DAWPL: A Simple Rust Based DSL For Algorithmic Composition and Music Production*

ARMIN NAMAVARI, Stanford University, USA

Additional Key Words and Phrases: algorithmic composition, music, music theory, rust, SuperCollider, DSL, metaprogramming

ACM Reference Format:
Armin Namavari. 2017. DAWPL: A Simple Rust Based DSL For Algorithmic Composition and Music Production. 1, 1 (December 2017), 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 SUMMARY

Algorithmic music composition languages provide a particularly interesting challenge for language, abstraction, and system design. The term "music" encompasses a wide variety of things, from the production noise from signal generators, to a notation widely understood by musicians that expresses melody and rhythm. In this project, we investigate DAWPL (Digital Audio Workstation Programming Language), a simple DSL (domain specific language) that uses Rust to cater to the expressiveness of standard music theory (within the context of DAWPL, we consider Jazz theory in particular) and the organizational scheme of a DAW. DAWPL provides an API through which users can generate, sequence, and process sounds. DAWPL code is translated to SuperCollider (a popular and well-established algorithmic composition language), which actually produces the sound. DAWPL provides a more syntactically intuitive way for musicians to interact with algorithmic composition (it also carries with it Rust's rich syntax and type system). Furthermore, DAWPL brings with it the ability to write extensions in Rust, which has a much richer and more flexible type system. We will analyze and compare the expressiveness of DAWPL and SuperCollider and comment on how their individual attributes reflect their desired use-cases. We will also comment on the overall structure of DAWPL (and its implementation) and reflect on how Rust's language features facilitate its implementation. Possible use-cases for DAWPL include providing artists the ability to "prototype" tunes and productions , build tools to facilitate composition (e.g. "Auto-improvisers"), generalize music composition patterns, or generate complicated and intricate tunes in a programmatic fashion.

2 BACKGROUND

Music itself has a rich and complex structure that requires a rich and complex notation and system of abstraction. Musical notation, like the kind one often sees on sheet music, more or less describes pitches, durations, and volumes. Pitches are described using the letters A through G (these correspond to the white keys on the piano). There are also tones between these notes that are described using modifiers calls "sharps" and "flats" as in "A flat", for instance. When the frequency of a note is doubled, the note retains its name, however, it is said to be in a different octave.

*We can add a note to the title

Author's address: Armin Namavari, Stanford University, Stanford, CA, 94305, USA, gang_zhou@wm.edu.

XXXX-XXXX/2017/12-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

The distance between two notes is called an interval and a sequence of notes played in succession according to a particular pattern of intervals is called a scale. The position of a note within a scale is called its degree. A particular pattern of degrees of a scale determine a chord. For instance, a major chord is defined as the first, third, and fifth degree of a major scale. Table 1 summarizes the hierarchy and relationship of notes, scales, and chords.

Table 1. Melodic Abstraction Hierarchy

Chord: A combination of notes, often defined relative to a particular scale
Scale: A sequence of notes given by an interval-sequence formula
Note: A tone of a certain frequency

There is a rich and complex theory of what orders of chords and patterns of notes sound "good". Songwriters and producers often use chord progressions (a particular sequence of chords) as the foundation for many songs. Melody is often dependent on the underlying chord progression. One can think of a chord progression as the context within which a melody exists.

It is also worth mentioning certain rhythmic abstractions, such as beats – a quantized unit of time, and durations – a specified amount of time. Rhythm dictates when we play certain notes and for how long. In DAWPL, users can indicate ryhthm using the following mapping between letters and note-durations (Q-quarter, H-half, W-whole, E-eighth, S-sixteenth). The default has one whole-note as corresponding to one second, however, the vector can be scaled via a map to change the tempo.

In addition to these traditional music theory abstractions, modern music production applications called DAWs (digital audio workstations) enable users to interact with and modify sound in a different way. Below is a screenshot of Logic (image source)

A DAW consists of clips, which represent audio signals and the effects applied to those signals. Those clips are sequenced within a particular track. All the tracks together comprise an arrangement. In the figure above, the colored boxes are the clips and the rows they are on are the tracks. The different tracks are the rows that these clips are in. One could argue that traditional music notation has a notion of tracks as there are different musical parts written for different instruments. This organizational scheme is familiar to music producers.

