Tonedef

Curtis Henkel - cah2196
Chatura Atapattu - cpa2116
Matt Duane - md2835
Kevin Ramkishun - kr2418

COMS W4115
Fall 2010
Project Proposal

Description

Tonedef is an imperative programming language designed to represent and manipulate the components of musical score. Its basic data types are chosen from the lexicon of music. Tonedef aims to provide a platform for constructing programs using abstractions that are familiar and useful to developers with at least a basic knowledge of music theory (e.g. what pitches are). By defining the most basic elements and operations of musical composition, Tonedef allows programmers to build code of varying complexity according to the depth of their own knowledge. For example, a simple program might allow a user to choose a key and one of several scale types (major, minor, chromatic, etc.) and then perform that scale; a more complex program might analyze a melody and produce an accompaniment of chords that complement the melody and play them both together.

Types

Tonedef will have the int, boolean, and string data types that most languages use, as well as several unique data types for the specifics of describing the components of music.

`step`	Represents the distance between two pitches, which are integers OR integers+1/2. The single-quote character (‘) represents a half-step in step literal tokens. These can be combined into a sequence data type.
`beat`	Represents the duration of a note, which is a rational number, n/d where n, d are integers. 1 whole-note — 1/8 eighth-note 1/2 half-note — 3 2 tied whole-notes 1/4 quarter-note — 3/4 a dotted half note
`pitch`	Represents the pitch of a note. It is comprised of an element of [A-G], an optional [#b] and an octave number. Progression of pitches in an octave follow this sequence { C, C#, D, D#, E, F, F#, G, G#, A, A#, B } where each pitch is a half step above the previous, and this sequence is equal to this sequence of alternate pitch names { B#, Db, D, Eb, Fb, F, Gb, G, Ab, Bb, Cb } (Note: D, F, G have no alternate pitch name). The octaves are numbered from lowest to highest. Pitch literals begin with the dollar-sign ($)

	character. A null pitch (similar in function to the empty string for strings) can be represented by ($_).
`note`	Composed of a pitch and a beat. Notes are constructed from a pitch or a pitch and a beat using the semi-colon (:) operator. `note a = $C0 ;` // a has a pitch of $C0 and a duration of zero `note b = $D1 : 1/4 ;` // b is a quarter-note at pitch $D1 `note c = pX : bY` // can construct from variables

`sequence`	A comma separated list of steps. Used to represent the changes in pitch through time or the distance between notes in a chord - it depends on how you apply it.
	
	`sequence major_scale = 0, 1, 1, ‘, 1, 1, 1, ‘ ;`
	`sequence major_chord1 = 0, 2, 3` ` ;`

`chord`	Collection of notes that occur simultaneously. Built from a list of notes;
`phrase`	Collection of chords in time, represented as a list of chords, each 1/16th of a second in time. The order of chords determines which notes are played at a given time.
`rhythm`	Series of beats. Built from a string of characters 1,0,- to describe a rhythm of beats where each character has the same duration in time and spaces are allowed but neglected by compiler/functions that use rhythm. The dash (-) character represents a sustaining of a note. `rhythm r = “1--- 1100 1-1-” ;` r represents a whole note, then 2 quarter notes, then a half rest, then 2 half notes.

Operators

Common basic arithmetic:

`id = expr`	Assignment to variable identifier.
`- expr`	Negation - unary operator on integers and steps.
`expr1 + expr2` `expr1 - expr2`	Addition/Subtraction - can add and subtract integers, steps, and beats where both expressions are same type (integers can be interpreted as both steps and beats).

`expr1 * expr2` `expr1 / expr2`	Multiplication/Division - can multiply and divide integers, beats, but not steps
`expr1 % expr2`	Modulo - can provide the modulo between two integers or beats, but not steps.
`expr1 <comp> expr2` `<comp> = { ==, >, <, <=, >= }`	Comparison - basic arithmetic comparison on int, bool, step, pitch, beat.

