CHAPTER EIGHT

The Architecture of Musicat

Previous chapters have described, at a high level, some of the particular aspects of musical listening that Musicat is intended to model. This chapter presents the architecture of the system and describes its inner workings in some detail. The ultimate reference for how Musicat works is the even lower level of the C♯ source code itself[11], but as a cognitive scientist, I generally find the ideas behind Musicat to be more interesting than the technical details of its implementation. This chapter aims for a level of detail somewhere above the level of source code while still providing enough information for someone to try implementing a similar system based on these ideas.

Optimistically, I think that some of the lowest-level details do not matter — I had to make many rather arbitrary decisions along the way when implementing my ideas, and I believe that many of them are not critical, in comparison with overall organization of the program. For instance, I chose a default "urgency" value for many codelets to be 20 (out of 100). But why not 30, or 10, or even 25, 21, or 20.17? Musicat has a large number of such parameters, not to mention other small decisions about how each individual codelet works.

[11] Musicat is written in the programming language C♯, pronounced "C sharp". This language does not inherently have anything to do with music — the fact that I used a programming language named after a musical note to write a program that models music listening is simply an amusing coincidence.

Sometimes it bothers me that I have had to make so many decisions during implementation, but I have taken comfort in the fact that other FARG programs such as Copycat have worked fine, with their similarly large numbers of parameters. Another surprising source of optimism was a brief discussion I had with Doug Lenat, when I asked him about choosing parameters in his Automated Mathematician (AM) program. He told me that he simply chose reasonable-sounding values for various weights and other parameters, and AM essentially functioned in the same manner across a range of parameter values (Lenat 2009). I suspect that this is also the case for Musicat, especially considering its stochastic architecture. Its overall plan of attack for making sense of music is critical, but hopefully the way it "listens" is stable with respect to minor differences in implementation.

Overview: Mental Processes to Model

Before describing the architecture, I first review some of the key mental processes Musicat is intended to model: internal representation of music, the flow of time, importance of rhythmic patterns, grouping structure, tension and resolution, tonal pitch structure, expectation, and last but of course not least, musical analogy-making.

- **Internal representation**: music is represented in human minds in a high-level way that throws out the raw perceptual details of incoming sound waves but maintains higher-level features in memory. Musicat is given music at the note level (which is already quite high-level compared with sound waves) and it generates structured representations of what it "hears". These representations, and the way they evolve in real-time, constitute the most significant "output" of Musicat (as seen in Chapter 5). Even though they represent *internal* cognition, the groups and analogies formed by Musicat are the objects of interest Musicat makes available *externally* for us to observe.

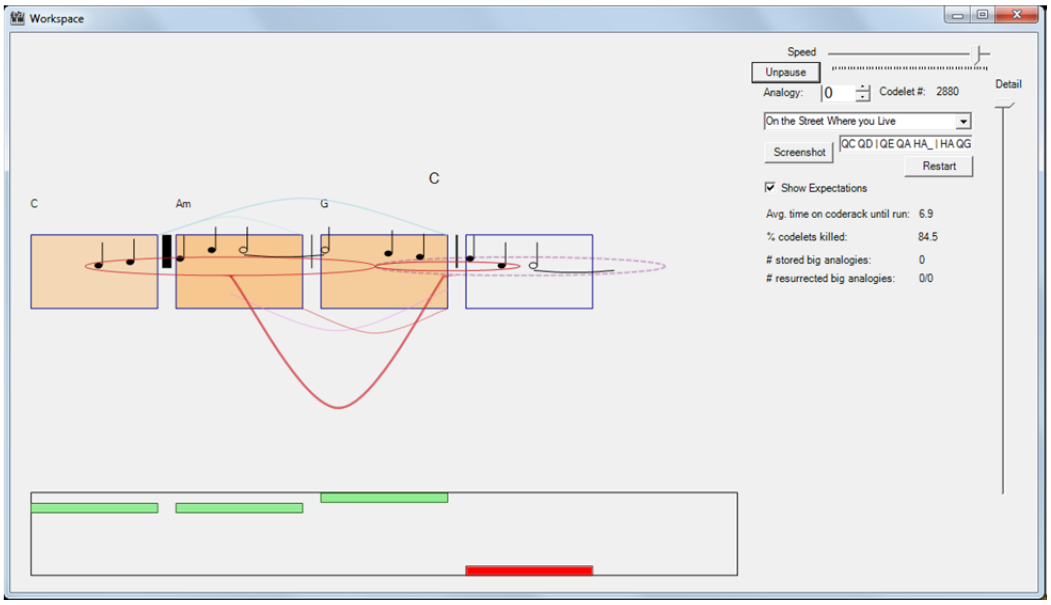
- **Flow of time**: music, in contrast to some other arts, is critically dependent on the passing of time. The human short-term memory system, in particular, places constraints on how we can hear and understand music. Musicat is provided notes one at a time in simulated real-time, and is limited in its ability to reinterpret notes and phrases that have faded into the past, outside the focus of short-term memory.
- **Rhythmic patterns**: rhythm is a particularly primal aspect of music, and likely contributes more to recognition of musical patterns than pitch (melodies are often recognizable by their rhythmic pattern alone, whereas people have more difficulty recognizing a pitch contour devoid of rhythm).
- **Grouping structure:** Gestalt grouping principles apply to music listening, causing notes to be perceptually grouped in much the same way as objects are grouped together in visual perception. Grouping in music is hierarchical and helps people understand chunked larger-scale structures that would otherwise exceed working-memory capacity.
- **Tension and resolution**: one of the strong clues to grouping structure in music, as well as an important way in which music evokes emotions, is the frequent increase and subsequent relaxation of musical tension. For example, cadences typically involve harmonic tension followed by relaxation to a stable chord, and entire phrases often take the shape of an arch: pitches rise, increasing tension, and then relax via downward motion at the end of a phrase. Musicat uses tension and resolution of various parameters as clues to grouping structure.

- **Tonal pitch structure**: pitch contour is important to melody understanding, and in Western tonal music in particular, the qualities of individual pitches in a harmonic context are quite salient. In Musicat, tonal structure is modeled through a combination of pitch contour and analysis of the stability or the tension inherent in pitches in tonal contexts.
- **Expectation**: grouping, and in general the understanding of rhythmic structure, is aided by expectation. For example, if an eight-measure phrase is established, listeners generally expect the next phrase also to span eight measures. These expectations help guide Musicat's generation of groups.
- **Analogy-making**: as discussed in previous chapters, musical perception (like the rest of perception) is critically dependent on analogies at many different levels. Analogies drive a great deal of Musicat's top-down processing.

Major Components

User Interface

Because the main output of Musicat is the program's internal representation of what it has "heard", the user interface is a significant component of the program. Two versions of the user interface were developed: a stand-alone program (Figure 8.1) and an "add-in" to the Visual Studio 2010 development environment for rapid feedback during development (Figure 8.2).



Analogy: 0 Codelet #: 2880

On the Street Where you Live

QC QD | QE QA HA_ | HA QG

[✓] Show Expectations

Avg. time on coderack until run: 6.9

% codelets killed: 84.5

stored big analogies: 0

resurrected big analogies: 0/0

Figure 8.1: Standard user interface for Musicat.

Standard Interface for Musicat

Earlier chapters explained how to interpret Musicat's output. In this section I discuss the input to the program. The primary user-provided input is the melody, represented using a simple text format. A set of characters such as `QG#5` represents a single note (in this case, a quarter note on the pitch G♯, in octave 5). Notes are separated by spaces, and measures are separated by a vertical bar. Each note is composed in the following manner (optional elements shown in square brackets):

Duration	Pitch	[Octave]	[Tie?]
`(S/E/Q/H/W)[.]`	`(A-G)[b/#]`	`[0-9]`	`[_]`

The duration symbols represent sixteenth, eighth, quarter, half, and whole notes, respectively. The optional dot after the duration is used to represent a dotted eighth, quarter, or half note. Pitch characters may be followed by a '`b`' or '`#`' to indicate flat or sharp notes. Note that when a note is initially provided to the program, its pitch is represented *internally* simply as an integer from 1–127, where each number corresponds to a different piano key[12], just as in the Musical Instrument Digital Interface (MIDI) standard. Importantly, the accidentals included in the input are used solely to specify the MIDI number, which means that "`C#`" and "`Db`" are equivalent inputs to the program, as they both are indicated by MIDI number 61 (in the middle octave of the piano). The octave number above specifies an octave in the same manner as in MIDI, so that C4 is Middle C (MIDI note 60), and C5 is the note C one octave higher (MIDI note 72). If the octave is not specified, it defaults to the octave of the preceding note (or octave 4, for the first note in the melody). Finally, the optional underscore indicates that the note is tied to the following note.

In the user interface, a dropdown box provides a list of named melodies that have been saved on disk. Selecting one of these loads the melody, avoiding the need to retype the long melody representation. Additionally, a separate utility program was developed to convert back and forth between the standard MusicXML format and Musicat's text format, greatly facilitating music entry by allowing use of a separate notation program such as Finale or Sibelius.

Once a melody has been entered or selected, clicking the "Run" button starts the "listening" process; this is the main loop of the program, described later in this chapter. This can be paused or stopped with buttons in the user interface. The speed of the run can also be changed with the slider bar in the top right corner of the window. This slider adjusts the duration of a very frequently occurring but also very short delay built into the program to slow things down. Additionally, it affects the frequency with which the screen is redrawn; the fastest speeds are attained with the screen is redrawn less often.

The detail slider at the right edge of the window adjusts how many structures are displayed in the workspace view to the left. A high detail level shows everything, while a low

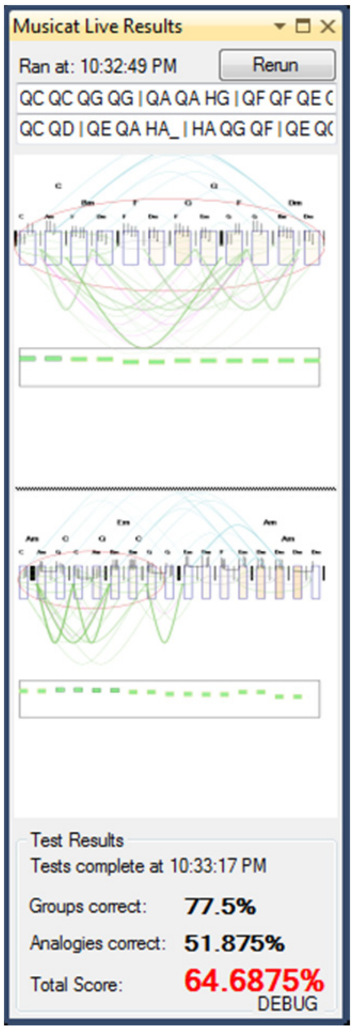
[12] Standard pianos have only 88 keys, indexed by MIDI note numbers 21–108.

detail level causes only the strongest structures to be displayed. Finally, a checkbox is provided to control whether or not future expectations (shown in purple) are displayed in the workspace.

Development Interface for Musicat

The standard interface shows the program acting on a single melody at a time. It takes at least several seconds to run, depending on the speed selected. The run time is not, as one might expect, primarily due to computation (running codelets). In fact, the bulk of the time is spent in redrawing the user interface many times per second to show the workspace as it changes.

Thus, to speed up development, I wrote a separate interface that does not redraw the screen while Musicat runs. Instead, it performs an entire run and then displays the final result. This is not as informative as watching a run in real time, but it is much faster. In order to speed development even more, I integrated this version of the user interface into my development environment, Visual Studio, as an “add-in” panel that is displayed right next to the code (Figure 2). This might seem like a technical detail, but it was actually extremely helpful in making modifications to the code and seeing the results nearly instantly. I set up the program so that it would re-run two entire melodies and display the



Ran at: 10:32:49 PM

QC QC QG QG | QA QA HG | QF QF QE (

QC QD | QE QA HA_ | HA QG QF | QE Q(

Test Results

Tests complete at 10:33:17 PM

Groups correct: 77.5%

Analogies correct: 51.875%

Total Score: 64.6875%

Figure 8.2:
Development interface.

results each time I recompiled any of the code. In Figure 2, the melodies are input as text at the top of the panel; the results are shown, stacked, below the text.

