Case study: Panola

What is Panola?

Panola is an acronym. It stands for **PA**ttern **NO**tation **LA**nguage. It allows for a concise specification of a supercollider pattern.

What problem does Panola solve?

When writing supercollider patterns using `Pbind`, information about notes, durations, dynamics and other properties are completely separated. Consider this example:

```
(
Pbind(
        \instrument, \default,
        \degree, Pseq([1,3,5,3,4,2,3,1,1,3,5,3,4,2,1], inf),
        \dur, Pseq([1,1,1,1,1,1,1,1,1,1,1,1,1,1,2]),
).play
)
```

Now imagine you decide you want to change the 9th note from a quarter note into two eighth notes. Have fun counting to the 9th note in each of the keys…

With Panola all this information is kept together. One specifies notes/chords with durations and additional properties all grouped together. The above example in Panola would read something like:

```
(
Panola("d4_4@amp[0.1] f a f g e f d d f a f g e d_2").asPbind.play;
)
```

What we see is a kind of text-based score. The first note is a d in the fourth octave ("d4"), which is a quarter note (_4) and which has an amplitude of 0.1. Note how properties are inherited by subsequent notes. If the octave or duration or amplitude don't change, we do not need to re-mention them.

Now if I want to change the 9th note into two eighth notes, I need to change only the Panola string itself.

```
(
Panola("d4_4@amp[0.1] f a f g e f d d_8 d_8 f_4 a f g e d_2").asPbind.play;
)
```

Or if we want a crescendo from the start of the phrase until the end, we can simply use animated properties:

```
Panola("d4_4@amp{0.1} f a f g e f d d_8 d_8 f_4 a f g e
 ↪  d_2@amp{0.4}").asPbind.play;
```

In the above specification, amplitude will smoothly increase from 0.1 to 0.4. This document is not intended as Panola tutorial - see the Panola documentation for many more examples and explanations.

Parsing Panola

When developing a parser for a non-trivial language, I prefer to work bottom-up. Each subparser can be fully tested before we move on to bigger and better things.

Parsing a note name

Note names in Panola are written using a single character (a, b, c, d, e, f or g). I also allow uppercase letters to be used, although I've never seen someone use those in a practical score.

The preferred way to parse a note name therefore is using a regular expression which matches exactly one character [a-gA-G].

```
(
var noteParser = RegexParser("[a-gA-G]");
)
```

Calling run

Whenever you want to try a parser on a piece of text, you can use its `run` method. A call to `run` with the text you want to parse as an argument returns a so-called *parser state*. The parser state contains information about how successful the parsing was (i.e. did an error occur?), as well as information that informs subsequent parsers about where in the given string they can try to continue.

Try to run this parser on a valid note name:

```
(
var ps; // ps is an abbreviation of parserState
var noteparser = RegexParser("[a-gA-G]");
ps = noteparser.run("a#");
ps.isError.debug("result 1 is error?");
ps.result.debug("result 1");
ps.prettyprint;
)
```

The above code gives the following result:

```
result 1 is error?: false
result 1: a
*************
*ParserState*
*************
target:  a#
index:  1
result:  a
*************
```

With a wrong note name, however, an error is signaled and the parser index remains at 0:

```
(
var ps;
var noteparser = RegexParser("[a-gA-G]");
ps = noteparser.run("h#");
ps.isError.debug("result 2 is error?");
ps.result.debug("result 2");
ps.prettyprint;
)
```

If you run the above code, you get

```
result 2 is error?: true
result 2: nil
regexParser: Error! at position 0 expected to match '[a-gA-G]' but found
↪  'h#[...]'
```

Note that prettyprint will just show an error message in case of a parsing error. If you want to see the complete contents of the parser state, use print instead.

```
(
var ps;
var noteparser = RegexParser("[a-gA-G]");
ps = noteparser.run("h#");
ps.print;
)
```

gives

```
*************
*ParserState*
*************
target:  h#
index:  0
result:  nil
isError:  nil
errorMsg:  regexParser: Error! at position 0 expected to match '[a-gA-G]' but
 ↪  found 'h#[...]'
```

Mapping the parsing result

Our specification for note names allows both lowercase and uppercase characters, without difference in meaning. To make our lives easier, we can transform the parser result to contain some contextual information, and to get rid of the uppercase note names by making them lower case.

