ChucK Library Extensions

Avneesh Sarwate
COS 2014
Spring 2014 Independent Work
Advisor: Brian Kernighan

Abstract

ChucK is a music programming language that supports algorithmic music composition and live coding. The purpose of this project is to create tools to facilitate these activities in ChucK. The library consists of three modules: a module of data structures, a module of musical structures, and a module for encapsulating and streamlining the use of live coding features already present in ChucK. This paper will provide a brief survey of other popular music programming tools, describe ChucK, describe the implemented library, and discuss the results of a user evaluation and future work.

1. Introduction

1.1. MUSIC PROGRAMMING TOOLS

Music programming languages have been present for almost as long as general purpose programming languages. Max Matthews developed what many consider the first programming language for music, called MUSIC, in 1957 at Bell Labs [6]. MUSIC and its descendants, known as the MUSIC-N family, were responsible for many of the paradigms that are now common in music programming languages, particularly that of the Unit Generator (UGen), and the separation between Score and Orchestra. UGens are "atomic, often predefined, building blocks for generating or processing audio signals." These building blocks can be things like filters or wave generators. The Score vs Orchestra paradigm is the separation of the representation of the musical content (notes, chords, etc) from the representation of the "instruments" or sounds used to play that content [6].

Currently, the most popular music programming language is Max/MSP [6]. Max/MSP is a graphical programming language, where each object is a block on a screen. These objects can either compute data or audio signals, and the inputs and outputs of different boxes can be connected by dragging a line from the output of one to the input of another. Max/MSP and its open source implementation, Pure Data, have a very large user community.

Many general-purpose programming languages have extensive libraries for musical applications. The Synthesis Toolkit is a C++ library for signal processing developed by Perry Cook and Gary Scavone [2]. It implements many lower level UGens, such as various filters and oscillators, as well as classes for physical modeling of instruments. Physical modeling synthesis is the synthesis of sound using circuit models that simulate the resonance and filtering properties of physical objects. Music21 is a Python library for symbolic music analysis developed by Christopher Ariza [1]. It provides a very rich representation of symbolic music data, as well as a host of features for parsing and analyzing symbolic music stored in a variety of different formats. Euterpea is a domain specific language embedded within Haskell with wide ranging functionality, from algorithmic music composition at a symbolic level to low-level sound synthesis [3].

There also exist several highly popular text-based music programming languages. Csound is one of the earliest computer languages still in common use. It was developed in the late 1980s by Barry Vercoe [6]. As a language in the MUSIC-N family, it fully embraces the paradigms of UGens and separate abstractions for Orchestras and Scores. Super Collider is another popular language for computer music [4]. It allows for real-time signal processing and algorithmic music creation, and also has libraries to support for musical live-coding, the practice of manipulating musical output through interactive, real-time coding.

1.2. GOALS

ChucK, the language focused on in this paper, is a music programming language developed at Princeton that allows for real-time signal processing and live coding [6]. Chuck has several interesting features (explained in section 2) that make it very attractive for live coding. However, ChucK does not provide any high-level abstractions for algorithmic music composition. The purpose of this project is to develop a library for ChucK that provides tools for algorithmic music composition and tools to simplify live coding in ChucK. The creating of these high level musical abstractions, when combined with tools to simplify live coding, could greatly increase the power of ChucK as a live coding tool.

1.3. OVERVIEW

Section 2 describes the ChucK programming language and its various features. Section 3 describes the library that has been implemented, and section 4 discusses a user evaluation and future work.

2. THE CHUCK LANGUAGE[1]

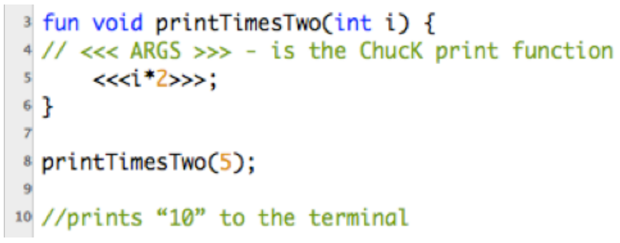
2.1. ARCHITECTURE

ChucK is a language designed with a focus on threading and concurrency. In ChucK, threads are called "shreds" and have special synchronicity properties when compared to threads from other languages (threading and concurrency are discussed more in section 1.5). ChucK source files define shreds that are run on the ChucK Virtual Machine. These shreds can be added either manually or programmatically and run in parallel on the virtual machine. This ability to

[1] This entire section borrows heavily from Ge Wang's 2008 PhD thesis. [7]

flexibly add and synchronize shreds, even while other shreds are running, is called “on-the-fly” programming. Each shred has its own namespace, but child shreds can access the global variables of their parent shreds. To allow for sharing of data between shreds, users must create a public class with static data. In doing so, the public class’s definition is added to the VM namespace rather than a shred namespace, allowing any running shred to access the public class’s static data. The latest available (as of May 2014) information about the system’s level implementation on ChucK is from 2008. Based on the literature and on code testing, it seems accurate to say that the ChucK Virtual Machine executes all ChucK instructions on a single core Virtual Processor.

2.2. OBJECT MODEL



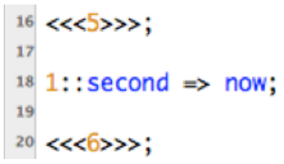
```
fun void printTimesTwo(int i) {
// <<< ARGS >>> - is the ChucK print function
    <<<i*2>>>;
}

printTimesTwo(5);

//prints “10” to the terminal
```

Figure 1

ChucK is a compiled, statically typed, object oriented language. Unlike Java, code in ChucK does not have to be contained in a class. Logic can be executed outside of a class, and functions can exist independent of classes (Figure 1). ChucK primitives include integers, floats, time (the representation of the current time in the VM), duration (a duration of elapsed VM time), complex numbers, and polar numbers. All classes in ChucK inherit from the Object class. Chuck supports only single inheritance, and does not implement interfaces or generic types. Unlike in Java, these primitives do not have object wrappers, thus they cannot be assigned to Object variables. Also, given an array of a subclass, it is not possible to cast it to an array of its superclass, which is possible in Java. ChucK allows an object variable to be cast to a superclass, and upon downcasting that variable back to its original class, preserves its values. However, unlike Java, ChucK seems to allow for arbitrary downcasting from a superclass variable to a subclass variable, regardless of what type the superclass variable was cast from, resulting in undefined behavior. This seems to be a flaw in the type system rather than a planned feature.

