SCANDAL: 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

TABLE OF CONTENTS

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

SCANDAL: 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*

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

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
instr 1
    kscale amp, knob1, 0, 10000
    kscale freq, knob2, 20, 2000
    oscil x8, #func1, amp, freq
    out x8
endin
```

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

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.

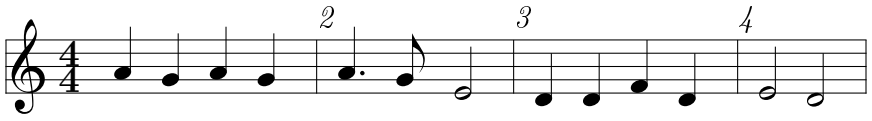
Listing 1.2. *Musica* algorithm that creates a simple melodic line.

```
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* $'\mathrm{A} = 440Hz$*, and* $''\mathrm{A} = 880Hz$*. 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* $\mathrm{h} = 440Hz$*, whereas* $\mathrm{a} = 220Hz$*.*

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

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
IMPLEMENTATION OVERVIEW

In this chapter, we present and discuss the tools and methodology utilized to build *Scandal*. We start discussing how to produce sound with the *Java* sound API, and particularly how it handles real-time audio after a thin layer has been added on top of it. We then give a thorough formal presentation on how the domain-specific language was designed, including the machinery involved in building its compiler.

2.1 The Real-Time Audio Engine

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, but increases latency, which 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 volatile 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.

```
private void play() throws Exception {
    SourceDataLine sourceDataLine = AudioSystem.getSourceDataLine(format);
    sourceDataLine.open(format, Settings.vectorSize * Settings.bitDepth / 8);
    sourceDataLine.start();
    while (running) {
        ByteBuffer buffer = performer.getVector();
        sourceDataLine.write(buffer.array(), 0, buffer.position());
    }
    sourceDataLine.stop();
    sourceDataLine.close();
}
```

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 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.

2.2.1 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.

2.2.2 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 and separation of tokens. *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.

2.2.3 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.

2.2.4 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, statically 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 dynamically 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.

2.2.5 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.

2.2.6 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);
    return (Runnable) loader
            .define(className, program.bytecode)
            .getConstructor()
            .newInstance();
}
```

CHAPTER 3
CONSTRUCTING THE ABSTRACT SYNTAX TREE

This section describes in detail every syntactical construct of *Scandal*. For each of them there is a corresponding Java class with the same name. What follows is restricted to stating their abstract syntax definitions, and how they are implemented in the parser from their corresponding concrete rules. The particularities of type-checking and generating bytecode will be discussed in the next chapters. Constructs that either have trivial implementations, or whose implementations are, *mutatis mutandis*, identical to other constructs are omitted, in which case only a representative is described. In the discussion below, productions begin with an upper-case letter to denote they are abstract counterparts to concrete rules with the same name beginning with a lower-case letter. Whereas all-capitalized rules referred to terminal symbols in the concrete syntax, here they refer to enumeration cases. Abstract all-capitalized rules that refer to types are cases of the Types enumeration, and all other abstract all-capitalized rules are cases of the Token enumeration.

3.1 The Program Class

The root of a *Scandal* AST is always a program, which contains declarations and statements as sub-nodes. Declarations in the AST are instances of Declaration, an abstract class that has two subclasses: AssignmentDeclaration, and ParamDeclaration. The former unfolds into two subclasses, FieldDeclaration, and LambdaLitDeclaration, while the latter has no subclasses. The primary difference between the two subclasses of Declaration is that parameter declarations define a type and a name without binding any value to that name at the time of declaration, while assignment declarations require that some expression be bound to the name at the moment the variable is declared. An instance of Program can contain any number of assignment declarations, field declarations, or lambda literal declarations, in any order, while parameter declarations only exist in the context of lambda literals. Statements are the other type of top-level construct in *Scandal*, and all statements in the AST are subclasses of Statement, which is also abstract. There are nine

different types of statements providing various functionalities to the language. Statements may be regarded as void-returning functions, and represent the imperative side of *Scandal*. Below are the abstract syntax rules for top-level constructs.

- Program := (AssignmentDeclaration | FieldDeclaration)*
- Program := (LambdaLitDeclaration | Statement)*

The parsing routine that constructs an instance of Program is very simple, and does so by checking whether the next token in the array of tokens produced by the scanner is in the FIRST set of a declaration. If so, it attempts to construct an instance of AssignmentDeclaration, consuming in the process all the tokens therein. If not, it attempts to construct a subclass of the abstract type Statement. That is done much that same way, by looking at the FIRST set of a statement. Listing 3.1 shows how a Program node is instantiated in the AST.

Listing 3.1. 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
```

2. Line 2 stores a reference to the first token, since this token will be consumed during the instantiation of sub-nodes, but is needed as an argument to the constructor of Program in line 9.
3. An instance of Program holds an array of nodes. These nodes, however, must be either a subclass of AssignmentDeclaration, or a subclass of the abstract class Statement.

Nodes are added in the exact order in which they appear in the *Scandal* program, regardless whether they are declarations or statements.

5. 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 legal option is that the next token initiates a statement, and parsing thereof is delegated to the statement routine in line 6.

8. An end-of-file token was included in the scanning process for convenience, and parse makes use of it by checking the next available token against the `EOF` kind. As soon as it finds it, it knows it has reached the end of the token array, and can thus stop looking for declarations and statements. If a particular token was expected, but `EOF` appeared prematurely, matchEOF throws an error.

3.2 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 being declared, 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 abstract 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. Similarly, types are converted after parsed into enumeration cases of Types, according to the keyword at hand.

- AssignmentDeclaration := Types Token.`IDENT` Expression
- FieldDeclaration := Types Token.`IDENT` Expression

- LambdaLitDeclaration := Types.`LAMBDA` Token.`IDENT` LambdaLitExpression
- LambdaLitDeclaration := Types.`LAMBDA` Token.`IDENT` LambdaLitBlock
- ParamDeclaration := Types Token.`IDENT`
- Types := `INT` | `FLOAT` | `BOOL` | `STRING` | `ARRAY` | `LAMBDA`

As shown in line 5 of Listing 3.1, declarations are parsed by looking at the very first token at hand, since FIRST(declaration) = type ∪ `KW_FIELD`. Listing 3.2 demonstrates how the various types of top-level declarations are constructed in the parser by the assignmentDeclaration routine. The sections below describe how each type of declaration differs from Declaration in their *Java* implementations.