In the first version we had:

```
var ps;
var noteparser = RegexParser("[a-gA-G]");
ps = noteparser.run("A#");
ps.result.debug("result");
```

which displays:

```
result: A
```

With a result mapper, we can transform the parse result and add some structure or record some contextual information:

```
(
var noteparser = RegexParser("[aAbBcCdDeEfFgG]").map({|result|
    (\type: \notename, \value: result.toLower) });
var ps = noteparser.run("A#");
ps.result.debug("result");
)
```

The parse result now looks like:

```
result: ( 'type': notename, 'value': a )
```

Much better already: the uppercase *A* has been transformed into a lowercase a, and how the parser now returns an event with extra information (a 'type' and a 'value'). Unless we explicitly add some extra information in the parse result about the fact that *A* was specified in uppercase, this information, because of the mapping function that calls .lowercase, is now irrevocably lost. But as it's not needed for Panola to function properly this is not a problem.

Moving forward, the Panola parser will make heavy use of mapping to document or simplify details of the internal structure of the input strings.

What about rests?

No music language can be complete without a specification of rests. In Panola a rest is notated with an *r* symbol. So why didn't we just add an "rR" in our note name parser? The reason has to do with semantics: to a note name one can assign many properties that make no sense in combination with a rest, like a pitch modifier (sharp, flat, …) or an octave indication.

So, for rests a separate RestParser is specified:

```
(
var restParser = RegexParser("[rR]").map({|result| (\type: \rest) });
)
```

Mapping again is used to add some extra information and structure into the parse result. We can try out if the rest parser works:

```
(
var restParser = RegexParser("[rR]").map({|result| (\type: \rest) });
var ps = restParser.run("r");
ps.result.debug("result");
)
```

which gives the desired result

```
result: ( 'type': rest )
```

Pitch alteration marks

A note name alone does not suffice to specify a pitch. Pitches can optionally be altered by decorating the note name with a sharp (#), a flat (-), a double sharp (x) or a double flat (–) Note: later also an octave specification will be needed, but in this section we concentrate on the alteration marks.

Choice

Small parsers like the ones we made so far can be combined together into bigger parsers. In this case we will make a small sharp parser, a flat parser a double sharp parser and double flat parser. Then a `Choice` parser will be used which can try different alternatives until one succeeds. The `Choice` parser will be wrapped in an `Optional` parser, because not every note name is decorated with a alteration mark.

Pitfall...

Here we encounter a subtle pitfall: if a pitch is decorated with a double flat ("–") then a parser for a single flat will succeed since it matches the first character of ("–"). Once the parser for a single flat has consumed the first "-" the double-flat parser will never succeed anymore since there's only one "-" left. The way we solve it here is to take into account the ordering in which different parsers will be tried. The `Choice` parser always tries out parsers from left to right, and stops trying as soon as one succeeds. So by making sure the double-flat parser runs before the flat parser, we can be sure not to mistake any double flats for a single flat followed by a minus sign.

Note that in some more complex cases it may be difficult to ensure the "correct" parser runs first. For such occasions, there's an alternative parser called `LongestChoice` which will try to match all the parsers and keep the one that consumes the most tokens. This is obviously less efficient that just using `Choice`.