Inevitably, producers and composers alike recognize patterns among their compositions and might even wonder if there is a way to automatically generate compositions based on a particular pattern or template. It turns out there are classes of programming languages that facilitate "algorithmic composition" i.e. programmatically creating and editing sound. Often, these languages, although powerful, have strange and unwieldy syntax. As

a particular case study, we will consider the algorithmic composition language SuperCollider. SuperCollider, although highly powerful, caters more to a DSP (digital signal processing) audience as opposed to musicians and producers. It does not have many fundamental music theory constructs built in and does not impose many organizational restrictions in terms of how code is written.

It becomes natural to wonder how one could tap into the power SuperCollider has to offer, without having to use SuperCollider syntax and having a richer system of abstractions layered on top. This is the problem DAWPL (DAW programming language) aims to solve. DAWPL seeks to provide a programming environment that is highly expressive for those coming from a music theory and/or music production background (as opposed to a DSP background). The interface DAWPL offers allows those with more of a music background to quickly and clearly express musical ideas in a way that more closely resembles what they are familiar with. Furthermore, being implemented in Rust, DAWPL provides users and extremely flexible and modular way to interact with the abstractions its API provides. This all ties back to a key challenge and compromise we discussed in CS242: expressiveness.

SuperCollider is overall more powerful, as it possess many features that allow it to interface with different audio technologies and low level audio. However, DAWPL provides a higher level interface that makes idioms and constructs such as chords, scales, notes, and tracks easily available to the composer, allowing them to express their ideas in a more straightforward manner.

Later in the paper, we will discuss and compare the expressiveness of DAWPL and SuperCollider. We will do this in the context of a simple code snippet comparison. We will discuss what trade-offs they make in the realm of expressiveness and how these trade-offs reflect their intended use cases. We will also include a discussion of DAWPLs structure and the structure of its Rust implementation.

3 APPROACH

Core to DAWPL is a set of music theory inspired abstractions and DAW inspired abstractions. The music theory abstractions that are implemented in the software are exactly those described in Table 1. In addition to these, DAWPL employs another 5-Layer abstraction hierarchy (with the theory components existing within the signal)

Table 2. The 5-Layer DAWPL Abstraction Hierarchy

Arrangement: a combination of tracks, the “big picture” of the composition
Track: a time sequenced series of clips
Clip: a combination of signals
Signal: A description of a sound e.g. a waveform or mp3 file
Audio: the physical production of sound by a device

Upon first thought, it may seem as if this system of abstraction poses a limitation. It actually doesnâĂŹt – all it really does is force the user to organize their code in a certain way – in the same way they would if they were dealing with a piece of DAW software. This leads to improved readability of code and a style standard. Through this model, DAWPL achieves a type of expressiveness that resonates with those familiar with this kind of software and this general music production paradigm. By realizing this paradigm within a programming language, we give ourselves the opportunity to programmatically generate and modify arrangements such as these – as opposed to how one manually might with the GUI interface of a DAW. Here DAWPL exhibits expressiveness by capturing a pattern for how many music arrangements are described.

3.1 Implementation of Core Datatypes

Using DAWPL, a user can easily declare notes, chords, and scales and use simple operations with them e.g. indexing into a scale, finding a particular voicing for a chord. A note is internally represented as a struct that contains a Name (enum) for the note name and an octave. There is a global mapping between notes and MIDI values (in both directions). MIDI is a system that represents notes as eight-bit numbers. It is easily understood by SuperCollider, but quite opaque to most who are well-versed in theory. In this way, DAWPL is more expressive than SuperCollider in that users can interact with notes as opposed to eight-bit numbers.

Scales are defined in the code by a base note of the scale and a scale type. The scale type is an enum that maps to an interval sequence formula. There are functions for indexing into scale by an offset from the base note as well as by "Arabic numbers" (a 1-indexed system of identifying chord degrees, which is more familiar to musicians).

All chords are defined relative to a major scale in terms of Arabic number voicings. These voicing formulas are stored as a global constant. In the future, I might modify the formula initialization code (for both chords and scales) with something that loads the formulas from a config file.