Listing 3.2. 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, e);                9
   if (isField) return new FieldDeclaration(firstToken, identToken, e);          10
   return new AssignmentDeclaration(firstToken, identToken, e);                  11
}                                                                                12
```

2. Observing that FieldDeclaration is the only subclass of Declaration that may possibly make use of a field flag, assignmentDeclaration begins parsing 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 in line 3.
4. Lines 4 to 7 store the type (first) and identifier tokens, consume the equals sign, and delegate the expression's parsing to the expression routine. Unassigned declarations

will fail the match(ASSIGN) call in line 6, and will cause a compilation error. Based on the type of expression received, a corresponding subclass of AssignmentDeclaration is constructed.

8. Line 8 checks if the parsed expression is an instance of LambdaLitExpression, in which case line 9 constructs a LambdaLitDeclaration. 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.
9. If not a lambda literal, then line 10 checks if a field flag was given, constructing a FieldDeclaration if so. If control reaches line 11, then a generic AssignmentDeclaration is returned instead.

3.2.1 The AssignmentDeclaration Class

Assignment declarations are the most general an common type of top-level declaration in *Scandal*. They correspond to all variable declarations that are not *special*, neither in the sense of featuring an expression that contains 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, an instance of AssignmentDeclaration is constructed by passing a type token and an identifier token to the superclass, then storing an expression property, the latter being what differentiates assignment declarations from the abstract type Declaration.

3.2.2 The LambdaLitDeclaration Class

The LambdaLitDeclaration class is a particular case of AssignmentDeclaration where the stored expression property is of type LambdaLitExpression. By defining a new type, instances of lambda literals are more easily separated from other nodes inside a Program. As shall be seen in the the chapter on type-checking, 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 an entire chain of bindings may need to

be unraveled until a name that is bound to an expression of type LambdaLitDeclaration is found. A lambda literal declaration is constructed 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 is called lambda. Naturally, every LambdaLiteralExpression inherits from Expression. At the time a lambda literal declaration is constructed, the isField boolean property is also immediately set to true.

3.2.3 The FieldDeclaration Class

Field declarations are possibly the simplest type of declaration in the AST. They mostly exist for convenience, in order not to clutter its superclass AssignmentDeclaration, which has a somewhat more involved implementation. Field declarations are constructed by calling the superclass' constructor with exactly the same arguments that were given to the constructor of FieldDeclaration, and by setting the isField boolean property to true.

3.2.4 The ParamDeclaration Class

ParamDeclaration is basically an unchanged implementation of the abstract type Declaration. It adds no properties to it, 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. The parsing routine for a parameter declaration simply stores and consumes the type and identifier tokens, then uses those to instantiate a ParameterDeclaration class.

3.3 Subclasses of Statement

Statements and declarations are the only types of top-level constructs that *Scandal* supports. Statements can occur freely inside any scope, and are essential for the language in that they provide much of its core functionality. As seen above in Listing 3.1, parsing statements is accomplished the same way declarations are, by looking at the FIRST set

of each statement. 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. Below are the abstract syntax rules for statements.

- Statement := ImportStatement | IfStatement | WhileStatement
- Statement := AssignmentStatement | IndexedAssignmentStatement
- Statement := PrintStatement | PlotStatement | PlayStatement | WriteStatement
- ImportStatement := Expression
- IfStatement := Expression Block
- WhileStatement := Expression Block
- Block := (AssignmentDeclaration | Statement)*
- AssignmentStatement := Token.`IDENT` Expression
- IndexedAssignmentStatement := Token.`IDENT` Expression_0 Expression_1
- PrintStatement := Expression
- PlotStatement := Expression_0 Expression_1 Expression_2
- PlayStatement := Expression_0 Expression_1
- WriteStatement := Expression_0 Expression_1 Expression_2

There are four varieties of statements in *Scandal*: compiler statements, control statements, assignment statements, and framework statements. Compiler statements are restricted to import statements at the moment. Control statements are if and while-loops. Assignments simply bind a new expression to a previously declared variable. 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 constructor of its superclass, and an instance of Expression, which is stored. 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.

3.3.1 The ImportStatement Class

Import statements are the simplest type of statement in Scandal. The ImportStatement class extends Statement, but adds no properties to it. Each import statement accepts a single expression of type string containing a path in the file system to a *.scandal* file. 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. A depth-first search of the DAG pre-compiles every program in it, that is, creates 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 imported programs contains a cycle, then execution will fail at runtime.

3.3.2 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. These are executed one 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 assignment declarations or statements. Parsing blocks is very similar to parsing

top-level constructs. The basic difference is that a block is surrounded by braces, thus parsing begins by consuming the left brace. It then looks for declarations or statements until a right brace is found, throwing an error if not. The parser routines that create these declarations and statements are exactly the same that create top-level constructs. Conditional statements are parsed by consuming the first token, asking the expression method in the parser for an expression, and finally asking the block method for a block.

3.3.3 Assignment Statements

The AssignmentStatement class extends Statement by including a declaration property and, naturally, overriding both decorate and generate methods. AssignmentStatement has a subclass that specializes in assigning float values to an array at particular indices. IndexedAssignmentStatement extends AssignmentStatement by including an expression property that holds the index at which the array is to be assigned. Indexed assignment statements are parsed alongside its superclass by observing that PREDICT(indexedAssignmentStatement) = PREDICT(assignmentStatement). Differentiating between indexed and non-indexed assignments is accomplished by consuming the identifier, then checking if the next token is a left bracket. An instance of IndexedAssignmentStatement is constructed by passing along the declaration and expression properties to the constructor of the superclass, then storing the index property.

3.3.4 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 only by implementing both decorate and generate methods. As with import statements, print statements take a single expression as input, hence a print statement is needed for each expression posted to 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. The token and expression are used to instantiate a PrintStatement class, whose constructor simply calls the superclass' constructor.

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 in the IDE will oversample the array to have the specified length. Plot statements are parsed by consuming the keyword and a left parenthesis, then calling expression on the parser three times, consuming the commas in between. Finally, the right parenthesis is consumed, and an instance of PlotStatement is returned, whose constructor uses the keyword token and first expression to construct the superclass, storing the three expressions as properties.

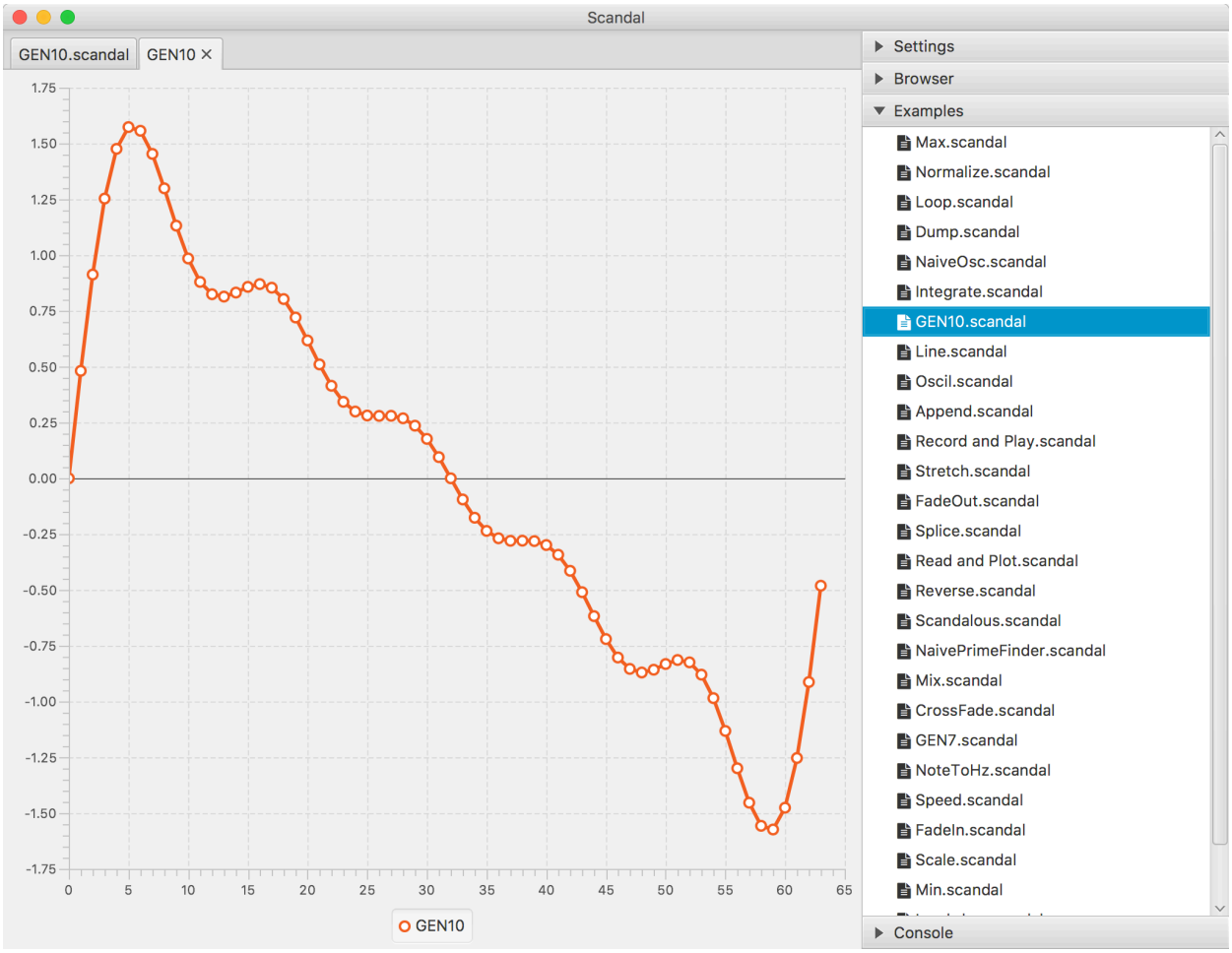


Figure 3-1. Plotting an array in *Scandal*.

Play statements take an array of audio samples and an integer number of channels, hence extend Statement by adding to it a second expression property. Parsing is done exactly the same way it is for plot statements, only there is one less expression to parse. 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 exact same pattern found in other framework statements.

3.4 Subclasses of Expression

The abstract class Expression has a very diverse family of subclasses, and extends Node by adding to it three static methods. These methods are type converters and will be discussed in the context of lambda expressions. Expression neither implements decorate or generate, nor adds any properties to Node, hence remaining a fairly general type of node. Given their diversity, and the fact that often expressions come as binary sub-trees of the AST, expressions are by very far the most difficult construct in the language to parse. The expression routine in the parser, given below in Listing 3.3, works by always looking for operator tokens, then forming a binary expression whenever applicable. Hence, even when a simple literal expression is given, parsing thereof goes through several steps before being delegated to the specific parsing routine for that particular type of literal. In particular, parsing expressions begins by assuming the expression given is a binary expression containing a lowest-precedence comparison operator. Binary expressions extend Expression by overriding its abstract methods and adding to it three properties, namely a left-hand side expression, an operation token, and a right-hand expression. Below are the abstract syntax rules for expressions, wherein the rules for operators are the same as the concrete rules, except that instead of strings of characters, the abstract counterparts are given by instances of Token.

- Expression := BinaryExpression | DerivedExpression | LiteralExpression
- Expression := ArrayExpression | FrameworkExpression | LambdaExpression

- BinaryExpression := Expression_0 Operator Expression_1
- Operator := ComparisonOperator | SummandOperator | FactorOperator

Listing 3.3. Parsing Expressions.