In addition to the graphical instant-run results, I also set up a test suite consisting of a set of melodies with desired resulting group structures and analogies. Each time the code is recompiled, the test suite is rerun: each melody is run through Musicat several times (to account for different results due to randomness), and the percentage of correct groups and analogies is computed across all runs. These percents are displayed at the bottom of the panel and averaged to give a total score (shown in red). The test suite takes a long time to run, but it does so in the background. The total score is a helpful guide in trying out new codelets or changing parameter values; I can make a change, recompile, and see instant results for two particular melodies in terms of the displayed grouping structure and analogies, and then if I wait a while longer I will see the effects of the change on the total score for the test suite.

Program Initialization and Assumptions

After a melody has been input by the user, Musicat makes some assumptions about it to simplify the problem. Future versions of Musicat would ideally avoid this phase — see Chapter 8 for discussion. The assumptions have to do with the key and the metric structure of the piece. Key is handled in a very simple way: the program assumes that the melody is in the key of C (either major or minor mode), so all input melodies must be given in C. Musicat does not know about modulation.

Meter is treated in a similarly simplistic manner: Musicat assumes that the entire melody is in the same time signature. This restriction was not present in earlier versions of the program, but the current version considers the measure to be a basic unit of time, so it is important that each measure have the same duration. Importantly, Musicat is restricted to making groups that start and end at a measure boundary, with one type of exception, described below. Admittedly, this is overly restrictive, as many melodies have phrases starting or ending mid-measure. Additionally, some small-scale structures (such as a simple group of four sixteenth notes) could be considered to be groups; however, Musicat focuses attention at a larger scale for its groups. Again, see Chapter 8 for discussion.

Exceptions to the rule of groups starting and ending at measure boundaries are found in melodies starting with a pickup note (or notes). Pickups are particularly common in 3/4 meter (for example, a 3/4 melody may begin with an unaccented quarter note on beat 3, followed by an accented downbeat), but they can occur in any time signature. Multiple measures can even serve as pickups at a higher metrical level (*e.g.*, in the First Movement of Mozart's 40th Symphony; see Figure 3, where the first strong beat at the hypermeasure level is shown with an accent mark[13] in the second full measure), but these more complex cases are not handled by Musicat.

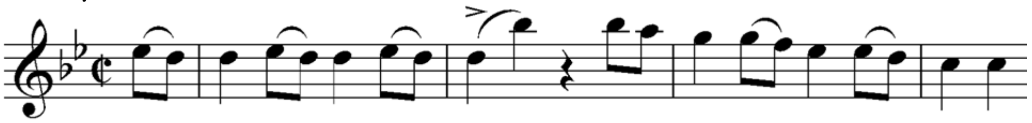


Figure 8.3: Hypermetric upbeat in Mozart's 40th Symphony, First Movement.

For melodies with pickup notes, Musicat uses a simple trick: in its internal representation, the bar lines of the input melody are shifted to the left by the duration of the pickup notes, so that the pickup notes seem to Musicat to be the start of a measure. These shifted measures are used when creating group structures. For example, with a one-beat pickup in a 3/4 piece, groups must all start on beat three of a measure and end after beat 2 of another. The original beat positions are preserved, however, for calculations involving things such as beat *strength*. See Figures 4–5 for an example of this shifting process, where metric accents are indicated with accent marks in the second figure, and grouping structures are

[13] The accent on this downbeat is not literally present in the melody line, but the strength of the beat is implied by several factors, including a low G in the cello and bass. See Bernstein (1976) for a detailed discussion.

indicated with slurs. The internal restriction to forming groups at bar lines in the shifted version of the melody (Figure 5) results in correct grouping structure in the unshifted version (Figure 4).

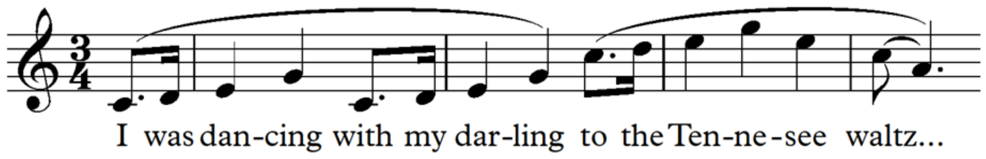


Figure 8.4: Tennessee Waltz melody with pickup beat.

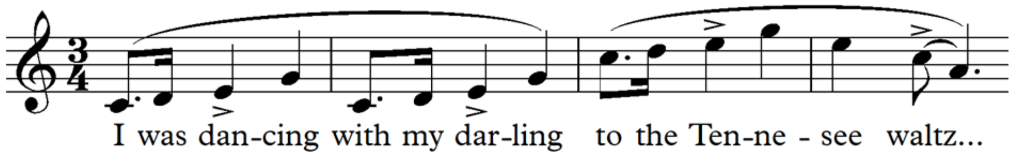

I was dan-cing with my dar-ling to the Ten-ne - see waltz...

Figure 8.5: Tennessee Waltz melody with shifted bar lines.

Main Program Loop

Once a melody has been given to the program, Musicat is ready to start simulating the listening process. But how exactly does Musicat work? How does it simulate real-time listening? We can look at the main program loop to get a high-level picture of the process. The program follows these steps until the processing of the melody is complete:

1. Add the next note to the Workspace.
2. Determine a number *k* of codelets to run for this note. This number is directly proportional to the duration of the current note (measured in sixteenth notes), and the constant proportion used allows for a simulation of different musical tempi.
3. Compute the quality, or happiness, of the representation of each measure in recent history.

4. Fill the Coderack with a mixture of random codelets and codelets selected in a top-down manner to act on any measure whose happiness is low.
5. Allow a small number of codelets to run, and then return to step 2 (unless *k* codelets have already been run).
6. By now *k* codelets have been run, so return to step 1 if there are still notes to process.
7. No more new notes remain to process. Musicat runs a set number of codelets (corresponding to a few measures of musical time) to wrap up its processing of the final notes.

Note that in step 2, the selection of *k* codelets results in simulated real-time processing. If our goal were to perform strict real-time processing, this would be insufficient; moreover, different codelets require different amounts of time to run. We might imagine a version of Musicat that uses asynchronous processing to add notes to the Workspace in real time, depending on the tempo of the music, and unrelated to computation time. However, because Musicat runs large numbers of codelets, we could in principle compute an average computation time (in seconds) per codelet (if we are careful to design codelets such that each one is guaranteed to run quickly and complete in a short time[14]), so choosing *k* proportional to note duration results, on average, in a real run-time that is proportional to the results we might attain with an asynchronous method. Also, for this model real time is inconsequential – if the program ran at half or 10% or 0.01% of real-time speed, that would still be acceptable, because the goal is cognitive modeling, not real-time application.

[14] Codelets are conceptually very small pieces of code. With care we can ensure an upper bound on time complexity of each codelet that is constant with respect to the length of the melody and the number of items in the Workspace.

Workspace

As in other FARG projects, the Workspace represents working memory and the perceptual structures built therein. The use of the Workspace in Musicat is perhaps more similar to that of Seqsee than of any other FARG project because both programs deal with temporal sequences. However, to aid in modeling the flow of time, Musicat's Workspace is not confined to structures presently available in working memory, but instead also contains older structures that have already faded from conscious awareness. A strict interpretation of the term "Workspace" might have required old structures to move into a longer-term memory, but in avoiding this transfer process for individual melodies, Musicat maintains a more blurry notion of the distinction between longer- and shorter- term memory. (A separate long-term memory that would be useful for making analogies between different pieces of music would be a useful addition in a future version of the program). In addition, the Workspace contains structures temporally located not only in the present and past, but also in the future: it may contain "hallucinated" structures that it expects to occur later.

Objects in the Workspace belong to roughly three categories: raw inputs, generated structures, and future expectations. How do objects get created in the Workspace? At the start of each run of the program, the Workspace is empty. The first action the program takes (Step 1 in the main program loop above) is to add the first note of the melody to the workspace. Then the Coderack is allowed to start processing by running codelets. Additional notes are added as time passes (as described above). Codelets add structures, destroy structures, analyze structures, and annotate the Workspace to guide other codelets; almost all of the objects in the Workspace, aside from the notes and measures of the melody, are generated by codelets.

The first category of objects — raw inputs — is made up of the unaltered notes added to the Workspace by the main loop of the program. These notes are held fixed in the

Workspace; they can't be modified or deleted. The second category comprises the bulk of the stuff of the Workspace: these are structures built by codelets, such as groups of notes or analogies between groups. All these structures are "hallucinated" by the program. They aren't as fixed as the raw input notes, but they can gain strength and become just as significant (or even more so). Finally, the third category is composed of structures such as groups or notes that are expected to occur in the future. These are also hallucinated structures, but they are much less tangible than other structures because they might never become "real" structures in the workspace. These expectations — either for certain groups to form, or for certain notes to occur — might be fulfilled or denied, depending on what comes next in the input stream.

Some additional objects in the Workspace are given to the program "for free" to reduce the complexity of the listening problem. These include the key, time signature, and measure-level metric structure of the piece. For example, all melodies are presented in C major or A minor, and the program is told which key is in use. Similarly, the time signature is given up-front, and it is assumed to be constant throughout a melody. Likewise bar lines are implicit: the program is given the metric position of the first note (and implicitly all successive notes), so it doesn't have to decide whether to interpret the first note as a downbeat in 4/4, or an upbeat in 3/4 time, for instance.

CODERACK

The Coderack is a critical component of the architecture, but is also the most straightforward one to understand. Just as in other FARG projects, the Coderack provides simulated parallel processing of codelets in much the same way as a modern computer operating system simulates multitasking of processes and threads. It stores a set of codelets to be run later. Each codelet is assigned an urgency by its creator, which is either another codelet or the main program loop. Only one codelet runs at a time, but because codelets perform small units of computation, many codelets run in a short span of time. At each iteration of the main program loop, the Coderack selects the next codelet to run. Higher-urgency codelets are more likely to be run next than lower-urgency codelets. Urgency is important for two reasons: it suggests a rough relative ordering of codelet execution, and it helps determine which codelets will eventually execute and which might be deleted without ever having run. Many codelets are posted to the Coderack, but not all posted codelets get a chance to run (since only a fixed amount of computation time is available). Periodically, the Coderack is cleaned up by removing the oldest codelets — that is, codelets that have been waiting on the Coderack for the longest time — one at a time until the number of waiting codelets is acceptably small (50 codelets, for example). Only 10–20% of codelets typically are ever actually run; the others are eventually purged in this cleanup process.

Choosing the next codelet to run is the central step in the description above. The basic algorithm for making this choice is the well-known "roulette-wheel" selection method, commonly used, for example, in genetic algorithms to decide which individuals will survive into the next generation (based on their fitness scores). Each codelet's urgency value is divided by the total urgency of all waiting codelets to determine the size of the "pie" allotted to that codelet in a random virtual spin of the roulette wheel.