This is how chord formulas are currently represented in the code:

```
lazy_static! {
    static ref CHORD_FORMULAS: HashMap<ChordType, Vec<ArabicNum>> = {
        // negative numbers will denote flat tones (relative to a major scale)
        [(ChordType::Maj7, vec![ArabicNum::Natural(1), ArabicNum::Natural(3),
            ArabicNum::Natural(5), ArabicNum::Natural(7)]),
         (ChordType::Min7, vec![ArabicNum::Natural(1), ArabicNum::Flat(3),
            ArabicNum::Natural(5), ArabicNum::Flat(7)]),
         (ChordType::Dom7, vec![ArabicNum::Natural(1), ArabicNum::Natural(3),
            ArabicNum::Natural(5), ArabicNum::Flat(7)]),
         (ChordType::Dim, vec![ArabicNum::Natural(1), ArabicNum::Flat(3),
            ArabicNum::Flat(5)]),
         (ChordType::Aug, vec![ArabicNum::Natural(1), ArabicNum::Natural(3),
            ArabicNum::Sharp(5)]),
         (ChordType::Maj6, vec![ArabicNum::Natural(1), ArabicNum::Natural(3),
            ArabicNum::Natural(5), ArabicNum::Natural(6)])]
         .iter().cloned().collect()
    };
}
```

There are three kinds of clips: instrument, audio file, and empty. (File has yet to be implemented) Each clip consists of a name and parameters relevant to that clip. An instrument, for instance, consists of its name (for the clip), the name of the instrument (some sort of synth/wave generator), a vector of notes (option types containing MIDI values), and a vector of floats that represent durations for the notes. Rests (i.e. the absence of a note) would be indicated using None. Empty clips are clips that just consist of rests – they allow for pauses within a track.

Tracks are a surprisingly simple data structure. They consist of only the track name and a vector of strings, which correspond to clip names. This setup facilitates the translation of the DAWPL abstractions to SuperCollider code later on.

Finally, the arrangement container consists of a vector of tracks and a vector of clips. The reason is set-up this way has to do with translations. When clips are translated by DAWPL into SuperCollider, they are given variable names. Variable names must all be declared within the same place in a particular SuperCollider scope. Therefore, it is convenient for the translator program to know the full and complete set of clips that will be used in the composition. It is also convenient for the programmer to organize their code in this way. By having the

complete set of a clips all in one place, someone reading the code does not have to search through all the tracks to figure out what clips are loaded and available.

3.2 Convenient Macros

There are additional convenient macros for declaring notes, chords, instrument clips, tracks, and rhythms. Rust has a powerful and flexible macro system that facilitated the definition and usage of these macros.

Below is an example of one such macro: the `track!` macro:

```
macro_rules! track {
    ($name:ident, $( $clip_name:ident),*) => {{
        let mut clip_names: Vec<String> = Vec::new();
        $(
            clip_names.push(String::from(stringify!($clip_name)));
        )*
        Track::new(String::from(stringify!($name)), clip_names)
    }}
}
```

This macro enables users to easily and conveniently assemble the track name and the names of associated clips into a track.

3.3 Translation

In order to translate a full arrangement from DAWPL (Rust) code to SuperCollider, I defined separate functions for translating each modular component. The `format!` macro in rust is also indispensable for compiling the entire SuperCollider program. Below is a brief code snippet of the arrangement translation code.

```
pub fn arrangement_to_super_collider(arrangement: &Arrangement) -> String {
    // Handle appropriate variable declaration
    let mut var_decl: String = "var ".into();
    let names = arrangement.get_names();
    for i in 0..names.len() {
        var_decl += &(names[i])[..];
        if i < names.len() - 1 {
            var_decl += ",";
        }
    }
    var_decl += ";";
    // add clip declarations
    let mut clip_decl: String = "".into();
    for clip in arrangement.get_clips_ref().iter() {
        clip_decl += &(clip_to_super_collider(&clip))[..];
        clip_decl += "\n";
    }
    // add track declarations
    let mut track_decl: String = "".into();
    let mut track_name_str: String = "[".into(); // list of track names
    for track in arrangement.get_tracks_ref().iter() {
        track_decl += &(track_to_super_collider(&track))[..];
```

```
        track_decl += "\n";
        track_name_str += &format!("{},", track.get_name());
    }
    track_name_str += "]";
    // put declarations together to form arrangement
    format!(k_arrangement_template!(), instruments=k_instruments!(),
        variable_declarations=var_decl, clip_declarations=clip_decl,
        track_declarations=track_decl, track_names=track_name_str)
}
```

An interesting quirk to notice is that instead of a string constant for the arrangement template, I had to use a macro. This is because `format!` expects a first argument that is a string literal or something that expands to it. As a result, I had to use another macro.