```
public Expression expression() throws Exception {                    1
   Token firstToken = token;                                         2
   Expression e0;                                                    3
   Token operator;                                                   4
   Expression e1;                                                    5
   e0 = comparison();                                                6
   while (token.isComparison()) {                                    7
      operator = consume();                                          8
      e1 = comparison();                                             9
      e0 = new BinaryExpression(firstToken, e0, operator, e1);       10
   }                                                                 11
   return e0;                                                        12
}                                                                    13
```

3. Lines 3 to 5 declare two expressions, one for each node of the assumed binary sub-tree of the AST whose root is returned in line 12, and an operator token.
6. Line 6 asks the comparison routine to provide the left-hand side expression, and line 7 tests in a while-loop to see whether the next token is a comparison operator. If so, the operator is stored and consumed in line 8, line 9 asks comparison for the right-hand side expression, and line 10 creates an instance of BinaryExpression with the two expressions. While the next token is still a comparison operator, the while-loop in line 7 keeps asking comparison for a new right-hand side, then substituting the current instance of BinaryExpression with another in which the left-hand side is the entire binary expression parsed last, and the right-hand side is the last expression returned from the comparison method. This method ensures binary expressions always associate equal-precedence operations from left to right.

12. Finally, line 12 returns the left-hand side expression, which may or may not be a binary expression.

The comparison routine does exactly the same expression does, only checking whether the operator token is at the precedence level of sums. Expressions are instantiated inside comparison by calling the summand routine. The summand routine, in turn, does also exactly the same as expression and comparison, only checking whether operator tokens are at the precedence level of products, which is the highest level of precedence among operator tokens in *Scandal*. Expressions are instantiated inside summand by calling the factor routine. The latter returns a leaf in the binary tree by looking at the set FIRST(expression), similarly to how declarations and statements are parsed. Some factors, however, require more than one token of look-ahead, namely those that begin with an identifier, with the exception of simple identifier expressions. Inside the factor routine, whenever a token of kind `IDENT` is seen, the token next to it needs to be considered so it can be determined whether the factor should be parsed as an indexed array, a lambda application, a lambda composition, or a simple identifier expression. Since two tokens of look-ahead are necessary and enough to parse factors, no other construct in the language requires more than two tokens of look-ahead, constructs are parsed from left to right and leftmost-derived, it follows *Scandal*'s grammar is LL(2).

3.4.1 Derived and Literal Expressions

Parsing derived expressions takes place in the factor method, as discussed above. For parenthesized expressions, factor simply looks for a left parenthesis, consumes it, recursively calls the expression method, then consumes the right parenthesis. Parenthesized expressions are crucial in order to resolve ambiguities, or force the compiler to construct a binary tree for an expression in a certain way. The expression $(1 + 1)/2$ will be parsed differently than $1 + 1/2$, however the AST has no knowledge of parenthesis, it is the way in which binary expression trees are constructed that determines how they are evaluated. Therefore there are no abstract syntax rules for parenthesized expressions, since

what they define is not a subtype, but a specific branching of a binary expression tree. Naturally, this tree may be trivial, or identical to a non-parenthesized tree. Such use of the parenthesized expression rule may have a purpose in improving readability of long mathematical expressions. Below are the abstract syntax rules for all derived expressions.

- DerivedExpression := UnaryExpression | IdentExpression
- UnaryExpression := (Token.`KW_MINUS` | Token.`KW_NOT`) Expression
- IdentExpression := Token.`IDENT`

Unary expressions are parsed by consuming the operator token, then calling expression in the parser recursively. The UnaryExpression class extends Node by adding this referenced expression as a property. The operator is stored as the firstToken property, which is inherited from the superclass. Identifier expressions extend Node by adding to it a declaration property, which is naturally a subclass of Declaration. Parsing is done in the factor method, as well. Since the *Scandal* grammar is LL(2), all of the cases in which an expression may begin with an `IDENT` token are first ruled out before an instance of IdentExpression is constructed.

All literal expressions extend Expression by adding no properties to it, since all that is needed is a value for the expression, which is in turn retrieved from the inherited firstToken property itself. For integers and floats, there is a risk the user will declare a number that is possibly too long. These cases are handled by the scanner at the earliest stage possible, and an informative compilation error is thrown if a bad number is given. In all literal expressions, types are assigned at their onset in the constructor, as type-inference mechanisms are unnecessary.

3.4.2 Array Expressions

For all but the ArrayItemExpression rule, array expressions are parsed in the factor method by looking at their first token. In the case of array item expressions, since FOLLOW(`IDENT`) = ε | `LBRACKET` | `LPAREN` | `DOT`, one more token of look-ahead is needed, namely a left brace. The other cases are those where a left parenthesis is seen, which

correspond to lambda applications, and those in which a dot is seen, corresponding to lambda compositions. After consuming the first token, and possibly the second, parsing follows the same procedure as above, by calling the expression routine recursively, and abstracting away parenthesis, brackets, commas, and keywords. Below are the abstract rules for array expressions.

- ArrayExpression := ArrayLitExpression | ArrayItemExpression
- ArrayExpression := ArraySizeExpression | NewArrayExpression
- ArrayLitExpression := (Expression)$^+$
- ArrayItemExpression := IdentExpression Expression
- ArraySizeExpression := Expression
- NewArrayExpression := Expression

Array literal expressions extend Expression by adding an ArrayList of expressions, and naturally overriding its abstract methods. Its type is set to Types.ARRAY in the constructor, similarly to how literal expressions are handled. Array item expressions extend Expression with two properties, namely an identifier expression that references the array, and an expression of type integer that stores the index information. Instead of storing an IDENT token, an IdentExpression is instantiated while constructing an ArrayItemExpression. The reason for that is to simplify decoration and code generation by delegating those tasks to the IdentExpression class, rather than repeating code inside ArrayItemExpression. Like with literal expressions, the type is always known to be float, hence set accordingly in the constructor. Array size expressions extend the superclass by including an expression of type array. Like before, the type is set to int in the constructor. The last type of array expression deals with creating a new array of zeros with a specified length. An instance of NewArrayExpression extends Expression by adding to it a size expression of integer type. Its type is too set to array in the constructor.

3.4.3 Framework Expressions

Parsing framework expressions is done in the factor method by simply looking at the first token at hand and constructing the appropriate subclass of Expression. Parenthesis, commas, and keywords are naturally all abstracted away. The abstract rules for framework expressions are omitted since they follow the usual pattern of listing all parameters to a framework routine as expressions. The simplest case of a framework expression is that of pi expressions. They extend Expression only by implementing its abstract methods. Like with literal expressions, its type is set to float right at construction, and decoration therefore does nothing. Cosine expressions are absolutely fundamental to audio signal processing, and are one of the cases where a pure Scandal implementation might not be worthwhile. They extend Expression by adding a phase property, which is a subclass of Expression of type float. Since cosine expressions always have type float in *Scandal*, the type is set in the constructor. Power expressions provide a convenient way to compute exponential expressions, and extend Expression with two properties, namely base and exponent. Floor expressions are identical to cosine expressions, only naturally differing in functionality. Read expressions provide a hook to the audio engine's framework.generators .WaveFile class, whose main purpose is to parse the contents of a *.wav* file in the file system and return them as an array of floats. ReadExpression extends the superclass by including two properties, a string path, and an integer format, the latter specifying a channel count. The format property is not overloaded, and accepts only integers. Finally, record expressions connect the DSL with the audio engine's framework.generators.AudioTask class. They extend Expression by adding a duration property, which has type integer but is overloaded to accept floats, as well.

3.4.4 Parsing Lambda Expressions

Parsing lambda literal expressions relies on looking at the first token at hand. Both variants feature a list of parameters separated by arrows, and each parameter is an instance of ParamDeclaration, hence begins with a type token. Since these are the

only two types of expression that begin with a type token, the first token is enough to determine a lambda literal expression. While creating an array of parameter declarations, arrows are abstracted away until no more type tokens are seen. At this point, if a left brace is seen, a lambda block is instantiated and used to construct an instance of LambdaLitBlock. Otherwise, expression is called in the parser and the result used to instantiate a LambdaLitExpression. In fact, LambdaLitBlock is a subclass of LambdaLitExpression.

Parsing applications and compositions requires a second token of look-ahead, since both expressions begin with an identifier token. As previously discussed, indexed arrays and identifier expressions are the other two constructs that also begin with an identifier token. What distinguishes all four is the token that follows: indexed arrays are followed by a left bracket, lambda applications are followed by a left parenthesis, lambda compositions are followed by a dot, and identifier expressions are not followed by brackets, parenthesis, or dots. Once a parenthesis is found, the identifier is stored in an instance of IdentExpression, both parenthesis and all commas are abstracted away, and an instance of LambdaAppExpression is constructed with the identifier expression and a list of expressions, one for each given argument. If a dot is seen, however, all the subsequent dots are abstracted away while building a list of IdentExpression. When no more dots are seen, the next token may be a parenthesis or not. If so, an instance of LambdaAppExpression is used to construct a LambdaCompExpression, otherwise a null pointer is passed *in lieu* of a LambdaAppExpression to the constructor of LambdaCompExpression. Below are the abstract syntax rules for lambda expressions.

- LambdaExpression := LambdaLitExpression | LambdaLitBlock
- LambdaExpression := LambdaAppExpression | LambdaCompExpression
- LambdaLitExpression := ParamDeclaration$^+$ Expression
- LambdaLitBlock := ParamDeclaration$^+$ ReturnBlock
- ReturnBlock := (AssignmentDeclaration | Statement)* Expression

- LambdaAppExpression := IdentExpression Expression$^+$
- LambdaCompExpression := IdentExpression$^+$
- LambdaCompExpression := IdentExpression$^+$ LambdaApp

3.4.4.1 Lambda Literal Expressions

The LambdaLitExpression class extends Expression by including a list of parameter declarations and a return expression, as seen above. In addition to that, it holds an integer property named lambdaSlot, which is used for naming the lambda in bytecode. Every lambda necessarily has at least one parameter, and necessarily returns a non-void expression. While constructing a lambda literal with expression, its type may be incorrectly inferred by the constructor of Node to be that of the expression's first token. Hence the expression type is set to Types.LAMBDA in the constructor of LambdaLitExpression. The LambdaLitBlock class is a type of lambda literal expressions in which an entire block of declarations and statements is featured before a return expression is given. Similarly to the superclass, the type of the expression is set to Types.LAMBDA already in the constructor. Lambda literals with blocks subclass LambdaLitExpression by adding to it a block property, that in turn is an instance of ReturnBlock. Return blocks extend Block by adding to Block a return expression property, and this return expression is used in the constructor of a LambdaLitBlock as an argument to construct the LambdaLitExpression super class.

3.4.4.2 Lambda Applications and Compositions

An application of a lambda expression in *Scandal* corresponds in the AST to an instance of the LambdaAppExpression class, which extends Expression by overriding its abstract methods and including four properties, namely an identifier expression called lambda, a list of expressions which are the arguments applied to this lambda, an immutable integer property named count, and a reference to an instance of LambdaLitExpression called lambdaLit. While constructing a lambda application expression, the size of the argument list is stored in the count property, as parameters might be added to this list in

the decoration phase. The application type is also set to Types.LAMBDA, although that too might change if the application is not partial. Lambda compositions correspond to instances of LambdaCompExpression, which extend Expression by including two properties, an array of identifier expressions, and a lambda application expression. The latter can be null, as explained in the parsing process. If so, the type of the composition expression is set to Types.LAMBDA while constructing, otherwise setting a type is deferred until decoration.

CHAPTER 4
CHECKING TYPES AND DECORATING THE AST

This chapter presents the type-checking rules for *Scandal* along with the challenges involved in enforcing these rules in the language's *Java* implementation. Although types are static in *Scandal*, many productions are overloaded to accept types that can be easily cast to the required type. Moreover, operators are in their majority overloaded to accept mismatching types, in which case the rules defined in this chapter describe what types should be expected in return. Given that every node in the AST inherits from the Node abstract base class, type-checking takes place alongside other bookkeeping tasks in every node's decorate method. This routine is always called on a node by the node's parent. In particular, inside an instance of Program, type-checking is completely delegated to each node in its node array. More precisely, the decorate routine iterates over the node array and, for each node, calls node.decorate, passing along the symbol table instantiated by the compiler. Every direct child node of a program, in turn, delegates decoration of their child nodes, and with very few exceptions, delegation is the default behavior for AST decoration and type-checking.

4.1 Decorating Declarations

