A programming language for exploring and creating music

Kevin Chen (kxc2103), Brian Kim (bck2116), Jennifer Lam (jl3953),
Edward Li (el2724)

September 30, 2015

## 1 Goals and Philosophy

We would like to create a language optimized for exploring and creating music. The syntax facilitates writing snippets of music, or tracks, as parts of larger compositions. Standard library functions enable the programmer to transform these tracks. By using multiple compiler backends, the language can be compiled into audio (MP3 and MIDI) or sheet music.

Our language revolves around the concept of tracks. A track is a variable length sequence of chords, each with a list of pitches and a length, plus metadata such as key signature, tempo, time signature. Tracks are the building blocks of a composition, because they allow the user to specify a few snippets of music, and then concatenate, overlay, transform, or otherwise reuse them while creating a song.

## 2 Usage

### 2.1 A Basic Example: Twinkle Twinkle Little Star

To demonstrate the features and syntax of our language, we include an example program that expresses “Twinkle Twinkle Little Star”:

```
1 intro = q:[ 1 1 5 5 6 6 ] . h:5
2 chorus = Rhythm intro : [ 4 4 3 3 2 2 1 ]
3 bridge = Relative 2 chorus
4 twinkle = intro . chorus . bridge . bridge . intro . chorus
5 Render twinkle "twinkle.mp3"
```

In line 1, the melody and rhythm of the first line of the song are assigned to `intro`. Pitches are specified as numbers relative to the key signature — since we have not specified a key signature, the default of C major will be used. Therefore, pitch 1 corresponds to C, and 5 corresponds to G. (The default time signature is 4/4, and the default tempo is 120 bpm.)

The square brackets in lines 1 and 2 are syntactic sugar for the musical list constructor. The `q:[...]` applies the quarter note length to each note in the list, and returns a track. Similarly, `h:5` describes a half note of pitch 5.

Our language syntax borrows heavily from OCaml: function calls do not require parentheses or commas. For example, in line 2, the standard library function `Rhythm` takes the argument `intro` without including C-like parentheses. `Rhythm` takes a track or list of notes and returns a list of the notes' lengths. The colon `:` is an operator that zips the returned rhythm to the notes, and returns a track.

In line 3, we demonstrate the use of the standard library function `Relative`. It shifts `chorus` up to a second (1 pitch up), and assigns the returned value to `bridge`.

Line 4 creates the song by concatenating the pieces we have made so far. Concatenating is represented by the PHP-like period operator. You may be wondering how we concatenated a track and a note in line 1. `h:5` actually creates a track with a single note. Ta-da!

The last line creates an audio file of our song named `twinkle.mp3`.

## 2.2 Advanced Musical Features

This example incorporates advanced musical features, such as chords, different octaves, and multiple tracks:

```
 1 // Constants from stdlib
 2 /* Nested
 3   /* comments */ */
 4 STD$key_signature = STD$C_minor
 5 STD$time_signature = STD$two_two
 6 STD$tempo = 150
 7
 8 intro = q:@-1[1 1 5 5 6 6] . (q+q/2):5@1 . (q/2):0
 9 chorus = Rhythm intro : [ 4 4 3 3 2 2  1,3,5  0]
10 chorus_harmony = Relative 3 chorus
11 bridge = Relative 2 chorus
12 outtro = Arpeggio q:1
13
14 melody = intro . chorus . bridge . bridge . intro . chorus
15     . outtro
16
17 padding = Rhythm intro : 0
18 harmony = padding . chorus_harmony . Repeat 3 padding
19     . chorus_harmony
20
21 Render (Parallel melody harmony) "twinkle.mp3"
```

The first three lines demonstrate the syntax of comments. We support both types of C-style comments (`//` and `/*...*/`) and extend the multi-line comment to allow nesting.

In lines 4–7, we demonstrate how to override the default key signature, time signature, and tempo. To change the key signature to C-minor, the programmer re-assigns the value of global variable `STD$key_signature`, declared in the standard library, to `STD$C_minor` (also declared in standard library). Likewise, `STD$time_signature`, `STD$two_two`, and `STD$tempo` are also standard library variables.

The last two items of line 8 ( `(q+q/2):5@1` and `(q/2):0` ) demonstrate how the user can specify custom note lengths. Because our note constants (`q` for quarter notes, `h` for half, and `w` for whole) are implemented as floating point numbers in the standard library, it is possible to perform arithmetic operations on these constants. For example, `(q+q/2)` is a dotted quarter note, and `(q/2)` indicates an eighth note.

Line 9 uses `@-1[...]` to reduce all the pitches that follow by one octave from the default (default octave assumes middle A is tuned to 440 Hz). This syntax is similar to `q:[...]` from the previous example.

Line 9 also demonstrates syntax to specify a chord, by typing a comma-separated list of notes: `1,3,5`.

Line 17 creates `padding`, a track of silence that is the same length as `intro`. This works because pitch `0` creates a rest. In line 18, this allows us to use `padding` to offset our harmony track so that it plays at the right time. Finally, we play `melody` and `harmony` in parallel and write the result to a file.

### 2.3 Turing Complete, Yo

Our language can also be general-purpose: for example, the implementation of the `Relative` library function will use if-then and for.

```
 1 fun Relative offset track = (
 2   if -1 <= offset and offset <= 1 then track
 3   else (
 4     direction = do -1 unless offset > 0 inwhichcase 1
 5     scale_size = STD$key_signature$size
 6     offset_track = {}
 7     for chord in track do (
 8       offset_tones = {}
 9       for i in (Range 0 chord$tones$length) (
10         pitch = chord$tones[i]$pitch + direction * ((offset - 1) % scale_size)
11         octave = chord$tones[i]$octave + direction * ((offset - 1) / scale_size)
12         offset_tones = offset_tones . pitch@octave
13       )
14       offset_chord = chord$duration : offset_tones
15       offset_track = offset_track . offset_chord
16     )
17   )
18 )
```

In line 1, we declare the function using OCaml-like syntax. The function body has multiple lines, so we must wrap it in parentheses.

This function does not have an explicit return statement — everything in our language has a return value. Functions, if-then, for, and while all return the last line executed. Thus, the function returns `track` (line 2) when the condition is true and returns `offset_track . offset_note` (line 18) otherwise.

Line 4 takes advantage of this feature by setting a variable based on the result of a conditional. We also introduce an innovative feature of our language: do–unless–in which case, a fun and exciting version of if-then-else that really spices up your programs.

Lines 10–11 perform arithmetic on integers representing pitch and octave. Our language supports `+ - * /` for integers and reals, and `%` for integers.

### 2.3.1 A note on list construction

Those two lines also use square brackets for list indexing: `tones[i]`. When applied to the end of a list name, the square brackets indicate indexing instead of list construction.

`{...}` and `[...]` both construct lists. The difference is that square brackets `[...]` indicate that the contents should be interpreted first as musical notes — this resolves the ambiguity between `{1 2 3}`, which is a list of integers, and `[1 2 3]`, a list of musical notes.