3.4 Reflections on the Development Process

My initial reason for choosing this approach was my fascination with how the abstractions a language has to offer define its usage and expressiveness. Abstractions fundamentally define the interface that a programmer uses to interact with a language, and therefore are a key aspect of expressiveness. Furthermore, it was a lack of appropriate abstractions in SuperCollider that made me want to implement a language that could offer greater musical conceptual richness to users.

Initially, I had considered writing a language from scratch, with custom syntax and everything. I quickly realized that, although that would make for an even more fun and exciting project, there simply was not enough time to develop and test a system that complex. Furthermore, as I mentioned in the paragraph above, the syntax was not as important as the fundamental abstractions for tackling this particular challenge. There also was not much of a point in reinventing the wheel when it came to syntax. Rust itself provides quite powerful syntax among other language features. It would therefore be a favorable environment within which I could situate DAWPL. Furthermore, the macro system Rust has gives programmers great power and flexibility in adding new syntax to the language, as I discussed two sections ago.

Another big challenge I encountered in realizing a music theory system within code was dealing with ambiguity. For instance, because of the cyclic nature of notes and scales, there can often be many different versions of the same chord (these are called “inversions”). One decision I had to make was regarding how these inversions should be represented. I opted to represent inversions by defining a base note and a starting index into the chord formula.

It is often these small, seemingly insignificant decisions that make the development process so difficult. It is rather easy to needlessly obsess over tiny details. Overall, I aimed for simplicity where possible, but never at the expense of expressiveness and explicitness.

In terms of improvements, I can do more error checking to provide users with helpful messages when they attempt to do something invalid within the language – or even something valid that may be compiled to invalid SuperCollider code. Furthermore, I might be able to benefit from caching translations for clips – this would be particularly relevant for large and complex arrangements.

4 RESULTS

Recall that the main metric we wish to analyze and compare between DAWPL and SuperCollider is expressiveness. In order to do this, we will perform a code comparison (for code that produces the same output). Through this comparison, we will point out and notice certain language features and abstractions and discuss their implications for expressiveness.

Below is DAWPL syntax:

```
let ii_chord = chord!(D4, Min7).play();
let V_chord = chord!(G3, Dom7).play();
let I_chord = chord!(C4, Maj7).play();
let mut melody_notes: Vec<i8> = Vec::new();
melody_notes.extend(&ii_chord);
melody_notes.extend(&V_chord);
melody_notes.extend(&I_chord);
let progression_clip = instr_clip!(prog, sine,
    vec![Some(ii_chord), Some(V_chord), Some(I_chord)], rhythm![W, W, W]);
let melody_clip = instr_clip!(melody, sine,
    melody_notes.into_iter().map(|n| Some(vec![n])).collect(),
    rhythm![Q, Q, Q, Q, Q, Q, Q, Q, Q, Q, Q, Q]);
let arr: Arrangement = Arrangement::new(
    vec![track!(progTrack, prog), track!(melTrack, melody)],
    vec![progression_clip, melody_clip]);
println!("Arrangement output: {}", arrangement_to_super_collider(&arr));
```

Here, we have the SuperCollider code it compiles down to:

```
(
SynthDef.new(\sine, {
    arg freq=440, atk=0.005, rel=0.3, amp=1, pan=0;
    var sig, env;
    sig = SinOsc.ar(freq);
    env = EnvGen.kr(Env.new([0, 1, 0], [atk, rel], [1, -1]), doneAction:2);
    sig = Pan2.ar(sig, pan, amp);
    sig = sig * env;
    Out.ar(0, sig);
}).add;
)

(
var prog,melody,progTrack,melTrack;

prog = Pbind(
    \instrument, \sine,
    \dur, Pseq([1, 1, 1]),
    \midinote, Pseq([[62, 65, 69, 72],[55, 59, 62, 65],[60, 64, 67, 71],]),
);


melody = Pbind(
    \instrument, \sine,
    \dur, Pseq([0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25]),
    \midinote, Pseq([[62],[65],[69],[72],[55],[59],[62],[65],[60],[64],[67],[71],]),
);
```

```
progTrack = Pseq([prog,]).do({arg currClip; currClip.play});


melTrack = Pseq([melody,]).do({arg currClip; currClip.play});


[progTrack,melTrack,].do({arg currTrack; currTrack.play})
)
```