Previous FARG programs have used two different modifications to the urgency value before it is used in codelet selection. In the current version of Musicat these modifications were not used, but it is important to understand why. The first modification was one used in the Tabletop program: the urgency is reduced if the codelet is a child of another codelet (as opposed to a child of the main program loop itself). Urgency is reduced for each generation. This strategy was used in Tabletop to prevent an "urgency explosion" problem, detailed by French (1992). I also implemented this strategy in Musicat, but I found it to cause worse performance, so eventually I removed it. I believe that in Musicat, codelets do not spawn other codelets as frequently as in Tabletop, and when they do, they are generally very useful. Additionally, the structure of the codelet-creation graph in Musicat is virtually loop-free (see Figure 6); the only loops in the graph are self-references, where a codelet may create child codelets with more specific parameters. Arrows in the figure represent parent codelets creating child codelets; for example, the **Suggest Parallel Analogy** codelet may generate **Create Analogy** codelets, which may in turn generate **Meta Grouper** or **Look for Contour Relationship** codelets. These last two types of codelets, however, do not themselves generate any more codelets directly. From the graph, we see that the longest chain of codelets (allowing only one self-loop, which is reasonable), has length four. At present only 15 codelet types can create other codelets; the other 25 codelet types just run and terminate without generating any children.

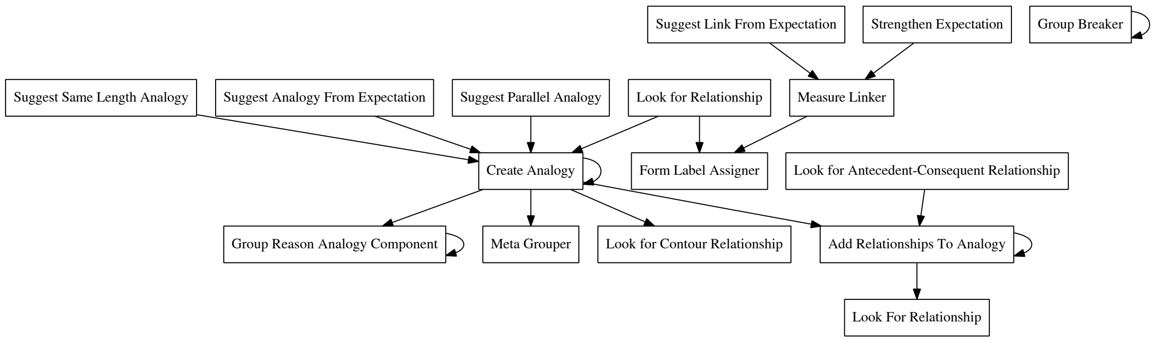

Suggest Link From Expectation
Strengthen Expectation
Group Breaker
Suggest Same Length Analogy
Suggest Analogy From Expectation
Suggest Parallel Analogy
Look for Relationship
Measure Linker
Create Analogy
Form Label Assigner
Look for Antecedent-Consequent Relationship
Group Reason Analogy Component
Meta Grouper
Look for Contour Relationship
Add Relationships To Analogy
Look For Relationship

Figure 8.6: Codelet-creation graph. Boxes at the top of the graph represent codelets that can generate codelets of the types below, shown by arrows.

The second potential modification to a codelet's urgency has to do with the current temperature of the Workspace: in other FARG programs, the distribution of codelet urgency values is modified to be flatter as temperature increases. That is, urgency plays less of a role as the temperature increases, which makes the codelet selection process more random. As temperature decreases, the urgency values are used more and more in the normal manner, and processing becomes less random. Note that in contrast to many other FARG programs, Musicat has no notion of global temperature, although it does have local temperature based on the strength of structures associated with each measure. Specifically, we can compute the happiness H of each measure and derive its temperature $T = 100 - H$. Thus the lack of global temperature doesn't actually pose a major problem; instead of global temperature, Musicat could use the average temperature of the measures of music in the recent time window (4 or 8 measures, for instance). If the recent measures have high average temperature, the Coderack could be made to act more randomly, ignoring urgency values. However, if we know that certain measures have high temperatures, and hence lower happiness values, simply using this temperature value to increase randomness in codelet selection would not be sufficient to drive codelet activity to the individual measures needing attention. Instead, Musicat uses the more focused mechanism of adding to the Coderack additional high-urgency codelets that act on measures with low happiness. Temperature values are still used in stochastic decisions made by codelets, such as deciding which of two competing groups wins a "fight", but for codelet selection, local high-temperature regions are used to indicate places to assign more computational effort. Urgency values are especially crucial when temperature is high precisely because urgency is the mechanism by which the main program loop tells the Coderack which codelets are the most important to run next, so Musicat does not weaken the effect of urgency as temperature increases. In other FARG programs, urgency-flattening due to high temperature made sense because temperature was computed globally; in Musicat, temperature in computed locally and thus has only a local effect.

Codelets

The Workspace provides a place for Musicat to build a representation of a melody, and the Coderack provides a mechanism for storing future codelets to run and for choosing what to do next, but the codelets themselves, of course, do the real work of Musicat, building up perceptual structures in the Workspace. Just as in other FARG models, each individual codelet performs a very simple, limited task (such as noticing that two particular measures have the same rhythmic pattern), and then it dies. Any codelet's action is very simple. Codelets often work at cross-purposes, performing conflicting actions in the Workspace. Paradoxically, it is precisely this frenzied struggle involving competing goals that results in fluid perception; the seeming chaos of a myriad of myopic single-minded codelets all performing their actions independently of each other provides the necessary substrate for emergent creative behavior. Ideally, the collective effect of many codelets operating in parallel is simulated perception that responds in a fluid manner to the changing structures in the Workspace.

Codelets may be loosely grouped into two broad categories: those that generally contribute to top-down perception and those that contribute to bottom-up perception, although there is some overlap, which I discuss later. Before describing these categories, I mention some features of codelets in general.

Features Common to All Codelets

Designing codelets is quite unlike the design of typical software. Several unique circumstances come up in their implementation, stemming from the stochastic nature of the program (codelets may run in any order). Implementing codelets thus has a lot in common with writing code capable of running in a multitasking computer environment. Codelets must be able to interact with a Workspace that is shared by all codelets in the system. Additionally, musical time is passing — each time a codelet completes, another unit of simulated musical time "ticks" — and codelets must focus their attention on structures that were created "recently" in musical time. This requirement of focusing on the present and recent past is enforced by other code in Musicat, so that individual codelets will not process structures occurring too far in the past.

To facilitate the arbitrary order of execution of codelets, each codelet is responsible for performing a series of validity tests at the start of its run. Recall that codelets in Musicat come from one of two sources: either they are created completely at random by the main program loop, or they are created in a more directed, top-down manner. In the first case (random creation), each codelet is created without any parameters specified (such as which measures to examine or which group to break); these parameters are filled in later when the codelet runs. This causes no problems related to arbitrary execution order. In the second case (top-down pressure), however, codelets, when created, are given some specific direction about which Workspace elements to use when they run. These parameters, such as "the group of measures 3 through 6" in an **Extend Group Right** codelet, may no longer be valid by the time the codelet runs. In this case, the group in question existed when the codelet was created but might have been destroyed by the time the codelet is picked to run. In this case, the solution is simply to make sure the group still exists before the codelet runs, and if not, to perhaps try to find a different group to operate on. In general, each codelet must evaluate a set of preconditions to verify that its set of parameters is still valid before it is given the green light to run. For codelets with many parameters, this can be surprisingly tricky (just as ensuring thread-safety of multithreaded code is tricky). However, as long as it is done carefully for each codelet, arbitrary run order works without any problems. Because each codelet runs atomically (codelets themselves do not run in parallel in this version of Musicat), these precondition checks also ensure the integrity of the Workspace, even though each codelet can change the contents of the Workspace. Note that it would be possible to implement Musicat to run multiple codelets in true parallel (*i.e.*, on multiple threads) but even more synchronization constructs would be required; because of the highly dynamic nature of the Workspace, this might not result in much performance improvement without a great deal of careful planning.

In order to enforce the idea that codelets must focus their attention on recent perceptual structures that still exist in working memory, codelets are expected to interact with the Workspace through a set of intermediate C♯ functions that can restrict their access as necessary. In terms of software design, each object in the Workspace (including the whole Workspace itself) is represented by a C♯ class that provides methods for codelets to interact with the objects. These methods take care of preventing access to objects that are too far in the past, and they do so in a consistent way for each type of codelet. As an example, imagine that a codelet tries to create a new group. The codelet may detect groups that would conflict with the proposed new group, so it first enters each conflicting group in a probabilistic competition with the proposed group, and removes the conflicting groups if they all lose their fights. The codelet would call a standard Workspace class method to remove the losing groups, but this method has added checks built in that stop the removal if other conditions are not fulfilled. For example, if the group to be removed occurs too far in the musical past, it will never be removed. Overall, a great deal of code for reading from and writing to the Workspace is encapsulated in Workspace-related objects.

Top-Down Codelets

The terms "top-down" and "bottom-up" can be slightly confusing. In each field that uses these terms, these terms tend to be used differently. Even within a single research group, such as FARG, the terms can be a bit ambiguous. For example, in his dissertation on Phaeaco, Harry Foundalis writes:

> There are bottom-up and top-down codelets. The former act directly on structures in the Workspace without any prior information about what

> should exist there. Top-down codelets, in contrast, carry out actions on Workspace structures with an eye to creating specific types of higher-level structures. (p. 132)

I use the terms slightly differently, although in a similar spirit. Specific codelets that have been posted to the Coderack in Musicat may be called "top-down" or "bottom-up", but codelet *types* generally do not have an intrinsic top-downness or bottom-upness to them. In Musicat, "top-down" instead refers to those codelets that have been generated as a result of context-based pressures, which I call "top-down" pressures. The goal of these pressures is generally to create higher-level structures, but the individual codelets generated may not necessarily have these specific goals.

For example, if Musicat has perceived an analogy between measures 1–2 and measures 5–6 in a melody, a **Suggest Parallel Analogy** codelet might be generated, with this known analogy as a parameter. The codelet might then look for an analogy between measures 3–4 and 7–8. But if there is a conflict, such as a group spanning measures 3–6, a **Group Breaker** codelet might be added to the Coderack. This codelet would still have a top-down genesis, although the more typical use of a **Group Breaker** would be to destroy random weak groups, in a manner that I would call "bottom-up". In general, particular codelets created in response to the big-picture context are called "top-down codelets", just as the pressures to create such codelets are called "top-down pressures".

Bottom-Up Codelets

Bottom-up codelets, in contrast to top-down, are not generated in response to contextual pressures from the system. Instead, the Coderack is periodically refreshed with a pool of codelets that randomly choose small-scale elements of the Workspace to analyze, where the random choice is biased towards analyzing the elements that were heard the most recently in musical time. A typical example of a bottom-up codelet is the **Measure Linker** codelet, which looks for simple rhythmic similarities between measures. If this type of codelet is generated by the main program loop, it selects a pair of recent measures at random to analyze. Most of the time a random pair of measures will not wind up getting linked by such a codelet. Occasionally, however, a useful link is formed, which stays in the Workspace and contributes to larger constructions. But, as discussed above, **Measure Linker** codelets may also be generated by other codelets such as **Suggest Link from Expectation** codelets, which would indicate pairs of measures that the **Measure Linkers** should to try to link; in this case, I would call these particular **Measure Linkers** "top-down", not "bottom-up".

Temperature

The notion of temperature was used in Copycat, as well as in several related FARG programs, to balance the need for undirected and chaotic action, when the program was far from a solution and needed to explore more "wild" ideas, with more focused and deliberate action, when a good solution was taking shape. The Workspace in these programs had a global temperature derived from the quality and cohesiveness of the Workspace's structures. Low-quality structures caused the temperature to go up and thus increased the randomness of codelet actions, while high-quality structures reduced temperature and thus randomness. Additionally, over time, the temperature was made to decrease gradually, as in simulated annealing, helping the program to settle down to some solution (even if of low quality).

Musicat also incorporates temperature, but two changes were necessary for temperature to make sense in the context of real-time listening. These have to do with the simulated-annealing idea and with a need for local (as opposed to global) temperature.

