Mission Sidequest 10.1: Beat Your Drums

Start date: 17 September 2017
Due: 24 September 2017, 23:59

Background Information

Instruments

In Source, we can represent *instruments* as functions that take two arguments, a MIDI note and a duration, and return a sound that plays the instrument at a pitch given by the note for the given duration. Hence, an instrument in Source has the following type:

$$\texttt{instrument} : (Number, Number) \rightarrow Sound$$

As introduced in Mission 10, we have several instruments pre-defined for you. These are `trombone`, `piano`, `bell`, `violin`, and `cello`. So if you want to play a piano at the MIDI note 60, with a duration of 2 seconds, you can write:

```
play(piano(60, 2));
```

This mission has **five** tasks.

Credit: This sidequest is inspired by cadet *Teo Siau Khian*.

Task 1:

You need to construct the `snare_drum` and the `bass_drum` in this task. Note the following:

- Generally for drum sounds, you do not need a Sustain or Release phase, and thus it suffices to talk about their AD envelope. Drums need an AD envelope with a rapid attack time of, say, 0.005 seconds and a decay time of, say, 0.495 seconds.
- Snare drums can be simulated by noise: You can get a snare drum by applying the AD envelope to a sourcesound consisting of randomized amplitudes.
- The sound of bass drums can be obtained by applying the AD envelope to an `aharmonic` cluster of sine waves of frequencies between 65 and 95Hz. Recall that in Mission 14, harmonics refer to the series of sound waves which have frequencies that are multiples of one another. Frequencies whose ratios are rational numbers with small denominators and numerators are also somewhat harmonic, even if the numerator is larger than 1. Aharmonic ratios have large numerators and denominators, and if the frequencies are all integer values, prime numbers are the most aharmonic. So for your bass drums, you can choose a cluster of sine waves with prime number frequencies between 65 and 95 Hz.

- The MIDI note makes no sense for drum instruments, so you may safely ignore the first argument.
- Since the duration of the sound is given by the envelope above, the instrument can be silent for the remaining time. If the given duration is less than 0.5 seconds, you should shorten the decay time accordingly, to avoid a click sound at the end. You can assume that the duration is always longer than 0.01 seconds.

Write the instruments `snare_drum` and `bass_drum` as functions that take a MIDI note and a duration as arguments, and return the corresponding drum sounds.

For the sake of consistency, you are also to represent silence as an instrument. Write a silence "instrument" as a function `mute` that takes a MIDI note and a duration as arguments, and returns a silent sound of the given duration.

You may find the following function useful.

`math_random()` : `returns a random number between 0 and 1`

The fact that the interval of possible return values of `math_random` goes from 0 (inclusive) up to but not including 1 (exclusive) is not important for this question.

The function `adsr` from Mission 10 is also defined for your convinience.

Task 2:

We can go further with our notion of instruments. In this task, you will write a function `combine_instruments` that takes in a list of instruments and return an instrument that when played, produce sound from all instruments at the same time.

`combine_instruments` : $list(instrument) \rightarrow instrument$

For example, the program

```
var snare_drum_and_bell = combine_instruments(list(snare_drum, bell));
play(snare_drum_and_bell(500, 1))
```

will play both snare_drum and bell being played together for a duration of 1 second.

Task 3:

A *simple rhythm* is a list of numerals, each representing a sound to be played. Mute is intuitively represented as 0.
For example, if the first sound plays a snare drum and the second sound plays a bass drum,

then the list `[1,0,2,0]` represents the simple rhythm of a snare drum followed by a mute sound followed by a base drum, followed by a mute sound.

Rhythms are often repeating in complex patterns. Instead of tediously repeating patterns, you invent a rhythm "language".

A rhythm is either:

- a simple rhythm, or
- a list of rhythms, or
- a pair whose head is a rhythm and whose tail is a number.