StrParser

Note that instead of using a `RegexParser` to match an alteration mark, we here use a simpler `StrParser`. `StrParser` can only do a literal matching of a string (case sensitive only at the time of writing this guide). It is prefered over `RegexParser` if applicable since it's less error prone than writing a regex, and a bit more efficient.

```
(
var noteModifier = Optional(Choice([
    StrParser("--").map({|result| (\type : \notemodifier, \value:
↪   \doubleflat) }),
    StrParser("-").map({|result| (\type: \notemodifier, \value: \flat) }),
    StrParser("#").map({|result| (\type: \notemodifier, \value: \sharp) }),
    StrParser("x").map({|result| (\type: \notemodifier, \value: \doublesharp)
↪   })
])).map({|result| result ? (\type: \notemodifier, \value: \natural) });
)
```

The above code parses the optional alteration mark, and if the `Optional` turns up with an empty result (i.e. there's no alteration mark present) the map function makes sure to decorate the parse result with a \natural note modifier. This ensures that every note will have a \notemodifier property which I expect will make using the information in the parse result easier later on.

Parsing the octave

In addition to an alteration mark, a note name needs an octave in order to be converted to a pitch. For the sake of efficient score creation, we want to make the octave specification optional, i.e. as long as you don't explicitly specify an octave, the previous one should be reused. For this reason, we will use the mapping functionality to record if an octave is explicitly present or not.

Code that uses this parse information later on to generate a supercollider pattern will have to take this information into account to use the correct octave. One possible approach would be to, after parsing is done, have a visitor run over the parse tree and resolve all octaves marked as `\value: \previous` to their correct numerical value.

```
(
var octaveParser = Optional(
    RegexParser("\\d\\d?").map({|result| (\type: \octave, \value:
↪   result.asInteger )})
).map({|result| result ? (\type: \octave, \value: \previous) });
)
```

Parsing the duration

Duration in Panola is indicated by using an underscore followed by a number. The number indicates a fraction, e.g. *a_4* is a quarter note; *a_8* is an eighth note (i.e. half as long as the quarter note). Durations can be marked up with some modifiers: you can one or more dots (as in regular notation). Every additional dot extends the duration with half of the previous duration, e.g. `a_4.` lasts a quarter note + an eighth note, whereas `a_4..` lasts a quarter note + an eighth note + a sixteenth note. In addition, one may want to make tuplets, so the duration can also be marked up with a positive multiplier and a positive divider. If you wanted to specify a triplet of eight notes you could write `a_8*2/3 a a`. Any ratio is allowed, so you can write `a_8*23/78` if you desire.

As with the octave, the duration is optional, and the previous one should be reused while no new one is specified. In addition there are some special rules: if you only specify a divider, the multiplicator is automatically reset to 1. If you only specify a multiplier, the divider is automatically reset to 1.

Many parser

Given that we can have zero or more dots as part of the duration, this is a good time to introduce a new kind of parser: the `Many` parser. The `Many` parser takes a parser *P* and returns a new parser that matches zero or more instances of *P*. The parse result that comes out of a `Many` parser is a list of parse results that come out of *P*. A map function that transforms the result of a `Many` parser therefore can address the result as an array.

Here the dots are completely optional, but in some cases you want to ensure that there's at least one match. In such cases you can use the `ManyOne` variant instead.

SequenceOf parser

To parse a multiplier, we need to parse a sequence of two different parsers: one parser matches the * sign, and another parser matches the digits that come after it. Same for the divider: one parser matches the / sign, and another parser matches the digits that come after it. For matching a sequence of parsers we use a `SequenceOf` parser. The `SequenceOf` parser takes a list of parsers *P, Q,…, Z* and tries to match them one after the other. As soon as one of these parsers fails, the complete `SequenceOf` fails. The parse result of a `SequenceOf` parser is a list of results, one entry for each parser *P, Q, …, Z* in the `SequenceOf` parser.