Whereas Copycat's goal for each program run was to solve a single analogy problem, with the complete Workspace structure constantly available in memory, Musicat creates many different structures and analogies in one run, and older structures fade into the past as new notes are “heard”. Copycat’s Workspace can be represented naturally by a diagram of fixed size in 2-D space (*e.g.*, as a rectangle with all the pieces of the analogy contained within), but Muscat’s Workspace is better visualized as a 2-D diagram where time has been mapped onto the *x*-axis and objects are flowing to the left as they gradually move from the present into the past. Because time is moving along naturally, the enforced gradual decay of temperature doesn’t make at much sense here. Musicat generates structures to make sense of recently-heard notes, but as new notes of the melody are perceived, old notes and structures stabilize as they leave the working-memory area, without the need for a simulated-annealing process.

The other difference in temperature between Copycat and Musicat is more significant. Using a single temperature value in Copycat is reasonable because its goal is to create one unified analogical mapping explaining the input and suggesting a solution. Why can’t a similar global temperature be defined for Musicat? It is tempting to think that Musicat’s goal is to create a single coherent musical parsing of a melody (such as the Time-Span Reduction in GTTM), and a temperature value could be computed that would describe the coherence of the generated structure. However, this picture is too perfect to be a good model of the human listening experience. Instead, some portions of a melody may indeed be heard as part of a strong structure, but other parts may be less understood, and the listener will simply move on and continue listening to newer notes without ever coming to a complete understanding of that part of the melody. Thus, Musicat computes temperature over small ranges of music; there is no global temperature.

Although there is no global temperature, the window of recently-heard music has special significance and we can make a rough analogy (for program-architecture purposes) between the music currently active in working memory and the entire Copycat Workspace, which also represents structures in working memory. Thus, in Musicat, the temperature of the range of measures in working memory (say, eight measures) is used to modify certain probabilistic codelet actions. As was explained earlier, urgency values are not modified by this temperature value. However, stochastic decisions such as determining which of two structures will win a "fight" are influenced by this temperature.

In Musicat, it is often easier to think of a "happiness" value instead of temperature. Recall that happiness and temperature are related by the equation $T = 100 - H$, where both H and T range from 0–100. For example, part of the main program loop computes the happiness of single measures in the recent history of the Workspace and creates codelets to act specifically on structures involving those measures with low happiness.

Sliding Time Window of Perception

As was mentioned above, Musicat focuses its attention on the most recent measures of a melody and the most recent structures in the Workspace. New structures come and go rapidly as codelets try out different ideas, but as time passes, Musicat's perceptual structures become more fixed. Eventually, structures that are old enough become unchangeable. This reflects our intuition that listeners can retroactively change their perception of musical structures but only for a short time. Presumably, this time span is determined by the length of time that structures persist in working memory, although I'm not aware of any research specifically on the subject of retroactive listening and working memory.

In Musicat, two distinct methods are used to make structures stabilize as time passes. First, groups that lie several measures in the past can receive a strength bonus based on how distant they are from the most recently perceived note. For example, imagine a piece where measures 1 and 2 are identical to measures 3 and 4. Also imagine that Musicat has generated a group G1 containing the first two measures, and a second group G2 containing measures 3 and 4. If measure 4 has just been completely presented to the program and measure 5 is starting, G1 and G2 might look identical, but G1 would likely have a higher score because it is further in the past. Note that this bonus is based solely on the temporal distance between the end of the group and the present moment; the age of the group, in terms of how long it has persisted in the Workspace after its creation, is not a factor here. Once enough musical time passes (4 or 8 measures, based on a constant parameter set in the code and increased slightly for large groups), any groups that exist are considered to be permanent, and are immune to further changes. This simulates the idea that eventually any groups constructed in the Workspace will stop being part of active working memory and exist only in a longer-term storage. This also helps ensure that Musicat's algorithms have a bounded time complexity, and makes real-time processing computationally feasible for melodies of unlimited length.

A second thing that helps older Workspace objects stabilize is the manner in which codelets decide which measures, groups, or analogies they should act on. Codelets choose objects to work on simply by asking the Workspace object to provide a recent object of the desired type at random. For example, a codelet may make a request such as "Find a recent measure". The Workspace will choose one probabilistically, according to a decay function that favors the most recent objects. In this way, most codelets act on very recent objects in the Workspace, but sometimes they will act on objects further in the past (as long as those objects have not become permanent).

One unresolved issue in the program is how to best deal with larger-scale structures in the Workspace. Listening happens in a mostly-linear fashion as time progresses, but in addition to a certain amount of retroactive listening that occurs as we reprocess music stored in working memory, we are also listening in a hierarchical way. It is a subtle blend of linear and hierarchical pressures that leads to hearing music as having a particular grouping. How do listeners (and how should Musicat) balance these two pressures? Sometimes a local linear structure (*e.g.*, a scale fragment) sounds like a group, but in other cases the same structure might be broken into two groups as a result of higher-level (hierarchical) grouping pressures. Fortunately we get some balance for free: time keeps moving, forcing decisions to be made quickly. But this also causes other difficulties: what happens when large structures have moved too far into the past?

Once a listener has perceived large objects such as phrases, periods, or entire sections as "chunks" in memory, it is possible to reason about these larger structures. Musicat can make long-distance analogies between chunks, but the creation of large-scale groups is difficult when some of the elements of the group are no longer in Musicat's short-term window of perception. Perhaps a mechanism for temporarily reactivating older groups would solve the problem. In Musicat, the Workspace represents both short-term working memory and longer-term memory (for structures which are far in the past and have achieved "permanent" status); this blending-together of short- and long-term memory is architecturally appealing, but a separate medium- or long-term memory space might help clarify some of the issues involved in listening to long melodies with large-scale structures.

Objects in the Workspace

The primary output from a run of Musicat is a representation of the "heard" melody. This representation is simply the contents of the Workspace at the end of a run (although the dynamic, changing representation as time flows during a run is of great interest as well). This section describes the various objects that can be found in the Workspace.

Notes

Notes are simple: as described above in the section on Musicat's user interface, they are made up of a symbolic representation of pitch such as "60", which stands for the pitch of Middle C, as well as rhythmic information. (Note that individual codelets may hear a note as "the tonic in C major" or "raised third degree of an A♭ minor chord", but in this version of the program — in contrast to earlier ones — these interpretations are created temporarily by codelets and not persisted as part of the Workspace structure.) The note's start and end time are given in units of sixteenth notes, relative to the beginning of the measure which contains it. Notes may also have a tie to the following note. When a note is tied, Musicat understands that the note is not rearticulated, so that operations requiring a list of attack points, for instance, ignore notes where the previous note was followed by a tie. Note that ties cause some subtle issues that must be handled carefully in the software. For instance, when shifting measures internally to account for pickup beats, sometimes ties will need to be created to account for long notes which cross a bar line after the shift operation.

Measures

Each note in the Workspace must be contained within a measure. A measure stores a list of its notes and a time signature. Optionally, a measure may store an associated pitch alphabet. Musicat's measures are the smallest elements which can be members of a group. Although codelets can examine the notes contained in a measure, in many ways the measure is the atomic musical unit in Musicat.

Rhythm-Based Measure Links

Rhythmic similarity plays a key role in Musicat. It is the most basic kind of similarity and forms the basis for more complex types of relationships between musical structures. Thus, there is a special object in the Workspace that indicates rhythm-based similarity, simply called a *measure link* internally, as opposed to the more descriptive "rhythmic measure link", because rhythmic similarity is the natural, default type of similarity Musicat notices. In other words, the "rhythmic" part is implied. (Other, more general links, which may involve pitch information, are called *relationships).* When two measures are found to be (rhythmically) similar to each other, a measure link may be created (by codelets) to represent the discovered similarity.

Measure links appear in the user interface as arcs above the staff. Like most objects in the Workspace, each measure link has an associated strength value in the range 0–100. For a measure link, this describes the amount of similarity discovered between the two measures in question. In addition, the strength can decay over time if the link is not used as a component of any larger structure. Thus, irrelevant links eventually fade away.

Relationships

"Relationship" objects in the Workspace (henceforth referred to as "relationships") are used to represent all types of similarity between pairs of objects in the Workspace. Each relationship has a strength value (just like measure links do). In addition, each relationship has a specific type. Relationships can be formed as a result of perceived similarity in melodic contour or metric position, or of the presence of a rhythmic measure link, or of two groups having the same initial notes, or for many other reasons.

Although any relationship is *conceptually* an analogy, in Musicat, there is a special *analogy* datatype reserved for describing a mapping between two large structures and their components (read more about the analogy datatype in the "Analogies" section below). For the program, then, relationships and analogies look different — they have different datatypes. To allow codelets that act on relationships to also act on analogies using the same code, I have adopted the following solution: when an analogy is created in the Workspace, an special type of relationship — *an analogy relationship* — is automatically formed in the workspace, in *parallel* with the analogy datatype An analogy relationship simply links two large objects and indicates that they are similar, so that other codelets can be aware of the relationship without unpacking all the details involved in the large data structure that describes an analogy. (The *analogy relationship* can be thought of as a "hack" that makes it simple for codelets to act on relationships without understanding the inner details of the relationships.) In general, a relationship represents a cognitive association between two objects, where the details and the reason for the association are not immediately accessible.

Bar Lines

In Musicat, a bar line is more than just the boundary point between two adjacent measures. Bar lines are represented explicitly in the Workspace because each bar line can have a perceived "thickness" assigned to it by codelets, which is modifiable over time.

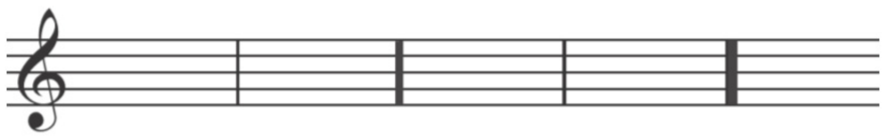


Figure 8.7: A typical sequence of bar line "thicknesses".

A bar line's thickness indicates how strongly an implied breakpoint might be heard between two adjacent measures based solely on their positions in the melody's metric hierarchy. Equivalently, we can say that bar-line thickness corresponds to the strength of the metric stress of the measure following the bar line. For example, after the second measure of a melody with a straightforward duple hypermeter, we would hear the bar line as slightly "thicker" than the bar line after the first measure. Then the bar line after measure four would be thicker yet, and the one after measure eight would be the thickest bar line encountered up to that point in the melody. The thicker a bar line is, the less likely it is that a group will be formed containing measures on both sides of it (unless the group is quite large itself). Other representations of hypermetric structure in the literature have used a metaphor of "height" or "strength" attached to the first beat of a measure, but we feel that bar-line thickness more clearly communicates the relationship between hypermeter and grouping boundaries. (Of course, as Rothstein points out (1989), grouping and hypermeter structure may differ by high-level (hypermetric) upbeats, such as shown in Figure 8.3, but Musicat has a strong preference for these two structures to line up exactly.)

ALPHABETS

Pitch alphabets (or simply "alphabets") were discussed earlier in the context of Larson's Seek Well program. They are used in Musicat to describe the implied harmonic context of a measure or group. An alphabet is simply an ordered collection of pitches. In Musicat, the pitches of an alphabet repeat in every octave (*e.g.*, a major chord with an added ninth is equivalent to a major chord with an added second). Musicat's alphabets differ slightly in implementation from those used in Seek Well, in which each regular alphabet was paired with a goal alphabet (a subset of the regular alphabet). In Musicat, each pitch in an alphabet has an additional binary flag marking the pitch as stable or unstable. Stable pitches correspond to pitches in a goal alphabet in Seek Well.