Decorating an instance of AssignmentDeclaration is a bit tricky. There are basically two tasks: checking that the variable is not being re-declared, and checking that the type of the given expression matches the type of the declaration. The declaration's type is set by the constructor of Node based on its first token. The decorate routine starts then by checking the top of the stack of symbol tables to see if there is no name clash, then inserts into the symbol table a key of identToken.text with value this. The next step is to assign a slot number to this variable, which is done by maintaining a property slotCount inside SymbolTable. The slotNumber property is set to the current slot count, and the latter is increased. A call to expression.decorate causes the expression to decorate itself and have a non-null type in most cases thereafter, except when the expression is an instance of

LambdaAppExpression, in which case a roadblock in the type-checking process occurs. Listing 4.1 provides a small snippet of *Scandal* code to illustrate the situation.

Listing 4.1. 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, there is no mechanism in the language to tell what is the parameterized type of that lambda expression. 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. In Listing 4.1, an instance of ParamDeclaration 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 or statement. In line 3, x is applied to f and the result stored in val. Since both x and val have type float, it can be inferred 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 parameter types of a lambda expression cannot be inferred 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 an application of a lambda declared as a parameter of a lambda literal, instead of expecting the call to expression.decorate to define a type for the expression, the AssignmentDeclaration class itself needs to decorate the expression. The process is very simple: after calling expression.decorate, a check is made to

determine if the expression is an instance of LambdaAppExpression. If so, the declaration of that application's lambda property is retrieved. If the declaration is an instance of ParamDec , then the expression at hand must be the application of a lambda declared as a parameter of a lambda literal expression, since parameter declarations only ever exist in the context of lambda literals. The decorate method 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. Finally, the expression property will have to have a type, which is checked against the declaration type. Naturally, in the special case the expression was decorated for being an application of a lambda parameter, this test never fails. If it does, a compilation error is thrown.

- AssignmentDeclaration:
  - + Must not be declared more than once in the same scope
  - + Type = Expression.type

Decorating and type-checking field declarations is easier and involves checking all symbol table scopes to see if the variable is not being redeclared. Naturally, there is only ever one scope to check, the zeroth. The variable is then inserted into the symbol table with a key given by identToken.text and a value of this. A call to expression.decorate is made, 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 have the same type by construction, here a compilation error occurs if the program tries to store, say, a float into an integer field. This is not inherent to the JVM, however, which would in fact accept that a different type be stored in a local variable slot or field. In *Scandal* this is illegal, hence decoration checks 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.

- FieldDeclaration:
  - + Must be declared in the outermost scope
  - + Must not be declared more than once
  - + Type = Expression.type

Lambda literal declarations have very different implementations of the Node abstract methods than those of its superclass 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. The decorate routine checks the entire stack of symbol tables to see whether the variable is 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 in the outermost scope, hence the stack only contains a single symbol table when the call to decorate is made. The next step is to insert the identifier into the symbol table, associating to it the instance of LambdaLitDeclaration at hand. Finally, calling lambda.decorate delegates decoration of the lambda expression to the LambdaLitExpression instance. The types of the declaration and expression are always equal to lambda, set in the respective constructors, and hence never need to be checked.

- LambdaLitDeclaration:
  - + Must be declared in the outermost scope
  - + Must not be declared more than once
  - + Type = Types.`LAMBDA`

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 shall be seen momentarily.

- ParamDeclaration:
  - + Must not be declared more than once

4.2 Decorating Statements

Decoration of import statements begins by checking that the current scope number in the symbol table is zero, since the linker routine in the compiler only looks for imports in the outermost scope of each pre-compiled program. If not, an error is thrown. Next, a call to expression.decorate is made and, after the expression has been decorated with a type, a check is performed to see whether that is indeed a string, throwing an exception if not.

- ImportStatement:
  - + Must be stated in the outermost scope
  - + Expression.type = Types.`STRING`

Decorating assignment statements involves looking for a declaration value whose key corresponds to the identifier held 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, an error is thrown. Decoration of the expression property is delegated to the corresponding subclass of Expression, as usual. The exact same caveat with type inference in instances of LambdaAppExpression applies here, so a check is made to determine whether the given expression is a lambda application of a lambda parameter inside a return block. If so, the AssignmentStatement class decorates the lambda application expression with its declaration's type. All that is left is to check if the type of the declaration corresponds to the type of the expression, and an error is thrown if there is a mismatch.

- AssignmentStatement:
  - Must have been declared in some enclosing scope
  - Declaration.Type = Expression.type

Decoration of an indexed assignment statement 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, a check must be made to determine if the declaration associated with the identifier has indeed type array, otherwise an error is also thrown. Decoration of the index property is delegated to the appropriate subclass of Expression, as usual. Contrary to most descendants of *C*, however, *Scandal* does allow arrays to be indexed by expressions of type float, in which case the integer part of the number is taken in the generation phase. Thus while checking the type of index, an error is thrown if is neither an integer nor a float. The expression property is decorated in a similar way, however here too care must be taken with higher-order functions. A check for those is performed exactly the same way it is in the superclass, decorating the expression property as needed. When decorating a lambda application inside a lambda block, the expression's type is naturally always set to Types.FLOAT. Similarly to the index property, the expression property is overloaded to accept integers, and in that case a conversion to float is performed in the generation phase before storing the expression's value.

- 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`

Decorating if and while-statements is identical. The expression property is asked to decorate itself, then checked to see whether it is of type boolean, an error being throw otherwise. After that, the block is asked to decorate itself. Decorating a block involves introducing a new scope by calling symtab.enterScope, asking each node in the block's nodes

array to decorate itself, then calling symtab.leaveScope. Import statements are disallowed in blocks, since the compiler only ever looks for those in the zeroth scope, and so are lambda literals and fields. If a global variable declaration or an import statement is given to a block, neither the parser nor the block will take 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.

- IfStatement:
  - Expression.type = Types.`BOOL`
- WhileStatement:
  - Expression.type = Types.`BOOL`

Framework statements are all decorated in a similar fashion by delegating decoration to each parameter, then checking types. Decoration of a print statement asks the expression to decorate itself, after which a check is performed to determine if the expression's type is either array or lambda. In both cases an error is thrown. Decoration of a plot statement is accomplished simply by asking each expression to decorate itself, then type-checking the first to be a string, the second to be an array, and overloading the third to accept integers and floats. Decoration of play statements follows the same pattern, and requires that the first expression be an array, and the second be an integer. The same pattern applies to write statements, which require that the three expressions be of type array, string, and integer, respectively.

- 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`

4.3 Decorating Expressions

Decorating and generating binary expressions is somewhat involved, mostly due to the great variety of operators and their supported types. In addition, operators in *Scandal* are overloaded to a certain extent, as to provide type polymorphism in binary expressions, which adds to the number of cases that must be considered while decorating and generating expressions, but also make the DSL more concise and readable. Decorating a binary expression begins with asking both expressions to decorate themselves, after which their types will be non-null. Given the binary-tree nature of these expressions, this is accomplished recursively, hence decorate may be calling on itself if one of the sub-expressions is a binary expression. For this reason, decorate needs to have a type before returning. In fact, almost all that is done inside decorate is to go over the different combinations of operators and expression types to determine what the overall type of the binary expression is. If given two expressions in any combination of integers or floats, arithmetic can be performed on those numbers. Note that arithmetic operators do not always have the same precedence amongst themselves. There are two sub-cases. If both expressions are of type integer, then the resulting binary expression is decorated with an integer type. Otherwise, if at least one of the expressions has type float, the whole binary expression will have type float. If instead any combination of integers, floats, or booleans is given, then logical or comparison operators can be used, remembering that booleans are implemented in bytecode as integers, and that logical operators also have different

precedence levels. In both cases, the binary expression will be of type boolean. Before returning, the caveat of having applications of lambdas that were declared as parameters of another lambda must be considered. Given that there is no parameterization of types in *Scandal*, these lambdas must be applied alone first in an assignment declaration or statement, which is the current inference mechanism for such parameterized types. So a check is performed on both expressions to see whether any of their declarations is a subclass of ParamDeclaration, and an error is thrown if that is the case. The very last thing we do while decorating a binary expression is to check if the binary expression's type is still null. If so, all tests above have failed. Since the binary expression must be decorated with a type, as mentioned above, an error is thrown whenever all type-checking cases fail.

- BinaryExpression:
  - ArithmeticOp := `MOD` | `PLUS` | `MINUS` | `TIMES` | `DIV`
  - (`INT` | `FLOAT`) ArithmeticOp (`INT` | `FLOAT`) → `FLOAT`
  - `INT` ArithmeticOp `INT` → `INT`
  - ComparisonOp := `AND` | `OR` | `EQUAL` | `NOTEQUAL` | `LT` | `LE` | `GT` | `GE`
  - (`INT` | `FLOAT` | `BOOL`) ComparisonOp (`INT` | `FLOAT` | `BOOL`) → `BOOL`

Decoration of unary expressions is simple, following the usual path of asking the expression property to decorate itself, after which type-checking proceeds as follows. If given a `MINUS`, and the expression is neither an integer not a float, an error is thrown. Else, if given a `NOT`, and the expression is not a boolean, an error is also thrown. Note that unary operators are not overloaded in *Scandal*, and only apply to their specific types. Before returning, the unary expression is naturally decorated with the same type as the given expression. Type-checking identifier expressions is similar to assignment statements in that a check for a name that has already been declared is made, and an error is thrown if it has not. That is done by asking the symbol table for the declaration value associated to the given identifier key, after which the declaration is stored in the declaration property. The last step is to set the expression's type to be the same as the declaration's.

- UnaryExpression:
  - Token.`MINUS` Types.`INT` → Types.`INT`
  - Token.`MINUS` Types.`FLOAT` → Types.`FLOAT`
  - Token.`NOT` Types.`BOOL` → Types.`BOOL`
- IdentExpression:
  - Variable must have been declared in some enclosing scope
  - Type = Declaration.type

Decorating literal expressions is trivial. In all of them, decorate is overridden for being abstract, but nothing is done there, since all that is needed is to decorate the expression with a type, which is always known, hence done in each class' constructor.

- IntLitExpression:
  - Type = Types.`INT`
- FloatLitExpression:
  - Type = Types.`FLOAT`
- BoolLitExpression:
  - Type = Types.`BOOL`
- StringLitExpression:
  - Type = Types.`STRING`

Decoration of array expressions follows the usual route. In all of them, the expression type is always known, hence set in the constructor. For array literal expressions, the decorate method iterates over the array of expressions, asking them to decorate themselves. For each of them, a check is made to determine that they have type integer or float, throwing an error otherwise. Ultimately, though, all values are cast to float when creating the array. Decoration of array item expressions asks the array property to decorate itself, then checks if it has type array, throwing an error if not. The same is done for the index property, only checking that it indeed has type integer or float. For array size expressions,

decoration is done only by asking the array property to decorate itself, then checking it indeed has type array. For new array expressions, decorate asks the size expression to decorate itself, then checks it is either an integer or a float.

- ArrayLitExpression:
  - (Expression)$^+$.type = Types.`INT` | Types.`FLOAT`
  - Type = Types.`ARRAY`
- ArrayItemExpression:
  - Expression_0.type = Types.`ARRAY`
  - Expression_1.type = Types.`INT` | Types.`FLOAT`
  - Type = Types.`FLOAT`
- ArraySizeExpression:
  - Expression.type = Types.`ARRAY`
  - Type = Types.`INT`
- NewArrayExpression:
  - Expression.type = Types.`INT` | Types.`FLOAT`
  - Type = Types.`ARRAY`

Framework expressions are also easily decorated, and type-checking arguments follows the usual pattern. For pi expressions, nothing is done inside decorate. Cosine expressions overload the phase property to accept integers and floats. In power expressions, both base and exponent properties are overloaded to accept integers and floats. Read expressions take a format argument that is of type integer and is not overloaded, similarly to play statements, which also have an integer-only property describing the channel count. Record expressions, on the other hand, do overload the duration property to accept integers as well as floats.

- PiExpression:
  - Type = Types.`FLOAT`

- CosExpression:
  - Expression.type = Types.`INT` | Types.`FLOAT`
  - Type = Types.`FLOAT`
- PowExpression:
  - Expression_0.type = Types.`INT` | Types.`FLOAT`
  - Expression_1.type = Types.`INT` | Types.`FLOAT`
  - Type = Types.`FLOAT`
- FloorExpression:
  - Expression.type = Types.`INT` | Types.`FLOAT`
  - Type = Types.`FLOAT`
- ReadExpression:
  - Expression_0.type = Types.`STRING`
  - Expression_1.type = Types.`INT`
  - Type = Types.`ARRAY`
- RecordExpression:
  - Expression.type = Types.`INT` | Types.`FLOAT`
  - Type = Types.`ARRAY`

4.4 Decorating Lambdas