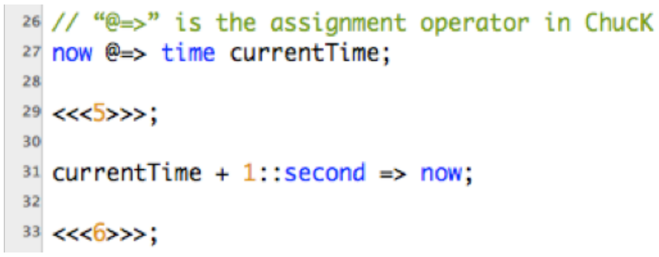
2.3. TIME CONTROL

ChucK allows programmers to explicitly control the time flow of their program. The virtual machine strictly keeps track of the flow of time in each running shred, allowing for deterministic synchronicity (described later). The "now" keyword is used to make a shred yield for a particular duration of time or until a specific time. For example, the code in Figure 2 prints the number 5, makes the shred wait for 1 second, and then prints the number 6. The code "1::second" is a duration value. This is behaviorally equivalent to Figure 3, which saves the current VM time, and makes the shred wait until the VM time is "currentTime+1::second". This time control is crucial for signal processing and synchronization, explained later on.



```
<<<5>>>;

1::second => now;

<<<6>>>;
```

Figure 2

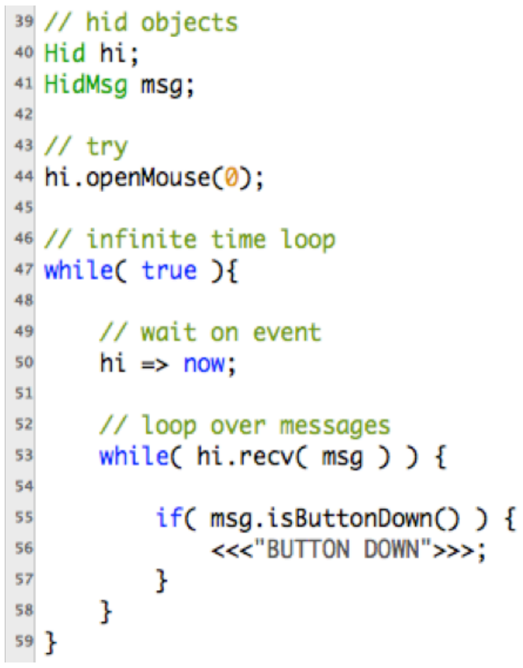


```
// "@=>" is the assignment operator in ChucK
now @=> time currentTime;

<<<5>>>;

currentTime + 1::second => now;

<<<6>>>;
```

Figure 3

2.4. EVENTS

Threads can also be forced to yield for events, which can be either synchronous triggers from ChucK or asynchronous input from external hardware or software. ChucK implements an Event class, with built-in subclasses for HID[2], MIDI, and OSC[3] [8]. In Figure 4[4], the shred is running in an infinite loop, but hangs (at line 50) until it receives a mouse click event, when it prints "BUTTON DOWN."

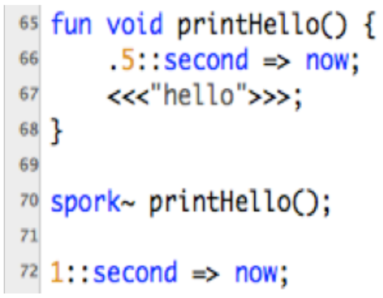


```
39 // hid objects
40 Hid hi;
41 HidMsg msg;
42
43 // try
44 hi.openMouse(0);
45
46 // infinite time loop
47 while( true ){
48
49     // wait on event
50     hi => now;
51
52     // loop over messages
53     while( hi.recv( msg ) ) {
54
55         if( msg.isButtonDown() ) {
56             <<<"BUTTON DOWN">>>;
57         }
58     }
59 }
```

Figure 4

2.5. THREADING AND SYNCHRONICITY

ChucK encourages the use of concurrency and multi-threaded, or in ChucK's case "multi-shredded" programs. Shreds can be created either manually, by adding ChucK source files to the virtual machine, as described above, or programmatically. Programmatically added shreds must be defined as a function and can be launched via the "spork~" function. Figure 5 sporks a the function "printHello()", which waits for .5 seconds and prints "hello".



```
65 fun void printHello() {
66     .5::second => now;
67     <<<"hello">>>;
68 }
69
70 spork~ printHello();
71
72 1::second => now;
```

Figure 5

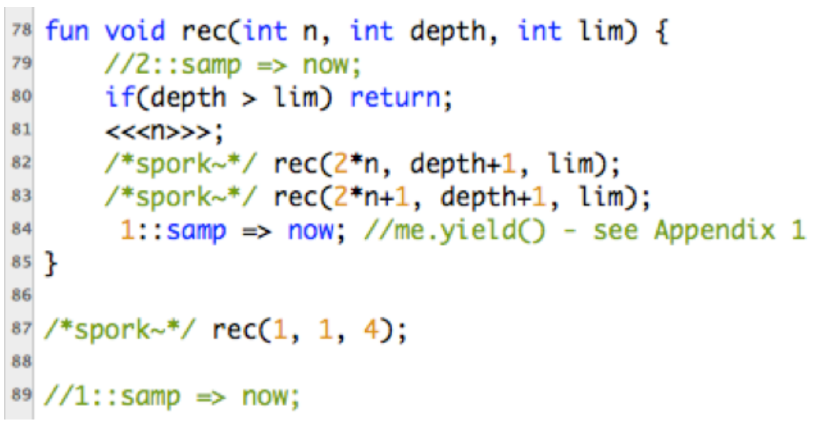
If line 72 in Figure 5 was removed, nothing would be printed to the output. This highlights the "Shreduler" (which is ChucK's shred scheduler) in action. When a shred is "sporked," it is entered into a queue of threads waiting to be run. A single active shred will own the processor and be executing instructions while others hang (and compute audio if necessary, explained later). The

[2] The Human Interactive Device (HID) protocol is a subset of the USB protocol (USB-HID) that governs the data provided by interactive hardware devices such as mice, keyboards, joystics, etc
[3] The OSC protocol is a UDP based networking protocol often used in computer music
[4] Code for Figure 4 adapted from the examples at http://chuck.cs.princeton.edu

active shred will only relinquish the processor once it executes an instruction for time to advance, waiting for an event, or if me.yield() is called (me is a keyword referring to the current shred, and yield() relinquishes the processor to other shreds scheduled to execute at the current time). Once a shred relinquishes the processor, the next shred on the queue will be shreduled to run (either from the start, or from wherever it was last waiting) until it too either exits or relinquishes the processor for the above reasons. Also, child shreds only live as long as their parent shreds are alive. In Figure 5, when line 72 is commented out, the parent shred exits before relinquishing the processor to its child, and thus nothing is printed. If the wait time is set to < .5 seconds there will be no output: the parent shred will relinquish the processor to wait for the child, then the child shred will relinquish the processor, but then the parent thread will become active and close before the child thread is schedule to become active and print.

