rp p lp lp p rp x2 p p s2 lp"
lt3 = lt1
lt4 = "rp rp p lp p rp p lp p lp x2 p p s2"
littletune = lt1 ++ " " ++ lt2 ++ " " ++ lt3 ++ " " ++ lt4
```

A Haskell function called `clayString2abcString` can be used to render a clay program as abc code, provided it is given an initial state to accompany the Clay program. Here, by way of example, are a couple of renderings of the `demo` program:

```
MxM> clayString2abcString demo c_major_state
"C1 E1 G2 G/2 F/2 E/2 D/2 C2"
MxM> clayString2abcString demo d_major_state
"D1 ^F1 A2 A/2 G/2 ^F/2 E/2 D2"
MxM>
```

To actually hear these renderings of the `demo` program, you could type:

```
MxM> clayplay demo c_major_state
MxM> clayplay demo d_major_state
MxM>
```

Some "Dynamic" State

Here are three little Haskell programs, each of which will write a Clay program:

```
gen_asc_scale mode n = clayList2clayString ( add_plays code )
  where code = make_list ( length mode ) "rp"

gen_dsc_scale mode n = clayList2clayString ( add_plays code )
  where code = make_list ( length mode ) "lp"

gen_stepwise_random_walk mode n = clayList2clayString ( add_plays ( f ++ r ) )
  where f = changes ( n * ( length mode ) )
        r = inversion f
```

The `gen_asc_scale` program will generate a Clay program to play an `n` octave scale ascending consistent with `mode`. The `gen_dsc_scale` program will generate a Clay program to play an `n` octave descending scale consistent with `mode`. The `gen_stepwise_random_walk` program will generate a random walk in the given mode of length equal to twice the given number, which is palindromic, and thus is invariant with respect to pitch. The following demo may be helpful in clarifying these words:

```
MxM> gen_asc_scale major_mode 1
"p rp p rp p rp p rp p rp p rp p rp p"
MxM> gen_asc_scale chromatic_mode 1
"p rp p rp p rp p rp p rp p rp p rp p rp p rp p rp p rp p rp p rp p"
MxM> gen_dsc_scale minor_mode 1
"p lp p lp p lp p lp p lp p lp p lp p"
MxM> gen_dsc_scale wholetone_mode 1
"p lp p lp p lp p lp p lp p lp p"
MxM> gen_stepwise_random_walk major_mode 2
"p rp p lp p lp p rp p lp p lp p lp p rp p lp p lp p lp p lp p rp p rp p
lp p rp p rp p lp p rp p rp p rp p lp p rp p rp p rp p rp p lp p lp p"
MxM> gen_stepwise_random_walk wholetone_mode 2
"p rp p lp p lp p rp p lp p lp p lp p rp p lp p lp p lp p lp p lp p rp p
rp p lp p rp p rp p rp p lp p rp p rp p rp p rp p"
MxM>
```

Perhaps another demo is in order?