ParserFactory

To make life a bit easier, scparco comes with a `ParserFactory` convenience class that offers some ready-made parsers you may often need. There's nothing special about the parsers in ParserFactory, they are made like any parser you'd specify yourself. The duration parser uses the `ParserFactory.makeFloatParser` and `ParserFactory.makeIntegerParser`. These parsers have a built-in map function that converts the parse result from string to a numerical value already. Nothing prevents you from adding a second map function to further transform the result, and this will be done here.

```
(
var durationParser = Optional(SequenceOf([
    StrParser("_"),
    ParserFactory.makeFloatParser.map({|result| (\type: \duration, \value:
↪   result)}),
    Many(StrParser(".")).map({|result| (\type: \durdots, \value:
↪   result.size)}),
    Optional(SequenceOf([StrParser("*"),
↪   ParserFactory.makeIntegerParser]).map({|result| (\type: \durmultiplier,
↪   \value: result[1])})),
    Optional(SequenceOf([StrParser("/"),
↪   ParserFactory.makeIntegerParser]).map({|result| (\type: \durdivider,
↪   \value: result[1])}))
])).map({
    |result|
    if (result.isNil) {
        (\dur : \previous, \durmultiplier: \previous, \durdivider: \previous,
↪   \durdots: \previous);
    } {
        var dur = ( \dur : result[1][\value], \durdots : result[2][\value]);
        // treat divider and multiplier as one: specifying only one of the
↪   two affects the other one
        if (result[3].isNil && result[4].isNil) {
            dur[\durmultiplier] = \previous;
            dur[\durdivider] = \previous;
        } {
            if (result[3].isNil) {
```

```
                // only dividider specified ->reset multplier to 1
                dur[\durmultiplier] = 1;
                dur[\durdivider] = result[4][\value];
            } {
                if (result[4].isNil) {
                    // only multiplier specified -> reset divider to 1
                    dur[\durmultiplier] = result[3][\value];
                    dur[\durdivider] = 1;
                } {
                    // everything specified
                    dur[\durmultiplier] = result[3][\value];
                    dur[\durdivider] = result[4][\value];
                }
            };
        };
        dur;
    };
});
)
```

Parsing note properties

Panola allows adding properties to each note. These properties can be anything you fancy, although some properties have a predefined meaning. When Panola generates a supercollider `Pbind`, all these user defined properties are present as key-value pairs in the Pbind. This mechanism e.g. makes it possible to send arguments to your synths from Panola. Other applications include using `Pbindf` to send the property value with OSC to some user interface code for real-time visualization while the pattern is being realized (played).

Here's an example of a quarter note (_4) in the third octave with 2 properties attached:

```
"a#3_4@amp[0.1]@mysynthknobvalue{34}"
```

Properties come in several flavors. Property values enclosed in `[ ]` brackets are *static* properties, i.e. their value remains the same until a new value for the property is specified. Properties enclosed in `{ }` on the other hand are *animated* properties. Their values linearly interpolate to the next specification of a value for this property. And properties enclosed in `^ ^` brackets are *one-shot* properties.

Static properties can be used to set up some fixed values, e.g. specify an amplitude value of 0.9 for all notes. Example:

```
"a@amp[0.9] b c d e f g"
```

Animated properties are used to specify gradually changing things (think: creating a *crescendo*). Example where amplitude gradually rises from 0.1 to 0.9:

```
"a@amp{0.1} b c d e f g@amp{0.9}"
```

One-shot properties are used to specify a value for a property that is used only once, after which the property falls back to its previous value. Think: defining an accent.

```
a@amp[0.1] b c d e@amp^0.5^ f g
```

In the above example, notes a, b, c and d have property amp=0.1, then note e has property amp=0.5, but notes f and g have property amp=0.1 again.

The property type can change during the score, for example

```
"a@amp{0.1} b c d e@amp[0.6] f g@amp[0.8] f e"
```

would gradually increase amplitude from a to e, then remain at 0.6 from e to f and jump to 0.8 for the final g, f and e notes.

To parse properties, we need something to parse the property name (like *amp*) and then something to parse the property argument and type. Map again is used to record the difference between static, animated and one-shot properties.

```
(
var propertynameParser = RegexParser("@[a-zA-z][a-zA-Z0-9]*").map({|result|
↪  (\type: \propertyname, \value: result.drop(1))});
var propertiesParser = Many(
    Choice([
        SequenceOf([
            propertynameParser,
            StrParser("{"),
            ParserFactory.makeFloatParser,
            StrParser("}")
        ]).map({|result| (\propertyname: result[0][\value], \type:
↪  \animatedproperty, \value: result[2])}),
        SequenceOf([
            propertynameParser,
            StrParser("["),
            ParserFactory.makeFloatParser,
            StrParser("]")
        ]).map({|result| (\propertyname: result[0][\value], \type:
↪  \staticproperty, \value: result[2])}),
        SequenceOf([
            propertynameParser,
            StrParser("^"),
            ParserFactory.makeFloatParser,
            StrParser("^")
        ]).map({|result| (\propertyname: result[0][\value], \type:
↪  \oneshotproperty, \value: result[2])}),
]));
)
```