GROUPS AND META-GROUPS

Simple Groups

The simplest kind of group in Musicat is a collection of adjacent measures. One-measure groups are possible in certain circumstances, although they occur quite rarely and are expected to grow to incorporate additional measures. Groups represent musical structures, such as phrases, that are heard as a unit. Each group maintains a list of "reasons" that it exists (*e.g.*, starting after a thick barline and ending on a long and stable note), as well as a list of "penalties" describing problems that afflict it (*e.g.*, spanning a thick bar line). These lists are created and modified over time by codelets. A group's strength is calculated based on these reasons and penalties (details will be given in the "Calculating Structure Strengths" section). Groups also maintain a current choice of pitch alphabet, assigned by and modifiable by codelets, which is used to aid understanding of notes of the group in a harmonic context.

Meta-groups

Groups can contain other groups. For example, if a two-measure group G1 spans measures 1–2 and another two-measure group G2 spans measures 3–4, then a group M can be created that contains both G1 and G2. I sometimes call a group that contains other groups a "meta-group", but for simplicity I use the term "group" for all kinds of groups, whether they contain groups or measures (or both). Elements such as measures or groups that are contained in a meta-group are called *children* of the meta-group; conversely, the meta-group is called the *parent* group of those elements.

Restrictions on Groups

The elements contained in a group must be temporally adjacent. For example, if we continue the previous example by adding another two-measure group G3 spanning measures 5–6, Musicat can make a group M2 that contains all three groups G1, G2, and G3, but it is not allowed to make a group M3 containing only groups G1 and G3.

Groups may contain other groups, but no groups may overlap temporally. Equivalently, we can say that each measure is the direct child of at most one group (measures may not have any parent group at all, but Musicat has internal pressures that push it to try to avoid this situation.) For example, given the group G1 above, it is not possible to create a group G4 that spans measures 2–3; the two groups would be in contention because they both contain measure 2. Such conflicts between rival groups occur extremely often during each run of Musicat and codelets are considering alternate grouping structures, creating rival groups temporarily in memory for consideration, but for all groups stored in the Workspace the non-overlapping rule applies strictly. Note that requiring non-overlapping groups is a significant restriction in the music domain, and allowing a small amount of overlap would be preferable in a future version of the program. Lerdahl and Jackendoff (1983) demonstrate in GTTM that a single note might simultaneously fill the dual roles of being the final note of one group and the first note of a second group. These two groups, then, would overlap by one note. In spite of this possibility for single-note group overlap, GTTM has a rule requiring non-overlapping groups (Grouping Well-Formedness Rule 4), but it is justified by a discussion of group elision, in which any passage including an overlap of the sort discussed here is seen as a syntactically-transformed version of an underlying musical structure in which the first group ends and then the note is repeated to start the second group. This idea is related to Chomsky's notion of the surface form of a sentence deriving, via syntactic transformations, from an underlying deep structure. In the current version of Musicat, groups must start and end at measure boundaries, so the sort of overlap just described in GTTM cannot occur. For styles of music where group overlap is natural and common, this may present a problem, but for the time being, requiring non-overlapping groups makes sense because groups in Musicat must occur at measure boundaries and in a relatively simple melody, it is unlikely that an entire measure would serve as both the start measure of one group and the end measure of another group.

Sequences

Analogy structures in the Workspace (see below) involve a mapping between two objects, along with their subcomponents. The musical notion of a sequence, on the other hand, often involves three or more copies of a musical pattern appearing successively (in *sequence*, of course). A sequence is thus a distinct object from an analogy for Musicat. Sequences also have a strict definition here: sequences require nearly-exact copies of a melodic pattern, with only minor tonal variations allowed to due transposition. For example, the pattern in the first measure might be repeated in the next measure, but with each note shifted one tone lower in an alphabet such as the major scale. If the same pattern were to continue for a third measure, Musicat might detect this as a sequence. See Figure 8.8 for an example. To be sure, for a human listener, hearing such a pattern as a sequence is indeed analogy-making in action, but treating the notion of sequence as a special circumstance with rather rigid rules was helpful in the implementation of Musicat. This definition of sequence handles many interesting cases, because many sequential patterns in music are quite strict and require only minimal amounts of flexible analogy-making to understand.

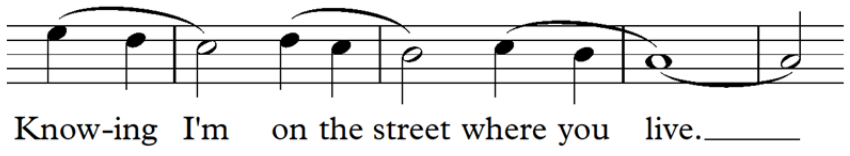


Figure 8.8: A typical sequence.

In Figure 8.8, the three-note pitch pattern (step down, step down) repeats perfectly three times, with each instance starting one step lower. However, there is a very small difference in the third repetition of the quarter-quarter-half rhythmic pattern: the final note is a tied whole note, lasting six beats instead of the expected two beats. Musicat allows the final repetition of a sequence to be different after the initial notes are played as expected. This allows changes at the end of a sequence, as in Figure 8.8, to be considered as part of the sequence. In the case of threefold repetitions like this one, a change at the end is often necessary to round out a phrase, which likely has a length (in measures) of four, eight, or some other power of two.

Analogies

As was discussed in Chapter 4, analogy-making plays a central and omnipresent role in music listening (not to mention the rest of cognition). In implementing a program such as Musicat, there is a danger of not coming anywhere close to the flexibility and complexity of human analogies, in that one needs to design highly-precise data structures in order for a computer to represent an analogy. Of course, the whole architecture of Musicat and of other FARG programs was designed to support fluid analogy-making; nevertheless, I am reminded of the vast oversimplifications necessary in my program whenever I open up the "Analogy.cs" file and see the scant 477 lines of code that define the Analogy class in the C# language. With that disclaimer in place, I will describe how Musicat represents analogies. But first, I will explain some small-scale linking structures in Musicat that are not part of the official "Analogy" class, even though I think of them as analogies.

At a low level, Musicat represents similarity between objects in the Workspace through rhythm-based measure links and a set of more general relationship objects[15] described above. One might wonder why these types of links are necessary, instead expecting the Analogy class to handle all types of connections between objects. Indeed, it would be possible to rewrite the code to make an all-encompassing Analogy class, but there are two reasons I avoided doing so. First, as a practical matter, an analogy is an inherently recursive

[15] In fact, the specialized measure links are transformed into relationship objects when necessary, so it is sufficient to talk about relationships alone for the rest of this section.

data structure: an analogy between two groups might be composed of smaller-scale analogies between children of those groups. Any recursive data structure eventually "bottoms out" in structures with no children of their own; for example, think of the leaf nodes of a binary tree. The "leaf nodes" in Musicat's analogies are precisely the low-level relationship links. Second, because relationships are so low-level, we can think of them as operating closer to the periphery of the nervous system than higher-level analogy-making. Certain types of auditory processing (such as resolving fundamental frequencies from a sound wave, or, perhaps, detecting repetitive rhythmic patterns) happen much more immediately in the brain than other, higher-level processing. Thus it struck me as reasonable that low-level relationships be detected with specialized code, while having higher-level relationships be found using a more general analogy mechanism.

An analogy in Musicat is essentially a mapping between a group on the left (I abbreviate this as "LHS", for "left-hand side") and a group on the right (RHS). Left and right are meant with respect to a musical score, so the LHS is a group further in the past than the RHS. The mapping is flexible: not all elements are required to be mapped from one side to the other; in mathematical terms, an analogy is not an isomorphism. It is simply a set of relationships, where half of each relationship corresponds to an element from the LHS of the analogy and the other half corresponds to an element from the RHS. Analogies at a more abstract level (I call them meta-analogies) are beyond the scope of this program. For example, Musicat can map the first four measures of "Sur le pont d'Avignon" onto the last four measures of the melody: the measures on the left are mapped to the measures on the right. But if Musicat were to notice that measure 1 is related to measure 2 by transposition, and furthermore were to notice that measure 3 is related to measure 4 by transposition, it could not use this knowledge to make a meta-analogy between (1–2) and (3–4). Another restriction on analogy-making is that all analogies are within a single melody, not between different melodies.

An analogy, just like groups, links, and so forth, has an associated strength value. One important difference between the method of computing the strength of an analogy, as opposed to that of a group, however, is that analogy strength is computed using a fixed list of criteria. An analogy's strength can change over time, however, as new relationships are added to the analogy. An incomplete analogy can be formed with a relatively low strength, and the birth of the analogy will trigger a search for completing the mapping, which will increase the analogy's strength, if the search is successful. Analogy strength is discussed in more detail in a following section.

Expectations

There are codelets in Musicat that can create groups, analogies, and measure links that are expected to occur in the future. Expectations are formed in a simple-minded manner: when a strong and relatively large structure has been formed that ends at a thin bar line, Musicat expects that structure to repeat. This is admittedly a simple-minded rule of thumb, but it reflects the intuition that musical patterns tend to repeat. Indeed, we can even think of this as an interpretation of "inertia" in Larson's theory of musical forces: a large structure sets up a trivial expectation that another structure just like it is on the way.

When an expectation is formed, it is used to focus the attention of future codelets. For example, if a group that includes measures 9–10 is expected to form, then once measure 9 is heard, codelets will be created to add measure 9 to a new group, and once measure 10 arrives, the codelets will try, with high urgency, to group measures 9 and 10 together.

Musicat's notion of expectations may seem surprising at first, because typical work on expectation is centered on expectation in the pitch and rhythm domains. These are certainly important and are a high priority for future work on Musicat. In this version of the program, however, consistent with its focus on understanding the higher-level structure of melodies, expectations involve grouping and analogy, not the pitches and rhythms of individual notes.

Calculating Structure Strengths

The Workspace is filled with the perceptual structures listed above, such as groups, relationships between measures or groups, analogies, expectations, and so forth. Each such structure has an associated strength that indicates how strongly committed Musicat is to the structure. These strengths (equivalently, "scores") range from 0 (extremely weak) to 100 (extremely strong).

The computation of structure strengths is central to how Musicat functions, and much of Musicat's idiosyncratic listening style emerges as a consequence of which types of structures tend to be scored highly. In the current implementation, we have tried to make Musicat favor the kinds of structures and patterns that are important in Western folk melody. Many of these, however, such as the primacy of rhythmic repetition, or gestalt grouping principles, are rather general and should apply to most musical cultures. Others, such as the tonal functions of the tonic and dominant scale degrees or the notion of melodic sequence, may be more culturally specific.

A common suggestion I've received for Musicat is that the scoring functions should be optimized for a particular musical corpus using machine-learning techniques, perhaps to simulate the effect of musical enculturation on a listener. While this is an interesting and tempting idea, it misses the basic point of this work, which is to model fundamental mechanisms of listening (in the general framework of analogy-making). Additionally, the overall architecture of the model, its types of codelets, and the types of perceptual structures it can make are probably more critical to its listening style than are the details of how it computes scores.

An analogy with computer chess programs may be useful to motivate the philosophy behind our design choices: the best chess programs use brute-force search as their main "trick" to achieve super-human performance. Deep Blue, IBM's computer that famously defeated world champion Garry Kasparov in 1997, relied on an optimized brute-force search capable of examining 200 million positions per second. In this type of search, the desirability of each board position in the game tree is computed with an evaluation function that assigns a score such as +0.8, indicating that the first player is ahead by nearly a pawn, or –8.5, indicating that the first player has lost nearly the equivalent of a queen. Interestingly, Deep Blue used a quite simple evaluation function that was optimized for speed, not for subtlety of positional understanding. It turns out that the brute-force search algorithm (combined with the hardware's raw speed) was the driving factor behind the computer's success at chess; I suspect that practically any quick-and-dirty evaluation function taking into account the value of the pieces on the board for each player would result in a similar level of chess-playing. Kasparov was so stunned by the subtle-looking nature of the computer's 37th move (Bishop moves to square "e4"), in game 2 of the 1997 match, that he later accused IBM of cheating.