```
MxM> clayString2abcString ( gen_asc_scale major_mode 1 ) c_major_state
"C1 D1 E1 F1 G1 A1 B1 c1"
MxM> clayString2abcString ( gen_asc_scale major_mode 2 ) c_major_state
"C1 D1 E1 F1 G1 A1 B1 c1 d1 e1 f1 g1 a1 b1 c'1"
MxM> clayString2abcString ( gen_asc_scale major_mode 1 ) d_major_state
"D1 E1 ^F1 G1 A1 B1 ^c1 d1"
MxM> clayString2abcString ( gen_asc_scale major_mode 2 ) d_major_state
"D1 E1 ^F1 G1 A1 B1 ^c1 d1 e1 ^f1 g1 a1 b1 ^c'1 d'1"
MxM> clayString2abcString ( gen_asc_scale minor_mode 1 ) c_minor_state
"C1 D1 _E1 F1 G1 _A1 _B1 c1"
MxM> clayString2abcString ( gen_asc_scale minor_mode 2 ) c_minor_state
"C1 D1 _E1 F1 G1 _A1 _B1 c1 d1 _e1 f1 g1 _a1 _b1 c'1"
MxM> clayString2abcString ( gen_asc_scale minor_mode 1 ) d_minor_state
"D1 E1 F1 G1 A1 _B1 c1 d1"
MxM> clayString2abcString ( gen_asc_scale minor_mode 2 ) d_minor_state
"D1 E1 F1 G1 A1 _B1 c1 d1 e1 f1 g1 a1 _b1 c'1 d'1"
MxM> clayString2abcString ( gen_asc_scale wholetone_mode 1 ) c_wholetone_state
"C1 D1 E1 ^F1 ^G1 ^A1 c1"
MxM> clayString2abcString ( gen_asc_scale wholetone_mode 2 ) c_wholetone_state
"C1 D1 E1 ^F1 ^G1 ^A1 c1 d1 e1 ^f1 ^g1 ^a1 c'1"
MxM> clayString2abcString ( gen_asc_scale dim7th_mode 1 ) c_dim7th_state
"C1 _E1 _G1 A1 c1"
MxM> clayString2abcString ( gen_asc_scale dim7th_mode 2 ) c_dim7th_state
"C1 _E1 _G1 A1 c1 _e1 _g1 a1 c'1"
MxM> clayString2abcString ( gen_asc_scale dim7th_mode 3 ) c_dim7th_state
"C1 _E1 _G1 A1 c1 _e1 _g1 a1 c'1 _e'1 _g'1 a'1 c''1"
MxM> clayString2abcString ( gen_dsc_scale chromatic_mode 1 ) c_chromatic_state
"C1 B,1 ^A,1 A,1 ^G,1 G,1 ^F,1 F,1 E,1 ^D,1 D,1 ^C,1 C,1"
MxM> clayString2abcString ( gen_dsc_scale myminorblues_mode 2 ) c_myminorblues_state
```

```
"C1 _B,1 G,1 _G,1 F,1 _E,1 C,1 _B,,1 G,,1 _G,,1 F,,1 _E,,1 C,,1"
MxM>
```

Of course, you can hear these scales by using the `clayplay` function.

The `clayplay` Function

The `clayplay` function will play melody determined by the string representation of a Clay program and an MxM state. The heart of this function is a transformer that changes the given Clay program, contextualized by the accompanying state, into a string of abc commands. The abc string returned is then converted to Euterpea code, which is played by the native Euterpea `play` command.

The transformation of Clay code to abc notation can be monitored by running the `clayString2abcStringMonitor` function. To illustrate:

```
MxM> clayString2abcStringMonitor demo c_major_state

Input = "p rp rp p rp rp x2 p s2 s2 p lp p lp p lp p lp x2 x2 p s2"

Applying "p"
((60,("C",[2,2,1,2,2,2,1]),1,1.0),"C1")
Applying "rp"
((62,("C",[2,2,1,2,2,2,1]),2,1.0),"")
Applying "rp"
((64,("C",[2,2,1,2,2,2,1]),3,1.0),"")
Applying "p"
((64,("C",[2,2,1,2,2,2,1]),3,1.0),"E1")
Applying "rp"
((65,("C",[2,2,1,2,2,2,1]),4,1.0),"")
Applying "rp"
((67,("C",[2,2,1,2,2,2,1]),5,1.0),"")
Applying "x2"
((67,("C",[2,2,1,2,2,2,1]),5,2.0),"")
Applying "p"
((67,("C",[2,2,1,2,2,2,1]),5,2.0),"G2")
Applying "s2"
((67,("C",[2,2,1,2,2,2,1]),5,1.0),"")
Applying "s2"
((67,("C",[2,2,1,2,2,2,1]),5,0.5),"")
Applying "p"
((67,("C",[2,2,1,2,2,2,1]),5,0.5),"G/2")
Applying "lp"
((65,("C",[2,2,1,2,2,2,1]),4,0.5),"")
Applying "p"
((65,("C",[2,2,1,2,2,2,1]),4,0.5),"F/2")
Applying "lp"
((64,("C",[2,2,1,2,2,2,1]),3,0.5),"")
Applying "p"
((64,("C",[2,2,1,2,2,2,1]),3,0.5),"E/2")
Applying "lp"
((62,("C",[2,2,1,2,2,2,1]),2,0.5),"")
Applying "p"
((62,("C",[2,2,1,2,2,2,1]),2,0.5),"D/2")
Applying "lp"
((60,("C",[2,2,1,2,2,2,1]),1,0.5),"")
Applying "x2"
```

