Building a DSL for Music Composition

Jacob Sonnenberg

May 17, 2016

Contents

1 Introduction

The purpose of this document is to outline what I learned in creating a small domain specific language for music composition. There are three main parts in this project: the engine the DSL is built on, the raw string data that's parsed into music data, and finally the macros which allow some convenience in working in the DSL in Elixir.

The later two represent two approaches to building (or extending) languages. Raw strings can be very expressive for many purposes when parsed since there's no limit on how complexly language is constructed, and provide only an unstructured way for extending a language. On the other hand macros are a more structured form of language extension and DSL construction.

The order topics are covered reflects the order I built them in. In Lisp tradition I built it from the ground up, creating the basis for the application and abstracting on top of that.

2 The engine

Underlying the higher level constructs for constructing pieces of music is the struct-based data representation. There are five data-types:

1. Piece
2. Rest
3. Chord
4. Staff
5. Note

The first three are simpler than the last two. `Piece`'s and `Chord`'s are just structural indicators on `Staff`'s and `Note`'s respectively. `Rest`'s are just placeholders for an empty location in a staff where a note typically exists.

2.1 Note Module

This module defines a note struct and methods to generate sound data based off of the fields of `Note` structs. As it stands it relies on the external program *SoX* to generate the note sounds. This back-end for generating the sounds can be changed with a small amount of effort because, fortunately, the procedures for performing that sound generation is relegated to only a couple modules.

The two most important functions in Note are `play` and `note_args`. `play` invokes the system call and `note_args` constructs the arguments that will be passed to the "play" program provided by SoX.

```
def play(note, opts \\ %{}) do
  System.cmd("play", note_args(note, opts))
end

defp note_args(note, opts) do
  ["-qn", "synth",
   to_string(4 / note.type),
```

```
    Map.get(opts, :type, "pluck"),
    process_note(note),
    "vol", Map.get(opts, :vol, "1")
  ]
end
```

2.2 Staff Module

The `Staff` module defines the majority of the behavior for groups of notes. The main function of this is to handle beats-per-minute associated with a collection of measures, or notes.

`Staff` is the main mover in the underlying engine as it groks most of the various data-types and understands what to do with them. The of structure of `Staff`'s are transformed to a list of time-annotated notes, the associated time data specifies how many milliseconds into execution the note is played.

The backbone of `Staff`, `tag_note_delays` was put together before I had a more complete understanding of what I was doing but nevertheless serves its function well enough. It serves to flatten the structure of the piece, everything under staff level becomes a 2-tuple with the first place denoting the time at which the note is played, and the second is the actual note itself.

2.3 Piece, Rest, and Chord Modules

These modules are very limited in scope and have a lot of room for improvement in the future. `Rest`'s and `Chord`'s are elements of `Staff`'s, and `Staff`'s are elements in `Piece`'s. They serve mostly to implement structural alterations to music pieces and are parsed out of existence early on.

3 The language

The language is made up of two major parts, macros and note-strings. Macros provide some convenience facility for writing in Elixir while the note-strings are bit-strings which are parsed to generate the data.

3.1 Note-Strings in the language

In Harpsi the actual playable data is decided primarily by note-strings. The note-strings are parsed into chords and notes and act as below staff level. Transformation from raw string to playable data structure is performed by the Parser. The backbone of the `Parser` is `process_word` which serves as a dispatch to the various rules defining how the notes are constructed, each case of the cond serve as the individual rules for note construction. The rules of course can be any elixir expression. Regular expressions serve as the rules for how notes are constructed.

The 'bottom' of the language is `process_note`, every word must decompose into single letters naming the notes A through G, or a rest. The language for note-strings is essentially a series of manipulations on `Note`, `Chord`, and `Rest` structs, and so exists in the small domain that currently allows. In the future this domain could be expanded by changing a finite number of (albeit yet undocumented) functions, and adding more fields to the two structs, and enhancing the engine. But it serves as a simple basis for my purposes that permits some limited amount of expression.

As for rules, a good example is `process_doctave`:

```
def process_doctave(word, opts) do
  cap = Regex.named_captures(~r/^(?<up_or_down>[<>])(?<word>.*)/, word)
  process_word(cap["word"], Map.put(opts, :octave,
opts.octave + (case cap["up_or_down"] do
 "<" -> -1
 ">" -> 1
       end)))
end
```

This one is changes the the octave by a shift of

Each rule should work on a certain foundation that could potentially be better enforced as macro. Implementing it is an exercise for the reader, but I'll note patterns that exist. The `word` arg is the individual note-string note in a staff. This is dissected by the rules via regex into the three basic structs, `Note` and `Chord`, and `Rest`. On the other hand `opts` is concerned with the rest of the data associated with the notes.

