A *JAVA* FRAMEWORK AND DOMAIN-SPECIFIC LANGUAGE FOR
MANIPULATING SOUNDS

By

LUIS F. VIEIRA DAMIANI

A THESIS PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

UNIVERSITY OF FLORIDA

2018

For my wife, Maria Inês.

ACKNOWLEDGMENTS

Acknowledgments go here.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Abstract of Thesis Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Master of Science

A *JAVA* FRAMEWORK AND DOMAIN-SPECIFIC LANGUAGE FOR
MANIPULATING SOUNDS

By

Luis F. Vieira Damiani

August 2018

Chair: Dr. Beverly Sanders
Major: Computer Science

Abstract goes here.

CHAPTER 1
LITERATURE REVIEW

Arguably the first notable attempt to design a programming language with an explicit intent of processing sounds and making music was that of *Music I*, created in 1957 by Max Mathews. The language was indented to run on an IBM 704 computer, located at the IBM headquarters in New York City. The programs created there were recorded on digital magnetic tape, then converted to analog at Bell Labs, where Mathews spent most of his career as an electrical engineer. *Music I* was capable of generating a single waveform, namely a triangle, as well as assigning duration, pitch, amplitude, and the same value for decay and release time. *Music II* followed a year later, taking advantage of the much more efficient IBM 7094 to produce up to four independent voices chosen from 16 waveforms. With *Music III*, Mathews introduced in 1960 the concept of a *unit generator*, which consisted of small building blocks of software that allowed composers to make use of the language with a lot less effort and required background. In 1963, *Music IV* introduced the use of macros, which had just been invented, although the programming was still done in assembly language, hence all implementations of the program remained machine-dependent. With the increasing popularity of Fortran, Mathews designed *Music V* with the intent of making it machine-independent, at least in part, since the unit generators' inner loops were still programmed in machine language. The reason for that is the burden these loops imposed on the computer [1, 15-17].

1.1 Software Synthesis Languages

Since Mathews' early work, much progress has been made, and a myriad of new programming languages that support sound processing, as well as domain-specific languages whose sole purpose is to process sounds or musical events, have surfaced. In [2], we see an attempt to classify these languages according to the specific aspect of sound processing they perform best. The first broad category described is that of *software synthesis languages*, which compute samples in non-real-time, and are implemented by use of a

text editor with a general purpose computer. The *Music N* family of languages consist of software synthesis languages. A characteristic common to all software synthesis languages is that if a toolkit approach to sound synthesis, whereby using the toolkit is straightforward, however customizing it to fulfill particular needs often require knowledge of the programming language in which the toolkit was implemented. This approach provides great flexibility, but at the expense of a much steeper learning curve. Another aspect of software synthesis languages is that they can support an arbitrary number of voices, and the time complexity of the algorithms used only influences the processing time, not the ability to process sound at all, as we see with real-time implementations. As a result of being non-real-time, software synthesis languages usually lack controls that are gestural in nature. Yet, software synthesis languages are capable of processing sounds with a very fine numerical detail, although this usually translates to more detailed, hence verbose code. Software synthesis languages, or non-real-time features of a more general-purpose language, are sometimes required to realize specific musical ideas and sound-processing applications that are impossible to realize in real time [2, 783-787].

Within the category of software synthesis languages, we can further classify those that are *unit generator languages*. This is exactly the paradigm originally introduced by *Music III*. In them, we usually have a separation between an orchestra section, and a score section, often given by different files and sub-languages. A unit generator is more often than not a built-in feature of the language. Unit generators can generate or transform buffers of audio data, as well as deal with how the language interacts with the hardware, that is, provide sound input, output, or print statements to the console. Even though one can usually define unit generators in terms of the language itself, the common practice is to define them as part of the language implementation itself. Another characteristic of unit generators is that they are designed to take as input arguments the outputs of other unit generators, thus creating a signal flow. This is implemented by keeping data arrays in memory which are shared by more than one UG procedure by reference.

The score sub-language usually consists of a series of statements that call the routines defined by the orchestra sub-language in sequential order, often without making use of control statements. Another important aspect of the score sub-language is that it defines function lookup tables, which are mainly used to generate waveforms and envelopes. When *Music N* languages became machine-independent, function generating routines remained machine-specific for a period of time, due to performance concerns. On the other hand, the orchestra sub-language is where the signal processing routines are defined. These routines are usually called instruments, and basically consist of new scopes of code where built-in functions are dove-tailed, ultimately to a unit generator that outputs sound or a sound file [2, 787-794].

The compilation process in *Music N* languages consists usually of three passes. The first pass is a preprocessor, which optimizes the score that will be fed into the subsequent passes. The second pass simply sorts all function and instrument statements into chronological order. The third pass then executes each statement in order, either by filling up tables, or by calling the instrument routines defined in the orchestra. The third pass used to be the performance bottleneck in these language implementations, and during the transition between assembly and Fortran implementations, these were the parts that remained machine-specific. Initially, the output of the third pass consisted of a sound file, but eventually this part of the compilation process was adapted to generate real-time output. At that point, defining specific times for computing function tables became somewhat irrelevant.

In some software synthesis languages, the compiler offers hooks in the first two passes so that users can define their own sound-processing subroutines. In any cases, these extensions to the language were given in an altogether different language. With *Common Lisp Music*, for example, one could define the data structures and control flow in terms of Lisp itself, whereas *MUS10* supported the same features by accepting Algol code. In *Csound*, one can still define control statements in the score using Python. Until *Music*

11

IV and its derivatives, compilation was sample-oriented. As an optimization, *Music V* introduced the idea of computing samples in blocks, where audio samples maintained their time resolution, but control statements could be computed only once per block. Of course, if the block size is one, than we compute control values for each sample, as in the sample-oriented paradigm. Instead of defining a block size, however, one defines a control rate, which is simply the sampling rate times the reciprocal of the block size. Hence a control rate that equals the sampling rate would indicate a block size of one. With *Cmusic*, for instance, we specify the block size directly, a notion that is consistent with the current practice of specifying a vector size in real-time implementations. The idea of determining events in the language that could be computed at different rates required some sort of type declaration. In *Csound*, these are given by naming conventions: variables whose names start with the character 'a' are audio-rate variables, 'k' means control rate, and 'i'-variables values are computed only once per statement. *Csound* also utilizes naming conventions to determine scopes, with the character 'g' indicating whether a variable is global [2, 799-802].

1.2   Real-Time Synthesis Control Languages

Some of the very first notable attempts to control the real-time synthesis hardware were made at the *Institut de Recherche et Coordination Acoustique/Musique* in the late seventies. Many of these early attempts made use of programming languages to drive the sound synthesis being carried out by a dedicated DSP. At first, most implementations relied on the concept of a *fixed-function* hardware, which required significantly simpler software implementations, as the latter served mostly to control a circuit that had an immutable design and function. An example of such fixed-function implementations would be an early frequency-modulation synthesized, which contained a dedicated DSP for FM-synthesis, and whose software implementation would only go as far as controlling the parameters thereof. Often, the software would control a chain of interconnected dedicated DSP's, which would in turn produce envelopes, filters, and oscillators. The

12

idea of controlling parameters through software, while delegating all signal processing to hardware, soon expanded beyond the control of synthesis parameters, and into the sequencing of musical events, like in the New England Digital Synclavier. Gradually, these commercial products began to offer the possibility of changing how exactly this components were interconnected, what is called a *variable-function* DSP hardware. Interconnecting these components through software became commonly called *patching*, as an analogy to analog synthesizers. The idea of patching brought more flexibility, but imposed a steeper learning curve to musicians. Eventually, these dedicated DSP's were substituted by general-purpose computers, wherein the entire chain of signal processing would be accomplished via software [2, 802-804].

Commonly in a fixed-function implementation there is some sort of front panel with a small LCD, along with buttons and knobs to manage user input. In the case of a keyboard instrument, there is naturally a keyboard to manage this interaction, as well. The purpose of the embedded software is then to communicate user input to an embedded system which contains a microprocessor and does the actual audio signal processing, memory management, and audio input/output. All software is installed in some read-only memory, including the operating system. With the creation of the *Musical Instrument Digital Interface* standard in 1983, which was promptly absorbed my most commercial brands, the issue of controlling sound synthesis hardware transcended the interaction with keys, buttons, and sliders, and became a matter of software programming, as one could easily communicate with dedicated hardware, by means of a serial interface, MIDI messages containing discrete note data, continuous controller messages, discrete program change messages, as well as system-exclusive messages. As a trend, many MIDI libraries were written at the time for general-purpose programming languages such as APL, Basic, C, Pascal, Hypertalk, Forth, and Lisp. In addition, most descendants of the *Music N* family of languages began to also support MIDI messages as a way to control dedicated hardware [2, 804-805].

The implementation of a software application to control variable-function DSP hardware is no mundane task, as it requires knowledge of digital signal processing, in addition to programming in a relatively low level language. Dealing with issues of performance, memory management, let alone the mathematics required to process buffers of audio samples, often imposes an unsurmountable burden to musicians. Many solutions were invented in order to work around this difficulties, including the use of graphic elements and controllers, but ultimately it was the concept of a unit generator, borrowed from software synthesis languages, that most influenced the creation of higher-level abstractions that were more suitable for musicians. This is notably the case of the *4CED* language, which was developed at IRCAM in 1980, and owed greatly to *Music IV* and *Music V*. The resemblance extended as far as to comprise a separate orchestra sub-language for patching unit generators, a score sub-language, and a third command sub-language for controlling effects in real-time, as well as to link both orchestra and score to external input devices such as buttons and potentiometers. The hardware these languages drove was IRCAM's 4C synthesizer. The result of nearly a decade of research at IRCAM culminated in *Max*, a visual programming language that remains to this day one of the most important real-time tools for musicians. *Max*, which will later be discussed in more detail, eventually transcended its hardware DSP and implemented itself in C the sound-generating routines. But that was not until the 2000's, ten years after it became a commercial software application, independent of IRCAM [2, 805-806].

Example 1.2.1. *[2, 809] Music 1000 is a descendant of the* Music N *family of languages that was designed to drive the Digital Music Systems DMX1000 signal processing computer, in which we can clearly observe the unit-generator concept in action. In Listing 1.1, a fnctn statement assigns to variable func1 an array of 512 samples using a fourier series of exactly one harmonically-related sine, whose (trivial) sum is normal-ized. The amplitude of 1000 is then meaningless, but a required argument. In fact, func1 takes a variable number of arguments, where for each harmonic partial, the user specifies a relative amplitude.*

The block that follows defines an instrument, in which the unit generator oscil takes as arguments the output of three other unit generators, which are respectively the wavetable previously computed, as well as amplitude and frequency parameters, whose values are in turn captured by two knobs attached to the machine. The knobs produce values between 0 and 1, and the subsequent arguments to kscale are scaling parameters. Finally, out is a unit generator that connects the output of oscil to the digital-to-analog converter.