And here, we have a SuperCollider program I wrote myself that produces the same sound (note the synth used comes from this youtube tutorial):

```
(
SynthDef.new(\sine, {
arg freq=440, atk=0.005, rel=0.3, amp=1, pan=0;
var sig, env;
sig = SinOsc.ar(freq);
env = EnvGen.kr(Env.new([0, 1, 0], [atk, rel], [1, -1]), doneAction:2);
sig = Pan2.ar(sig, pan, amp);
sig = sig * env;
Out.ar(0, sig);
}).add;
)

// a Simple ii-V-I chord progression
(
var twoChord, fiveChord, oneChord, allChords;
twoChord = [62, 65, 69, 72];
fiveChord = [55, 59, 62, 65];
oneChord = [60, 64, 67, 71];
allChords = [twoChord, fiveChord, oneChord];
c = Pbind(
\instrument, \sine,
\dur, 1,
\midinote, Pseq(allChords, 3).trace,
).play;

m = Pbind(
\instrument, \sine,
\dur, 0.25,
\midinote, Pseq(twoChord ++ fiveChord ++ oneChord, 3).trace,
).play;

)
```

4.1 Initial Observations

Notice that the DAWPL program consists of 16 lines of code, whereas the SuperCollider program I wrote consists of 28 lines of code. The compiled output from DAWPL has 27 lines of code. Many of the additional lines in the SuperCollider code come from the definition of the instrument i.e. the sine wave oscillator with various effects applied to it.

DAWPL (in its current state) provides no mechanism to define such an oscillator directly. One would have to do so in SuperCollider and include it as an instrument macro. In terms of expressiveness for defining low-level DSP components such as synths and oscillators, SuperCollider wins by a long shot – there is simply no way to do so in bare DAWPL. It is worth mentioning, however, that one of the main goals of DAWPL is abstracting away all of the gory details of the low-level DSP that happens under the hood. After all, this language is intended for those with more of a musical background. DAWPL emphasizes music theory over frequencies and waveforms.

Another thing to note is the fact that the definitions of the chords in the DAWPL syntax are based on chord names. However, the definitions of the chords in the SuperCollider syntax are based on MIDI values. Musicians do not often have the MIDI values of different notes memorized – MIDI is most commonly used as a data format that devices like electronic keyboards use to communicate with DAWs and other audio components. Furthermore, the SuperCollider code uses raw time durations to describe rhythm whereas the DAWPL code uses commonly understood note durations e.g. “W” for whole notes and “Q” for quarter notes. The macros in the DAWPL code also enhance readability and make it easier to express musical ideas.

4.2 Subjective Experience: Coding in DAWPL vs. Coding in SuperCollider

Before I discuss my experience coding in SuperCollider and DAWPL, I will quickly mention some of the abstractions that SuperCollider has and what roles these play in the given examples.

One abstraction we have already mentioned is the ability to deal with low level DSP e.g. defining new synths. This enables us to define new instruments and interact with sound at a very fundamental level. The Pbind abstraction allows users to sequence variations of parameters on these synths through a SuperCollider pattern construct. I would however argue that Rust contains constructs that are far more powerful and expressive than the pattern constructs present within SuperCollider. There is some support for scales. However, SuperCollider’s scale interface is not particularly modular and is still mainly numerical (as opposed to allowing users to use note name syntax).

SuperCollider also has some syntactic quirks that make it hard to get used to. For instance, one must declare all variables at the beginning of a scope using the var keyword before one uses them. Furthermore, there is a strange backslash syntax used for specifying key value pairs for varying parameters within synths (this is quite common for other constructs within SuperCollider).

Overall, it was rather annoying to have to manually look up MIDI values for the notes I was interested in. The pattern syntax is also rather opaque and hard to pick up. Furthermore, the SuperCollider interpreter does not always present the most helpful error messages. These difficulties mainly accounted for the additional difficulty I had with the SuperCollider syntax.