This shredding model assures that the shredding order is deterministic. Because shreds must themselves relinquish the processor before another shred runs, shreds will never preempt each other. This means that any block of code between two processor-relinquishing commands (time advancements, event waits, or me.yield()) will be run without interference from other shreds. Shred execution is naturally mutually exclusive, and there is no need for synchronization primitives such as locks or semaphores; blocks that must stay "atomic" must simply avoid putting a processor-relinquishing command in the block, and the natural mutual exclusion prevents any other shred from changing shared memory while that block is being executed.

The code in Figure 6a demonstrates the various cases of



```
fun void rec(int n, int depth, int lim) {
    //2::samp => now;
    if(depth > lim) return;
    <<<n>>>;
    /*spork~*/ rec(2*n, depth+1, lim);
    /*spork~*/ rec(2*n+1, depth+1, lim);
     1::samp => now; //me.yield() - see Appendix 1
}

/*spork~*/ rec(1, 1, 4);

//1::samp => now;
```

Figure 6a

the shreduler in action. The recursion in this function defines a perfect binary tree of size 15, with nodes numbered according to the diagram (Figure 6b). If we were to run the code without sporking lines 82 and 83 and

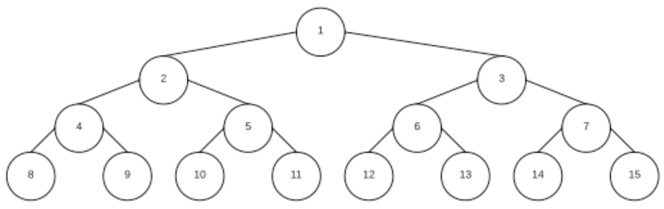


Figure 6b

simply having normal function calls instead, the order of the tree traversal would be a pre order traversal, printing the numbers 1-15 in the order of “1-2-4-8-9-5-10-11-3-6-12-13-7-14-15”. However, when we DO spork the calls in lines 82 and 83, we see that the order of the output is the numbers 1-15 in increasing order. The shred queueing occurs as follows:

1. First, rec(1, 1, 4) shredules rec(2, 2, 4) and rec(2, 3, 4) and then yields while waiting 1::samp to elapse.
2. The yield of rec(1, 1, 4) then lets rec(2, 2, 4) run, putting rec(4, 3, 4) and rec(5, 3, 4) on the queue before yielding itself.
3. The shred rec(2, 3, 4), being next on the queue is then scheduled next, putting rec(6, 3, 4) and rec(7, 3, 4).
4. This leaves the queue to be (from front to back) rec(4, 3, 4) rec(5, 3, 4) rec(6, 3, 4) rec(7, 3, 4). This pattern then continues down the tree.

If we comment out line 84, we see that the recursion halts after printing “1.” This is because rec(1, 1, 4) returns before relinquishing control to the shreds. If we uncomment line 79, we see again that the output is only 1, but this time rec(1, 1, 4) relinquishes control to rec(2, 2, 4), but then returns before rec(2, 2, 4) and rec(3, 2, 4) are finished waiting, killing the children and halting the recursion.

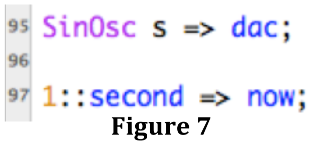
5.1 Events (contd.)

All Events have a method ".signal()", which schedules the next shred waiting on that event to run now. Events also have a method ".broadcast()" that schedules all threads waiting on that event to run now.

2.6. UNIT GENERATORS

ChucK also implements several Unit Generators (UGens). These include both signal processing modules, such as various oscillators and filters, as well as physical modeling instruments created by Perry Cook and Gary Scavone. The physical modeling instruments and some of the other UGens are implemented using the Synthesis Toolkit (STK) by Perry Cook and Gary Scavone.

Unit Generators can be linked to each other via the ChucK operator. Figure 7 connects a sine wave oscillator (with default frequency 220) to the Digital Audio Converter (whatever audio output is specified in the ChucK preferences), advances time 1 second, and then returns. The second line highlights the fact that UGens only calculate and output audio signals when ChucK time passes. We could also have a shred calculate audio while waiting for an event. As mentioned before, this event could also come from an external hardware device, such as a MIDI keyboard or JoyStick.

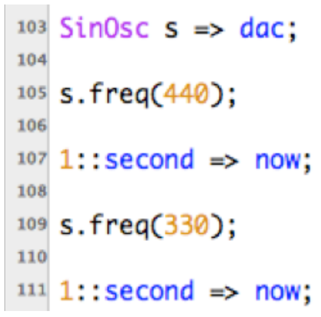


```
SinOsc s => dac;

1::second => now;
```

Figure 7

Several UGens can be linked together to create a signal chain. ChucK signal chains calculate audio based on a "pull" model. The DAC will request a sample from the UGens that feeds into it, and then those UGens will request audio from their preceding UGens, and so on. Another endpoint called a "blackhole" can be used to "pull" samples and calculate audio without actually playing it. This can be used to modulate one audio signal with another without needing to play both signals.

The code in Figure 8 shows a sinusoidal oscillator outputting audio at 440 hz for 1 second and then at 330 hz for another second before the shred exists. Additional logic could be added after and before the calls to advance time in lines 107 and 111. With regards to VM time, ChucK shreds execute logic as if it were occurring instantaneously, never advancing time without an explicit call to do so. This means that, if the logic before line 107 or between lines 107 and 111 took a long time to compute, the next sample of audio calculated in that shred would be delayed, throwing the audio calculation off from the sample rate and potentially creating problems if the audio is being played in real-time. However, this kind of delay does not break the determinism of ChucK's execution – if the audio signal was being written to a file, the audio file would still have the "correct" data. To handle cases where a ChucK shred is taking up a large amount of time on the processor, the VM issues a prompt to the user to either let the shred keep running or abort it.



```
103 SinOsc s => dac;
104
105 s.freq(440);
106
107 1::second => now;
108
109 s.freq(330);
110
111 1::second => now;
```