Listing 1.1. *Music 1000* algorithm that produces a sine wave.

```
fnctn func1 , 512 , fourier , normal , 1 , 1000          1
instr 1                                                  2
    kscale amp, knob1, 0, 10000                          3
    kscale freq , knob2, 20, 2000                        4
    oscil x8, #func1 , amp, freq                         5
    out x8                                               6
endin                                                    7
```

1.3   Music Composition Languages

Between the 1960's and the 1990's, many programming languages were devised to aid music composition. As a noticeable trend, one can define two categories among those languages, namely those that are *score input languages*, and those that are *procedural languages*. The main difference between the two categories is that, in the former, some representation of a musical composition is already at hand, hence score input languages provide a way to encode that information. This could be a score, a MIDI note list, or even some graphical representation of music. In the latter category, the language provides, or helps define procedures that are used to generate musical material, a practice that is often called *algorithmic* music composition. One outstanding characteristic of score input languages is how verbose and complex they can become, depending on the musical material they are trying to represent. This difficulty influenced the devising of many alternatives to textual programming languages, such as the use of scanners in the late 1990's by Neuratron's *PhotoScore*, an implementation which was predicted by composer

15

Milton Babbitt as early as in 1965. Before the advent of MIDI, however, programming languages were indeed the user interface technology of choice, or lack thereof, to design applications meant for analyzing, synthesizing, and printing musical scores. With the widespread adoption of the MIDI standard in the mid-1980's, whereby one can input note events by performing on a MIDI instrument, combined with the advancements in graphical user interfaces of the mid-1990's, the creation and maintenance of score input languages has faced a huge decline. What is even worse, the paradigm of a musical score is itself inadequate for computer music synthesis, in that a score is more often than not a very incomplete representation of a musical piece, often omitting a great deal of information. It is the job of a musical performer to provide that missing information. In this sense, *procedural languages* are much better suited for computer performance, but that comes at the cost of replacing the score paradigm altogether [2, 811-813].

In 2018, a few *score input languages* remain, despite the vast predominance of graphical user interfaces as a means to input notes to a score. *MusiXTex* is a surviving example that compiles to LATEX, which in turn compiles to PDF documents. It was created in 1991 by Daniel Taupin. The language has such unwieldy syntax, that often a preprocessor is required for more complex scores. One famous such processors is *PMX*, a *Fortran* tool written by Don Simons in the late 1990's. Another was *MPP*, which stands for MusiXTex Preprocessor, created by Han-Wen Nienhuys and Jan Nieuwenhuizen in 1996, and which eventually became *LilyPond*, arguably the most complete surviving score input language today. *LilyPond* has a much simpler syntax than that of *MusiXTex*, however not nearly as simple as *ABC* music notation, a language that much resembles *Musica* and which is traditionally used in music education contexts. A package written by Guido Gonzato is available in LATEXwhich can produce simple scores in *ABC* notation. Its simplicity comes, however, at the expense of incompleteness. Finally, it is worthwhile to mention a music-notation specific standard that has emerged in the mid-2000's, namely the *MusicXML* standard. Heavily influenced by the industry, it was initially meant as an

object model to translate scores between commercial applications where the score input method was primarily graphical, and whose underlying implementation was naturally object-oriented. *MusicXML* is extremely verbose, and borderline human-readable. It is, however, very complete, to the point of dictating what features an object-oriented implementation should comprise in order to be aligned with the industry standards. In recent years, many rumors have surfaced to make *MusicXML* an Internet standard, such as that of *Scalable Vector Graphics*, however nothing concrete has been established.

Listing 1.2. *Musica* algorithm that creates a simple melodic line.

```
4'AGAG / 4.A8G2E / 4DDFD / 2ED                                              1
```

Example 1.3.1. *[2, 812] Musica was developed at the Centro di Sonologia Computazionale in Padua, Italy, and is particularly interesting in its interpreter compiles programs into* Music V *note statements. In the snippet above, all numbers indicate note duration, that is, 4 is a quarter-note, 8 is an eighth-note, and 2 is half-note, with dots indicating dotted durations. The letters indicate pitch, and the apostrophe indicates octave such that 'A = 440Hz, and "A = 880Hz. Finally, the slash indicates a measure. The code below, on the other hand, shows an example of the very same musical material expressed in* MusiXTex. *One can immediately notice the difference in implementation by the sheer amount of code required to express basically the same symbols. In the code above, many commands, despite verbose, are quite self-explanatory. Some others, however, are not. The \qu command means a quarter-note with a stem pointing upward, whereas the \Notes command actually means how notes should be spaced. The more capital letters, the more spacing between the notes, that is, \NOTes is more spaced out than \NOtes. Finally, in addition to supporting the same apostrophes as* Musica *for defining octave,* MusiXTex *also supports other letters, as well as capitalizations thereof. In the example above, we have* h = 440Hz, *whereas* a = 220Hz.

Listing 1.3. *MusiXTex* algorithm whose output is shown in Fig. 1-1.

17

Figure 1-1. Typesetting music with *MusiXTex*.


```
\begin{music}                                              1
  \generalmeter{\meterfrac44}                              2
  \startextract                                            3
  \Notes \qu{h g h g} \en \bar                             4
  \Notes \qup{h} \cu{g} \hu{e} \en \bar                    5
  \Notes \qu{d d f d} \en \bar                             6
  \Notes \hu{e d} \en                                      7
  \endextract                                              8
\end{music}                                                9
```

One of the greatest contributions of *procedural composition languages* to the field of music composition is arguably the concept of algorithmic composition, in particular when the realization of the musical algorithm is not restricted to human performers. In such circumstances, the composer is capable of exploring the full extent of musical ideas a computer can reproduce. Naturally, the composer must often trade off the ability to represent those ideas via a score, in which case the algorithm itself becomes the representation. If, on one hand, reading music from an algorithm is somewhat unfamiliar to most musicians, the representation is nonetheless formal, concise, and consistent. Furthermore, it lends itself to be analyzable a much larger apparatus of analytical techniques and visualization tools, hence is equally beneficial a representation to music theorists. A machine is capable of representing all sorts of timbres, metrics, and tunings that humans cannot, but it needs to be told exactly what to do. Unlike a human performer, who interprets the composer's intents, a purely electro-acoustic algorithmic composition must address a human audience without relying on a middle-man. Hence the programming language of choice becomes an invaluable tool for the composer. In addition to all that, another important aspect of

algorithmic composition is how it is capable of transforming the decision-making process of a composer. Instead of making firm choices at the onset of a musical idea, a composer can *prototype* many possible outcomes of that idea before deciding. One example is assigning random numbers to certain parameters, this postponing the decision making until more structure has been added to the composition. In fact, this postponing may be final, thus an algorithmic composition may be situated within a whole spectrum of determinism. A fully stochastic piece fixes no parameter, as opposed to a fully deterministic composition. Some of the notable techniques of electro-acoustic music composition also include spatialization, where the emission of sounds through speakers positioned at specific spatial locations constitutes a major musical dimension in a composition; spectralism, where the spectral content of sounds are manipulated by an algorithm; processing sound sources in real time, very often capturing a live performance on stage; and sonification of data nor originally conceived as sound [2, 813].

1.4   Libraries

Many domain-specific languages that deal with sound synthesis, processing, and music composition are *extensible* in the sense that they provide a hook for code written in the implementation language to be executed in the context of the DSL. This feature can render a DLS a lot more flexible, at the expense of annulling the very purpose of the DSL, which can be a good trade-off if the latter's implementation is incomplete. An early example would be *Music V*, which could accept user-written subroutines in *Fortran*. *Music 4C* had its instruments written in *C*, and *Cscore* was a *C*-embedding of *Cmusic*. Other examples are *MPL*, which could accept routines written in *APL*, and *Pla*, whose first version was embedded in *Sail*, and whose second version was embedded in *Lisp*. In the particular case of *Lisp*, embeddings include *MIDI-LISP*, *FORMES*, *Esquisse*, *Lisp Kernel*, *Common Music*, *Symbolic Composer*, *Flavors Band*, and *Canon*. *Music Kit* was embedded in the object-oriented *Objective-C* [2, 814].

Besides domain-specific languages, a variety of libraries exist for general-purpose programming languages that also deal with aspects of sound synthesis, processing, and music composition. In languages like *Haskell*, these libraries may carry such syntactical weight, with so many specifically-defined symbols, that they do in fact resemble more a DSL that a library, even though such terming would not be technically correct.

CHAPTER 2
THEORETICAL FRAMEWORK

In this chapter, we present and discuss the tools and methodology utilized to build *Scandal*. We start discussing how to produce sound in the Java Runtime Environment, and particularly how it deals with real-time audio. We next give a formal presentation on how the domain-specific language was designed, and attempt to classify it among other programming languages and music-DSL's. Finally, we address the methodology involved in building its compiler, appending to the latter a short discussion on how a general-user interface was designed to provide an integrated development environment experience for the end user, as well as alternative command-line methods for compiling and running *Scandal* programs.

2.1   How *Scandal* Manages Sound

The JRE System Library provides a very convenient package of classes that handle the recording and reproduction of real-time audio, namely the javax.sound.sampled package. In it, we find two classes, TargetDataLine and SourceDataLine, that deal respectively with capturing audio data from the system's resources, and playing back buffers of audio data owned by the application. An instance of SourceDataLine provides a write method that takes three arguments: an array of bytes to written to a Mixer object, an integer offset, and an integer length. In *Scandal*, we do not specify a Mixer object, hence we make use of one provided by the System. There are two main aspects of the write method that need to be addressed. Firstly, it blocks the thread in which it lives until the given array of bytes has been written, from offset to length, to the Mixer its SourceDataLine contains; secondly, if nothing is done, it returns as it has no more data to write. In order to have real-time audio, one then needs to be constantly feeding this write method with audio samples, for as long as one wants continuous sound output, even if these buffers of audio samples contain only zeros, i.e., silence. It immediately follows that one must specify exactly how many samples are sent at a time, naturally with consequences to the system's performance.

This parameter is commonly referred in the industry as the *vector size*. The trade-off is measured in terms of latency: a large a vector size helps slower systems perform better, or can allow more complex processing, or even increase polyphony. Latency, however, is bad for any live application, including the generation of MIDI notes, and the recording of live sound from a microphone. A good, low-compromise vector size is usually set to 512 samples, and normally these sizes will be powers of two. In order to specify the preferred vector size, as well as many other environment settings, *Scandal* refers to a static class named Settings, which contains a static property Settings.vectorSize.

The aforementioned two characteristics of the write method within a SourceDataLine are managed by *Scandal* by the class AudioFlow. In order to prevent write from prematurely returning, an instance of AudioFlow contains a boolean property named running, which is set to true for as long as real-time audio is desired. The fact that write blocks its thread, however, is managed by any class that contains itself an instance of AudioFlow as a property. The latter are in *Scandal* the implementors of the RealTimePerformer interface, which is a contract that contains four abstract methods: startFlow, stopFlow, getVector, and processMasterEffects. The role of the startFlow is to merely embed an AudioFlow within a new Thread object and start this new thread. This guarantees the thread that manages audio is different than the main Application thread, hence resolving the thread-blocking issue. Once a new Thread is started in *Java*, however, one cannot in general interrupt it. In order to stop the audio thread, we set the property running inside an AudioFlow to false via the stopFlow method, which causes the write method inside the AudioFlow to return. Hence one cannot resume an audio process in *Scandal* at this point, even though doing so is perfectly possible in *Java*. The reason for that is not that of a design choice, but rather the fact that the domain-specific language is at its infancy, and many important features that go beyond a proof-of-concept are yet to be implemented. The getBuffer method is called by the AudioFlow every time it needs to write another vector of audio samples. It

is the responsibility then of any RealTimePerformer to timely compute the next Settings .vectorSize samples of audio data. Finally, the processMasterEffects routine is called from within getVector to further process the buffer of audio samples. This is usually done while the samples are still represented as floats, hence before converting them to raw bytes.