4.3 Advantages of SuperCollider

Although I found enough wrong with SuperCollider that I felt compelled to create my own language to layer on top of it, there are some components of it that are quite beneficial.

The SuperCollider IDE is probably one of the biggest pluses the language has. There is a documentation search window embedded within it, as well as auto-completion for the text editor. It is also a fairly well established language and has a reasonably large community (being the niche language it is). There is also fantastic Youtube

tutorial series on SuperCollider that serves as a great introduction to the language (I link to this in the References section).

As mentioned before, SuperCollider enables users to deal with raw DSP audio constructs. This makes it possible to discover and create new exciting sounds. SuperCollider can also interface with other audio technologies and protocols such as OSC. OSC can be used to interface SuperColliders with audio GUIs and devices.

In general, SuperCollider can be thought of as a more general purpose language whereas DAWPL caters specifically to having a music theoretic perspective on production in addition to a DAW-style organizational scheme.

4.4 Overall Analysis

Although SuperCollider has more expressiveness in the realm of low-level audio characteristics, such as frequencies, amplitudes, MIDI numbers, etc. it is worth mentioning how cumbersome it was to write the initial SuperCollider code, as well as the strange syntax, and low readability. SuperCollider allows us to define raw audio signal waveforms, and synths, however, it lacks a simple intuitive system of abstractions to let us deal with melody and rhythm in a traditional way. I had to manually look up MIDI numbers to correspond to the tones of individual chords. We are forced to specify time durations instead of expressing rhythm in a simpler and more intuitive way that resembles actual music notation.

DAWPL does not provide the same level of access to low level audio features, however, there are clearer connections to it and actual music notation. Users do not have to deal with raw MIDI numbers (although they technically can if they want to) – they can instead use actual note names and chord names. One can see how the chord names used in the DAWPL code matches the chords that annotate standard music notation. Users can specify rhythm in terms of quarter, eighth, half, whole, and sixteenth notes, instead of specifying a time in seconds. Users are also forced to organize their code in a simple and modular way, reminiscent of how a DAW setup looks.

It is worth mentioning why both of these systems lie where they do on the tradeoff space and what that means for how they are supposed to be used. SuperCollider is designed with DSP in central focus: this is for people working on experimental sound projects, developing DAW plugins, dealing with potentially complex hardware setups, doing DSP music research, creating new synthetic electronic instruments etc.

DAWPL is designed with music theory and organization in focus: its best use cases are for people wishing to compose more traditional/conventional music (as opposed to experimental) and who wish to specify the production of their music in a programmatic manner. Note that both languages can be used for algorithmic composition, however, DAWPL facilitates algorithmic composition with notes and beats instead of frequencies and durations. It would therefore be difficult to use DAWPL to create experimental synths.

The DAW organizational structure of DAWPL (see 5-layer-abstraction-hierarchy) enforces a beneficial and modular decomposition of a piece – this makes a piece more readable and understandable so that other composers can "remix" it and so that the user themselves can add more complexity in a comfortable way (think: the benefits of decomposing code in general).

In summary, DAWPL has higher expressiveness for music theory constructs and lower expressiveness for DSP audio constructs (for which SuperCollider has higher expressiveness). The DAWPL organization scheme forces the user to express their compositions in a more readable and modular manner.

4.5 Reflection on DAWPL and its Future Development

I would say that I made DAWPL more or less into what I wanted it to be in terms of expressiveness (as I have explained in the previous section). I am interested in also developing out certain features to enable it to work better with SuperCollider as opposed to being an exclusive replacement for it. With automatic code generation of

modular components, one can already use DAWPL to enrich their SuperCollider code. It was easier to express ideas in DAWPL because the abstractions DAWPL offered were more familiar and traditional. Furthermore, DAWPL inherits Rust-syntax, which is often heralded for being clear and expressive (this was another contributing factor to why it was easier to write code in DAWPL). The macro system Rust offers makes it simple to add more to DAWPL and make it even simpler and intuitive to understand.

5 REFERENCES

I used this Youtube channel to gain a better understanding of SuperCollider. I made extensive use of the Rust documentation and forums to develop DAWPL. In particular, I looked up things related to macros, string formatting, and aspects of the type system Rust supports (e.g. traits). I also looked through the SuperCollider documentation a good amount.