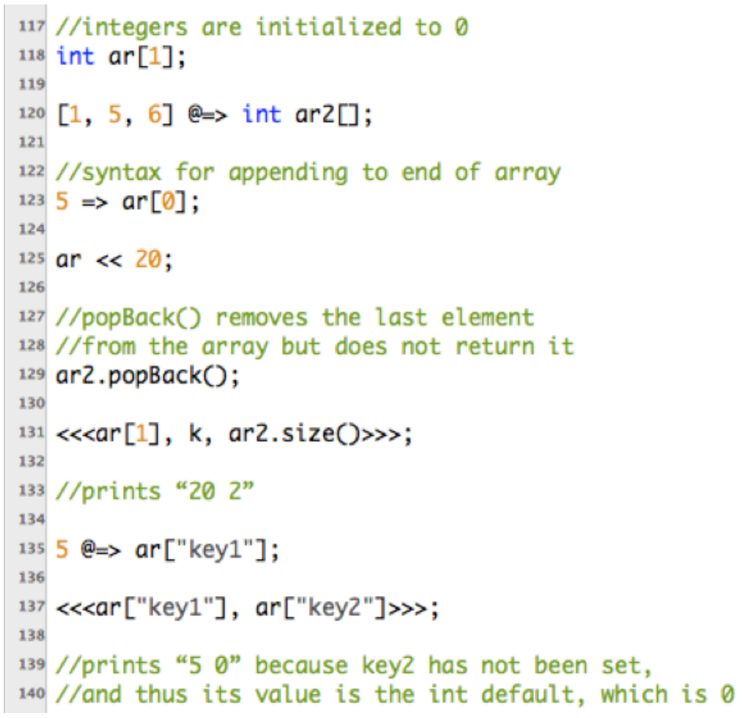
Figure 8

2.7. BIG PICTURE

The end result of ChucK's shred model of execution is that logical execution is actually never parallel, while audio computation is highly parallel. No matter how many shreds are running, only one "active" shred is ever executing instructions in code, and logical flow is never parallel. Yet, all "inactive" shreds with UGen's connected to a blackhole/daw are continually calculating audio in parallel. This allows for the writing of massively threaded code (allowing for optimized parallelization of audio calculation) without ever needing to worry about thread safety in terms of program logic.

2.8. CURRENT LIBRARY (and basic data structures)

ChucK currently has several libraries for convenience. It supports IO from audio, text, and MIDI files, and also supports Serial IO. It also has libraries for string manipulation and regular expressions on strings. ChucK also includes a math library encapsulating commonly used functions. ChucK also includes classes for sending OSC and MIDI messages and Event classes for receiving messages on these protocols. It also includes a base Event class meant to be extended into custom events. ChucK's only built in data structure is a combination of an array, a partial implementation of an expanding stack, and a string map, but is simply called an "array" in the documentation. It is used and declared like a normal array, but can have items append to its end and popped off (but not returned) of the end Figure 9. It also has a separate namespace for string keys and can be used like a hash table (Figure 9 lines 135 – 140).



```
117 //integers are initialized to 0
118 int ar[1];
119 
120 [1, 5, 6] @=> int ar2[];
121 
122 //syntax for appending to end of array
123 5 => ar[0];
124 
125 ar << 20;
126 
127 //popBack() removes the last element
128 //from the array but does not return it
129 ar2.popBack();
130 
131 <<<ar[1], k, ar2.size()>>>;
132 
133 //prints "20 2"
134 
135 5 @=> ar["key1"];
136 
137 <<<ar["key1"], ar["key2"]>>>;
138 
139 //prints "5 0" because key2 has not been set,
140 //and thus its value is the int default, which is 0
```

Figure 9

3. NEW LIBRARIES

ChucK's features are very well suited to quickly prototype different signal chains and to intuitively and easily manipulate digital audio at a range of resolutions. Its on-the-fly programming capabilities also make the ChucK environment much more interactive than those of many other programming languages. These features make it a perfect platform for musical live coding.

However, ChucK lacks objects to represent and play discrete musical structures such as chords or melodies, which is an obvious drawback when it comes to musical programming. It also lacks the standard libraries of algorithms and data structures that are built into most developed programming languages, which also hinders the development of musical algorithms in ChucK. The aim of this project is to develop ChucK libraries for these features in order to make music making in ChucK even easier. The appendices show an example of music written using these new libraries, consisting of a melody algorithmically generated over a static chord progression.

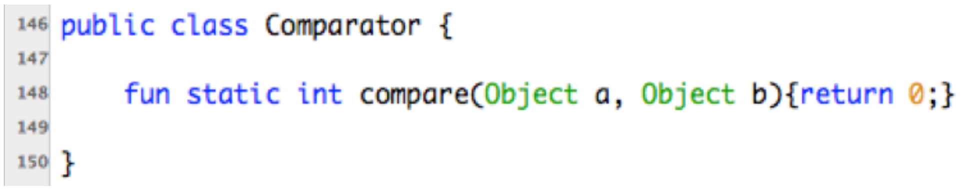
There are three main modules in this library:

1. A collection of data structures to facilitate the development of musical algorithms.
2. Classes for the representation of more "traditional" notated music.
3. Utility classes to help create musical programs for both non-live and live coding.

3.1. DATA STRUCTURES

The two collection data structures implemented in this library are Python-style lists (the List class) and BST maps (the SymTable class). These we designed to be collections to all data types, including ChucK primitives and user-defined classes. ChucK does not support generic typing, so when lists and maps are given user-defined types to hold, they store and return them as Object type objects. It is up to the user to cast them back to their original type.

We've implemented a base Comparator class (Figure 11) that defines a method ".compare(Object a, Object b)", which returns an integer (it should be implemented so that compare(a, b) returns 1 if a > b, -1 if a < b, and 0 if a == b). This comparator is meant to be subclassed for user-defined types, and the subclassed implementation must cast a and b back to their original type before performing the comparison. Because BSTs organize themselves by comparisons on keys, a Comparator subclass is a necessary argument when instantiating a BST containing user-defined types as keys. If users wish to sort a list containing items of a user-defined type, they must provide the list a Comparator subclass for that type. Because ChucK does not support multiple inheritance, we decided it would be more flexible to have a separate Comparator class than to force all types used with our collections to be subclasses of some "Comparable" class that implements the method compare.



```
public class Comparator {

    fun static int compare(Object a, Object b){return 0;}

}
```

Figure 11

In the implementation here, an attempt was made to have only a single class facing the user, no matter what type the user was storing in the collection. As a result, when the collections are initialized, they must be done so through a constructor-like function (ChucK does not support true constructors) that takes one or more strings as arguments to determine what type(s) are allowed to be stored in the collection. For example, the function call "List.newList("int")" returns a list of integers, and the function call "List.newTable("int", "Object")" returns a new map that takes integer keys and Object (and thus any user defined type) values.

In the following documentation, the notation used is as follows: "foo(int arg1, [string arg2])" means that the function "foo" has a mandatory integer argument and an optional string argument, and "foo(type arg1)" means foo is overloaded to take arguments of multiple types, depending on what type(s) are specified for the collection. Also, "foo[type]()" means that the function name's root is "foo" and that its suffix used depends on the type of the element the data structure holds (the suffixes are Int, String, Float, Obj). For example, a list of integers would have the functions popObj(), popString(), and popFloat() publicly accessible, but calling them would result in an error.