The constructor of an AudioFlow takes, in addition to a reference to a RealTimePerformer , a reference to an AudioFormat object. The latter is part of the javax.sound.sampled package and is how we ask the AudioSystem for a SourceDataLine. Instead of constructing AudioFormat objects, however, the Settings class contains static members Settings.mono and Settings.stereo that are instances of AudioFormat defining a mono and stereo format, respectively. In addition to a channel count argument, AudioFormat instances are constructed by specifying a sampling rate, and a bit depth (word length) for audio samples. Those are, too, static properties in the Settings class, namely Settings.samplingRate and Settings.bitDepth. Listing 2.1 gives the specifics of maintaining a SourceDataLine open inside an instance of AudioFlow . The latter implements, in turn, the Runnable interface, hence needs to override a run method. Inside this run method, we call the private play subroutine that is given below:

Listing 2.1. Writing buffers of audio data inside the play subroutine.

```java
private void play() throws Exception {                                              1
    SourceDataLine sourceDataLine = AudioSystem.getSourceDataLine(format);          2
    sourceDataLine.open(format, Settings.vectorSize * Settings.bitDepth / 8);       3
    sourceDataLine.start();                                                         4
    while (running) {                                                               5
        ByteBuffer buffer = performer.getVector();                                  6
        sourceDataLine.write(buffer.array(), 0, buffer.position());                 7
    }                                                                               8
    sourceDataLine.stop();                                                          9
    sourceDataLine.close();                                                         10
}                                                                                   11
```

Inside the play subroutine, we acquire a SourceDataLine object from the AudioSystem with the specific format that the RealTimePerformer passed while constructing this AudioFlow. In order to open the data line, we need specify a buffer size in bytes, hence we multiply the vector size by the word length in bits, divided by eight, as there are eight bits per byte. We then start the data line and keep writing to it for as long as the RealTimePerformer maintains the running property inside its AudioFlow set to true. At each call to write, we ask the performer for a new vector. Filling the vector causes its position to advance until its length, hence the position method inside the ByteBuffer class will in fact return the length value we desire. The rest of the play subroutine simply releases resources before returning, at which point the audio thread is destroyed.

2.2   How *Scandal* Handles MIDI

2.3   The Structure of the Compiler

In a broad perspective, the compilation process of *Scandal*'s DSL has the following steps:

1. A path to a *.scandal* file is passed as an argument to the constructor of the compiler and a linker subroutine is called, in order to resolve any dependencies;

2. The code is passed through a scanner, which removes white space and comments while converting strings of characters to tokens. Any illegal symbol will cause the scanner to throw an error, interrupting the compilation process;

3. The tokens are parsed and converted into an abstract syntax tree, during which many tokens are discarded. If the order of the tokes does not match any of the constructs that *Scandal* understands, the parser throws an error and interrupts the compilation process;

4. The root of the AST begins the process of *decorating* the tree, in which name references are resolved, types are checked, and variable slot numbers are assigned, whenever applicable. To keep track of names, a LeBlanc-Cook symbol table is kept.

If types do not match, or names cannot be referenced, the offending node in the AST throws an error, aborting the compilation;

5. Again starting from the root of the AST, each node generates its corresponding bytecode, making use of the org.objectweb.asm library as a facilitator. For any node that is a subroutine, its body is added following its declaration. No errors are thrown in this phase, and the root node returns an array of bytes containing the program's instructions in *Java* bytecode format;

6. Every *Scandal* program implements the Runnable interface. After the compiler receives the program's bytecode, it dynamically loads that bytecode as a *Java* class on the current (main) thread, causing the *Scandal* program to be executed.

2.3.1 The Linking Process

The main entry point to the compilation process is given by the Compiler class, whose constructor requires a path to a *.scandal* file. This class contains a link routine that is called before each compilation to resolve dependencies, and which is given in Listing 2.2 below. The Compiler class has a property named imports, which is an array of paths to other *.scandal* files upon which the program at hand depends. It also holds a path property, which was passed to its constructor, and which is used as an argument to link's first call. A *Scandal* program may have at its outermost scope import statements, which take a single string as a parameter, which in turn represents a path to a *.scandal* file in the file system. Any code contained in the file may depend on this imported path's content. Similarly, the imported path's content may depend itself on other imports, and so on, provided there is no circularity, that is, nothing imports something that depends on itself. We may regard then the linking process as a directed graph, in which arrows point toward dependencies. Since we do not allow cycles, this is a directed acyclic graph. It may very well be the case that more than one import depend on a particular file, in which case we certainly do not want to import that code twice. In order to import each dependency exactly once in an order that will satisfy every node of the DAG that points to it, we need

to somehow sort the array of imports. It is easy to see that this is no different than the problem of donning garments, in which one must have her socks on before putting her shoes, and where some items may call for no particular order, such as a watch [3, 612]. The solution for this problem is to topologically sort the array of imports. Since it is a DAG, however, that is very easily accomplished by a depth-first search of the graph, which is exactly what Listing 2.2 accomplishes recursively.

Listing 2.2. The linking process of a *Scandal* program.

```
private void link(String inPath) throws Exception {              1
    if (imports.contains(inPath)) return;                        2
    Program program = getProgram(getCode(inPath));               3
    for (Node node : program.nodes)                              4
        if (node instanceof ImportStatement)                     5
            link(((ImportStatement) node).expression.firstToken.text);  6
    imports.add(inPath);                                         7
}                                                                8
```

The if-statement in line 2 of the link routine deals with the base case of the recursion, namely the case in which we have already discovered that vertex. If we are seeing a vertex for the first time, line 3 converts the code into an AST, so we can check for any import statements therein. That is, in turn, accomplished by the for-loop in line 4, which checks each node in the AST's outermost scope for import statements. For each one it finds, line 6 recursively calls the link routine with the path extracted from that import statement. Since any code upon which we might depend needs to appear *before* our own, the first vertex that is finished needs to go in front of the list, and so on. To be precise, this is a *reverse* topological order. If the chain of imports given by the user contains a cycle, then no topological order exists, and the *Scandal* program will throw a runtime error. This is not ideal, and future versions of *Scandal* will throw a compilation error instead. In order to do so, however, more structure needs to be added to the compiler, so that we may check for backward edges in the linking process, although this feature remains unimplemented.

2.3.2 The Scanning Process

The design of the entire complier takes full advantage of *Java*'s object-oriented paradigm. In order to convert strings of characters from the input file into tokens, we first define a particular *type* of token for each individual construct in the DSL. This is accomplished by the Token class, which contains a static enumeration Kind, that in turn defines a type for each string of characters the DSL understands. The constructor of Token takes a Token.Kind as input, and each instance of Token contains, in addition, a text property, which holds the particular string of characters for that token's kind, as well as other properties that are convenient when throwing errors, namely that token's line number, position within the input array of characters, position within the line, and length. The Token class also contains methods for converting strings into numbers, as well as convenience methods for determining whether the kind of a particular instance of Token belongs to a particular *family* of tokens, i.e., whether a token is an arithmetic operator, or whether it is a comparison operator, and so on.

What the Scanner class accomplishes is the conversion of an array of characters into an array of instances of Token. The mechanism is conceptually very simple: we scan the input array from left to right and, whenever we see a string of characters that matches one of the DSL's constructs, we instantiate a new Token and add it to the array of tokens we hold, in order. In the process, we skip any white space found. These can be tab characters, space characters, new lines, hence *Scandal*, unlike *Python* or *Make*, makes no syntactical use of line breaks or indentation. The only role white spaces play in a *Scandal* program is that of improved readability. *Scandal* also supports two kinds of comments: single-line, which are preceded by two forward slashes, and multi-line, where a slash *immediately* followed by a star character initiates the comment, and a start immediately followed by a slash terminates it. Unlike *Java* or *Swift*, comments are not processed as documentation, and are thus completely discarded. Their only purpose is to document the *.scandal* file in which they are contained. String literals in *Scandal* are declared by

enclosing the text between quotes, and single apostrophe are neither allowed, nor in the language's alphabet anywhere. Besides token kinds that bear syntactical relevance, there is an additional *end-of-file* kind that exists for convenience, and is placed at the end of the token array right before the scan method returns. Checking for illegal characters, or combinations thereof, such as a name that begins with a number, for example, is all the checking the Scanner class does. All syntactical checking is delegated to the parsing stage of compilation.

2.3.3   The Parsing Process

The main purpose of the parsing stage is to convert the concrete syntax of a *Scandal* program into an abstract syntax tree, where constructs are hierarchically embedded in one another. An instance of the Parser class is constructed by passing a reference to a Scanner object. The process is unraveled by invoking the parse method, which returns an instance of the Program class. A Program is a subclass of Node, an abstract class that provides basic structure for every node in the AST. In particular, Program is the node that lies at the root of the AST. For each construct specified by the concrete syntax of *Scandal*, there is a corresponding construct specified by its abstract syntax. More often than not, the abstract construct will be simpler, sometimes with many tokens removed. The job of the parser is to facilitate the process of inferring meaning from a given program, and it does so by going, from left to right, through the array of tokens passed by the scanner and, whenever it sees a sequence of tokens that matches one of the constructs in the concrete syntax, it consumes those tokens and creates a subclass of Node that corresponds to the construct at hand. It follows, for every acceptable construct in the DSL, there is a subclass of Node that defines it. Some nodes are nested hierarchically in others, and ultimately all nodes are nested in an instance of Program, hence why the parsing stage ultimately constructs a tree.

Structuring a program hierarchically is essential for inferring the meaning of complex expressions that have some sort of precedence relation among its sub-expressions. That is

the case of arithmetic operations, in which, say, multiplication has precedence over addition, and exponentiation has precedence over multiplication. As an example, *Supercollider* evaluates $3 + 3 * 3$ to 18, since it parses the expression from left to right without regard for the precedence relations among arithmetic operators. This is counterintuitive, and does not correspond to how mathematical expressions are evaluated in general. We would like, instead, the expression $3 + 3 * 3$ to evaluate to 12, in which case we cannot take it from left to right. Rather, we must first evaluate $e_2 = 3 * 3$, *then* evaluate $e_1 = 3 + e_2$. It is easy to see that, no matter how complex the expression might be, we can always represent it as a *binary* tree by taking the leftmost, highest-precedence operator and splitting the expression in half at that point. We then look at each sub-expression and do the same, until we reach a leaf. Note that the AST is not, in general, a binary tree. If two operators have the same precedence, we associate from left to right, that is, $1 - 2 + 3 = (1 - 2) + 3 = 2$, which also corresponds to how mathematical expressions associate. Complex expressions are dealt in *Scandal* by the BinaryExpression class, and the fact that instances of BinaryExpression may contain other instances of BinaryExpression simply means we must construct them recursively. We shall discuss in detail each syntactical construct of *Scandal* in the sections that follow, along with their concrete and abstract syntax definitions, and parsing routines.

