PLT Language Proposal: MIDILC

9/29/10

Ben Mann (BJM2122)
Ye Liu (yl2428)
Fred Lowenthal (fml2106)
Akiva Bamberger (ab2928)

Describe the language that you plan to implement.

The language, hereafter referred to as MIDILC (pronounced MIDDLE C, standing for MIDI Language Compiler), allows programmers to compose music. It compiles into MIDI format and has syntax that is similar to Java, but changes the basic primitives and the meaning of various operators. The basic primitives are:

Number
- immutable, limited to [-128,128]

Note
- immutable
- stores pitch, volume, and duration

Chord
- immutable
- stores simultaneous notes with equal durations

Sequence
- mutable
- stores start and "current time"
- list of notes and their associated start times
- index of when each note was added to this sequence

The langauge is strongly typed. Each of the above primitives supports the + operator, which increments the duration of each note and the .+ operator, which increases the pitch of each note. The same goes for - and .- to decrement these properties. Each type can be safely cast up in the following order: Number → Note → Chord → Sequence. The standard library, written in the language itself, supports major and minor chords, arpeggios, repetition, and other such basic and often used concepts. Note durations are specified in terms of whole notes (w), halves (h), quarters (q), eigths (e), and sixteenths (s). Sequences can either be appended to, which advances the "current time" by however long the appended sequence is, or else something can be inserted into a sequence at a given offset using bracket notation, as in arrays for java. Sequences also support slice notation (as in Matlab and Python), which returns a new sequence containing all of the notes that start within the specified boundaries. Methods are specified in the same way as in Java. To specify which sequence to compile, return is used in the scope of the main method, which by default is the outermost scope.

In order to actually write the MIDI files, the compiler will interface with a Java program using

the javax.sound.MIDI package.

Explain what problem your language can solve and how it should be used.

Composing music on a computer is often done using GUIs that allow the user to drag and drop notes, hear their compositions as they are creating them, and finally compile into some music file format, such as MP3 or MIDI. Unfortunately these systems are designed for musicians and often take input from an electronic keyboard, which means composing still requires a high degree of musical expertise. Composing harmony and chords requires thought each time these structures should be used.
In addition, songs often have frequent recurring themes. Manually reusing these themes requires precision and dedication. If pieces of a song could be manipulated automatically for reuse and slight modification, song production speed could increase dramatically.
The MIDILC language allows programmers to algorithmically generate notes, chords, and sequences of notes and chords by writing functions. This allows for writing interesting compositions that minimize time spent rewriting basic MIDI manipulation routines and implementing primitive musical constructs. The language works optimally with compositions that make consistent use of simple motifs as it encourages reuse of simple mathematical operations on notes and chords.

Describe an interesting, representative program in your language.

One interesting program that could be implemented in our language is the creation of counterpoints based on a given melody. Given a melody sequence M containing a series of W whole notes N, one could easily manipulate the notes in M to create sequences of counterpoints to the melody. One way to do to so is to use the notes of a chord in the key of each note. First, the user would use library functions such as Major(), Minor(), or implement their own chord-creating method to obtain a chord based on N. Each of these methods should return a Chord object containing the notes of the given chord. Now, the user would call a function that generates an arpeggio from the input chord, separating the notes making up the chord so that they would play sequentially instead of simultaneously. Given these notes, the user could build a new sequence with permutations of the notes, as whole, half, or quarter note as needed, to be the counterpoint for the original melody. Finally, when all the notes of melody M have been used to provide chords, and all the chords have given rise to permutations of notes in the chords, the user would merge the two sequences using brace notation on the earlier sequence (M[0]). Now, when the sequences are written to midi and played back, there will be a counterpoint in addition to the melody.

Give some examples of its syntax and an explanation of what it does.

// Basic objects are Note, Chord, and Sequence

```
// This function takes a note, forms, the root chord, and arpeggiates it
Sequence arpeggiate( Note root )
{
	// This segment creates a major chord given a root note
	Chord notes = ( root, root .+ 4, root .+ 7 )

	// This segment creates a new sequence and appends the three notes to it
	// As each note is added, the sequence moves the pointer forward in time
	Sequence arpeggio
	arpeggio += root
	// The .+ operator specifies a pitch addition
	arpeggio += root .+ 4
	arpeggio += root .+ 7

	return arpeggio
}

Chord majorChord( Note root )
{
	Chord notes = ( root, root .+ 4, root .+ 7 )

	return notes
}

// This function transposes a chord to a new key, specified by a note
Chord transposeChord( Chord oldChord, Note newKey )
{
	Number keyDifference = newKey .- oldChord[0]
	Chord newChord = oldChord .+ keyDifference

	return newChord
}

Sequence oneFourFiveProg( Note root )
{
	Sequence progression

	Note root2 = root + 5
	Note root3 = root + 7

	Chord cOne = majorChord( root )
	Chord cFour = transposeChord( cOne, root2 )
	Chord cFive = transposeChord( cOne, root3 )
```

```
        progression += cOne
        progression += cOne
        progression += cFour
        progression += cOne
        progression += cOne
        progression += cFive
        progression += cFour
        progression += cOne

        return progression
}

// Predefined notes are specified using the note, accidental (optional),
// octave (optional-default middle octave), and duration (optional-default quarter note)

Sequence cProgression = oneFourFiveProg( C3q ) // predefined quarter note
Sequence eProgression = oneFourFiveProg( E4h ) // predefined half note
Sequence gProgression = oneFourFiveProg( G7q )

Sequence theSong
// As sequences are appended to the “theSong” sequence, the notes within them are merged and
made // relative to the “theSong” start
theSong += cProgression
theSong += eProgression
// Slice notation can be used to return a subsequence containing only the notes from the specified
time   // range
theSong += theSong[2q:5q]
theSong += gProgression

setTempo( 120 )
playSong( theSong )
```