3.1.1 Lists

ChucK supports the following methods (in the following descriptions, the "front" of the list is the 0th index):

- (static) newList(string typeStr) – This function takes a string (which must be from the set {"string", "int", "float", "Object"}) that determines what type of elements the list will contain. It returns an empty list configured to throw errors if items of any type other than the specified type are added to the list
- size() – returns the number of (key, value) pairs in the map. Runs in *O(1)* time.
- enqueue(type item) – add an element to the front of the list. Runs in *O(1)* time.
- dequeue[Type]() – removes and returns the item from the back of the list. Runs in *O(1)* time.
- push(type item) – adds an item to the back of the list. Runs in *O(1)* time.
- pop[Type]() – removes and returns the item from the back of the list. Runs in *O(1)* time.
- insert(int index, type item) – inserts the item at the index. Runs in *O(N/2)* time

- remove[Type](int index) – removes and returns the item at the index. Runs in *O(N/2)* time
- set(int index, type item) – sets the element of the list at the index to be item. Runs in *O(1)* time.
- get[Type](int index) – returns the item of the list at the index. Runs in 1 time.
- setComparator(Comparator comp) – sets comp to be the Comparator used when sorting the list (only if the list is holding objects).
- sort() – sorts the list in either natural order (for primitives), or according to the order specified by the comparator. Runs in *O(N lg(N))* time.

The list is implemented as a circular buffer. A "head" variable contains the index of the first item in the list, and a "tail" variable contains the index of the last item. The implementation of the list as a circular buffer allows both head and tail indices to move forwards and backwards, allowing all insertions and removals on ends of the list to occur in constant time. Also, the circular buffer implementation allows the insert and remove operations to run in N/2 time rather than N time: when an insert or remove occurs at index i, the list shifts the smaller of the two segments [0, i], [i, size], rather than simply [i, size] by default.

3.1.2 Maps

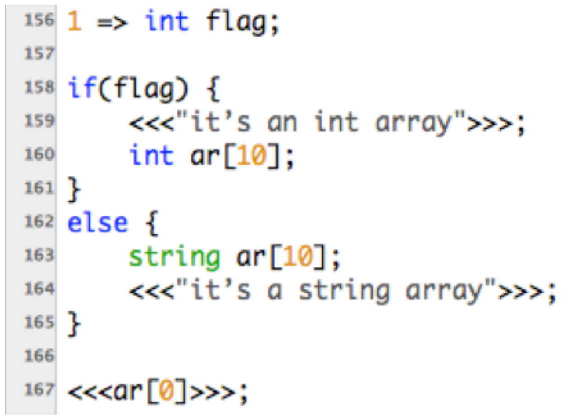
Maps are implemented as Left Leaning Red Black Trees because of their guaranteed *O(lg(N))* performance. Map methods are:

- (static) newTable(string keyStr, string valStr, [Comparator c]) This function takes two strings (which must be from the set {"string", "int", "float", "Object"}) the first of which determines the type of the keys and the second of which determines the type of the values. It returns an empty map that is configured to throw errors if given keys or values of any type other than the ones specified in the arguments. The Comparator argument must be provided only if the key is of a user-defined type.

- size() – returns the number of (key, value) pairs in the map. Runs in 1 time.
- put(type key, type val) – adds the (key, val) pair to the map. Runs in lgN time.
- contains(type key) – checks if the key is in the map. Runs in lgN time.
- get(type key) – returns the value paired with key in the map. Runs in lgN time.
- remove[Type](type key) – removes key and its paired value from the map. Runs in lgN time.
- keys() – returns a list of all keys in the table.

3.1.3 Challenges in Implementing Generic Collections

One of the main challenges in implementing these data structures was to provide a single, simple interface for users while still allowing the collections to hold arbitrary types of data. The solution taken here was to create several parallel singly-typed implementations of the data structures, one for each type (for Lists) or each type combination (for maps), and then wrap them in a single interface. For example, the List class is a wrapper around the classes IntList, StrList, FltList, and ObjList, which hold integers, strings, floats, and Objects, respectively.



```
1 => int flag;

if(flag) {
    <<<"it's an int array">>>;
    int ar[10];
}
else {
    string ar[10];
    <<<"it's a string array">>>;
}

<<<ar[0]>>>;
```

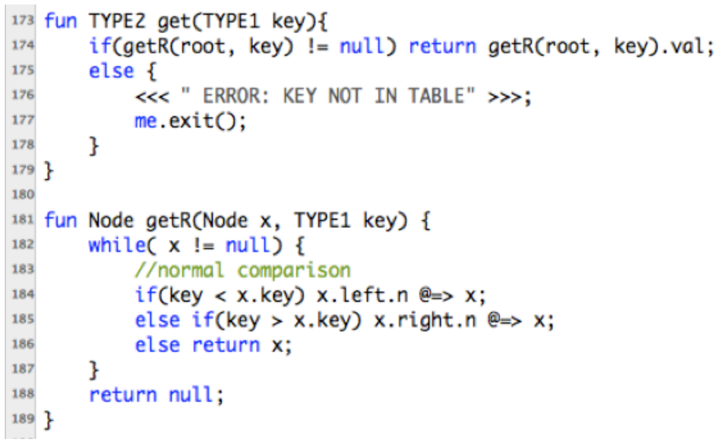
Figure 12

There were two main challenges that led to creating parallel implementations. Firstly, because of ChucK's static typing, the internal structures which store the data in the collections would have to be duplicated for each type that is allowed to be stored. For example, Lists are implemented on top of arrays, but code of the form Figure 12 is not legal in ChucK. Because of this, the List must contain an array for each type (integer, string, float, and Object), and then only use the relevant array after it is created and its element type is set via newList().

Secondly, if we wanted to present a single class and uniform interface to the user, all of the methods that insert into a collection would need to be overloaded to be able to handle each supported element type. Also, for methods that returned an item from a collection, a different method name would need to be implemented for each type of element the list could contain. This is because even though the get methods could share input types, their output types could be different depending on what type the collection is holding. This variability in return types prevents two functions with the same name from being overloaded. For example, if we had a map m with integer keys and Object values, calling m.get(i) (where i is an integer) would return an object, while calling this same method on a symbol table with integer keys and string values would return a string. Thus, we would need different names for functions returning different values. The solution to this issue was the root-prefix naming scheme mentioned above.