Decorating lambda literal expressions is not too complicated. Because new local variables are introduced with a lambda literal expression's list of parameters, decoration begins by pushing a new scope onto the symbol table stack. This allows names to be declared such that they will not clash with names already in the zeroth scope, however does not protect the lambda literal expression from *seeing* the zeroth scope. Given that names in the bottom scope are local variables in the context of the *Java* class' `run` method, and lambda bodies are scoped in an altogether different method within the same *Java* class, any name declared in the zeroth scope is actually invisible to the lambda body.

Of course, unless these names were declared as fields. After introducing a new scope, decorate iterates over the parameter list, assigning to each parameter a slot number, and asking each parameter to decorate itself. The parameter slot numbers are needed for local variables and are assigned sequentially from zero to the size of the list minus one. After that, the return expression is asked to decorate itself, and the symbol table is asked to leave the previously introduced scope. Finally, the lambdaSlot property is set to be equal to the symbol table's lambdaCount property, and the latter is incremented by the size of the parameter list. The reason for assigning a lambda number to each lambda literal is because lambda expressions are accessed by name in bytecode, and the naming convention is the word *lambda* followed by a number. In addition, the Function interface naturally curries the parameters in a lambda expression and, per *Scandal*'s design choice, parameters in a lambda have a traditional right-associative currying. This means there is a method body that responds to all parameters, another that responds to all but the first, and so on until the last method body that responds only to the very last, rightmost parameter. Therefore the leftmost body is associated to the lambdaCount slot, the one to its right is associated to the lambdaCount+1 slot, and the rightmost body is associated to the lambdaCount + params.size−1 slot. In effect, that is exactly how these curried methods are accessed as partial applications, separately.

Decorating lambda literals with blocks is similar, but a bit more complicated than decorating lambda literals with expressions. As before, a new scope is introduced and decorate iterates over the parameter list, assigning parameters slot numbers and asking them to decorate themselves. Before asking the block to decorate itself, however, the fact that the block may contain declarations must be considered. Since declarations inside a return block are local variables to a method, they need slot numbers. The symbol table passed along has a running slot count for variables that are local to the run method, however, hence a temporary change to this value is needed. Every local variable inside the lambda block is assigned an incremental slot number that is offset by the number of

arguments given in a lambda application. If, for example, a lambda has three parameters and three local variables, the parameters will have slot numbers ranging from zero to two, and the local variables will have slot numbers ranging from three to five. So the current slot count is stored in a local variable, changed to the size of the parameter list, the block is decorated, and finally the running slot count in the symbol table is set back to its previous state, using the value previously stored. After that, the topmost scope is popped, and the lambda count is increased in the symbol table by the size of the parameter list, similarly to lambda literals with expressions. Decoration of return blocks differ from the Block superclass in that they do not introduce a new scope, since that task is accomplished by the lambda literal that contains the return block. It also differs in that the return expression is asked to decorate itself.

Decorating a lambda application is not at all straightforward. The main complication is finding the declaration of a lambda literal that contains the body of the method to which the lambda application's list of arguments should be applied. In other words, the name reference held by the lambda property may not necessarily point to the declaration of a lambda literal, in which case a whole chain of name references needs to be unraveled until a declaration whose expression actually contains a method body is found. Decoration begins by asking lambda, which is an identifier expression, to decorate itself, after which it will be decorated with a declaration. A check is made to determine if the declaration is a parameter declaration, in which case decoration can go no further, since this is an application of a lambda that is a parameter of another lambda. In this case, the type of the application expression, as well as the parameterized type of its declaration, is inferred and set by an assignment declaration or statement, as discussed previously. All that is left to do then is to iterate over the parameter list, asking for each parameter to decorate itself, and return.

If the lambda declaration is not a parameter declaration, it might still not point to the declaration of a lambda literal, but at least it can be determined that it either lives inside

run, or is a field. Decoration then traverses the AST from the application node backwards to its ancestors until a lambda literal is reached. There are basically three cases. In the first, the declaration of lambda points to an identifier expression. This case happens whenever a lambda expression is copied locally to a variable. The declaration then is set to point to the declaration of *that* identifier expression, and traversal keeps looking, moving onto the previous ancestor. In the second case, the declaration points to a lambda composition. A lambda composition holds an entire list of identifier expressions, all of which may or may not point to the declaration of a lambda literal. In this case, there may be many parents to a node, but the very last is picked. Whatever lambda literal to which it ultimately points, must have a parameter list that matches this application's argument list, which is exactly what is needed in order to decorate this lambda application. If neither an identifier nor a composition, then the declaration must be pointing to another application. In this case, it cannot be known yet whether this application is a partial application, but it can be known for sure that the application to which the declaration points indeed is a partial application, for if it were total, this application could not exist, as there would be no more parameters to be fixed. But given this situation, it can be inferred that this application's argument list must be smaller that the parameter list of the lambda literal sought, since some partial application already fixed some of its parameters, and this application is a step further down the chain. Decoration then iterates over the argument list of the application to which the declaration points from right to left, that is, from its last down to its first element, and prepends to this application's list of arguments any of those arguments that its list does not yet contain. A sanity check is then performed to determine if some of these arguments have not already been added.

After traversing the AST without errors, the declaration pointer is guaranteed to have arrived at the declaration of a lambda literal, hence the latter's expression is stored in the lambdaLit property for further use during generation. Only at this point does decoration of the argument list actually begin. The decorate method iterates over the argument list,

however the original argument list might have grown in the traversal process. That is why the size of the original list was stored in the count property, so decorate will restrict to *only* the parameters that were originally in the list. That is accomplished by offsetting the iterator by the current argument list size minus its original size, which represents exactly the number of parameters that might have been added during traversal. By prepending arguments to the argument list, the latter now has been made the exact same size as the parameter list in lambdaLit, but there is no actual need to decorate arguments that were not meant for this particular application, hence why the list is offset. Moreover, the index of each original argument in the argument list now aligns properly with the corresponding parameters in the lambdaLit parameter list. Decoration then goes over the original parameters asking them to decorate themselves and checking that their types are the same as the types of the corresponding parameters in lambdaLit. Finally, a check is made to see if the type of this lambda application expression is indeed lambda. That is done by checking whether the current count of the argument list is the same as the parameter count in lambdaLit. If so, then the application has fixed all the parameters in lambdaLit, hence the type of the lambda application is set to the type of the lambdaLit's return expression. If there are less arguments than lambdaLit has parameters, then this is a partial application, hence the current type definition is kept.

Decorating a lambda composition consists at the moment of just asking each composed lambda to decorate itself, then checking whether the lambdaApp property is null and, if not, asking it to decorate itself. The lambda application will in turn decorate its parameters and check that input types are correct. The last step is to set the type of the entire composition expression to the return type of its lambdaApp, if the latter is not null. This is an incomplete implementation, and the reasons for that shall be addressed in the following chapters.

- LambdaLitExpression:
  - \+ Type = Types.`LAMBDA`

- LambdaLitBlock:
  - Type = Types.`LAMBDA`
- LambdaAppExpression:
  - Expression_0.type = Types.`LAMBDA`
  - The type of each argument must match the type of the corresponding parameter in the original lambda literal
  - Type = Types.`LAMBDA` if the number of arguments given is less than the total number of parameters
  - Type = the original lambda's return type if the number of arguments matches the total number of parameters
- LambdaCompExpression:
  - (Expression)$^{+}$.type = Types.`LAMBDA`
  - The return type of each expression must match the input type of the next
  - Argument types must match the first expression's input types
  - Type = Types.`LAMBDA` if no arguments are given
  - Type = the return type of the last expression if arguments are given

CHAPTER 5
GENERATING TARGET CODE

This chapter discusses the mechanisms necessary to translate a decorated *Scandal* AST into *Java* bytecode. This target language is characterized by running on the *Java Virtual Machine*, a multi-platform interpreter and just-in-time compiler. Bytecode is written as text, but ultimately converted into a binary representation in order to facilitate execution by the JVM. In order to compile a *Scandal* AST into bytecode text and binary representations, the *Scandal* compiler makes use of a *Java* library called org.objectweb.asm, or ASM for short. This library provides many facilities, including an IDE plug-in that takes a *Java* class and displays its corresponding bytecode implementation, as well as the necessary ASM calls to generate that bytecode. In fact, this method represents most of the work flow in implementing *Scandal*. The entire work flow is roughly:

1. Make choices for the language's concrete syntax.
2. Introduce new symbols to the Token and Scanner classes, as needed, and test.
3. Write a corresponding AST class, parsing routine, and test.
4. Describe type-checking rules, implement decorate inside the AST class, and test.
5. Write *Java* code with the same functionality, and copy its ASM plug-in output.
6. Paste the ASM plug-in output inside the AST class' generate method, substituting names, constants, and method signatures by properties belonging to the AST class, make adjustments, and test.

Often adjustments and optimizations are necessary, as the ASM plug-in output contains many unnecessary calls to its API. These are usually routines that add comments and line numbering to the bytecode class, but that can cause significant bloat in the compiler. Although not necessary, minor optimizations and code re-orderings are usually made, such as duplicating the top of the stack instead of making another method call, for example. Also, for every *Scandal* type that is overloaded, a check is inserted with the necessary type conversion, as types are not overloaded in the JVM in general.

The *Java Virtual Machine* is stack-based, that is, operands are stored in a stack data structure. Operations consist mainly of pushing operands onto the stack, then calling a method. If the method is void, it will pop from the stack the operands it takes as arguments and leave the stack in the same state it was before those operands were pushed. If the method returns something then it will leave, in addition, a result on top of the stack, that is, after popping its required arguments. Stacks in the JMV are further bundled into entire frames, which contain data and local variables the stack can access. Bytecode is oblivious to most details pertaining to the implementation of frames, except for providing the frame's local variable count and maximum stack height. Naturally, these *details* are fundamental for memory allocation in the JVM. Fortunately, computing these values is delegated entirely to the ASM library, as previously discussed.

5.1 Generating a Program

Generating bytecode in the Program class is a lot more complex than the corresponding decoration phase, since the overall structure for the entire underlying *Java* class needs to be provided. This global task is accomplished inside the generate method by creating an instance of org.objectweb.asm.ClassWriter. The latter, which is stored locally in a property called 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, the JRE is set to version 1.8, access to the class is made public, and the class is defined as a subclass of java.lang.Object that implements the java.lang.Runnable interface. This is all accomplished by calls to the ASM API, which manages writing a bytecode class with the appropriate instructions. Next, three instances of org.objectweb.asm.MethodVisitor are acquired by calling cw.visitMethod, one for each method in the *Java* class. The methods are namely init , run, and main.

The init method basically goes through the node array and, if the particular node is an instance of LambdaLitDeclaration, a call is made to node.generate, passing the appropriate instance of MethodVisitor and the compiler's symbol table as arguments. What the generate

method does inside a LambdaLitDeclaration is somewhat complicated, and its explanation deferred to the moment the LambdaLitExpression class is discussed. Before visiting run, generate goes once again over all nodes in the node array and, if they are either an instance of LambdaLitDeclaration or an instance of FieldDeclaration, it calls cw.visitField. The latter method creates fields in the Java class, which correspond to global variables in the *Scandal* program. Every field is marked as static, since no use is made in *Scandal* of *Java*'s object-oriented paradigm.

In addition, lambda fields are marked as final, as previously discussed, and cw is passed along to the instances of LambdaLitExpression for which field declarations are being created, asking them to create method bodies for their respective lambda literal expressions. This is accomplished inside each lambda literal expression by an overloaded generate method, which takes, instead of a MethodWriter, an instance of ClassWriter, namely cw. Each lambda literal expression uses cw to create its own MethodWriter, which will write the lambda's corresponding method to the *Java* class. These instances of LambdaLitExpression are accessed through the corresponding lambda property inside a LambdaLitDeclaration class, 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, generate goes yet once more over the array of nodes, and this time it generates any node that is *not* an instance of LambdaLitDeclaration, for obvious reasons. It does, however, visit instances of FieldDeclaration, since cw.visitField only created the fields, but never assigned any values to them. Since unassigned declarations are only allowed in *Scandal* when declaring lambda parameters, there is always some value to assign to those fields at initialization, and generate inside FieldDeclaration takes care of exactly that. Finally, generate visits the main method in the *Java* class, which is shown in Listing 5.1.

Listing 5.1. Using the ASM framework to construct a main method.