Combining everything

Everything we've done so far can now be combined into a parser for a note. I like to work step by step, writing some test code in each step (not shown in this tutorial).

```
(
var noteAndModAndOct = Choice([
    SequenceOf([noteParser, noteModifier, octaveParser]).map({
        |result|
        (\type: \note,
            \notename: result[0][\value],
            \notemodifier: result[1][\value],
            \octave: result[2][\value])
    }),
    restParser
]);

var noteAndModAndOctAndDur = SequenceOf([
    noteAndModAndOct,
    durationParser
]).map({|result| (\pitch : result[0], \duration: result[1] ) });
```

```
var noteAndModAndOctAndDurAndProp = SequenceOf([
    noteAndModAndOctAndDur,
    propertiesParser]).map({|result|
        (\type: \singlenote,
         \info : ( \note : result[0], \props : result[1] ) );
    });
)
```

From notes to chords

Chords in Panola are defined as a list of notes between angular `< >` brackets. In case of chords, Panola will typically only use the properties attached to the first note and reuse them for the other notes in the chord. This is mainly because realizing supercollider patterns with separate properties for every note in a chord is more difficult to implement. If you need multiple notes sounding together with different properties for each, like in polyphonic music e.g., you can make multiple Panola strings (one for each voice) and play them in parallel.

makeBetween parser

Since parsing *something* between delimiters is a common use case, scparco's `ParserFactory` provides a `makeBetween` parser. This makeBetween parser is a bit special, in that it takes a parser that matches the delimiters, and returns a function (not a parser!). This function needs to be called with another Parser as argument. This parser must match whatever lives between the delimiters.

In other words, the general way of using `makeBetween` is:

```
(
var f = ParserFactory.makeBetween(
    StrParser("("),
    StrParser(")"));
var parser = f.value(NoteParser); // parser that parses a note between ( and
↪ )
)
```

However the delimiter parsers and content parsers can be arbitrarily complex, which make this a powerful building block. Note: Parsers made by makeBetween throw away the delimiters in the parse result.

handling whitespace

Something that hasn't been discussed so far is how to handle white space. To tolerate whitespace, you need to make parsers that match whitespace: you have full control over where whitespace is allowed. If you forget to properly handle whitespace parsing can fail unexpectedly.

Thinking back about the previous section with the `makeBetween` parser: we might be tempted to extend the parser as follows:

```
var f = ParserFactory.makeBetween(
    SequenceOf([StrParser("<"), ParserFactory.makeWs]),
    SequenceOf([ParserFActory.makeWs, StrParser(">")]));
```

However, typically the parser that specifies the content that lives between the delimiters will also have some provisions to parse whitespace, so specifying two of them might be overkill.

As a rule of thumb, I tend to specify whitespace only at the end of a parser, never at the beginning (unless we're talking about the very start of the target sequence). If you don't

care about the whitespace itself, it can be removed from the parse result using a map function. Applying this rule of thumb to the parsing of notes between brackets gives:

```
(
var betweenChordBrackets = ParserFactory.makeBetween(
    SequenceOf([StrParser("<"), ParserFactory.makeWs]), // white space at the
 ↪  end of SequenceOf
    StrParser(">")); // no whitespace before closing bracket...

var chordParser = betweenChordBrackets.(
    ManyOne(
        SequenceOf([
            noteAndModAndOctAndDurAndProp,
            ParserFactory.makeWs // ...because whitespace after note is
 ↪  handled here already
        ]).map({|result| result[0] }); // remove whitespace from result
)).map({|result| (\type: \chord, \notes : result) });
)
```