2.3.4 Decorating the AST

The idea of representing a program as a tree has many advantages, chief among them being the fact we can traverse the tree to infer its meaning. This is often non-trivial, and is necessary as many constructs are name-references to other constructs, and require that we look back to how they were originally declared if we are to make sense of them. In *Scandal*, every subclass of Node overrides the abstract method decorate, which in turn takes an instance of SymbolTable as an argument. The latter is a class that implements a LeBlanc-Cook symbol table [4]. Several nodes in the DSL define new naming scopes, Program being the node that holds the zeroth scope. These nodes are namely those that

have Block, or its subclass LambdaBlock as members. IfStatement and WhileStatement both have Block as a child node, whereas LambdaLitBlock points to a LambdaBlock, who differs from Block in that it has a return statement. LambdaBlock only exist in the context of a lambda literal expression, however instances of Block, inside if or while-statements or on their own, may exits arbitrarily, always defining new naming scopes. Every time we enter a new scope in *Scandal*, we have access to variables that were declared in outer scopes, but the converse is not true. Also, every time we enter a new scope, we have the opportunity of re-declaring variables' names without the risk of clashing with names already declared in outer scopes. For each scope, we hold a hash table whose keys are the variables' names, and whose values are subclasses of the abstract type Declaration. In order to *remember* as we enter new scopes, and *forget* as we leave them, an instance of SymbolTable holds a Stack of name-declaration hash tables, since stacks are exactly the kind of data structure that gives us this last-in, first-out behavior. In order to trigger the whole process of decorating the AST, the Compiler class instantiates a SymbolTable, and passes that as an argument to the instance of Program that was returned by the parser. Listing 2.3 shows how this is done in the compile method inside Program. Since every node overrides the decorate method, this instance of SymbolTable is passed down along the entire tree. Nodes that introduce new naming scopes have the responsibility of pushing a new hash table onto the stack, then popping it before returning from the decorate method.

In addition to resolving names, the decoration process is crucial for type-checking expressions and statements in the DSL. Even though the *Java* bytecode instructions are explicitly typed, languages that compile to bytecode do not need to be. That is the case of *Scala* and *Groovy*, in which types can be inferred, or declared explicitly. Furthermore, there is a degree of latitude to which types can actually *change* in the bytecode implementation. The JVM only cares that, once a variable is stored in a certain slot number as, say, a float, that is, using the instruction FSTORE, that it be retrieved too as a float, that is, using the instruction FLOAD. It is perfectly possible to use the same

slot number to, say, ISTORE an integer value. The only consistency the JVM requires

is that, for as long as that slot holds an integer, the value can only be retrieved by an

ILOAD instruction. This requirement naturally extends to method signatures, which are

also explicitly typed in the JVM. Hence, like *JavaScript*, one can theoretically change

the type of a variable attached to a name after it has been declared; unlike *JavaScript*,

however, types have to be assigned to arguments when declaring a method, and that

method signature is immutable. It is still possible to overload a method to accept multiple

signatures, but overloaded names are still *different* methods, with altogether different

bodies. The same applies to non-primitive types, that is, types that are instances of

a class in the JVM. To store or retrieve non-primitive types, we use the ASTORE and

ALOAD JVM instructions, respectively. Hence it is also theoretically possible to overwrite

non-primitive types. However, method signatures that take non-primitives require a fully-

qualified class name, hence are immutable as above. A fully-qualified name is the name

of the class, preceded by the names of the packages in which it is contained, separated by

forward slashes. For Compiler, for example, we have language/compiler/Compiler.

Listing 2.3. Triggering the compilation process of a *Scandal* program.

```
public void compile() throws Exception {                          1
    imports.clear();                                              2
    code = "";                                                    3
    link(path);                                                   4
    for (String p : imports) code += getCode(p);                  5
    symtab = new SymbolTable(className);                          6
    program = getProgram(code);                                   7
    program.decorate(symtab);                                     8
    program.generate(null, symtab);                               9
}                                                                 10
```

Scandal is, by design choice, strongly typed. There are many reasons for that. The

main reason is that the only kind of method it supports is that of a lambda expression,

even though *Scandal* is not a pure functional language. These lambda expressions define themselves their own parametrized sub-types, hence a lot of what the language *is* hinges on type safety. It is also a design choice to make *Scandal* accessible as an entry-level language, that is, directed toward an audience interested in learning audio signal processing in more depth, without the implementation hiding inherent to the unit-generator concept. Having types explicitly defined can help inexperienced programmers better debug their code, as well as help them understand the underlying implementation of the language. Type inference is, in essence, another way of hiding implementation, which has advantages, but also drawbacks. It is notoriously difficult to report errors and debug large projects in an IDE with languages that are not strongly typed. That is certainly the case with *JavaScript*, of which *TypeScript* is a typed superset aimed exactly at facilitating development within an IDE. *Scandal* is fully integrate into its IDE, where reporting compilation errors to the programmer is a lot more informative, hence educational, than throwing runtime errors and aborting execution. For all these reasons, type-checking is one of the main jobs the decoration process accomplished. It can become rather involved, especially when it comes to composing partial applications of lambda expressions. We shall describe the intricacies of type checking alongside each of the DSL's constructs in the sections that follow.

2.3.5 Generating Bytecode

Similarly to the decoration process, bytecode generation is triggered from the root of the AST, that is, an instance of Program received from the parser, and which has been already decorated, and passed down to every node of the tree by a common abstract method each subclass of Node overrides. In this case, this common method is called generate , and it takes two arguments. The first is an instance of org.objectweb.asm.MethodVisitor, and the second is the the decorated instance of SymbolTable. MethodVisitor is part of the ASM library, which is a convenient set of tools aimed at facilitating the generation of *Java* bytecode. As the name suggests, it visits a method within the bytecode class and

adds statements to it. As can be seen in line 9 of Listing 2.3, a null pointer is passed to the very first call to generate, since at that point we have not created any methods in the bytecode class yet. Every *Scandal* program compiles to a *Java* class, which in turn implements the Runnable interface. Inside the class, there are three methods: init, where we create the method bodies of lambda literal expressions, which are always fields in the *Java* class; run, which is a required override of the Runnable interface, and where we create all *Scandal* local variables and statements; and main, where we instantiate the class and call run. Inside Program, the generate method creates three instances of MethodVisitor, one for each aforementioned method.

If and only if a child node is an instance of LambdaLitDeclaration, a Node used to declare a name and assign to it a lambda literal expression, this child node is passed an instance of MethodVisitor that lives inside the init method. The immediate implication of this design choice is that lambda literal expressions are always global variables in a *Scandal* program, thus accessible everywhere. However, they must be declared at the outermost scope of the program, and will throw a compilation error if declared elsewhere. A similar design pattern applies to nodes that are instances of FieldDeclaration, a Node used to declare field variables in the *Java* class, which in turn correspond to global variables in the *Scandal* program. For both LambdaLitDeclaration and FieldDeclaration nodes, we need to add field declarations in the *Java* class, which is accomplished by instantiating, for each of these nodes, a org.objectweb.asm.FieldVisitor. This is only ever done inside Program hence, as a consequence, global variables in a *Scandal* program must always be declared at the outermost scope. Similarly to instances of LambdaLitDeclaration, instances of FieldDeclaration in inner scopes throw a compilation error. Every descendant of the root node that is *not* an instance of LambdaLitDeclaration receives as a parameter to its generate method an instance of MethodVisitor that lives inside the run method of the *Java* class. This includes instances of FieldDeclaration, which are only declared by a FieldVisitor, and

whose assignment is done inside the run method, along with all other declarations and statements.

Unlike instances of FieldDeclaration, instances of LambdaLitDeclaration are marked as *final* in the *Java* class, hence cannot be reassigned. The reason is simple: once reassigning a variable that points to a method body, the latter may become inaccessible. In *Scandal*, one can create references to lambdas inside the run method, which are not instances of LambdaLitDeclaration, that is, which do not specify a method body. These references are rather instances of the superclass AssignmentDeclaration, and can be freely reassigned, even to lambdas that have different parameters, i.e., method signatures, that that of the original assignment. Reassigning references to lambdas come allow for great code re-usability. There is a third subclass of Node which can only be used at the outermost scope, namely ImportStatement. The reason is, besides clarity and organization of *Scandal* code, because the link routine inside Program only looks for import statements within the outermost scope of a program's AST. In all three such nodes, checking whether that particular instance lives in the outermost scope is a simple matter of asking the passed instance of SymbolTable whether the current scope number is zero.

2.3.6   Running a *Scandal* Program

Every Program node holds an array of bytes corresponding to a binary representation of the compiled *Scandal* program. This array is created right before the generate method returns. The Compliler class naturally holds a reference to an instance of Program, and utilizes the latter's bytecode property to dynamically instantiate the *Scandal* program as a *Java* class, that is, from an array of bytes stored in memory, rather than from a *.class* in the file system. Within the IDE, a path to a *Scandal* program is used to instantiate a Compiler. After calling the compile method, the resulting bytecode is used to define a subclass of java.lang.ClassLoader, namely DynamicClassLoader, which is capable of dynamically instantiating a byte array as a *Java* class, as opposed to the instance returned by the static method ClassLoader.getSystemClassLoader(), which can only load classes from the file

system. Once defined, we construct and instantiate the program, finally casting the result to Runnable, as illustrated in Listing 2.4. The getInstance method is called from the IDE by the tab that currently holds the pogram's text editor, which is an instance of ScandalTab. After retrieving the instance of Runnable, the ScandalTab simply puts is on a new Thread. Starting the thread then causes the *Scandal* program to execute.

Listing 2.4. Obtaining an instance of a *Scandal* program.

```
public Runnable getInstance() throws Exception {              1
    ClassLoader context = ClassLoader.getSystemClassLoader();     2
    DynamicClassLoader loader = new DynamicClassLoader(context);  3
    return (Runnable) loader                                      4
        .define(className, program.bytecode)                      5
        .getConstructor()                                         6
        .newInstance();                                           7
}                                                                 8
```

2.4 The Syntax of *Scandal*

In this section, we describe in detail every syntactical construct of *Scandal*. For each of them, we state their concrete and abstract syntax definitions, how one is converted into the other in the parser, as well as the particularities of type-checking and generating bytecode. We omit some constructs that, either have trivial implementations, or whose implementations are, *mutatis mutandis*, identical to other constructs, in which case we describe only a representative. In the discussion that follows, terminal symbols are in all-capital letters, productions in the concrete syntax begin with a lower-case letter, and their counterparts in the abstract syntax begins with an upper-case letter. We here present terminal symbols in the syntactical context in which they appear, and a complete list of terminal symbols can be found in Sec. **??**.