```
private void addMain(ClassWriter cw, SymbolTable symtab) {                1
```

```
    MethodVisitor mv = cw.visitMethod(ACC_PUBLIC + ACC_STATIC, "main",          2
        "([Ljava/lang/String;)V", null, null);                                  3
    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
```

2. This is the standard main method in *Java*, which is always public and static, takes an array of strings, and returns nothing. The bytecode syntax for arrays is that of a left bracket, followed by the type. Line 2 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 a return type. Hence a void method that takes a String[] in *Java* becomes ([Ljava/lang/String;)V, where the left bracket means an array of whatever type follows, and the colon separates arguments. Naturally, java/lang/String is a string, and V stands for the void type.
4. The JVM is stack-based, so line 4 creates a new instance of the *Java* class, whose name is stored in the compiler's symbol table, and the result is left on top of the JVM stack.
5. Line 5 duplicates whatever is on top of the stack, since the newly created *Java* class will be needed twice, namely for a call on it to init, and another on 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.
8. Finally, a return statement is added to the main method's body, which is omitted in void *Java* methods, but required in bytecode.
9. A bytecode method requires that the maximum number of elements the stack will have, as well as the total number of local variables in the method be computed.

ASM will do that automatically by passing ClassWriter.COMPUTE_FRAMES as an argument to the constructor of ClassWriter. The two arguments to mv.visitMaxs are the maximum stack size, and the total number of local variables. Zeros are passed here since ASM is computing them later, but the call must be made nonetheless.

5.2 Generating Declarations

Overriding the generate method inside an assignment declaration is straightforward, and consists of making a call to expression.generate, which causes the expression value to be left on top of the JVM stack, after which a switch statement determines the expression type, and uses the appropriate JVM instruction to store the variable. For integers and booleans it uses ISTORE, for floats it uses FSTORE, and for all other types in *Scandal* it uses ASTORE. Similarly, the generate method in lambda literal declarations calls lambda.generate, which causes a lambda literal expression to be left on top of the JVM stack. This expression is then bound to the identToken.text property using the PUTSTATIC bytecode instruction. The overridden generate method in field declarations is identical to that of LambdaLitDeclaration. In the generate method, parameter declarations do nothing, since there is no value that can be bound to the declaration's identifier at the moment. Naturally, these values will exist in the context of a lambda application.

5.3 Generating Statements

Even though import statements belong to the AST, no bytecode is ever generated for them. The generate method still needs to be implemented for being abstract, but simply nothing is done there. The generate method for assignment statements, on the other hand, 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, an expression value is always left on top of the JVM stack. Whenever assigning to a parameter variable inside a lambda literal expression or block, if the expression's type corresponds to a primitive type in *Scandal*, this primitive sitting on top of the JVM stack is wrapped into the appropriate *Java* class. That is done by testing whether the declaration property inside the assignment statement class is an instance of ParamDeclaration, which immediately tells the variable is a parameter to a lambda. Otherwise, a test is performed to determine if the declaration is an instance of FieldDeclaration, in which case the PUTSTATIC bytecode instruction is used to assign the expression to the variable. If both tests fail, then the variable must be local, either in the context of run or local to a lambda body, hence the appropriate store call is made based on the variable type, similarly to assignment declarations. Here a slot number is needed, but that is conveniently stored in the declaration property, which computed it during its decoration phase.

Generating an indexed assignment statement is done a little differently from the AssignmentStatement superclass in it involves loading the array first on top of the JVM stack, which is done by asking the declaration property for a slot number and calling ALOAD. After that, the index property is asked to generate itself. If the index is a float, the F2I instruction is called, which effectively pops the float and pushes its integer part on top of the JVM stack. Similarly, the expression property is asked to generate itself and, if needed, converted to float by calling the I2F JVM instruction. Finally, the FASTORE instruction is used to assign the value on top of the stack to the array at the desired index, eventually popping the three topmost elements from the JVM stack. The generate method naturally differs between both types of conditional statement. The entire process of generating an if-statement is described in Listing 5.2.

Listing 5.2. 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
```