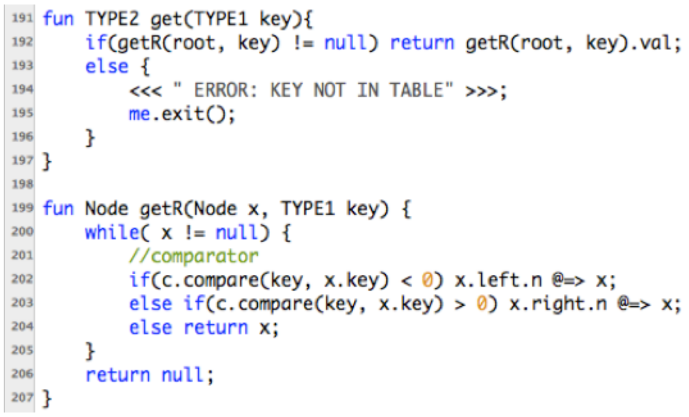
Because ChucK's limitations necessitate both independent storage and independent function calls for each type the collections can hold, we thought it would be more modular to completely separate out the implementations and then wrap them in a wrapper class. To do so, we created template files for each class, which were complete implementations of the class, but with the type key words for their elements were replaced with special keywords. We then used a Python script to generate the individual implementations. Finally, we used the Python script to generate the overloaded function definitions for the wrapper class.

The code in Figure 13 shows the template implementation of the get method for maps. The string “TYPE1” is the placeholder for the type-name of the key, and “TYPE2” is the placeholder for the type-name of the value. Because there is a difference in syntax when performing comparisons with the comparator vs with the built in comparison operators, each data structure has two different templates – one for built-in operator comparisons (Figure 13a) and one for comparator comparisons (Figure 13b).



```
fun TYPE2 get(TYPE1 key){
    if(getR(root, key) != null) return getR(root, key).val;
    else {
        <<< " ERROR: KEY NOT IN TABLE" >>>;
        me.exit();
    }
}

fun Node getR(Node x, TYPE1 key) {
    while( x != null) {
        //normal comparison
        if(key < x.key) x.left.n @=> x;
        else if(key > x.key) x.right.n @=> x;
        else return x;
    }
    return null;
}
```

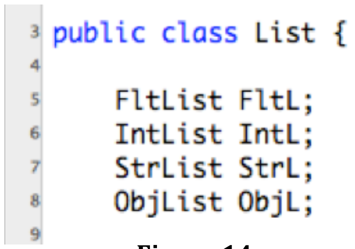
Figure 13a



```
fun TYPE2 get(TYPE1 key){
    if(getR(root, key) != null) return getR(root, key).val;
    else {
        <<< " ERROR: KEY NOT IN TABLE" >>>;
        me.exit();
    }
}

fun Node getR(Node x, TYPE1 key) {
    while( x != null) {
        //comparator
        if(c.compare(key, x.key) < 0) x.left.n @=> x;
        else if(c.compare(key, x.key) > 0) x.right.n @=> x;
        else return x;
    }
    return null;
}
```

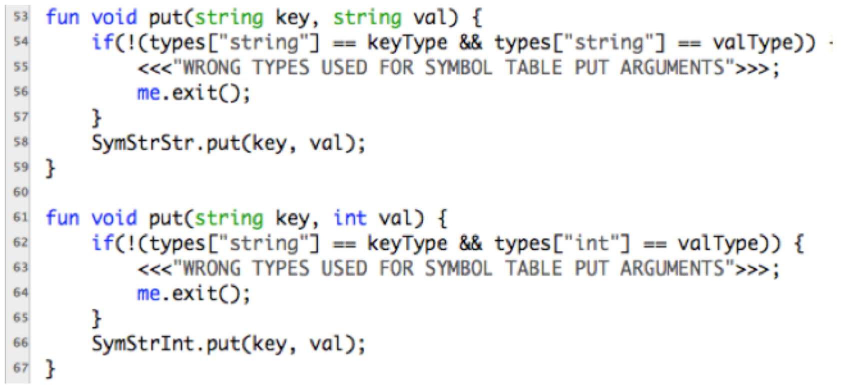
Figure 13b

Then, for each data structure, we created a wrapper class which instantiated an instance of each implementation (Figure 14) Function overloading allowed us to use one function name for a single kind of action over all of the different argument



```
public class List {

    FltList FltL;
    IntList IntL;
    StrList StrL;
    ObjList ObjL;
```

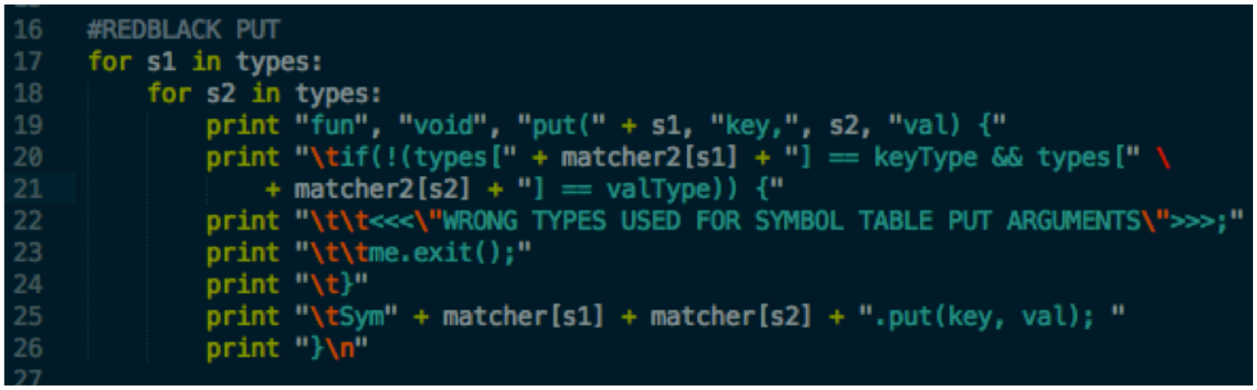
Figure 14



```
fun void put(string key, string val) {
    if(!(types["string"] == keyType && types["string"] == valType))
        <<<"WRONG TYPES USED FOR SYMBOL TABLE PUT ARGUMENTS">>>;
        me.exit();
    }
    SymStrStr.put(key, val);
}

fun void put(string key, int val) {
    if(!(types["string"] == keyType && types["int"] == valType)) {
        <<<"WRONG TYPES USED FOR SYMBOL TABLE PUT ARGUMENTS">>>;
        me.exit();
    }
    SymStrInt.put(key, val);
}
```