Pitfall: infinite loops

One has to be careful with optional whitespace parsers not to introduce infinite loops. Here's an example that will send supercollider into an infinite loop:

Side-note: `makeSepBy` is a parser that has not been discussed yet. Similar to the idea of `makeBetween`, it will create a parser that parses a list of things separated by another thing (like a list of numbers separated by commas). When you call `makeSepBy` it expects a parser as argument. This parser matches the separators (the commas) and it returns a function. This function will return a parser when you call it with a Parser that matches whatever is between the separators.

```
(
ParserFactory.makeSepBy(ParserFactory.makeWs).(
    ParserFactory.makeSepBy(StrParser(",")).(ParserFactory.makeFloatParser)
).run("3,3,3 4,4,4").result // hoping to see [[3.0,3.0,3.0],[4.0,4.0,4.0]] as
 ↪  result
)
```

Why does the above specification hang up supercollider? `makeSepBy` uses *optional* whitespace (`makeWs`) instead of *mandatory* whitespace (`makeWsOne`) as a separator. Since the separator itself is completely optional (and so an empty string is a valid separator), it will match infinitely many "empty" separators without advancing the parser state index.

A way to write the above without causing an infinite loop would have been:

```
ParserFactory.makeSepBy(ParserFactory.makeWsOne).(
    ParserFactory.makeSepBy(StrParser(",")).(ParserFactory.makeFloatParser)
).run("3,3,3 4,4,4").result
```

Parsing a mixture of notes and chords

With everything we've developed so far, it is now easy to parse multiple notes and chords:

```
(
var notelistParser = ManyOne(Choice([
    SequenceOf([chordParser, ParserFactory.makeWs]).map({|result|
 ↪  result[0]}), // eat whitespace
    SequenceOf([noteAndModAndOctAndDurAndProp,
 ↪  ParserFactory.makeWs]).map({|result| result[0] }) // eat whitespace
]));
```

```
)
```

Note: the use of `ManyOne`, which enforces that at least one note or chord is present.

Handling repeats

A feature of Panola not discussed so far is its syntax to handle repeats. A fragment can be repeated as follows: `( a- b- c d- )*3`.Repeats can be arbitrarily nested: `( a3_4 (b c#_8*2/3)*2 d)*3`

This introduces a new difficulty: how do you handle nested parsers?

Considering that this is a note: `a_4` and this is a repeated note list: `(b d)*3`, Take a look at the following valid Panola string:

```
"a_4 b (c (b d)*3 )*2 a"
```

It starts with two notes `a_4 b`, then there's a repeated note list `(c (b d)*3 )*2`, and ends with another note `a`. The repeated note list however itself consists again of a list of notes and/or repeated note lists, more specifically a single note `b` and the repeated note list `(b d)*3`.

In what follows, call such panola string a mixed note list. Mixed, because it mixes *normal* notes/chords with *repeated* notes and chords.

Thunk and forwardRef

This problem asks for a recursive solution and that's what we'll use. But there's a problem: we want to parse a mixed list of notes and repeated note list. The mixed list therefore depends on the notion of a repeated note list. But the repeated notelist can again contain a mixed list. We have a chicken and egg problem! Both definitions refer to each other (mutual recursion) and supercollider cannot handle that transparently.

In such case we need to resort to a trick, which is to defer evaluation of parsers until everything is initialized and the parsers can refer to each other without problem. This deferring can be accomplished using the ParserFactory's `forwardRef` method in combination with supercollider's `Thunk` object. `Thunk` creates an unevaluated function - i.e. you can put stuff inside that doesn't exist yet and allows to evaluate it at a later time. `forwardRef` then disguises that unevaluated function as a regular parser: a parser that evaluates its `Thunk` when the actual parsing has started. At that moment, all Parsers have been initialized already, and parser can refer to each other without problem.