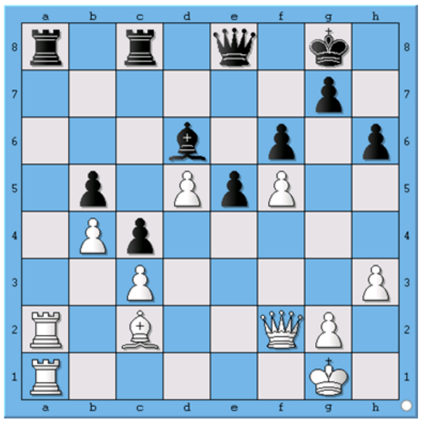


Figure 8.9: White (Deep Blue) to move, just before move 37 (Be4).
Game 2, Kasparov versus Deep Blue, 1997.

These days, however, most strong chess programs (including free programs that I can download and run on my laptop) will, after only a few seconds of computation, come up with the same bishop move as Deep Blue did, which strongly suggests that brute-force search using a wide variety of evaluation functions will lead to this move. Of course, variations in evaluation functions to score positional elements (such as king safety) more highly than aggressive piece placement would naturally result in more defensive play, although even a version of Deep Blue using a more defensive evaluation function would still be capable of brilliant but reckless-looking attacks whenever brute-force search could prove the attack's soundness. Similarly, although the evaluation functions that compute structure strengths are important in determining Musicat's listening style, the overall effectiveness of Musicat as a music listener may be more determined by the program's architecture than by the details of its scoring algorithms. (To be clear, Musicat does not use brute-force search like Deep Blue — Musicat *avoids* brute force! Both programs, however, do make use of evaluation functions.)

The following sections go into more detail about how various structures are scored, starting with simple rhythmic similarity between measures and proceeding through relationship, group, sequence, analogy, and expectation scoring.

Scoring Rhythmic Measure Links

A rhythmic measure link can be formed between two measures that are rhythmically similar. The strength of the link is based simply on the number of note attack-points (articulations) that the two measures have in common. Specifically, the score of the link between two measures m_1 and m_2 is inversely related to the size of the symmetric difference between the two measures' attack point sets, normalized by the total number of attack points.

$$symDiff(m_1, m_2) = \#missing\ attacks\ in\ m_1 + \#missing\ attacks\ in\ m_2$$

$$score(m_1, m_2) = 100 \times (1 - \frac{symDiff(m_1, m_2)}{\#attacks_{m_1} + \#attacks_{m_2}})$$

For example, given the measures (H QQ) and (QQQQ), the attack on beat 2 in the second measure is missing in the first measure, so the symmetric difference is 1. There are three attacks in measure 1 and four in measure 2, so the score is given by

$$100 \times \left(1 - \frac{1}{3+4}\right) = 100 \times \frac{6}{7} \approx 86$$

This is a high score: most of the attacks are present in both measures. (The maximum possible score, achieved only when the two rhythms are identical, is 100.) If we change the second measure to (QQ H), however, then the symmetric difference is 2, the total number of attack points is 6, and the score is approximately 67. In this case, the "correct" part of the score come from both measures having attacks on beats 1 and 3; the missing attacks on beats 2 and 4 in the first and second measures, respectively, reduce the score.

Scoring Relationships

Relationships between measures or groups are the key objects used in forming analogies between groups. Recall that relationships in Musicat are used to represent links that have been found between two structures, not the details of the links. This allows Musicat to easily build more complex analogy objects from simpler relationship objects. The score for a relationship is based on its type, and is calculated by the codelet that creates the relationship in the first place. Details follow for each relationship type.

Types of Relationships

Identical Rhythm, Similar Rhythm

The simplest relationship types — "Identical Rhythm" and "Similar Rhythm" — mirror the primal "rhythmic measure links" in the Workspace. For each such link, an Identical Rhythm relationship is formed if the link had a score of 100; otherwise, a Similar Rhythm relationship is formed.

Start Identical Rhythm, Start Similar Rhythm

Two measures or groups can be assigned a Start Identical Rhythm relationship if they start with identical rhythmic material but diverge later. The "start" of a group or measure is defined as either half the duration or eight beats, whichever is shorter. The score is computed just as for a Rhythmic Measure Link, described above, but using only the start region for the computation.

Analogy

Recall from the "Objects in the Workspace" section above that when an analogy is formed, an Analogy *relationship* is automatically created in the workspace to mirror the analogy and is used to represent it in higher-level analogies. Unlike other relationship types, an Analogy relationship has a strength that is dynamically computed each time it is used, by recomputing the strength of the source analogy (the analogy object that gave rise to the Analogy relationship). The relationship strength is identical to this computed analogy strength.

Melody Contour

A Melody Contour relationship can be created if two measures or groups have a similar-enough melodic shape. The contour of a selected phrase is represented by a string of symbols representing the difference in pitch between adjacent notes. Five symbols are possible in a contour string (Table 2):

`U`	Step up (1 or 2 semitones)
`d`	Step down (1 or 2 semitones)
`U`	Leap up (> 2 semitones)
`D`	Leap down (> 2 semitones)
`–`	Sideways motion (repeated note)

Table 2: Contour symbols.

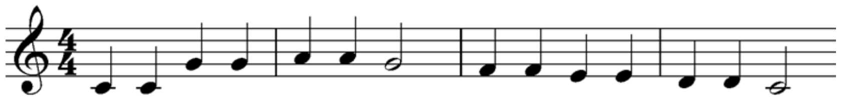


Figure 8.10: Twinkle, Twinkle.

For example, the first four measures of the "Twinkle, Twinkle, Little Star" melody (CCGG | AAG | FFEE | DDC) (Figure 8.10) would be described by the following string:

```
-U-u-dd-d-d-d
```

Observe that for a melody of N notes, there are only N−1 transitions between notes; this melody has 14 notes and 13 symbols in its contour string. This becomes more apparent when we consider the same melody broken into a pair of two-measure phrases of 7 notes each (“CCGG | AAG” and “FFEE | DDC”): the second instance of the symbol “d” will not appear because the transition between the two measure-pairs is not represented, so now there are 12 symbols total (Figure 8.11 & Figure 8.12).

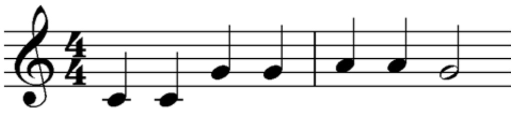


Figure 8.11: Measures 1–2.

`-U-u-d`

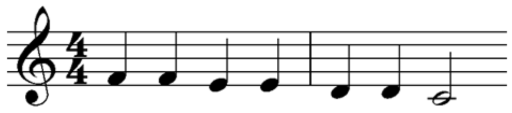


Figure 8.12: Measures 3–4.

`-d-d-d`

To compute the similarity between two melodic contours, such as between “`-U-u-d`” and “`-d-d-d`”, Musicat first computes an “edit distance” between strings, where each insertion, deletion, or substitution operation adds 1 unit of distance. The distance between these strings is 2; it takes one substitution to replace the “`U`” with a “`d`” and another to replace the “`u`” with a “`d`”. Another one-time operation is allowed in edit-distance computation, also with a cost of 1 unit of distance: one contour string can be replaced with its “vertical” inverse, where each “`U`” or “`u`” turns into “`D`” or “`d`”, respectively, and vice versa. In our example, the string “`-U-u-d`” could be replaced with “`-D-d-u`”, and then compared with “`-d-d-d`”; however, this new string still has distance 2 from the second contour string, for a total distance of 3, so the inverse route isn’t helpful in this case. In contrast, the two melody fragments “CDEFG” and “GFEDC” yield the contours “`uuuu`” and “`dddd`”, and without the inversion operator they would have distance 4, but allowing this operation results in a distance of 1 between these contours.

Once the edit distance has been computed, the similarity score (which is also the strength of the Contour relationship) is based on the distance divided by the maximum possible distance:

$$ContourSimilarity(x,y) = 100(1 - \frac{dist(x,y)}{maxDist})$$

where

$$maxDist = max(length(x), length(y))$$

The contour similarity of "CDEFG" and "GFEDC" is thus:

$$100\left(1-\frac{1}{5}\right) = 80$$

The similarity between the two Twinkle, Twinkle measure groups ("CCGG | AAG" and "FFEE | DDC") is:

$$100\left(1-\frac{2}{7}\right) \approx 71$$

Note that this last score is higher than one might expect at first based on the melodies' shapes; the similarity comes from the three pairs of repeated notes in each group.

Antecedent → Consequent Tonality

Groups may be perceived (by codelets) as ending in either a tonic or a dominant tonal function. For example, a stepwise descent to the tonic might be perceived as having a strong tonic function (implying a I chord), while a relatively long note on the dominant (G) or the supertonic (D) might be perceived as having strong dominant function (implying a V chord). An Antecedent–Consequent Relationship may be generated for a pair of adjacent groups of which the first has been perceived as dominant and the second as tonic. The strength of the relationship is simply the average of the perceived dominant and tonic strengths.

Scoring Groups

On the Difficulty of Grouping

Grouping in Musicat is different than in Copycat and other related projects because in listening there is time-based pressure to quickly form mental representations and one doesn't have the opportunity to carry out much retroactive modification of these quickly-formed representations. Fortunately, listeners make many quite sophisticated groups using relatively superficial cues. The key consists in having good heuristics and choosing the correct cues so that initial groupings are very good.

In music, an extreme change in one of many different parameters (such as pitch, dynamics, timbre, or duration) can be a strong indicator for forming a group boundary (according to GTTM's Grouping Preference Rule 3). In Musicat's microdomain, the only relevant factors in this list are pitch and note duration. Also, rhythmic gaps (relatively large time intervals between successive note attacks) are very strong clues to grouping boundaries. For this reason, Musicat's codelets suggest group boundaries at large pitch leaps, rhythmic gaps, and places where note density changes rapidly.

Grouping is a difficult problem in Musicat because there are in general many different possible groupings, and moreover, grouping can be hierarchical. For meta-groups, additional rules come into play. First, according to Grouping Preference Rule 4 in GTTM, higher-order group boundaries are suggested whenever the boundaries detected as described in the previous paragraph are relatively strong. A very large rhythmic gap suggests a group boundary at a higher level. Additionally, meta-grouping should occur in such a way that components have roughly equal size, and there should only be a few of these components (2, 3, or 4, ideally, but not 1 (GTTM, Grouping Preference Rule 1), and probably not 6 or 7). Finally, meta-groups should be grouping parallel structures — this is where analogy comes into play.

On the Difficulty of Group Scoring

Of particular difficulty to implementing Musicat is that to compare any two rival groups we need a concrete, computational way to judge the relative strength of two groups. I have experimented with many possible approaches. The strength of a group is needed in several different circumstances in Musicat:

1. in fighting it out with another group to see which one will exist;
2. in calculation of the happiness of a structure (for structures, such as analogies, of which the group is a component);
3. in the probabilistic selection of whether or not to create a group;
4. in the probabilistic selection of whether or not to destroy a group.

Item 1 is the easiest to handle: in the case where we are comparing two objects, it is easy to let them "fight" by using a simple formula to add up points for various features that influence group strength, then to compare the two strengths, and flip a coin biased in a way that depends on the strengths of the two objects. In this scenario, the units or scale of the strength score is irrelevant; if necessary, we can normalize the score to a desired range by dividing by the sum of the strengths of the two groups. In contrast, items 2–4 involve an isolated group or a group in a situation where we would need its strength to be computed on some sort of absolute scale (between 0 and 100, say). Here are three possible methods to derive a meaningful group strength within the desired range 0–100:

1. Design the scoring function carefully as a weighted sum of range-restricted subcomponents, so that the minimum is 0 and the maximum is 100. This is the approach taken in Copycat.
2. Design a very simple strength function (such as the weighted sum of an arbitrary number of factors) that can become arbitrarily small or large. Track the statistics of recently scored groups, using a moving window of the past N strengths that have been computed. Normalize each calculated strength by computing its z-score relative to the statistics in the recent window.
3. Method 2 has the disadvantage of comparing apples to oranges; scores of a short group of just 4 eighth notes could be compared with scores assigned to a huge group spanning 16 measures. So method 3 is a refinement of that idea, where instead of statistically comparing all groups with each other in the same pool, many different moving windows of statistics are maintained — one for each particular context in which strengths are computed. The tricky part is deciding what the context is.