Figure 15

types. The code shown in Figure 15 shows a sample of the overloading in the put method for the Map. Due to the high amount of code replication required, most of the wrapper class was also generated using Python. The code in Figure 16 shows the Python code for generating the overloaded put functions.



```
#REDBLACK PUT
for s1 in types:
    for s2 in types:
        print "fun", "void", "put(" + s1, "key,", s2, "val) {"
        print "\tif(!(types[" + matcher2[s1] + "] == keyType && types[" \
            + matcher2[s2] + "] == valType)) {"
        print "\t\t<<<\"WRONG TYPES USED FOR SYMBOL TABLE PUT ARGUMENTS\">>>;"
        print "\t\tme.exit();"
        print "\t}"
        print "\tSym" + matcher[s1] + matcher[s2] + ".put(key, val); "
        print "}\n"
```

Figure 16

3.2 MUSICAL REPRESENTATION

Several classes were implemented that allowed for the representation of discrete notated music. They were Chord (representing chords), Note (representing single notes, a subclass of Chord), Voice (a list or time-series of chord types), and Part (a collection of Voice objects meant to be played simultaneously). Since many ChucK users may not be strong programmers, the focus on implementing this module was to keep the complexity of the API as low as possible. This naturally leads to a loss of detail in the musical representation, but we believe the current design simplicity outweighs the loss of musical detail. Competent programmers who desire more functionality could extend the classes presented here.

3.2.1 Chord

A Chord is implemented as a list of integers. The methods and public variables for the Chord class were:

- size() – returns the number of notes in the chord.
- noteList – this List object holds the integers that represent the notes of the chord. The integers represent the MIDI pitch values, where 60 corresponds to middle C and every integer up or down from there corresponds to a semi-tone up or down from middle C (thus, 59 is the B below middle C and 72 is one octave up from middle C).
- duration – this float variable represents the duration of the chord as how many whole notes long it is. For example, a duration of .25 is a quarter note, while a duration of 3 is 3 whole notes.
- (static) newChord([int[] notes], [duration whole]) – this function returns a new chord. Both arguments are optional. If an integer array is passed to the function, those integers become the notes of the chord. If it is not provided, the chord has no notes. If a duration is provided, that duration becomes the duration of the returned chord. If it is not provided, the duration of the returned chord defaults to .25.
- addPitch(int p) – adds the pitch to the chord, or does nothing if the pitch is already in the chord.
- removePitch() – removes the pitch from the chord, does nothing if the pitch is not in the chord.
- play([duration whole]) – if duration is given, this plays the chord for that duration. If it is not, this plays the chord for .5 seconds.

3.2.2 Note

A note is a subclass of chord. Its additional methods are:

- (static) midi(int n) – creates a note with the midi pitch value specififed
- (static) pitch(string s) – takes a string name of a note, and returns a note with that pitch. Strings include the note name and octave, such as "C#5", "B7", or "Ab4"

3.2.3 Voice

A voice is implemented as a list of chords. It is analogous to the staff of a single instrument in a musical score, a time series of all notes that instrument plays. Its methods and public variables are:

- setInstrument(int inst) – The integer argument specifies which StkInstrument sound to set the instrument to. Numbers 0-14 are all different preset instruments. If none is given, the default instrument sound (0 – mandolin) is used.
- play([duration whole], [int inst]) – the optional argument whole specifies the duration of the whole note from which the fractional durations of the chords are derived. If none is given, the default is set to 1 second. The optional integer argument is used to set the instrument voice, just as in setInstrument()
- setMidiInstrument(int device, int channel) – this function lets users play voices through MIDI software or hardware (such as GarageBand or physical synthesizers). The first argument specifies which MIDI device to open (calling "chuck --probe" in the terminal prints out the list of all MIDI devices detected by ChucK and the indices to open them with). The second argument specifies what MIDI channel to play the voice on.

- midiModeOn() – sets the voice to play audio using the MIDI device specified in SetMidiInstrument() rather than the built in StkInstruments.
- midiModeOff() – sets the voice to play audio using the built in rather than a MIDI device
- Since a Voice is a List of Chords, it implements wrappers around all of the List functions to allow for manipulation of the internal Chord list. The specific function calls are the same as the List calls, but without the type suffixes, and all items are Chords

3.2.4 Part

A Part is a List of Voices. If a Voice is a time series of notes for a single instrument, a part is the collection of the time series of multiple instruments playing at once (eg, a Guitar Voice, a Bass Voice, and a Piano Voice all playing together).

- play([duration d]) – the optional argument duration specifies, for all voices in the Part, the duration of the whole note from which the fractional durations of the chords are derived.
- Since a Voice is a List of Chords, it implements wrappers around all of the List functions to allow for manipulation of the internal Voice list. The specific function calls are the same as the List calls, but without the type suffixes, and all items are Voices

3.3 UTILITY CLASSES

Live coding presents several challenges for inexperienced programmers, among them are inter-shred synchronization (at a musical level, not a systems level) and data sharing. The classes in this module are minimal, extensible tools to simplify live coding, as well as the non-live development of code later used in live coding.

3.3.1 Gapper

Earlier in the paper we mentioned that long computations performed in ChucK could delay signal-processing calculations, leading to glitchy real-time output. The Gapper class provides a way to perform massive computations without holding on to the virtual processor for too long. The Gapper class implements a function stepAndCheck(), which yields the current shred after every K calls, for some integral K. The public methods of the Gapper class are:

- (static) newGapper(int lim) – this returns a new Gapper with an internal counter instantiated to 0 and a "count limit" of K. The count limit is the number of times stepAndCheck() can be called before it yields the thread.
- stepAndCheck() – this function increments the internal counter every time it is called. If the internal counter is equal to the "count limit," then the function sets the internal counter to 0 and yields the current thread.

A Gapper object should be instantiated outside of a large block of computation (e.g. before entering a loop or a recursive function), and stepAndCheck should be called at every iteration of the computation (eg at the beginning of the loop or recursive call). A Gapper should be used for large computations where the final result is not needed immediately or in real-time. For example, a Gapper could be used when running a long optimization algorithm to generate a melody that is saved to be played later rather immediately.

There can often be times when no active threads are using the processor (i.e, all threads are waiting on time to advance or for an event). The Gapper allows shreds running computations to fill in this time without holding on to the processor for too long and adversely affecting the real-time signal processing.

3.3.2 StartSync

ChucK allows shreds to be manually added to the Virtual Machine. These shreds can begin computing audio immediately, and thus, clicking slightly too slowly when adding a new shred could lead to music being unsynchronized. The StartSync class helps fix this problem by using events to synchronize timing between shreds. The public methods of the SartSync class are:

- (static) sendSignal(string msg) – sends the string as a signal to all shreds waiting on a signal from the StartSync class
- (static) waitForSignal(string msg) – waits until it receives a signal from the StartString class. If the signal from the class matches the function argument, the function returns, allowing the shred that called it to proceed. If not, it continues waiting until it receives a signal with a string that matches the function argument.