Most rules are built like

```
def process_<case>(word, opts) do
  cap = Regex.named_captures(<regex recognizing case>, word)
  process_word(<transformation on cap["word"]>,
    <transformations of the structs>)
end
```

with a little imagination, one could construct a macro for defining these rules in a structured way by templating the above snippet and adding an entry to a list of callback functions invoked by `process_word`. More generally, the rules should have one of three things in the tail place of the function. That is,

1. A recursive call back to `process_word` with modified parameters
2. A `Note` struct
3. A `Rest` struct

Chord not included in the list because they're only a structure requiring constituent notes, of course.

3.2 Macros in the language

All the macros exist in in Lang. The foremost actor is the `piece` macro which reflects the `Piece` struct. When writing in Harpsi the `piece` macro provides a manipulable environment for writing `Staff`'s of music and building the whole structure of the playable `Piece`.

As stated there are two variable dimensions, the 'buffer' of music and the 'environment' the notes are created in. In `Lang` you'll find I use two agents to model this behavior in a unhygienic way, requiring a set of functions to handle an ad-hoc, unspecified behaviors for constructing the buffer and maintaining the environment. Agents are "simple abstractions around state", some shared state is kept in it so the state is accessible at different points in macro expansion

The buffer agent simply accumulates the musical structure and returns a list of `Staff`'s, and the environment agent tracks the state of the environment as a stack. Management of the environment is especially straight forward. The environment is initialized with the `start_env` function.

```
def start_env(), do: Agent.start_link(fn ->
  [%{bpm: 120, octave: 4, type: 4}] end)
```

This starts an agent with an initial element in the stack which serves as the 'default' environment. The environment is maintained with a set of three functions:

```
def push_env(env, attr_map) do
  new = Map.merge(get_env(env), attr_map)
  Agent.update(env, &[new | &1])
end

def get_env(env), do: Agent.get(env, &(&1)) |> hd
```

```
def pop_env(env), do: Agent.update(env, &tl/1)
```

The functional requirements are minimal and the behavior is pretty intuitive. It's a simple stack that implements push, pop, and peek. The `Agent` must be cleaned up after use.

```
def stop_env(env), do: Agent.stop(env)
```

If built with the proper initialization and cleanup of the environment and changes to it, macros built with these two simple tools allow for some flexibility in potential language constructs. A obvious pattern is the closure, a language construct that clearly marks the beginning and end of some modification to the environment. In the language of Harpsi this is of course is a vocabulary limited by the underlying data-structures and what can be done with them.

The macro `bpm` in Lang is a closure with a predefined item in the environment the construct will manipulate.

```
defmacro bpm(n, do: inner) do
  quote do
    push_env var!(env, Lang),
      %{bpm: unquote(n)}
    unquote(inner)
    pop_env var!(env, Lang)
  end
end
```

Of course, `bpm` manipulates the beats per minute of a `Staff`. Such specific operations should generally be avoided because maintaining a language can become cumbersome if the domain grows too large. Instead favor generic interfaces to achieve the effect of a battalion of special cases.

```
defmacro w_opt(kwl, do: inner) do
  quote do
    push_env var!(env, Lang),
      Enum.into(unquote(kwl), %{})
    unquote(inner)
    pop_env var!(env, Lang)
  end
end
```

The structure of the piece of music is encoded into macros as well. Witht the `piece` macro and `staff` macro for composing staffs and notes respectively. `piece` only initializes the two helper agents and puts the result of processing the macro's body into a `Piece` struct. `staff` similarly reflects the underlying `Staff` structs, generating a struct based on the environment it's constructed in, and the string passed to the macro.

```
defmacro staff(inner) do
  quote do
    put_buffer var!(buffer, Lang),
      %Staff{bpm: get_env(var!(env, Lang))[:bpm],
     octave: get_env(var!(env, Lang))[:octave],
     measures: process_notestring(unquote(inner), get_env(var!(env, Lang)))}
  end
end
```

`Staff`'s are collected into the shared `buffer` variable, and placed in a `Piece`.

4 Lessons learned