2. Line 2 asks the expression to generate itself, after which its value will be left on top of the JVM stack.
3. Line 3 loads a constant of value true, given by the ICONST_1 instruction, on top of the stack. Interestingly, integer constants are used in the bytecode implementation of booleans, but not in *Java*. Nor in Scandal, mostly for clarity.
4. An instance of org.objectweb.asm.Label is then created which, upon a call to mv.visitLabel in line 7, creates a bytecode label instruction.
5. Before visiting the label, though, both elements on top of the stack are compared by making a call to an IF_CMPNE jump, and giving it the label 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 at least one of them has value true, namely the one in line 3, the jump is performed only if the condition is false.
6. What is jumped over is exactly the contents of the block, thus the block is generated in line 6 still before the label is visited. Generating a block requires simply iterating over its nodes array, asking each node to generate itself.

Generating while-loops is analogous, but requires two labels, since the loop keeps jumping back to the first label while the condition is true. Naturally, the first label is visited first, to establish that jump location. Then, the expression and a constant of value true are pushed, followed by a call to IF_CMPNE, as with if-statements, passing as an argument the second label, which is yet to be visited. Next, the block is visited and a GOTO instruction is added, giving it the first label as an argument, which causes the condition to be re-evaluated. If still true, the block is repeated. If not, a jump to the

second label is performed, whose visit is the last step in generating a while-loop. The actual bytecode implementation often involves more instructions. Generation thereof is gladly delegated to the ASM library.

Generating print statements depends on whether the *Scandal* program is running inside the IDE, or in the command line. A call to Platform.isFxApplicationThread is made to check. If running on the command line, mv is used to load System.out, the expression is asked to generate itself on top of it, and finally mv is used again to call println on System .out. If running on the IDE, mv is used to load language.ide.MainView.console, which is an instance of javafx.scene.control.TextArea. The top of the stack is then duplicated, for reasons explained momentarily. The expression is then generated on top of that, and a test is made to see if the expression is not of type string, in which case a call is made to 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, a call is made to TextArea.appendText, which pops the two topmost elements, namely the expression and the duplication of the console. A new-line character is then pushed, and a call is made to TextArea.appendText, in order to create a line break. This causes the top of the stack to be popped twice, hence why the console was duplicated.

The generation phase for plot statements consists of checking, similarly to print statements, whether the program is running on the IDE. If not, no bytecode is generated, hence one cannot see plots when running a *Scandal* program from the command line. Otherwise, we mv is used to push a new instance of language.ide.PlotTab, and all three expressions are pushed on top of it. If the last expression has type float, the F2I instruction is used, and finally init is called on the PlotTab, causing it to be displayed on the IDE's main view.

Like before, generation of play statements involves checking whether the program is running inside the IDE. If not, mv is used to push a new instance of AudioTask onto the stack, which is duplicates, since it need to be used twice. A call to init on the AudioTask follows, which pops the topmost of the two. Both expressions are then generated, and a

call to play on the AudioTask is made. If the program is running on the IDE, however, a new instance of language.ide.WaveTab is pushed instead. Next, a string is pushed containing the class name, which the symbol table holds, both expressions are generated, and call is made to 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, and 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 code-generation routine in write statements uses mv to push a new instance of framework.generators.AudioTask, which is duplicated. After calling init on the AudioTask, all expressions are pushed and export is called on AudioTask, which effectively saves the audio buffer as a *.wav* at the specified path.

5.4 Generating Expressions

Generating binary expressions is unsurprisingly involved, given that operators are *not* overloaded in bytecode at all, hence a fair amount of work is needed to guarantee operators are matched with compatible types. The basic mechanism is to load the left expression, then create a switch case for each kind of operator. The procedure for all arithmetic operators is very similar. After pushing the left expression, a check is made to see if the binary expression has type float. If so, then at least one of the expressions must have type float. So while the left expression is still on top of the stack, generate checks if it is *not* a float, and if so calls the I2F instruction. Next, the right expression is pushed and the same check is performed. The last step is to call the appropriate JVM instruction to deal with the float case of the arithmetic operator at hand. If, on the other hand, the binary expression has integer type, then necessarily *both* expressions also have integer type, so generate just pushes the second expression and calls the appropriate JVM instruction. In the cases where the operation involves a logical operator, the JVM can only perform those operations on integers, including naturally integer representations of booleans. So here the left expression is loaded and cast to integer if it is a float, the same for the right expression, and the logical operator's instruction is called. Comparison

operators are more tricky, since generate cannot count on the overall type of the binary expression to make necessary conversions, and the JVM does have separate instructions for integers and floats, requiring both sides to have the same type. Every binary expression whose operator is a comparison has type boolean, but still, JVM instructions are not overloaded. Hence the top of the stack needs to have agreeing types before a particular instruction is called. However, observing that after calling the operator instruction, the top of the stack is always left with an expression of type boolean, instead of testing every possible case, generate casts both expressions to float, as needed, and only resorts to float comparisons.

Generation of unary expressions starts by pushing the expression on top of the stack. It then looks at the expression type and covers three cases. If it is an integer, then from type-checking it is only possible that the given operator is a `MINUS`, hence generate calls the INEG instruction. The same applies for floats, only that the FNEG instruction is called. Code generation is surprisingly more involved for the boolean case. If given a boolean expression, then it needs to be negated. That is accomplished by creating two labels, visiting an IFEQ jump instruction, and giving it the first label as an argument, which causes the top of the stack to be popped. If the stack contained a zero, that is, if the expression evaluates to false, then a jump to the first label is performed, where a ICONST_1 is pushed onto the stack. If the stack contained a true instead, then a ICONST_0 is pushed, and generate visits a GOTO jump, giving it as an argument the second label. At the second label location, nothing is really done, and generate just leaves the false value on top of the stack and returns.