For example, StartSync could be used to create a simple metronome that could be shared between multiple shreds (by calling StartSync.sendSignal("metronome")). Shreds wishing to use that metronome would simply need to wait for the signal "metrnonome" wherever they wish to synchronize with the metronome.

3.3.3 Storage

Sharing data between shreds requires creating a public class with static data that is accessible by all shreds. This class does just that by building on the map data structure to create a simple string-keyed map to save values of integers, floats, strings, and arbitrary objects. The documentation uses the same root-suffix notation as in the Map class. The public methods are:

- put(string key, type value) – puts the (key, value) pair into storage
- get[Type](string key) – returns the (key, value) pair where the value has a type of [Type]
- contains[Type](string key) – checks if there exists a (key, value) pair where the value has a type of [Type]

4. EVALUATION AND FUTURE WORK

4.1. USER EVALUATION

We evaluated the usability of the interface by having users experiment with the library for a 90 minute session and then collecting feedback. None of the participants were familiar with ChucK, so the first 20 minutes were spent giving a quick tutorial on the ChucK language, and the scope of the test was limited to the Musical Representation and Data Structure modules. Users were asked to create a single voice melody, a multi-voice melody, and then to experiment by trying to create their own algorithmic music. Five users were tested; of them one had minimal programming experience (no classes in CS or significant projects attempted), two had moderate programming experience (had taken several CS courses but no industry experience), and two had significant programming experience (CS majors with industry experience). Two of the users (one non-experienced and one significantly experienced in CS) were highly experienced musicians. The other three users had basic musical experience (instrument lessons in grade school).

All users found the APIs for the two modules intuitive and easy to use. However, some found the API to be a bit verbose, and said it would be worth a slight increase in complexity in order to reduce the amount of code needed to create chords and melodies. The users provided several useful suggestions to the API in this regard. In particular, users suggested that the ability to define note sequences and rhythmic sequences separately and then “zip” them together into a Voice object would be useful.

4.2. FUTURE WORK

The first steps for future work will be to create a library of examples of how these different classes can be used. Much of the power and expressiveness of musical live coding comes from being able to change even the deepest parameters of the music generating code on the fly, and thus too much encapsulation can limit the expressiveness of the medium. Instead, acquainting users with “best pratices” or “techniques” allows them to use the design patterns and strategies of others without stifling the freedom of live coding. Effort will also be made to make the library more terse to facilitate faster coding.

A related project by the author, Variator, attempts to create an algorithmic music library for Python by building “algorithmic primitives” – simple transformative and generative functions for music that could quickly be combined into much more powerful algorithms [5]. Porting these functions over to ChucK is another branch of future work that will be soon undertaken.

5. CONCLUSION

ChucK is a powerful tool for music composition and performance. We hope that the development and sharing of this library will show users the ease of ChucK and grow the community around it. The code is hosted at https://github.com/AvneeshSarwate/ChucKLib

6. References

[1] Ariza, C., Cuthbert, M. 2000. "Music21: A toolkit for computer-aided musicology and symbolic music data". *Proc. Conf. International Society for Music Information Retrieval (ISMIR) 2000.*

[2] Cook, P., Scavone, G. 1999. "The Synthesis Toolkit." *Proc. International Computer Music Conference (ICMC) 1999.* Beijing, China.

[3] Hudak, Paul. *The Haskell School of Music*. New Haven, CT: Yale University Department of Computer Science, 2014.

[4] McCartney, James. "Rethinking the Computer Music Language: SuperCollider." *Computer Music Journal* 26: 61–68.

[5] Sarwate, A., and R. Fiebrink. 2013. "Variator: A creativity support tool for music composition."*Proceedings of New Interfaces for Musical Expression (NIME)*, Daejeon, South Korea, May 27–30, 2013. (poster with paper in proceedings)

[6] Wang, Ge. "A history of programming and music." In*The Cambridge companion to electronic music*. Cambridge: Cambridge University Press, 2007. .

[7] Wang, Ge. 2008. The ChucK audio programming language: A strongly-timed and on-the-fly environ/mentality. PhD thesis, Princeton University, Princeton, NJ, USA.

[8] Wright, M. 1997. "Open Sound Control-A New Protocol for Communicating with Sound Synthesizers," *Proceedings of the 1997 International Computer Music Conference,* Thessaloniki, Greece, 1997.

Appendix I: Chord Progression

```
//creates an array of chords
Chord c[9];

//creates the individual chords in the progression
Chord.newChord([60, 63, 67], .125) @=> c[0];
Chord.newChord([63, 67, 70], .25) @=> c[1];
Chord.newChord([60, 63, 67], .25) @=> c[2];
Chord.newChord([60, 63, 70], .25) @=> c[3];
Chord.newChord([63, 67, 72], .25) @=> c[4];
Chord.newChord([60, 65, 74], .25) @=> c[5];
Chord.newChord([60, 63, 67], .125) @=> c[6];
Chord.newChord([58, 62, 65], .25) @=> c[7];
Chord.newChord([60, 63, 67], .25) @=> c[8];

Voice v;

//adds chords to the voice
for(0 @=> int i; i < 9; i++){
    v.push(c[i]);
}

//plays the chord progression over and over
while(true) {

    //signals the start of the chord progression
    StartSync.sendStartSignal("chords");
    v.play(5, 1.7::second);
}
```

Appendix I: Random Melody

```
//defines the C minor pentatonic scale
[55, 58, 60, 63, 65, 67, 70, 72, 75, 79] @=> int cMinPent[];

//defines a rhythm
[.25, .125, .125, .125, .125, .125, .375, .125, .125, .25, .25] @=> float
rhy1[];

//defines another rhythm
[.375, .375, .375, .375, .25, .25] @=> float rhy2[];

//a variable to store the index of the wanted pitch
int pitchInd;

//a variable to store the wanted rhythm
float rh[];

Voice v;

//waits till the chord progression is back to the beginning
StartSync.waitOnSignal("chords");

while(true) {

    //randomly picks a rhythm
    if(Math.randomf() > .5) rhy1 @=> rh;
    else rhy2 @=> rh;

    for(0 @=> int i; i < rh.size(); i++) {

        //performs a random walk to select the pitches of the melody
        pitchInd + Math.random2(-3, 3) => pitchInd;
        Math.abs(pitchInd % cMinPent.size()) @=> pitchInd;

        //adds the Notes to the voice
        v.push(Note.midi(cMinPent[pitchInd], rh[i]));
    }

    //plays the voice
    v.play(5, 1.7::second);

    //clears the voice to get ready for the next random melody
    v.clear();
}
```