This gives us the final elements required to complete our mixed note list parser:

```
(
var betweenRepeatBrackets = ParserFactory.makeBetween(
    SequenceOf([StrParser("("), ParserFactory.makeWs]),
    StrParser(")");
);

var mixedNotelist = ParserFactory.forwardRef(Thunk({
    ManyOne(Choice([repeatedNotelist, notelistParser])).map({|result|
↪   result.flatten(1); });
}));

var repeatedNotelist = SequenceOf([
    betweenRepeatBrackets.(mixedNotelist),
    ParserFactory.makeWs,
    StrParser("*"),
    ParserFactory.makeWs,
    ParserFactory.makeIntegerParser,
```

```
    ParserFactory.makeWs
]).map({
    // unroll the loop already - not sure if this is a good idea (memory
 ↪  consumption!)
    // but it's easier to evaluate later on
    |result|
    var parseRes = [];
    var repeat = result[4];
    repeat.do({
        parseRes = parseRes.addAll(result[0]);
    });
    parseRes.flatten(1);
});
)
```

Here the map function is used to already unroll the repeated sections which is not particularly efficient in terms of memory usage but which make postprocessing the parse result easier.

Conclusion

Although not trivial, it is still much easier to read and understand (as well as much more concise and robust!) than the official implementation as it exists in the quark v0.1.1.

> **Note:**
>
> Panola quark v0.1.1 does not support the *one-shot* properties proposed in this document.
> These, along with some other features, may be added in a future version of Panola.

The work is not finished here. After parsing a Panola string still code is needed to convert the parse result to a supercollider pattern. Such code falls outside the scope of this document, but may become part of a future (re)implementation of Panola based on the scparco parser generator library.

Appendix A: Panola source code

The source code for the Panola quark as it exists today (using an inferior hand-crafted parser) can be found on https://github.com/shimpe/panola.