Of course, if a rhythm is a simple rhythm, it is played as described above. If a rhythm is a list of rhythms, say, `list(R1, R2, ..., Rn)`, it is played by sequentially composing the component rhythms `R1`, `R2`, ..., `Rn`. If a rhythm is a pair `pair(R, n)` whose head is a rhythm `R`, and whose tail is a number `n`, then it represents the same rhythm as `list(R, R, ..., R)`, where the rhythm `R` appears `n` times.

A possible rhythm is for example:

```
var my_rhythm = pair(list(pair(list(1,2,0,1), 2), list(1,3,0,1,3,1,0,3)), 3);
```

To test your understanding of this rhythm language, draw the box-and-pointer diagram of `my_rhythm` (not graded). Can you figure out what simple rhythm it corresponds to? (not graded)

In this task, you are to write a function `simplify_rhythm` that takes a rhythm as argument and returns an equivalent simple rhythm.
For example, `simplify_rhythm(my_rhythm);` should evaluate to the list

```
[1,2,0,1,1,2,0,1,1,3,0,1,3,1,0,3,1,2,0,1,1,2,0,1,1,3,0,1,3,1,0,3,1,2,0,1,1,2,0,1,1,3,0,1
```

You may find the following given functions helpful:

- `is_list(x)`: returns `true` is the given argument is a list, and `false` otherwise.
- `is_number(x)`: returns `true` if the given argument is a number, and `false` otherwise.

Task 4:

Now, you are to actually generate some simple percussion arrangements out of a number of given sounds.

Write a function `simple_percussions` that takes the following three arguments:

- `distance` : the time between the start times of each subsequent sound;
- `list_of_sounds` : a list of sounds to be played;
- `rhythm` : a rhythm, as defined in Task 2.

The function `simple_percussions` returns a sound representing the given `rhythm` played at the speed indicated by the `distance` using the given `list_of_sounds`. At the beginning, the resulting sound contains the first sound of the rhythm, then after `distance` seconds from the beginning comes the second sound of the rhythm, after `2 * distance` seconds from the beginning comes the third sound of the rhythm, and so on. Each numeral in the rhythm represents the sound in the given `list_of_sounds` at the respective position. For example, the numeral `"2"` represents the sound in `list_of_sound` at position 2, where you start counting at 0.

For example,

```
var my_snare_drum = snare_drum(70, 2);
var my_bass_drum = bass_drum(80, 2);
var my_bell = bell(72, 2);
simple_percussions(0.5, list(my_snare_drum, my_bass_drum, my_bell), list(1,2,1,0,3,1,0))
```

will return a percussion sound that plays the sequence `my_snare_drum` - `my_bass_drum` - `my_snare_drum` - `my_mute` - `my_bell` - `my_snare_drum` - `my_mute`, in which each of the sounds start 0.5 seconds apart from the next.

Note: This task is not as easy as simply applying `consecutively` to a list of sounds. A sound may be longer than the given `distance`, and you need to allow it to finish its envelope, while the next sound is already playing.

Task 5:

`simple-percussions` only allows us to play one sound at one beat. In this task, you will write a function `percussions` that takes arguments similar to `simple_percussions`, except the `rhythm` encodes the combinations of percussion instruments being played. For example, the combination of the 1st, 3rd and 4th instrument is encoded with the number $2^0+2^2+2^3 = 13$.

```
percussions(0.5, list(my_snare_drum, my_bass_drum, my_bell), list(1, 2, 4, 0, 5, 6));
```

Will play (my_snare_drum)-(my_bass_drum)-(my_bell)-(my_mute)-(my_snare_drum+my_bell)-(my_bass_drum+my_bell).

Submission

To submit your work to the Source Academy, place your program in the "Source" tab of the online editor within the mission page, save the program by clicking the "Save" button, and click the "Submit" button. Please ensure the required function from each Task is included in your submission. Note that submission is final and that any mistakes in submission requires extra effort from a tutor or the lecturer himself to fix.

IMPORTANT: Make sure you've saved the latest version of your work by clicking the "Save" button before finalizing your submission!