2.4.1 Top-Level Productions

At the topmost level of a *Scandal* program, there are basically only two kinds of constructs that are allowed, namely declarations and statements. The legal declarations

at this level are further subdivided into three: assignment declarations, field declarations, and lambda literal declarations. In the productions that follow, the star symbol represents a *Kleene* star, and or-symbols and parenthesis are not tokens in the language. As a rule, terminal symbols will be given by their names, like `OR`, to avoid confusion with symbols in the language's grammar. Below are the production rules for program:

— types := `KW_INT` | `KW_FLOAT` | `KW_BOOL` | `KW_STRING` | `KW_ARRAY` | `KW_LAMBDA`

— declaration := assignmentDeclaration | fieldDeclaration

— declaration := lambdaLitDeclaration | paramDeclaration

— program := (assignmentDeclaration | fieldDeclaration)*

— program := (lambdaLitDeclaration | statement)*

In the AST, Declaration is an abstract class that has two subclasses: AssignmentDeclaration, and ParamDeclaration. The former has two subclasses, FieldDeclaration, and LambdaLitDeclaration. The primary difference between the two subclasses of Declaration is that the latter defines a type and a name without binding any value to that name at the time of declaration, while the former requires that some expression be given at the moment the variable is declared. It follows every variable declaration in *Scandal* must be initialized, except when they are parameters of a lambda literal, in which case they actually cannot be initialized. A program can contain any number of assignmentDec, fieldDec, or lambdaLitDec, in any order, while a paramDec only exist in the context of a lambda literal. As will be seen below, every declaration begins with a type token, or with a field flag, followed by a type token. After parsed, types are converted into an enumeration case, according to the keyword at hand. The abstract syntax of Program is then:

— Types := `INT` | `FLOAT` | `BOOL` | `STRING` | `ARRAY` | `LAMBDA`

— Declaration := AssignmentDeclaration | FieldDeclaration

— Declaration := LambdaLitDeclaration | ParamDeclaration

— Program := (LambdaLitDeclaration | FieldDeclaration)*

–   Program := (AssignmentDeclaration | Statement)*

The parsing routine for program is very simple, and constructs an instance of Program by checking whether the next token in the array of tokens produced by the scanner is in the FIRST set of declaration. If it is, we attempt to construct an instance or subclass of AssignmentDeclaration, consuming in the process all the tokens therein. If not, we attempt to construct a subclass of the abstract type Statement. That is done much that same way, by looking at the set FIRST(statement). Listing 2.5 shows how a concrete program is converted into a Program node in the AST.

Listing 2.5. Parsing topmost-level constructs in *Scandal*.

```
public Program parse() throws Exception {                                1
    Token firstToken = token;                                           2
    ArrayList<Node> nodes = new ArrayList<>();                          3
    while (token.kind != EOF) {                                         4
        if (token.isDeclaration()) nodes.add(assignmentDeclaration());  5
        else nodes.add(statement());                                    6
    }                                                                   7
    matchEOF();                                                         8
    return new Program(firstToken, nodes);                             9
}                                                                       10
```

In Listing 2.5, we construct an instance of Program by first creating an array of nodes. These nodes, however, must be either a subclass of AssignmentDeclaration, or a subclass of the abstract class Statement. Line 5 checks whether the next unconsumed token is in the FIRST set of a declaration. If so, further parsing is delegated to the assignmentDeclaration routine. If not, the only other option is that the next token initiates a statement, and parsing thereof is delegated to the statement routine in line 6. Nodes are added in the exact order in which they appear in the *Scandal* program, regardless whether they are declarations or statements. An end-of-file token was included in the scanning process for convenience, and here we make use of it by checking the next available token against the

EOF kind. As soon as we find it, we know we have reached the end of the token array, and can thus stop looking for declarations and statements. If we were expecting a particular token, but EOF appeared prematurely, we throw an error.

Inside an instance of Program, type-checking is completely delegated to each node in the node array. More precisely, inside the decorate routine, we iterate over the node array and, for each node, we call node.decorate, passing along the symbol table instantiated by the compiler. Generating bytecode, on the other hand, is a lot more complex, since we need to provide the overall structure for the entire *Java* class. That is accomplished inside the generate method by creating an instance of org.objectweb.asm.ClassWriter. The latter, which we call cw, manages the creation of the *Java* class itself, including the generation of the byte array used to instantiate and run the *Scandal* program. In particular, we set the JRE to version 1.8, make the access to the class public, and define it as a subclass of java/lang/Object that implements the java/lang/Runnable interface. We then create three instances of MethodVisitor by calling cw.visitMethod, one for each method in the *Java* class. The methods are namely init, run, and main. In init, we basically go through the node array and, if the particular node is an instance of LambdaLitDeclaration, we call node.generate, passing the appropriate instance of MethodVisitor and our symbol table as parameters. What the generate method does inside a LambdaLitDeclaration is somewhat complicated, and we defer its explanation to the moment we discuss the LambdaLitExpression class. Before visiting run, we go once again over all nodes in the node array and, if they are either an instance of LambdaLitDeclaration or an instance of FieldDeclaration, we call cw.visitField. This method creates fields in the Java class, which correspond to field variables in the *Scandal* program. Every field is marked as static, since we make no use of *Java*'s object-oriented paradigm. In addition, lambda fields are marked as final, as previously discussed, and we take the opportunity to pass cw along to the instances of LambdaLitExpression for which we are creating field declarations, and ask them to create a method body for the lambda literal expression. This is accomplished inside each lambda literal expression by

an overloaded generate method, which takes, instead of a MethodWriter, the instance of ClassWriter, namely cw, and uses that to create its own MethodWriter, which will correspond to the lambda's method. The instances of LambdaLitExpression are accessed through the lambda property inside the LambdaLitDeclaration, and the particularities of creating method bodies for lambdas will be discussed momentarily. The next step is to add a body for the *Java* class' run method. To do so, we go yet once more over the array of nodes, and this time we generate any node that is *not* an instance of LambdaLitDeclaration, for obvious reasons. We do visit instances of FieldDeclaration, since cw.visitField only created the field, but never assigned any value to it. Since we only allow unassigned declarations in *Scandal* when declaring lambda parameters, we have something to assign to that field, and generate inside FieldDeclaration takes care of that.

Listing 2.6. Using the ASM framework to construct a main method.

```
private void addMain(ClassWriter cw, SymbolTable symtab) {          1
    MethodVisitor mv =                                               2
        cw.visitMethod(ACC_PUBLIC + ACC_STATIC, "main", "([Ljava/lang/String;)V"   3
            , null, null);
    mv.visitTypeInsn(NEW, symtab.className);                         4
    mv.visitInsn(DUP);                                               5
    mv.visitMethodInsn(INVOKESPECIAL, symtab.className, "<init>", "()V", false)    6
        ;
    mv.visitMethodInsn(INVOKEVIRTUAL, symtab.className, "run", "()V", false);       7
    mv.visitInsn(RETURN);                                           8
    mv.visitMaxs(0, 0);                                              9
}                                                                   10
```

Finally, we visit the main method in the *Java* class, which is shown in Listing 2.6. This is the standard main method in *Java*, which is always public and static, takes an array of strings, and returns nothing. Line 3 uses cw to create an instance of MethodVisitor, namely mv, with exactly these properties. Bytecode syntax for method signatures is given by a parenthesized list of argument types, followed by the return type. Hence a void

39

method that takes a String [] in *Java* becomes ([Ljava/lang/String;)V, where the left bracket means we have an array of whatever type follows, and the colon separates argument types. Naturally, java/lang/String is a string, and V stands for the void type. The JVM is stack-based, so in line 4 we create a new instance of the *Java* class, whose name is stored in our symbol table, and leave it on top of the stack. In line 5 we duplicate whatever is on top of the stack, since we will need to use our newly created *Java* class twice, namely to call on it init, and then run. These two calls are made in lines 6 and 7, respectively. Notice both method signatures take no arguments and return nothing, hence are equivalent to ()V in bytecode. Finally, we add a return statement to the main method's body, which is omitted in void *Java* methods, but required in bytecode. A bytecode method requires that we compute the maximum number of elements the stack will have, as well as the total number of local variables in the method. ASM does that for us, and we asked it to do so by passing a ClassWriter.COMPUTE_FRAMES as an argument while constructing cw. The two arguments to mv.visitMaxs are the maximum stack size, and the total local variables. We pass zeros since we are not computing them, but the call must be made nonetheless.

2.4.2   Subclasses of Declaration

The class Declaration is an abstract type that extends Node by adding three instance variables, namely a Token to hold the name we are declaring, an integer to hold its slot number, and a boolean property to distinguish whether this is a field or not. Slot numbers are not necessary for fields, hence are only used in the context of the *Java* class' run method. Declaration branches out into two non-abstract subclasses, ParamDeclaration and AssignmentDeclaration. The latter has itself two other subclasses, FieldDeclaration and LambdaLitDeclaration. As discussed above, the difference is that ParamDeclaration only occurs inside a LambdaLitDeclaration; FieldDeclaration and LambdaLitDeclaration only occur at the outermost scope; and AssignmentDeclaration occurs anywhere. Below are the production rules for the concrete syntax of declarations in *Scandal.*

−   paramDeclaration := types IDENT

- assignmentDeclaration := types `IDENT ASSIGN` expression

- fieldDeclaration := `KW_FIELD` types `IDENT ASSIGN` expression

- lambdaLitDeclaration := `KW_LAMBDA IDENT ASSIGN` lambdaLitExpression

- lambdaLitDeclaration := `KW_LAMBDA IDENT ASSIGN` lambdaLitBlock

Listing 2.7. Parsing Top-Level Declarations.

```
public AssignmentDeclaration assignmentDeclaration() throws Exception {          1
   boolean isField = token.kind == KW_FIELD;                                      2
   if (isField) consume();                                                        3
   Token firstToken = consume();                                                  4
   Token identToken = match(IDENT);                                               5
   match(ASSIGN);                                                                 6
   Expression e = expression();                                                   7
   if (e instanceof LambdaLitExpression)                                          8
      return new LambdaLitDeclaration(firstToken, identToken, (LambdaLitExpression) e);   9
   if (isField) return new FieldDeclaration(firstToken, identToken, e);          10
   return new AssignmentDeclaration(firstToken, identToken, e);                  11
}                                                                                12
```

As shown in line 5 of Listing 2.5, we parse declarations by looking at the very first token at hand. In particular, we have FIRST(declaration) = type ∪ `KW_FIELD`. Observing that FieldDeclaration is the only subclass of Declaration that may make use of a field flag, we can begin parsing declarations by first checking whether the first token of a declaration is indeed a field flag, setting a local boolean property accordingly, and consuming the `KW_FIELD` token, as shown in line 2 of Listing 2.7. We then store the type and identifier tokens, consume the equals sign and delegate the expression rule to the expression routine. Based on which type of expression we receive, we create the corresponding subclass of AssignmentDeclaration. Even though lambda literal declarations are always fields in the *Java* class, the field flag is not necessary, but *can* be used without errors, since all that determines an instance of LambdaLitDeclaration is that the expression it contains is a subclass of LambdaLitExpression. Unassigned declarations, on the other hand, will fail the match(ASSIGN) call in line 6, and will cause a compilation error. Below are the abstract

41

syntax rules for declarations. In these abstract rules, an `IDENT` has a different meaning than its concrete counterpart. Here it is an instance of Token, rather than a character string.