Appendix B: full code

Here's the full code required to parse a Panola string as developed in this document:

```
(
var noteParser = RegexParser("[aAbBcCdDeEfFgG]").map({|result| (\type:
 ↪  \notename, \value: result.toLower) });
var restParser = RegexParser("[rR]").map({|result| (\type: \rest) });
var noteModifier = Optional(Choice([
    StrParser("--").map({|result| (\type : \notemodifier, \value:
 ↪  \doubleflat) }),
    StrParser("-").map({|result| (\type: \notemodifier, \value: \flat) }),
    StrParser("#").map({|result| (\type: \notemodifier, \value: \sharp) }),
    StrParser("x").map({|result| (\type: \notemodifier, \value: \doublesharp)
 ↪  })
])).map({|result| result ? (\type: \notemodifier, \value: \natural) }); //
 ↪  map missing modifier to \natural sign
```

```
var octaveParser = Optional(
    RegexParser("\\d\\d?").map({|result| (\type: \octave, \value:
↪  result.asInteger )})
).map({|result| result ? (\type: \octave, \value: \previous) }); // map
↪  missing octave to \previous

var durationParser = Optional(SequenceOf([
    StrParser("_"),
    ParserFactory.makeFloatParser.map({|result| (\type: \duration, \value:
↪  result)}),
    Many(StrParser(".")).map({|result| (\type: \durdots, \value:
↪  result.size)}),
    Optional(SequenceOf([StrParser("*"),
↪  ParserFactory.makeIntegerParser]).map({|result| (\type: \durmultiplier,
↪  \value: result[1])})),
    Optional(SequenceOf([StrParser("/"),
↪  ParserFactory.makeIntegerParser]).map({|result| (\type: \durdivider,
↪  \value: result[1])}))
])).map({
    |result|
    if (result.isNil) {
        (\dur : \previous, \durmultiplier: \previous, \durdivider: \previous,
↪  \durdots: \previous);
    } {
        var dur = ( \dur : result[1][\value], \durdots : result[2][\value]);
        // treat divider and multiplier as one: specifying only one of the
↪  two affects the other one
        if (result[3].isNil && result[4].isNil) {
            dur[\durmultiplier] = \previous;
            dur[\durdivider] = \previous;
        } {
            if (result[3].isNil) {
                // only dividider specified ->reset multplier to 1
                dur[\durmultiplier] = 1;
                dur[\durdivider] = result[4][\value];
            } {
                if (result[4].isNil) {
                    // only multiplier specified -> reset divider to 1
                    dur[\durmultiplier] = result[3][\value];
                    dur[\durdivider] = 1;
                } {
                    // everything specified
                    dur[\durmultiplier] = result[3][\value];
                    dur[\durdivider] = result[4][\value];
                }
            };
        };
        dur;
    };
});

var propertynameParser = RegexParser("@[a-zA-z][a-zA-Z0-9]*").map({|result|
↪  (\type: \propertyname, \value: result.drop(1))});
var propertiesParser = Many(
    Choice([
        SequenceOf([
            propertynameParser,
```

```
            StrParser("{"),
            ParserFactory.makeFloatParser,
            StrParser("}")
        ]).map({|result| (\propertyname: result[0][\value], \type:
↪  \animatedproperty, \value: result[2])}),
        SequenceOf([
            propertynameParser,
            StrParser("["),
            ParserFactory.makeFloatParser,
            StrParser("]")
        ]).map({|result| (\propertyname: result[0][\value], \type:
↪  \staticproperty, \value: result[2])}),
        SequenceOf([
            propertynameParser,
            StrParser("^"),
            ParserFactory.makeFloatParser,
            StrParser("^")
        ]).map({|result| (\propertyname: result[0][\value], \type:
↪  \oneshotproperty, \value: result[2])}),
]));

var noteAndMod = Choice([
    SequenceOf([noteParser, noteModifier]),
    restParser
]);
var noteAndModAndOct = Choice([
    SequenceOf([noteParser, noteModifier, octaveParser]).map({
        |result|
        (\type: \note,
            \notename: result[0][\value],
            \notemodifier: result[1][\value],
            \octave: result[2][\value])
    }),
    restParser
]);

var noteAndModAndOctAndDur = SequenceOf([
    noteAndModAndOct,
    durationParser
]).map({|result| (\pitch : result[0], \duration: result[1] ) });

var noteAndModAndOctAndDurAndProp = SequenceOf([
    noteAndModAndOctAndDur,
    propertiesParser]).map({|result| (\type: \singlenote, \info : ( \note :
↪  result[0], \props : result[1] ) ); });

var betweenChordBrackets = ParserFactory.makeBetween(
    SequenceOf([StrParser("<"), ParserFactory.makeWs]),
    StrParser(">"));

var chordParser = betweenChordBrackets.(
    ManyOne(
        SequenceOf([
            noteAndModAndOctAndDurAndProp,
            ParserFactory.makeWs
        ]).map({|result| result[0] }); // remove whitespace from result
)).map({|result| (\type: \chord, \notes : result) });
```

```
var notelistParser = ManyOne(Choice([
    SequenceOf([chordParser, ParserFactory.makeWs]).map({|result|
↪   result[0]}), // eat whitespace
    SequenceOf([noteAndModAndOctAndDurAndProp,
↪   ParserFactory.makeWs]).map({|result| result[0] }) // eat whitespace
]));

var betweenRepeatBrackets = ParserFactory.makeBetween(
    SequenceOf([StrParser("("), ParserFactory.makeWs]),
    StrParser(")");
);

var mixedNotelist = ParserFactory.forwardRef(Thunk({
    ManyOne(Choice([repeatedNotelist, notelistParser])).map({|result|
↪   result.flatten(1); });
}));

var repeatedNotelist = SequenceOf([
    betweenRepeatBrackets.(mixedNotelist),
    ParserFactory.makeWs,
    StrParser("*"),
    ParserFactory.makeWs,
    ParserFactory.makeIntegerParser,
    ParserFactory.makeWs
]).map({
    // unroll the loop already - not sure if this is a good idea (memory
↪   consumption!)
    // but it's easier to evaluate later on
    |result|
    var parseRes = [];
    var repeat = result[4];
    repeat.do({
        parseRes = parseRes.addAll(result[0]);
    });
    parseRes.flatten(1);
});
```