Sick Beets

A Programming Language to Make **Sick Beets**

plt-project-proposal.sb | Stephen Edwards | February 8, 2017

Manager: Courtney Wong (cw2844)
Language Guru: Angel Yang (cy2389)
System Architect: Kevin Shen (ks3206)
Tester: Jin Peng (jjp2172)

Introduction

Sick Beets is a programming language that allows users to compose music by generating .wav files. Sick Beets is inspired by the structured nature of music, which makes it easy to represent a composition piece by defining attributes of notes such as pitch or duration. Using Sick Beets, users can concatenate, overlay, and transpose a series of notes to digitally encode their compositions.

Usage

Sick Beets can be used to easily create music that involves multiple parts and instruments. Built in types allow the user to make interesting melodies and rhythms, and combine small sequences of notes to form a piece. Sick Beets can then output .wav files with the composed song.

To create a piece in Sick Beets, the user can first create a tune, which is a series of notes played by a single instrument. Each note is represented by its letter value. Compositions are not created by specifying key signatures, but by specifying the keys that are being played (ie c#, b flat, a). Then, the user can create a phrase, which is the overlapping of one or more tunes. This means that one phrase can have different instruments playing at the same time. Phrases are then concatenated together to create a track. Tracks are then exported as .wav files.

Syntax

Built-In Types

Name	Example	Explanation
int	8	An integer value
string	beets	Simple text word
note	f#(-1)	First letter represents pitch Second optional character represents sharp or flat Third optional integer designates shift in octaves
notes	[c f#(-1) bb1]	Sequence of notes
duration	hqi, q	Letters designate whole, half, quarter notes, etc. Can be combined without spaces to produce various durations (whole: w, half: h, quarter: q, eighth: i etc.)
rhythm	[q q q q q q h]	Sequence of durations to be applied to each corresponding note
tune	[c e g]:[q q h]	The combination of a notes object with a rhythm object, joined with an : operator
phrase	{melody, bass}	Overlapping of one or more tunes creates a phrase object.
track	intro . chorus	A sequence of phrase objects creates a track object. Create a sequence using the . operator.
instrument	$kick, $clap, $violin, etc.	The instrument specified for a tune. It is designated as an instrument with the $ symbol. Default is piano.

Operators

Operator	Explanation
=	Assigns right operand value to left operand. = following another operator such as +,*,/,-, . performs the operation on the left & right operands and then assigns the resulting value to the left operand.
+	Adds values of left and right operands.

-	Subtracts value of right operand from value of left operand.
*	Multiplies values of left and right operands.
/	Divides value of left operand by value of right operand.
&&	Boolean operator AND.
||	Boolean operator OR.
:	Applies the list of rhythms in the right operand to the notes specified in the left operand.
.	Concatenate the right operand to the left operand.

Control Flow

Control flow will include if-then-else statements, for loops, and while loops. The syntax will look like Python. An example is shown in our Example Code Usage section.

Comments

Single-line comments are designated by `//`. Multiline comments are enclosed by `/* */`.

Example Code Usage

example.sb

```
1  notes = [ g f# e(-1) d e e ]
2  rhythms = [ q q q q i hqi ]
3  melody = notes : rhythms
4  kick = [ c c r c c r ] : [ q q h q q h ] $kick
5  clap = [ r c r c ] h $clap
6  chorus = {melody, kick, clap}
7  we_will_rock_you = chorus . chorus
8  export we_will_rock_you "we_will_rock_you.wav"
9
10 // iterating across instruments
11 song = {}
12 tune = [ c d e d ] q . [ c ] w
13 instruments = [ $guitar $saxophone $flute ]
14 for instrument in instruments:
15     song .= {tune instrument}
16 export song "instruments.wav"
```

Lines 1 to 8 demonstrate the creation of a simple song, and the hierarchy of notes, rhythms, tunes, phrases, and tracks.
Lines 1 and 2 specify the notes and the rhythms for 6 notes by using array-like structures. Notes are designated with the scale letters for an individual note (c, d, e, f, g, a, b), an optional # or b symbol to specify sharps or flats, and an optional octave offset. Rhythms are designated by reserved letters for each duration: w for whole, h for half, q for quarter, i for eighth, s for sixteenth, t for thirty-second. This design decision was based on how the music library, CFugue, defines a note's duration.
Lines 3 to 5 create tunes in different manners, using different instruments.
Line 3 combines a predefined series of pitches to a predefined series of rhythms using the : operator. If the number of pitches and rhythms don't match, an error will be thrown. A tune's default instrument is a piano.
Line 4 merges a series of pitches to a series of rhythms and overrides the default instrument by adding the type of instrument after the notes and rhythms are specified.
Line 5 takes in a series of pitches and one rhythm designation so it applies the rhythm to all the notes to create the tune. It also overrides the default instrument.
Line 6 layers multiple tunes to create a phrase named chorus. Effectively, this plays the tunes simultaneously.
Line 7 concatenates phrases into a track.
Line 8 exports the track into a .wav file with the specified name.

Lines 10-16 show how a user can use a control flow statement to create a track and iterate through different instruments for a single tune.
Line 10 demonstrates the syntax for comments.
Line 11 creates a new phrase object, song.
Line 12 creates a tune by concatenating two sets of notes and rhythms.
Line 13 instantiates an array of instrument objects.
Lines 14-15 concatenates the phrase objects {tune instrument} to the end of song. Each phrase object consists of the same tune, but with a different instrument used.
Line 16 exports the song into a .wav file.