– ParamDeclaration := Types Token.`IDENT`

– AssignmentDeclaration := Types Token.`IDENT` Expression

– FieldDeclaration := Types Token.`IDENT` Expression

– LambdaLitDeclaration := Types.`LAMBDA` Token.`IDENT` LambdaLitExpression

– LambdaLitDeclaration := Types.`LAMBDA` Token.`IDENT` LambdaLitBlock

2.4.2.1 The ParamDeclaration Class

ParamDeclaration is basically an unchanged implementation of the abstract type Declaration. It adds no new properties, thus consisting of basically a type Token and an identifier Token. Since it is not abstract, it must override decorate and generate, the two abstract methods in Node that provide functionality to all nodes in the AST. Parsing a parameter declaration is absolutely straightforward, and done in the context of a LambdaLitExpression. We simply store and consume the type and identifier tokens, then use those to instantiate a ParameterDeclaration.

When a lambda literal expression is decorated, a new scope in the symbol table is introduced, so that parameter names do not clash with local variables in the *Java* class' run method. Thus inside an instance of ParamDeclaration, the decorate method checks only the symbol table at the top of the stack of symbol tables to see whether any two parameters have the same identifier, in which case it throws a compilation error. If not, it inserts the identifier into the topmost scope of the symbol table, associating to the identifier the instance of Declaration at hand. Since parameter declarations are local variables inside a lambda body, they need to have the slotNumber property set so they can be accessed. This is accomplished, however, by the lambda literal expression class before the decorate method is called on a parameter declaration, as we shall see momentarily. In

the generate method, we do nothing, since there is no value we can bind to the declaration's identifier at the moment. Naturally, these values will exist in the context of a lambda application.

2.4.2.2 The LambdaLitDeclaration Class

The LambdaLitDeclaration class is a particular case of AssignmentDeclaration where the expression is of type LambdaLitExpression. By defining a new type, we are able to separate more easily instances of lambda literals from other nodes inside a Program, as previously seen. As we shall see, having a specific type for lambda declarations is also invaluable when the underlying implementation of a lambda is obscured by partial applications and compositions, in which case we need to unravel an entire chain of bindings until we find a name that is bound to an expression of type LambdaLitDeclaration. We construct a lambda literal declaration by taking a LambdaLitExpression, and passing it to the constructor of the superclass. It follows the superclass' expression property is just a reference to this lambda literal expression property, which we call lambda. Naturally, every LambdaLiteralExpression inherits from Expression. At the time we construct a lambda literal declaration, we immediately set the isField boolean property to true, as discussed above.

Lambda literal declarations have very different implementations of the Node abstract methods than those of AssignmentDeclaration, hence both decorate and generate are overridden. The bodies of these methods are substantially simpler than the superclass' implementation, given the restricted nature of this type. Inside decorate, we check the entire stack of symbol tables to see whether we are not being redeclared, in which case an error is thrown. Checking only the topmost scope symbol table would work exactly the same, since lambda literal declarations are only allowed at the outermost scope, hence the stack only contain a single symbol table when we call decorate. If we are being declared for the first time, then we insert the identifier into the symbol table, associating to it the instance of LambdaLitDeclaration we have. Unlike parameter declarations, we now have an expression to decorate, which we do by calling lambda.decorate, hence delegating decoration to the

LambdaLitExpression instance. Similarly, the generate method calls lambda.generate, which causes a lambda literal expression to be left on top of the JVM stack. We then bind this expression to the identToken.text property and return.

2.4.2.3   The FieldDeclaration Class

Field declarations are possibly the simplest type of declaration. They mostly exist for convenience, in order not to clutter its superclass AssignmentDeclaration, which has a somewhat more involved implementation. We construct them by calling the superclass' constructor with exactly the same arguments, but in addition we set the isField boolean property to true. We also override both decorate and generate. In the former, and similarly to LambdaLitDeclaration, we check all symbol table scopes to see if we are not being redeclared. Naturally, there is only ever one scope to check. If all is fine, we insert ourselves into the symbol table with a key given by identToken.text and a value of this. We then call expression.decorate, after which the expression will be decorated with a non-null type, a property that is common to every node. Unlike LambdaLitDeclaration, in which the declaration and lambda always had the same type by construction, here we can have runtime errors if the program tries to store, say, a float into an integer slot. We then simply check whether the expression's type is the same as the declaration's. The latter never needed to be decorated, and was set when the constructor of Node was called, since it could be inferred from the declaration's first token alone. The overridden generate method is identical to that of LambdaLitDeclaration.

2.4.2.4   The AssignmentDeclaration Class

Assignment declarations are the most general an common type of declaration in *Scandal*. They correspond to all variable declarations that are not *special*, neither in the sense of declaring a method body, nor in the sense of being global to a *Scandal* program. In other words, they live inside the Java class' run method, as well as inside a lambda's body. Since the isField property is false by default, we construct a AssignmentDeclaration by passing a type token and an identifier token to the superclass, then storing our own

44

expression property, the latter being what differentiates us from the abstract type Declaration
. The decoration process is similar to those of the two subclasses of AssignmentDeclaration,
but a bit trickier. We start by checking the top of the stack of symbol tables to see if there
is no name clash, then insert into the symbol table a key of identToken.text with value this.
We then assign a slot number to this variable, which we do by maintaining a property
slotCount inside SymbolTable. We assign our slot number to the current slot count, then
increase the latter. We call expression.decorate, similarly to what we do in field and lambda
literal declarations, to decorate the expression, which will cause it to have a non-null type
in most cases, except when the expression is an instance of LambdaAppExpression, in which
case we hit a small roadblock in the type-checking process. To understand the situation,
we provide in Listing 2.8 a small snippet of *Scandal* code.

Listing 2.8. Type inference in *Scandal*.

```
lambda id = float x −> x                                        1
lambda higherOrder = float x −> lambda f −> {                    2
    float val = f(x)                                            3
    return val                                                  4
}                                                              5
```

When a lambda expression in *Scandal* has a parameter of type lambda, we have no
mechanism in the language to tell what parameter types pertain to said lambda. The
corresponding construction in *Java* is an instance of the Function interface, which is a
parameterized type. A lambda expression in *Java* that takes a float and returns a float has
type Function<Float, Float>, for example. It has been a design choice so far in *Scandal* not
to introduce parameterized types, and attempt instead at some yet crude type inference
mechanism. It is a long-term goal to actually make *all* types inferred. In Listing 2.8, we
have an instance of ParamDeclaration that defines a variable f of type lambda. Inside the
block, the mechanism of choice to in fact *declare* what parameter types f has is through an
assignment declaration. In line 3, we apply x to f and store the result in val. Since both x

45

and val have type float, we are now at a position to determine with that f takes a single argument of type float, and returns also a float. Note that omitting line 3 and putting return f(x) instead would cause a compilation error, exactly because we cannot infer parameter types of a lambda expression inside a lambda literal if there is no application thereof.

Given the discussion above, whenever an assignment declaration calls expression. decorate, and this expression happens to be a lambda application inside a lambda literal, instead of expecting the call to expression.decorate to define a type for the expression, we actually decorate the expression ourselves. The process is very simple: after calling expression.decorate, we check if the expression is an instance of LambdaAppExpression. If so, we look for the declaration of that application's expression. If the declaration is an instance of ParamDec, we know our expression is an application of a lambda is a parameter of lambda literal expression, since parameter declarations only ever exist in this context. We then trust the programmer will make use of the lambda literal expression correctly, by applying to it a lambda expression that has the same parameter types as those inferred inside the block. If not, a runtime error will occur. This is obviously not a complete implementation and, as discussed previously, future versions of *Scandal* will throw a compilation error instead. Finally, the expression property will have to have a type, which we check against the declaration type. Naturally, in the special case we decorate the expression ourselves, this test never fails. If it does, we throw a compilation error. Overriding the generate method inside an assignment declaration is straightforward. We make a call to expression.generate, which causes the expression value to be left on top of the JVM stack, after which we switch over the expression type, using the appropriate JVM instruction to store the variable. We end this section with a summary of the type-checker rules for all subclasses of Declaration.

− AssignmentDeclaration:

  + Must not be declared more than once in the same scope

46

+ Type = Expression.type

− FieldDeclaration:

+ Must be declared in the outermost scope

+ Must not be declared more than once

+ Type = Expression.type

− LambdaLitDeclaration:

+ Must be declared in the outermost scope

+ Must not be declared more than once

+ Type = Types.`LAMBDA`

− ParamDeclaration:

+ Must not be declared more than once

2.4.3 Subclasses of Statement

Statements and declarations are the only types of top-level constructs that *Scandal* supports. Even blocks are not allowed on their own, and may only occur in the context of a statement or lambda literal expression. That said, statements can occur freely inside any scope, and are essential for the language in that they provide much its core functionality. In many cases, statements are what allow *Scandal* to remain restricted to its domain, providing hooks to *Java* routines that, if ported to the DSL, would challenge its very purpose. In this sense, much of what it means to improve the language's domain, like its ability to produce user-interface elements, for example, will depend on how new types of statements are introduced. As a language designer, one cannot help but wonder about the trade-offs of such endeavor. A good balance between introducing new features while maintaining scalability is always delicate. On one hand, new features are necessary to provide more functionality. On the other, every bit of functionality that *can* be achieved in terms of the language alone, even if more difficult to implement, must be given priority. If the language only grows in terms of its *Java* underlying implementation, the structure of the compiler and its production rules become increasingly harder to maintain. This

47

philosophy has been the main driving force behind giving *Scandal* a functional flavor. It is also diametrically opposite to the concept of unit generators, although one can still hide implementation in *Scandal*, but the converse is true in general for strict unit-generator languages. As seen above in Listing 2.5, parsing statements is accomplished much the same way we parse declarations, by looking at the set FIRST set of statement. Below are the productions for the concrete syntax of *Scandal* statements.