To implement method 3, we require explicit knowledge of which alternate structures the current proposed group is competing with. In Copycat, competing structures are destroyed because only one consistent view of the workspace is maintained at a time. In Tabletop, however, the notion of having alternative complete structures present in the workspace, with only one current Worldview, allows the program to store the history and the necessary statistics.

Although I used method 3 for some time I have returned to the simple approach of method 1, with a slight modification. I have found the current approach to be sufficient when I have a simple enough strength function.

Basic Formula

Each group is associated with a list of "group reasons" that describe why it is a group. This list is crucial to the group's existence; without the list of reasons, the group would simply have a score of 0 and would vanish from the workspace. The problem with the simple weighted-sum formula (as used in Copycat) is that Musicat does not compute an exhaustive list of "features" of each group. Most typical AI approaches involving scoring possible representations, such as (Lartillot, 2004), require all features to be computed. In contrast, Musicat uses only those features that have been brought into perceptual focus by codelets, so each group might have a different collection of known features. This is especially true during the initial creation of a group, before additional codelets have run to examine it in more detail.

Musicat uses a slightly modified version of the simple weighted sum discussed above (method #1) to address the problem of scoring a group without computation of the exhaustive feature list. After computing the weighted sum of scores of all currently existing group reasons, we subtract a weighted sum of scores of group penalties, and then we run the sum through a squashing (*e.g.,* sigmoid) function to ensure that the result will remain in the range 0–100.

$$GroupScore = \sigma\left(\sum_{R_i \in GroupReasons} w_i R_i - \sum_{P_j \in GroupPenalties} w_j P_j\right)$$

with sigmoid function $\sigma(x) = 100\left(\frac{2}{1+e^{-2x/100}} - 1\right)$, and coefficients w_i and w_j determined simply by looking up the weight associated with each type of group reason or group penalty.

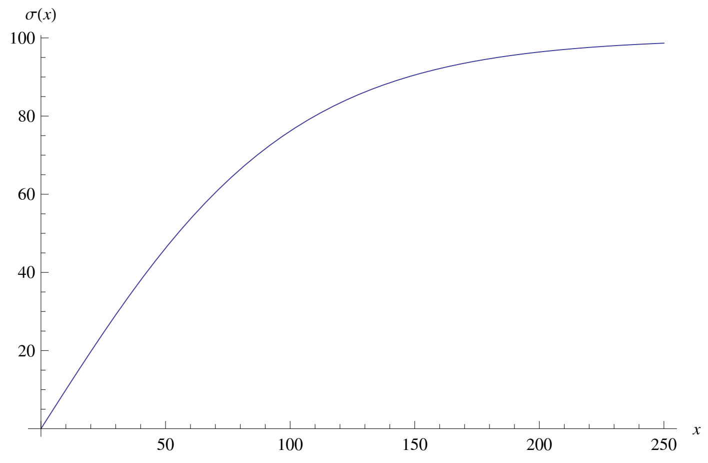


Figure 8.13: Sigmoid function.

The sigmoid allows us to be less strict about the weights in the linear sum: we can remove the constraint that the weights sum to 1. The downside is that if weights are too large, group scores might all become too high and might all squash down to essentially the same value near 100. Note that the function used here is just the right-hand half of the more typical S-shaped sigmoid. Often just one or two group reasons are enough to justify a group's existence, so I used a function that behaves nearly linearly near 0; there was no need to include the part of the S-shape that acts as a minimum activation threshold (this is used, for example, in many neural net models as well as in other FARG programs, such as Phaeaco).

Scoring Sequences

Because a sequence is a subclass of the Group class in Musicat, scoring a sequence is the same as scoring a group. A sequences can have a list of group reasons just as can any other group, but it can also have a special sequence group reason with a score determined by the number of times the sequence's motif repeats and by the length of this motif.

Scoring Analogies

An analogy in Musicat is a collection of relationships between two groups and their components. Analogies, like groups, have an associated score (or equivalently, strength). These scores are important for several different reasons, including:

- focusing codelet attention;
- determining happiness in the workspace;
- providing support to the groups involved;
- providing support to higher-level analogies;
- determining whether an analogy persists or gets destroyed in the workspace.

These ways of using analogy scores were important influences on my choice of how to score an analogy. I finally settled on the idea of a weighted sum of the following four factors (weights given in parentheses):

1. Size of the analogy (20%)
2. Completeness of the mapping (35%)
3. Strength of component relationships (35%)
4. How long the analogy has survived in the Workspace (10%)

Size of Analogy

Taking into account analogy size (in terms of number of measures) in the scoring of analogies is surprisingly subtle. On the one hand, small analogies may involve identical or nearly-identical components that are near to each other in time. Consider a sequence of three quarter notes, C-D-E, followed by three other quarter notes, E-F-G. It is easy for people to make an analogy between these two short groups. But now consider the larger-scale structure of the song "On the Street Where you Live". Its first 16 measures are repeated nearly exactly, except that the final two measures are modified to reach a stronger cadence. Is the analogy between the first 16 measures and the next 16 measures stronger than the analogy between C-D-E and E-F-G? The smaller analogy is more immediate and can thus be perceived more directly, whereas the longer analogy involves many more notes being mapped onto each other. Which should be scored more highly? We can also consider a non-musical analogy: in search of one, I looked out the window and noticed a tall tree covered with green leaves. I looked at one particular leaf and then saw a nearly identical leaf just below it, and of course, all the thousands of leaves on the tree were each very similar to each other, comprising a huge number of small leaf–leaf analogies. Then, "zooming out", I noticed another large tree next to the first. It looked like the same kind of tree, and it was trivial to make an analogy between these two trees. In this case, I couldn't even see most of the second tree; it wasn't visible through the window, but I could see a few branches and many leaves. These two trees have quite different structures when I look at the detailed patterns of their branches and leaves, but they are clearly two highly analogous trees. Is the tree–tree analogy stronger than any individual leaf–leaf analogy?

The answer for Musicat is "yes". Larger analogies are scored more highly than small analogies. Larger analogies are more difficult to form because they may involve more components, and because by definition they span larger segments of time. Musicat should pay attention to these large-scale analogies, should make sure they are given adequate codelet resources to flesh them out fully, and should recognize that the groups involved in a large-scale analogy are very strong.

Completeness of the Mapping

An analogy is a mapping between two groups and their components. Musicat enforces a time-ordering on the mapping so that the first element of group A must map onto the first element of group B, the second element of group A onto the second element of group B, and so forth. Subject to this constraint, pairs of elements (one from each group) are put into correspondence. If any such pair is found for which no relationships exist in the Workspace to justify the correspondence, the analogy's score is reduced. The maximum possible score is reduced by the percentage of unmapped components found in the analogy.

Strength of Component Relationships

Just as a group's score is computed as a function of a weighted sum of group reasons, an analogy's score is derived from the relationships that form its basis. For each mapped pair of items in the analogy, a weighted sum of the relationships associated with the pair is computed and then this is run through a sigmoid function, just as in group scoring. Then the average such score for all pairs of analogy components is computed.

How Long the Analogy has Survived

Groups become more stable over time; as new notes are "heard", the age bonus associated with a group is increased. Since analogies are the most complex objects in the Workspace, they also need a mechanism to help them stabilize over time. Each analogy receives a bonus based on how much time (measured in codelets) has passed since it was created.

Scoring Expectations

Groups

The strength of an expected group is based on the strength of the source group it was derived from. The base strength is multiplied by a weakening factor (such as 0.5) according to the intuition that group expectations are more tentative than already-discovered groups. A bonus term based on the strength of any expected measure links associated with the group is added to the base strength score.

Analogies

As was the case with groups, the strength of an expected analogy is based on the strength of the source analogy, multiplied by a weakening factor such as 0.5. An expected analogy is further weakened if the size of the source groups is a "strange" size, in that the group size does not fit in well with that implied by the thicknesses of relevant barlines.

Measure Links

Expected measure links are assigned the same strength as the original measure links they are derived from.

Creating Structures with Codelets

This section describes in broad strokes how Musicat's codelets work to create structures in the Workspace. For readers desiring more specifics without reading the source code, Appendix B gives more detail for each individual codelet type.

Each section below describes, for each Workspace structure, some of the codelets that directly contribute to creating the structure.

Rhythm-based Measure Links

Measure links based on rhythmic similarity are among the simplest structures for Musicat to create. A **Measure Linker** codelet will examine two measures, selected probabilistically, with a preference for recent measures (unless otherwise specified by the

Measure Linker's parent codelet). The rhythmic similarity between the measures is computed and a link may be created, with probability of creation reflecting the similarity.

Measure Link Breaker codelets, on the other hand, examine measure links and may destroy the links, with a probability again based on the similarity score. However, these codelets leave certain links alone: if a measure is involved in only one link, that link will be spared destruction. Additionally, if a link is stronger than other links involved with either of these measures, the link is spared. Only the "weakest links" have a chance of being destroyed. Note that any measure can link to many other measures, and allowing a great many links to persist in the Workspace is generally not a serious problem because they do not interfere with other structures. This is in contrast with structures such as groups, which cannot overlap. An overabundance of measure links may reduce Musicat's focus, so Measure Link Breaker codelets are necessary. Removing measure links *quickly* is not necessary, however, so the codelets are carefully designed to only remove the weakest, most unnecessary links.

Relationships

Several types of Relationship objects can be created in the Workspace. Generally, each relationship type is created by a separate codelet designed to look for relationships of that type. Relationships generally involve either rhythm or pitch separately (although with analogy-based relationships, both may be involved)

Rhythm

The general "Look for Relationship" codelet tests two components from the Workspace, which can be measures or groups, and checks for an already-detected similarity between them. For example, it might find a rhythmic measure link, and then might create, with probability based on the link strength, a "Similar" relationship (or an "Identical" relationship, if the rhythms are identical). More interestingly, it might find that an analogy between the two components already exists in the Workspace, and then create a "Similar" relationship based on the analogy. If these operations seem trivial, recall that relationship objects are important because they represent connections between two objects in a simple way that can be used to build up more complex analogies and other relationships. Generating a relationship from an analogy is precisely how Musicat implements "chunking" in memory.

The "Look for Starting Relationship" codelet looks for rhythmic similarity between the first parts of the components. It restricts the comparison to the first half of the shorter of the two components involved (with a two-measure maximum size) and computes rhythmic similarity. A "Start Similar" or "Start Identical" relationship may be created, again with probability based on the degree of similarity, and added to the Workspace.

Pitch

Now let us consider codelets operating in the domain of pitch, instead of rhythm. the "Look for Contour Relationship" codelet computes the melody contour of two groups and then may choose to create a "Contour" relationship linking the groups. As a reminder to the reader, contour relationships are displayed in the user interface as pink curves connecting the groups in question.

A rather different type of relationship is created by the "Look for Antecedent Consequent Relationship" codelets. This looks for two groups where the first ends on a pitch that might have a dominant chord function, and the second ends on the tonic. Of course, this is an extremely surface-level and simplified pitch relationship to find in a melody, and a true relationship between antecedent and consequent phrases has many more components. This is just one heuristic that Musicat uses in making connections between groups.