```
((60,("C",[2,2,1,2,2,2,1]),1,1.0),"")
Applying "x2"
((60,("C",[2,2,1,2,2,2,1]),1,2.0),"")
Applying "p"
((60,("C",[2,2,1,2,2,2,1]),1,2.0),"C2")
Applying "s2"
((60,("C",[2,2,1,2,2,2,1]),1,1.0),"")

Output = "C1 E1 G2 G/2 F/2 E/2 D/2 C2"

MxM>
```

To actually hear the melody, simply type:

```
MxM> clayplay demo c_major_state
```

The Scalething Program

Description of the Program

The "scale thing" program will play the scale for one octave ascending, will play the scale ofr one octave descenidng, and then will play a random walk in the scale of length equal to 8 times the scale's length, beginning and ending on the same mid-register tonic. (Actually, the random walk is far from random. The first half is generated randomly, but the second half is merely an "inversion" of the first half. Thus, the walk is palindromic (although most listeners probably won't pick up on that fact.))

Haskell Definition of the Program

```
gen_scalething mode = one ++ " " ++ two ++ " " ++ three
  where one = ( double_time ( gen_asc_scale mode 1 ) )
        two = ( double_time ( gen_dsc_scale mode 1 ) )
        three = ( double_time ( gen_stepwise_random_walk mode 4 ) )

gen_asc_scale mode n = clayList2clayString ( ( add_plays ascCode ) ++ dscCode )
  where ascCode = make_list ( n * ( length mode ) ) "rp"
        dscCode = make_list ( n * ( length mode ) ) "lp"

gen_dsc_scale mode n = clayList2clayString ( ( add_plays dscCode ) ++ ascCode )
  where dscCode = make_list ( n * ( length mode ) ) "lp"
        ascCode = make_list ( n * ( length mode ) ) "rp"

gen_stepwise_random_walk mode n = clayList2clayString ( add_plays ( f ++ r ) )
  where f = changes ( n * ( length mode ) )
        r = inversion f

double_time program = "s2 " ++ program ++ " x2"
```

Demo of Making Midi Files for Scalething Exercises

Here is a demo that involves writing a number of "scalething" exercises as midi files:

```
MxM makemidi ( gen_scalething major_mode ) c_major_state "c_major_scalething.mid"
MxM makemidi ( gen_scalething minor_mode ) d_minor_state "d_minor_scalething.mid"
MxM makemidi ( gen_scalething wholetone_mode ) d_wholetone_state "d_wholetone_scalething.mid"
MxM makemidi ( gen_scalething wholetone_mode ) d_wholetone_state "d_wholetone_scalething.mid"
MxM makemidi ( gen_scalething wholetone_mode ) d_myminorblues_state "d_myminorblues_scalething.mid"
MxM makemidi ( gen_scalething x1_mode ) c_x1_state "c_x1_scalething.mid"
MxM makemidi ( gen_scalething x2_mode ) c_x2_state "c_x2_scalething.mid"
MxM makemidi ( gen_scalething major7th_mode ) c_major7th_state "c_major7th_scalething.mid"
```

```
MxM makemidi ( gen_scalething dim7th_mode ) c_dim7th_state "c_dimth_scalething.mid"
MxM makemidi ( gen_scalething chromatic_mode ) c_chromatic_state "c_chromatic_scalething.mid"
MxM
```

Webpage Containing Sound Files for Some Scalething Exercises

https://www.cs.oswego.edu/~blue/arts_forever/music/MxM/Scalething/index.html