— statement := importStatement | assignmentStatement | indexedAssignStat

— statement := ifStatement | whileStatement | printStatement

— statement := playStatement | plotStat | writeStat

— importStatement := `KW_IMPORT LPAREN` expression `RPAREN`

— assignmentStatement := `IDENT ASSIGN` expression

— indexedAssignStat := `IDENT LBRACKET` expression `RBRACKET ASSIGN` expression

— ifStatement := `KW_IF` expression block

— whileStatement := `KW_WHILE` expression block

— block := `LBRACE` (assignmentDeclaration | statement)* `RBRACE`

— printStatement := `KW_PRINT LPAREN` expression `RPAREN`

— playStatement := `KW_PLAY LPAREN` expression `COMMA` expression `RPAREN`

— plotStat := `KW_PLOT LPAREN` expression `COMMA` expression `COMMA` expression `RPAREN`

— writeStat := `KW_WRITE LPAREN` expression `COMMA` expression `COMMA` expression `RPAREN`

Except for assignment statements, whose first token is an identifier, every statement in the language begins with a keyword. The statement routine in the parser contains a switch with cases for `IDENT` and every statement keyword, and defaults to throwing an error if the first token given does not match any of these kinds. Depending on the keyword, the parser instantiates one of the particular subclasses of Statement. There are basically four varieties of statements in *Scandal*: compiler statements, assignment statements, control statements, and framework statements. Compiler statements are

48

restricted to import statements at the moment. As the language grows in terms of itself, likely there will be need in the future for compiler statements that aid in API documentation, such as double-star comments that are used to generate a web page, say. Assignments simply bind a new expression to a previously declared variable. Control statements are if and while-loops. Switches, repeat-while-loops, and for-loops are not yet included, but intended. Framework statements define the specific domain of the language, and consist of four hooks to the *Java* audio engine, namely routines to print to the console, playback a buffer of audio data with a given number of channels, plot a decimated array of floats, and write a *.wav* file to disk. Framework statements may be regarded as void methods, since they never return anything. There exist other hard-wired routines in *Scandal* that do return some expression, and are thus treated as such and discussed later. The Statement class extends Node but is itself abstract. It is constructed with an instance of Token, that in turn is used to call the superclass' constructor, and an instance of Expression. Every statement in *Scandal* contains at least one expression, but their purposes vary according to the statement type. In converting from concrete rules into abstract ones, assignments, parenthesis, commas, and braces are all discarded, and most statements are in essence a collection of expressions. The exceptions are conditional statements, which also contain blocks. Below are the abstract syntax rules for statements.

–   Statement := ImportStatement | AssignmentStatement | IndexedAssignStat

–   Statement := IfStatement | WhileStatement | PrintStatement

–   Statement := PlotStatement | PlayStatement | WriteStatement

–   ImportStatement := Expression

–   AssignmentStatement := IDENT Expression

–   IndexedAssignStat := IDENT Expression_0 Expression_1

–   IfStatement := Expression Block

–   WhileStatement := Expression Block

– Block := (AssignmentDeclaration | Statement)*

– PrintStatement := Expression

– PlotStatement := Expression_0 Expression_1 Expression_2

– PlayStatement := Expression_0 Expression_1

– WriteStatement := Expression_0 Expression_1 Expression_2

2.4.3.1 The ImportStatement Class

Import statements are the simplest type of statement in Scandal. The ImportStatement class extends Statement, but adds no properties to it. Even though they belong to the AST, no bytecode is ever generated for import statements. We still need to implement generate, but we simply do nothing there. The decoration phase begins by checking the current scope number in the symbol table is zero, since we can only import in the outermost scope. If not, we throw an error. Next, it calls expression.decorate and, after the expression has been decorated with a type, we check to see whether that is indeed a string, throwing an exception if not, and return. Import statements are used by the compiler as described in Listing 2.2, wherein all import statements in a chain of linked programs form a DAG. We depth-first search the DAG by pre-compiling every program, that is, creating an AST for them without decorating or generating bytecode. Paths are then extracted from import statements in order, which effectively creates a reverse topological sorting of the graph. If the chain of programs contains a cycle, then execution will fail at runtime. As discussed above, future versions of Scandal will check for backward edges in the DAG, and throw a more informative compilation error instead.

At the moment, each import statement accepts a single expression of type string, but extending this implementation to allow comma-separated expressions would be fairly trivial. In designing a language, one must consider whether such implementation would *actually* improve the language. *Java* only supports single import statements and that can cause some clutter in the document header. Good IDE's will collapse import statements, which mitigate the cluttering somehow. With *Go*, on the other hand, one is allowed to

have multiple import statements, declared as comma-separated strings. More often than not, though, these strings are quite long, and eventually are broken into multiple lines. In *Scandal*, import statements take paths to files in the file system, which may be fairly long strings. The syntax is very similar to *Go*, however only single import statements are allowed, like *Java*, and the ability to collapse import statements in the IDE will likely take precedence over multiple import statements in the future.

2.4.3.2 Assignment Statements

The AssignmentStatement class extends Statement by including a declaration property and, naturally, overriding both decorate and generate. In the decoration phase, we look for a declaration value whose key corresponds to the identifier we hold in firstToken.text , starting from the innermost (topmost) scope, and descending to the bottom of the stack of symbol tables, as needed. Contrary to assignment declarations, if *no* declaration is found, we throw an error. We then delegate decoration of the expression property to the corresponding subclass of Expression. The exact same caveat with type inference in instances of LambdaAppExpression applies here, so we check whether we are a lambda application of a lambda parameter inside a return block, and decorate the lambda application expression with our declaration's type. After that, we check if the type of our declaration corresponds to the type of our expression, and throw an error if there is a mismatch.

The generate method is somewhat more complicated, since the Function interface in *Java* only handles classes and not primitive types. This restriction applies in *Scandal* to integers, floats, and booleans, which are implemented as primitives, and must therefore be wrapped in a *Java* class, namely Integer, Float, and Boolean, in order to be assigned to parameters in a lambda literal expression. The other three types in Scandal, namely strings, arrays, and lambdas, all correspond to class types in *Java*, hence do not require any conversion. Before assigning to any variable, we always leave an expression value on top of the JVM stack. Whenever assigning to a parameter variable inside a lambda block,

51

if the expression's type correspond to a primitive in *Scandal*, we wrap the primitive on top of the JVM stack into the appropriate *Java* class. We check that by testing whether our declaration property is an instance of ParamDeclaration, which immediately tells us we are inside a lambda block. Otherwise, we test if the declaration is an instance of FieldDeclaration, in which case we use the correct bytecode instruction to assign to the variable. If both tests fail, then we must be a local variable, either in the context of run or local to a lambda body, hence we need to make the correct store call, and provide a slot number. The former is determined by the type property, while the latter is retrieved from the declaration property.

AssignmentDeclaration has a subclass that specializes in assigning float values to an array at particular indices. Naturally, we need to extend AssignmentDeclaration by including an expression property that holds the index at which we are assigning. We parse indexed assignment statements alongside its superclass by observing that $\text{PREDICT}(\text{indexedAssignmentStatement}) = \text{PREDICT}(\text{assignmentStatement})$, and differentiate between indexed and non-indexed assignments by consuming the identifier, then checking if the next token is a left bracket. An instance of IndexedAssignmentStatement is constructed by passing constructing the superclass, then storing the the index expression. Decoration is similar to the superclass in it checks the symbol table to see whether the identifier has been declared, throwing an error if not. In addition, we must check if the declaration associated to the identifier has indeed type array, otherwise an error is thrown. We then delegate the decoration of the index property to the appropriate subclass of Expression, as usual. Contrary to most descendants of C, however, we do allow arrays to be indexed by expressions of type float in *Scandal*, in which case we simply take the integer part of the number in the generation phase. Hence we check the type of index, throwing an error if its neither an integer nor a float. We decorate the expression property in a similar way, however here too care must be taken with higher-order functions. We check for those exactly the same way we do in the superclass, decorating the expression property as needed.

Like indices, we overload the expression property to accept integers, as well, converting to float in the generation phase before storing. When decorating a lambda application inside a lambda block, we naturally always set it to Types.FLOAT.

Generating an indexed assignment statement is done a little differently from the superclass in it involves loading the array first on top of the JVM stack, which we do by asking the declaration property for a slot number. After that, we ask index to generate itself. If the index is a float, we call the F2I instruction, which pops the float and pushes its integer part on top of the JVM stack. Similarly, we ask expression to generate itself and, if needed, convert it to float by calling the I2F JVM instruction. Finally, the FASTORE assigns the value on top of the stack to the array at the desired index, eventually popping the three topmost elements from the JVM stack.

2.4.3.3   Control Statements

Both control statements in *Scandal* extend Node by including a block property. The basic functionality of a block is to introduce a new scope of declarations and statements, that are executed once or more times, depending on the type of conditional, should the test following the conditional keyword succeed. The Block class, in turn, extends Node by including an array of nodes, similarly to the Program class. Like the latter, these nodes may be declarations or statements, but unlike the zeroth scope, we disallow import statements, since the compiler only ever looks for those in the topmost scope, as well as lambda literals and fields. The restriction for fields is aimed at forcing clarity among declarations. Again here, it would be trivial to allow them, but it is arguably confusing to declare a global variable inside an inner scope, so we forbid it. The restriction for lambda literals, however, is not deliberate, but rather the result of an incomplete implementation. A fair amount of bookkeeping goes into allowing lambda literal declarations into an inner scope. For one, these would need to be local variables, and currently all lambda literals are global. Moreover, local lambdas can always be achieved by copying a field, hence no

additional functionality would be achieved, thus we omit their implementation in this proof of concept.

Parsing blocks is very similar to parsing top-level constructs. The basic difference is that a block is surrounded by braces, thus we start by consuming the left brace. We then look for declarations or statements until we reach the right brace. The parser routines that create these declarations and statements are exactly the same that create top-level constructs. If a global variable declaration or an import statement is given to a block, the parser will take no notice of it. The error will be thrown by the respective AST nodes upon decoration, when the node itself asks the symbol table for its current scope number, throwing an error if the latter is not zero. Decorating and generating the block is similarly delegated to each node in the nodes array. While decorating, we introduce a new scope by calling symtab.enterScope, ask each node in the nodes array to decorate itself, then call symtab.leaveScope. While generating we iterate over the nodes array, asking each node to generate itself.

We parse conditional statements by consuming the first token, asking the expression method in the parser for an expression, and finally asking the block method for a block. Decorating if and while-statements is identical. We first ask the expression property to decorate itself, then check whether it is of type boolean, otherwise we throw an error. Finally, we ask the block to decorate itself. The generate method naturally differs between both types of conditional statement. For an if-statement, we first ask the expression to generate itself, after which its value will be on top of the JVM stack. Next, we load a constant of value true on top of the stack. Interestingly, integer constants are used in the bytecode implementation of conditional statements, but not in *Java*. Nor in Scandal, mostly for clarity. We then create an instance of org.objectweb.asm.Label which, upon calling mv.visitLabel creates a label instruction in bytecode. Before visiting the label, though, we compare both elements on top of the stack by calling creating an IF_CMPNE jump, and giving it the label we have just created as an argument. When called, it will pop both

topmost elements and jump to the label if these elements are not equal. Since we know one of them has value true, we perform the jump only if the condition is false. What we jump over is exactly the contents of the block, thus we generate the block before we visit the label. The entire process is described in Listing 2.9.