Language specific operators:

`note ^ step`	Raise note - raises the “note” by “step” steps and returns a new note. Negative steps will lower the note.
`note ^^ int`	Raise note by octave - raises/lowers the “note” by “int” octaves
`pitch : beat`	Note creation - creates a note from pitch and beat. Notes can be used as either side and are interpreted as the “pitch-component” or “beat-component” of the note.
`chord|note + chord|note`	Chord addition - creates a new chord with the notes in the two operands
`pitch - pitch` `note - note`	Distance between pitches - returns a step equaling the distance between the two pitches
`note :: sequence`	Apply sequence to build a chord relative to note. Each step element of sequence is interpreted as a step from note; therefore, order of the steps does not matter.
`note << sequence`	Apply sequence to build a phrase starting at note with all notes in phrase having same beat as note. The steps in sequence are interpreted relative to the previous note in sequence, note the initial note.
`phrase @@ phrase`	Appending phrases - creates new phrase of left phrase followed by right phrase.
`phrase ** phrase`	Combining phrases - creates new phrase of the 2 phrases of notes merged (i.e. notes starting at same time).
`phrase >> beat`	Shift phrase - adds beats of rest to the front of phrase.

`phrase << rhythm`	Apply rhythm - creates new phrase of notes with same pitches and order as phrase, but with beats as specified by rhythm. Useful on phrases built by the note<<sequence operator. `phrase p = noteX << sequenceY << rhythmZ ;`
play phrase	outputs the musical info represented by phrase to speakers

Control

if-then-else	The standard if-then-else conditional execution of code blocks
for-loop	The standard for loop syntax. for (initial; condition; next) { block }
while-loop	The standard while loop syntax. while (condition) { block }
function	Keyword for defining functions similar to C, but with an explicit function keyword to aid parsing. return_type function identifier (arg list) { block }

foreach-in-loop	Loop mechanism for iterating over the time points of a phrase, which are each a chord instance, which can have 0 or more notes. Also for iterating over the notes in a chord. Example,
	
	`foreach (chordY in phraseX) {`
	`  foreach (noteZ in chordY) {`
	`  //code`
	`  }`
	`}`

Sample Code

```
// defines a function to play an ascending major scale starting at
// the note parameter
void function play_major_scale (note n) {
   sequence major_scale = 0, 1, 1, `, 1, 1, 1, `;
   phrase ph = n << major_scale;
   play ph;
}
```

```
// defines 5 pitches of interest
int num = 5;
pitch[] parray = { $C5, $D4, $E3, $F2, $G1 };

// loops on the pitches. makes an eighth-note at that pitch
// then plays a major scale of eighth notes starting at that pitch
for (int i = 0; i < num; i++){
   note x = parray[i] : 1/8 ;
   play_major_scale ( x );
}
```

```
// defines a function that takes a phrase and an integer
// and return a new phrase that is the same notes but raised
// by num octaves;
phrase function up_octaves ( phrase ph , int num) {
   phrase ph2 ;
   foreach ( chord c in ph ){
      chord c2 ;
      foreach (note n in c) {
         note n2 = n ^^ num ;
         c2 = c2 + n2;
      }
      ph2 = ph2 @@ c2 ;
   }
}

// combines a phrase with itself raised 2 octaves, then plays it
phrase high = up_octaves ( low, 2 );
phrase both = high ** low;
play both;
```

```
// this seq. represents a minor arpeggio up and a minor scale down
sequence s1 =  0, 1`, 2, 3`, -1, -`, -1, -1, -1, -`, -1 ;

// this rhythm plays the first 3 notes as quarter notes then
// a whole note, then 6 eighth notes, then a half note
rhythm r1 = "1-1-1-1- ------11 11111---";

// constructs a phrase from the sequence and rhythm starting at
// pitch Eb and plays it
phrase ph1 = $Eb2 << s1 << r1 ;
play ph1;
```