Bar Lines

The thickness of the bar lines in the Workspace is modified continually by **Measure Hierarchy** codelets. Each of these codelets selects a bar line at random from a window representing short-term memory, and chooses one of five actions at random to perform, using this bar line. Essentially, this means that each **Measure Hierarchy** codelet chooses the role of one of five completely different codelet types at run time; the five separate types are contained in the same **Measure Hierarchy** wrapper for historical reasons.

Bar-line thickness contributes to the strength of group boundaries. Conversely, strong group boundaries contribute to bar-line thickness. Each time Musicat encounters a new bar line as time moves forward, the line is initialized with the thickness expected if the music were following a typical binary pattern of alternating strong and weak measures, with this alternation happening recursively at higher and higher levels (longer and longer time spans, with each power-of-two length corresponding to a thicker bar line). Codelets can then modify this default thickness based on the endpoints of large groups in the Workspace.

Bar-line thicknesses are integer-valued, with the thickness of a bar line roughly corresponding to the expected length of groups having a boundary at that bar line. For a typical binary structure, the thickness would be the base-2 logarithm of the group length. For example, the thickness of the bar line between the first and second measures would typically be “0”, corresponding to minimum thickness, the thickness between measures 2 and 3 would be “1”, and the thickness between measures 4 and 5 would be “2”. The value 2 is the base-2 logarithm of the expected group length, 4 measures; a 4-measure group at the start of the piece would end just before this thickness-2 bar line.

Measure Hierarchy codelets must assign integer thicknesses to bar lines. However, it seems natural to program codelets to make gradual changes to thickness. Thus, the Workspace maintains a real-valued bar line thickness behind the scenes, simultaneously with the integer thickness. Some codelets will modify the real-valued thickness, while others will make the more significant change of modifying the integer-valued thickness by adding or subtracting 1, thus moving the integer thickness closer to the real-valued, hidden version. This allows for gradual modification by many codelets, with sporadic jumps from one integer thickness value to another.

The five **Measure Hierarchy** codelet possibilities are:

1. move integer thickness closer to the real-valued thickness;
2. decrease real-valued thickness towards zero;
3. apply a boost to real-valued thickness, based on the size of a group ending before the bar line;
4. add random noise to the real-valued thickness;
5. if the bar line is relatively thick (thickness greater than 1), move the real-valued thickness of the next bar line towards 0.

Groups and Meta-groups

In Musicat, groups can rapidly come into and go out of existence. Recall that the strength of a group depends on the list of group reasons that support the group; at the moment of its creation, a group may just have one such reason, and hence a relatively low strength. Such a group may easily be destroyed by a **Group Breaker** codelet. A group may rapidly increase in strength, however, if codelets discover good reasons that are added to the group's description. Codelets that discover group reasons are described below. First, however, I answer the question of how groups initially come into existence.

Groups always consist of adjacent measures, so most codelets that create groups do so by examining adjacent measures and seeing whether a group is justified. The same principle applies to meta-groups: a meta-group must be composed of adjacent groups. The basic grouping codelet is the **Proximity Grouper**. A **Proximity Grouper** codelet chooses two adjacent measures or groups in the Workspace and checks whether there are sufficient grounds to form a group (or meta-group) from the two elements. If a group is proposed, it is assigned a relatively weak initial strength of 50 (out of 100) and, like all new structures, must survive a standard probability test (a random number is generated and must be less than the structure's strength — in this case it amounts to a coin toss because the probability is 50%); moreover, the proposed new structure must compete with (and destroy) any existing incompatible structures before being added to the Workspace. If a new group is formed, an initial group reason is added to get it started with a non-0 strength. This is the very simple "Number of Elements" reason, which awards points to a group for having 2, 3, or 4 components, with a higher score given for 2 or 4 rather than 3 components. In this case, points are awarded because two adjacent elements have been grouped together,

The majority of elements available to the **Proximity Grouper** codelet are measures. Although it can create meta-groups, measures typically account for at least half of the possible group components, as we see if we consider the grouping structure of a melody as a complete binary tree structure — half the nodes in such a tree are "leaf" nodes, or measures in the case of a melody. Because of this distribution, **Proximity Groupers** don't try to create meta-groups as often as one might expect. To correct the problem, I designed a separate codelet type whose exclusive goal was to create meta-groups. The **Meta Grouper** codelet works much like the **Proximity Grouper**, but using groups as components. An optional parameter for the **Meta Grouper** codelet exists to specify a link to an existing analogy between adjacent groups. Other codelets may use this parameter to add pressure to form a meta-group based on an analogy. If an analogy has been specified and a meta-group is created, it will start with an "Analogy Support" group reason in its description.

Other codelets try to create groups of adjacent elements based on more than mere proximity. Each **Measure Sameness Grouper** codelet chooses a rhythmic measure link from the Workspace, with probability based on both the strength of the link and the recency of the most recent measure involved in the link. If the endpoints of the link are adjacent measures, the codelet attempts to group the measures together. If successful, the new group will start with either a "Components Identical" or "Components Similar" group reason. A group created in this way will thus start with a higher initial strength value than one created by a **Proximity Grouper**.

Another type of codelet, **Find Sequence**, looks for three or more elements in a row that have the same pitch contour and attempts to make a special Sequence group out of those elements. The final element in a sequence is allowed to be longer than the initial elements without penalty, reflecting the special case that in a sequence of three or more items, often the final item is extended to round out the musical phrase so that it has an even number of measures (recall Figure 8.8). Just as with groups formed by **Measure Sameness Grouper** codelets, sequences are created with an initial "Sequence" Group Reason, resulting in an initial group strength that is higher than for basic proximity-based groups.

Analogies

As we have seen, groups in Musicat often are created by codelets for very mundane reasons (such as mere the proximity of two elements) and then are either destroyed or strengthened by other codelets. Analogies, on the other hand, require stronger support before they come into existence, and are given more time to solidify before they risk destruction by other codelets. In particular, the **Destroy Old Analogy** codelet attempts to destroy only analogies with low strength that were created several measures earlier.

The **Create Analogy** codelet is, unsurprisingly, the standard codelet responsible for analogy creation. Of course, to say one codelet is responsible is slightly misleading, as it depends on the prior actions of many other codelets to create groups and relationships of various sorts. The job of the **Create Analogy** codelet is to suggest a candidate analogy once the necessary components for a mapping between two structures are in place. The codelet may have been posted to the Coderack along with parameters specifying which two elements to include in a potential new analogy. If these were not specified, the codelet starts by choosing (probabilistically, as usual) an existing relationship that links an earlier group or measure to a recent group or measure in the Workspace. The earlier element is denoted the Left Hand Side (LHS) and the recent element is called the Right Hand Side (RHS).

Once the LHS and RHS of the potential analogy are identified, a temporary analogy object is formed. The subcomponents of each side are identified and any existing relationships between an LHS subcomponent and an RHS subcomponent are added to the temporary analogy. Not all relationships are valid: within an analogy, relationships compete with each other, much as a new potential group competes with any existing groups with which it would overlap. For example, if a relationship has been added to the analogy that maps element A of the LHS has onto element C of the RHS because both elements have the same pitch contour, it is not valid to add to the same analogy a second relationship in which element A is mapped onto element D of the RHS because they have a similar rhythm. The mapping must be consistent.

Once the relationship-based mapping from LHS to RHS is established, the **Create Analogy** codelet tries to add the analogy to the Workspace. The probability of adding the analogy is determined by the strength of the analogy. As is the case with groups, once the analogy is added to the Workspace, other codelets can attempt to strengthen the analogy.

During the process of creating a temporary analogy described above, **Create Analogy** codelets post many other codelets to look for additional relationships, groups, or analogies. For example, a lack of possible relationships to add to the mapping suggests places to look for new relationships; new codelets are posted to search in the right places. The **Add Relationships to Analogy** and the specialized **Add Nonrhythm Relationships to Analogy** codelets provide this focused search for relationships missing in the mapping structure of an analogy. Similarly, new **Meta Grouper** codelets are created to look for a higher-level group encompassing both the LHS and RHS.

Sometimes the placement of an analogy in the Workspace suggests another analogy. Specifically, when the LHS and RHS are separated in time, and a thick bar line occurs before the RHS, it is natural to expect a second, parallel analogy The **Suggest Parallel Analogy** codelet looks for situations of this sort. Rather than creating an analogy itself, this codelet simply posts new **Create Analogy** codelets with parameters indicating where an analogy is hypothesized, as well as new **Proximity Grouper** and **Meta Grouper** codelets to create groups that would be necessary parts of the hypothesized analogy.

Analogies are the most complex objects in the Workspace. While watching Musicat run, we observed that sometimes a very promising (to human eyes) analogy forms but is later destroyed, despite its apparent strength. A certain amount of creation and destruction is fundamental to the FARG architecture. For a complex object, however, the difficulty of creation makes complete destruction and re-creation infeasible, especially given Musicat's real-time nature. Indeed, the program focuses on objects in recent musical time, so if a complex older object is destroyed, it is unlikely to be recreated before the musical elements in question have disappeared too far in the past to access.

The Tabletop program also faced the issue of the difficulty of re-creating complex objects. The solution adopted by French (1992) was to introduce the idea of a "Worldview", distinct from the Workspace (see Chapter 2). I used a similar strategy in Musicat. The **Store Strong Analogy** codelet selects a recent strong analogy and has a chance to add the analogy (including all its consistent groups and relationships) to a permanent record for later recall in case it is destroyed. The **Resuscitate Analogy** codelet can attempt to bring back a previously-discovered analogy from memory all at once. This allows a complex analogy to be revived when the normal mechanisms of gradually building-up groups and relationships might not be sufficient given the real-time pressure of the listening domain.

Expectations

Codelets related to expectations can be divided into two types: those that create expectations, and those that use the expectations later on to influence grouping and analogy-making in the Workspace.

The **Generate Expected Group Copy** codelet creates a basic kind of expectation: its job is to add an expected group to the Workspace. It does this by finding a strong group that has ended just before a relatively thick bar line, and attempts to generate an expectation for a group of the same size to come after the bar line. The structure of the group is also expected: any subgroups and measure links associated with the group are also turned into future expectations. In addition, a high-level analogy is suggested (*i.e.*, added to the list of expectations) to link the presently-existing source group with the future expected group. The **Generate Expected Analogy Copy** codelet does the related job of generating an analogy that is expected to occur in the future, between two expected groups. It does this by looking for expected groups that have already been generated by the **Generate Expected Group Copy** codelet, and then building expectation for analogies between the groups if there were analogies between the relevant source groups that generated the expectation. Basically, these two codelet types work together to create expected future groups and expected analogies between future groups.

Several more codelet types use the generated expectations to try to create real structures when the time comes. The **Suggest Link From Expectation** codelet does this for measure links. This codelet always focuses on the most recently heard measure, and looks for expected links that end on this measure. If such links are found, one is selected at random and the codelet attempts to build a real link between the suggested measures. The **Suggest Group From Expectation** codelet selects an expected group, ignoring any groups which exist entirely in the future. If at least the first measure of an expected group is a measure that has already been "heard" by the program, then the expected group can be used to create a real group. Note that when such a group is formed, it often contains just a single measure. For example, if an expected group spans measures 9–12, once measure 9 is heard, a codelet might try to form a real group starting on measure 9, and only containing that single measure. Another type of codelet, **Extend Group Right**, is responsible for attempting to extend groups to the right, especially when the extension is suggested by the existence of an expected group extending to the right. Finally, the **Suggest Analogy From Expectation** selects, at random, an expected analogy in which the measures that would make up the right hand side of the analogy have been heard (*i.e.*, the entire analogy must exist in the "heard" portion of the Workspace, not in the future.) This codelet simply posts a **Create Analogy** codelet to the Coderack to attempt to build the specified analogy.