Listing 2.9. Generating If-Statements.

```
public void generate(MethodVisitor mv, SymbolTable symtab) throws Exception {    1
    expression.generate(mv, symtab);                                              2
    mv.visitInsn(ICONST_1);                                                        3
    Label label = new Label();                                                     4
    mv.visitJumpInsn(IF_ICMPNE, label);                                            5
    block.generate(mv, symtab);                                                    6
    mv.visitLabel(label);                                                          7
}                                                                                  8
```

Generating while-loops is analogous, but requires two labels, since we jump back to the first label while the condition is true. Naturally, the first thing we do is visit the first label, to establish that jump location. We then load the expression and a constant of value true, call IF_CMPNE, as with if-statements, passing as an argument the second label, which is yet to be visited. We then visit the block and add a GOTO instruction, giving it the first label as an argument, which causes the condition to be re-evaluated. If still true, we repeat the block. If not, we jump to the second label, whose visit is the last step in generating a while-loop. The actual bytecode implementation often involve more instructions, the generation thereof we gladly delegate to the ASM library. A more complete implementation of these two types of control statement would include handling the cases where, instead of a block, we allow a single statement to be executed should the condition succeed. Adding this functionality is trivial, but would improve readability since we would able to drop the surrounding brackets. A similar type of syntactic sugar has been implemented for lambda literal expressions. A LambdaLitExpression takes an array of parameters and returns an expression, whereas its subclass LambdaLitBlock contains, in

addition, a ReturnBlock. When the body of a lambda literal consists of a return expression alone, we are able to drop the surrounding brackets and possible express the entire lambda in a single line.

2.4.3.4 Framework Statements

Print statements provide the functionality of printing to the IDE's console the value of strings, floats, integers, and booleans. They extend Statement by implementing both decorate and generate. As with import statements, print statements take a single expression as input, hence one needs a print statement for each variable one wishes to see posted on the console. Print statements are parsed by consuming and converting the print keyword into a Token, then asking the parser's expression routine for a subclass of Expression. We then use the token and expression to instantiate a PrintStatement, whose constructor simply calls to the superclass' constructor. Decoration asks the expression to decorate itself, after which we check if the expression type is either array or lambda, in which case we throw an error.

Generating a print statements depends on whether we are running inside the IDE, or as a command-line tool, so we call Platform.isFxApplicationThread to check. In the latter case, we use mv to load System.out, then ask the expression to generate itself on top of the JVM stack, and finally use mv again to call println on System.out. In the former case, we use mv to load language.ide.MainView.console, which is an instance of javafx.scene.control.TextArea, and duplicate the top of the stack, for reasons we will explain momentarily. We then generate the expression on top of that, and test to see if the expression is of type string, in which case we call String.valueOf, since TextArea can only append to its contents a value of type java.lang.String. To actually append the string to whatever the console is displaying at the moment, we call TextArea.appendText, which pops the two topmost elements, namely the expression and the duplication of the console. We then push a new-line character and call TextArea.appendText, in order to create a nice line break. This causes the top of the stack to be popped twice, hence why we duplicated the console.

Plot statements are absolutely fundamental to any digital signal processing development environment. Currently, Scandal supports static linear plots but, as the language evolves, there will be need for many other types of plots. These include logarithmic plots, plots of the complex plane, three-dimensional plots, spectrograms, as well as dynamic plots such as oscilloscopes. The PlotStatement class extends Statement by including two more expressions. The three expression properties are namely a string to display a title for the plot, an array of points to be plotted, and an integer defining the number of points to be plotted. Since arrays of audio samples can be quite long, the last parameter is used to decimate an array if it is longer than the specified number of points. If it is shorter, the PlotTab class will oversample the array to have the specified length.
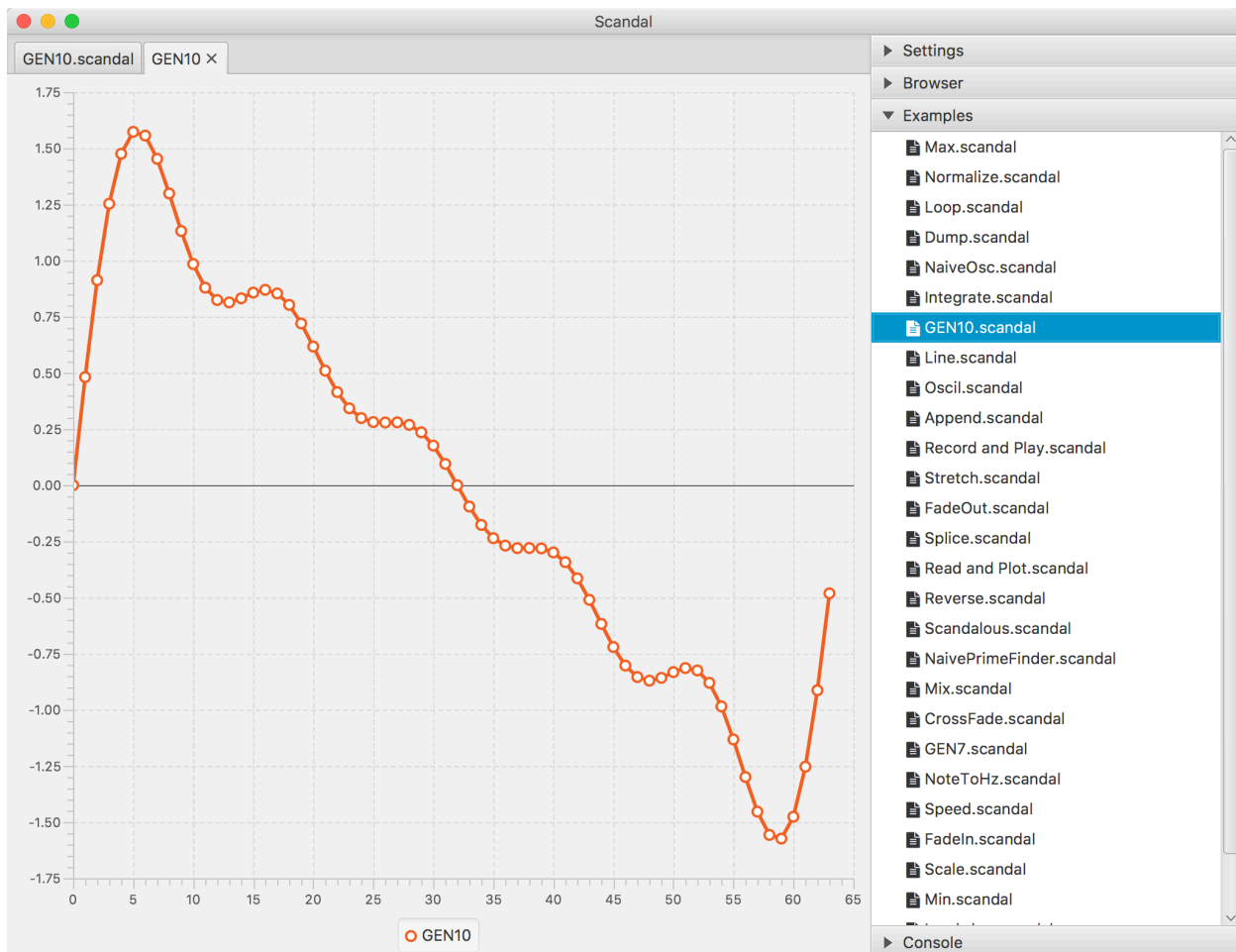

Figure 2-1. Plotting an array in *Scandal*.

We parse plot statements by consuming the keyword and a left parenthesis, then calling expression on the parser three times, consuming the commas in between. We next consume the right parenthesis, and return an instance of PlotStatement, whose constructor uses the keyword token and first expression to construct the superclass. Decoration of a plot statement is accomplished simply by asking each expression to decorate itself. We then type-check the first to be a string, the second to be an array, and overload the third to accept integers and floats. The generation phase checks, like we did with print statements, if we are running on the IDE. If not, we do not generate any bytecode, hence one cannot plot when running a *Scandal* program from the command line. Otherwise, we use mv to push a new instance of language.ide.PlotTab, and push all three expressions on top of it. If the last expression has type float, we give the F2I instruction, and finally call init on the PlotTab, causing it to be displayed in the main view. We provide a thorough discussion of *Scandal*'s IDE in the next chapter.

Play statements take an array of audio samples and an integer number of channels, hence extend Statement by adding to id a second expression property. Parsing is done exactly the same way it is for plot statements, only we have here one less expression to parse. Decoration follows the same pattern, and requires that the first expression be an array, and the second be an integer. We here choose not to overload the integer expression, although it may be useful in the future to use floats in order to define a surround configuration, such as 5.1, or 8.4 speakers, where the decimal part corresponds to the number of sub-woofers in a speaker configuration. Like before, we check in the generation phase whether we are running inside the IDE. If not, we use mv to push a new instance of AudioTask onto the stack, then duplicate it, since we are using it twice. We then call init on the AudioTask, which pops the topmost of the two. We then generate both expressions, call play on the AudioTask, and return. If we are on the IDE, however, we push instead a new instance of language.ide.WaveTab. We then push a string containing the class name, which the symbol table holds, generate both expressions, and call init on

the WaveTab. The string containing the class name is used as the tab's label. The WaveTab class in the IDE is a subclass of PlotTab, hence displays a plot of the array of samples, decimated to 1000 samples, in addition to playing it back, hence there one never needs to plot and play in a *Scandal* program.

The last type of framework statement in the language deals with saving a *.wav* file to disk. Write statements take three arguments, namely the array containing audio samples, a string containing a path in the file system, and an integer number of channels. Parsing follows the same methodology of other framework statements, so does decoration. We do not overload the channel count, but we might in the future to support surround configurations, similarly to play statements. The code-generation routine uses mv to push a new instance of framework.generators.AudioTask, which we duplicate. We will discuss the audio engine's framework package in the next chapter. After calling init on the AudioTask, we push all expressions and export on AudioTask, which effectively saves the audio buffer as a *.wav* at the specified path. We summarize below the type-checker rules for all subclasses of Statement.

− ImportStatement:

+ Must be stated in the outermost scope

+ Expression.type = Types.STRING

− AssignmentStatement:

+ Must have been declared in some enclosing scope

+ Declaration.Type = Expression.type

− IndexedAssignmentStatement:

+ Must have been declared in some enclosing scope

+ Declaration.Type = Types.ARRAY

+ Expression_0.type = Types.INT | Types.FLOAT

+ Expression_1.type = Types.INT | Types.FLOAT

– IfStatement:

   + Expression.type = Types.`BOOL`

– WhileStatement:

   + Expression.type = Types.`BOOL`

– PrintStatement:

   + Expression.type != Types.`ARRAY` | Types.`LAMBDA`

– PlotStatement:

   + Expression_0.type = Types.`STRING`

   + Expression_1.type = Types.`ARRAY`

   + Expression_2.type = Types.`INT` | Types.`FLOAT`

– PlayStatement:

   + Expression_0.type = Types.`ARRAY`

   + Expression_1.type = Types.`INT`

– WriteStatement:

   + Expression_0.type = Types.`ARRAY`

   + Expression_1.type = Types.`STRING`

   + Expression_2.type = Types.`INT`

2.4.4 Subclasses of `Expression`

REFERENCES

[1] C. Roads, M. Mathews, *Computer Music Journal* **4**, 15 (1980). Available from: http://www.jstor.org/stable/3679463.

[2] C. Roads, *The Computer Music Tutorial* (The MIT Press, 1995).

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms* (The MIT Press, 2009).

[4] R. P. Cook, T. J. LeBlanc, *IEEE Transactions on Software Engineering* **9**, 8 (1983). Available from: https://doi.org/10.1109/TSE.1983.236164.

BIOGRAPHICAL SKETCH

Bio goes here.