Generating identifier expressions resorts to three cases. The first case is when the declared variable is a field, which is known from the isField property every declaration contains. If so, generate use mv to push the value associated with the field's name. The second case deals with the possibility that the variable is a parameter inside a lambda, which happens whenever the declaration property is an instance of ParamDeclaration. If

so, then surely the value of the expression is not a primitive, since the Function interface requires values to be passed as classes, as discussed previously. The generate method then does two things, namely it uses ALOD to push the expression's value, then calls Expression .getTypeValue, a static method that converts from classes left on top of the stack to their primitive values. Naturally, this only applies to integers, floats, and booleans. If neither a field, nor a parameter, then generate loads the variable normally. For integers and booleans, it uses ILOAD, for floats FLOAD, and for all other types it uses ALOAD. Except for fields, a slot number is always needed to load variables, which is retrieved from the declaration property every identifier expression contains.

Code generation of literal expressions is easy. For integers and floats, generate calls respectively Integer.parseInt and Float.parseFloat, using the result as an argument to mv. visitLdcInsn, which effectively pushes the value onto the stack. There is no risk of finding bad numbers here, as a check has already been made during scanning. For booleans, generate tests the given keyword, and loads the appropriate constant, whereas for strings, it uses mv to load the firstToken.text property directly.

Generating array literal expressions requires constructing the array, element by element, on the JVM stack. To do so, generate uses mv to push the size of the list of floats onto the stack, then calls the NEWARRAY instruction with a T_FLOAT argument. It then iterates over the list of floats and, for each expression, duplicates the top of the stack, pushes an index value onto the stack, asks the expression to generate itself, converting it to float as needed, and finally calls the FASTORE instruction, which stores the expression's value onto the new array at the specified index. At the end of the each iteration, the call to FASTORE pops the topmost two elements, hence at the end of the entire process, generate effectively leaves the filled-up array on top of the stack. Generation of array item expressions asks the array property to push itself onto the stack, then asks the index property to do the same. If the index has type float, generate calls F2I, as usual. Finally, generate uses mv to call FALOAD, leaving a float on top of the stack. Generation of array

size expressions is also easy, asking the array property to load itself, then using mv to call the ARRAYLENGTH instruction. Code generation of new array expressions loads the size property on top of the stack and converts it to integer, as needed, finally calling the NEWARRAY instruction with a T_FLOAT argument, similarly to array literal expressions.

Code generation of pi expressions is trivial, with generate simply asking mv to load *Java*'s Math.PI on top of the stack. Generation of cosine expressions asks the phase property to leave a value on top of the stack, then converts it to double using either the F2D or the I2D instruction, since the method signature for *Java*'s Math.cos takes a double and returns a double. It then uses mv to invoke Math.cos, finally casting the result back to float using D2F. For power expressions, the method signature for Math.pow in Java takes two doubles and returns one, so generate converts back and forth, similarly to what is done in cosine expressions. Floor expressions follow the same pattern, only by invoking *Java*'s Math.floor method instead. For write expressions, code generation asks mv to create a new instance of WaveFile, which is duplicated. It then generates the path property and calls init on WaveFile. Next, generate pushes the format property and calls WaveFile.get, which leaves an array of floats on top of the stack. Code generation of record expressions is very similar to read expressions, where generate instantiates an AudioTask, duplicates it, and calls init. It the pushes the duration property onto the stack, which is given in milliseconds, and casts it to integer, as needed, finally calling AudioTask.record. The latter will block execution of the Scandal thread for the entire duration, capture input from the preferred audio input device in Settings, then leave an array of floats on top of JVM the stack.

5.5 Generating Lambdas

Code generation of lambda literal expressions is a lot more involved than expressions in general, and accomplished in two stages. In the first stage, an instance of Program calls the generate method overridden from Node, giving it as an argument the instance of MethodVisitor associated with the *Java* class' init method. The purpose of the code generated inside init is to associate the lambda declared as a field with a method body.

Every lambda literal implements the Function interface, which is a *Java* functional interface, so called whenever they comprise a single abstract method, and any number of methods containing bodies. Thus an object that implements the Function interface needs to be instantiated for each lambda literal declared. This object will in turn implement the interface's abstract method, which is matched to the expression or block held inside each LambdaLitExpression. This object can thereafter be treated as a variable. Whenever a lambda expression is applied arguments or composed with other lambda expressions, the JVM is basically calling methods on this class. Note that the one abstract interface method is however a static and synthetic method of the class that *defines* the implementor of the functional interface, and not part of the implementing object itself. For this reason, type-checking inside the JVM is deferred until runtime, hence generate uses the INVOKEDYNAMIC instruction to call java.lang.invoke.LambdaMetafactory, which does all the heavy lifting, instead of doing all the above wiring explicitly. In *Java*, lambda expressions are essentially syntactic sugar for objects that implement functional interfaces.

Inside the first stage of code generation, a string is formed containing the lambda's method signature. This method signature consists of a single input type and a return type. The reason for a single input parameter, despite the size of the parameter list, is the right-associated currying of lambda expressions described above, in which a function of n variables that returns a value is seen as a function of one variable that returns a function of $n - 1$ variables, and so on until a function of one variable that returns a value, when the last variable is reached. It follows the required method signature always has as input type the type of its very first argument, always given as a *Java* class, as Function accepts no primitive types. The return type depends on the size of the argument list, naturally. If only one argument is given, then the expression's return type is the type of the return expression, otherwise the return type is java.util.function.Function. Having formed this string, generate uses it as an argument to mv.visitInvokeDynamicInsn, which effectively wires

the lambda body pertaining to the *Java* class to the apply method in the object that implements the Function interface, making use of LambdaMetafactory.

The second phase of code generation consists of writing the bytecode for the lambda body itself. The LambdaLitExpression class overloads the generate method in Node to take as parameters an instance of SymbolTable and an instance of ClassWriter. The latter is used to create an instance of MethodVisitor that includes the lambda body as a method in the *Java* class. If the lambda expression has more that one parameter, generate actually needs to instantiate a method visitor, as well as include a method in the *Java* class for each parameter in the list, as discussed previously. These methods all have three flags, namely private, static, and synthetic, and are named by incrementing lambdaSlot. In only the very last of the methods does generate ask for the return expression to generate itself. For all the others, it pushes onto the stack all parameters up the the current, and calls the apply method inherited from the Function interface, which leaves an instance of Function on top of the stack. For all these methods, generate gives the ARETURN instruction and uses ASM to compute the sizes of the JVM frame and local variable count.

The first stage of generation in a LambdaLitBlock is identical to the LambdaLitExpression superclass, hence it is not overridden. The second phase of code generation, the one in which the redirected method body is written to the bytecode class, is not identical but very similar to the superclass'. The only difference is that, instead of asking a return expression to decorate itself at the body of the lambda associated to the rightmost parameter, generate asks for the lambda block to generate itself. Code generation of return blocks also differ from the superclass implementation in that, in addition, they ask the return expression to generate itself before returning.

The code generation phase of lambda application expressions is simpler, and begins by asking the lambda property to generate itself. Next, generate iterates over the argument list and, like in decoration, offsets the list of parameters by its current size minus count, asking each argument to generate itself. If the argument passed is a literal, generate casts it to

the corresponding *Java* class, since the Function interface does not accept primitive types. Also, for each argument, generate calls Function.apply, which effectively calls the method associated to each parameter of the original lambda literal. The last step of generating each parameter consists of verifying that the returned value left on top of the JVM stack corresponds to what is expected, which is done by calling the CHECKCAST instruction. There are three possibilities. In the case we are the application of a lambda that is a parameter to another lambda, generate simply trusts that the assignment declaration or statement that has this lambda application as its expression has already decorated it with a type, so generate uses the this.type property to make the check. Otherwise, generate can use lambdaLit.returnExpression.type for the very last argument being visited, or Types.LAMBDA for all others. After generating and applying all arguments, generate leaves a Java class on top of the stack, but a literal should be left if the expression type corresponds to one of *Scandal*'s literal types. If so, generate casts the result back to a primitive and returns.

Generating lambda compositions is, on the other hand, rather straightforward, and begins by pushing the very first expression onto the stack. Next, for each lambda other than the first, generate pushes it onto the stack and calls Function.andThen, which consumes the top two elements and leaves the result on top of the stack. The last step of code generation consists of checking whether the lambdaApp property is null. If not, generate loads its arguments, check-casting and applying each one, and converts the result to a primitive, as needed, exactly the same way arguments in a LambdaAppExpression are loaded and applied. Otherwise, generate simply leaves the last result of Function.andThen on top of the stack and returns.

CHAPTER 6
CASE STUDIES AND CONCLUSION

In this chapter, a variety of audio signal processing and music composition applications of *Scandal* is presented. These are meant as illustrations of how the language can be used, but also to show how it can differ from other languages whose domains are specific to audio signal processing and music. The chapter is concluded with a discussion of all roadblocks so far, and possible directions for the future development of *Scandal*.

6.1 Breakpoint Functions

Generating breakpoint functions is absolutely fundamental to musical applications. A primary use is to create fade-ins and outs to avoid undesirable clicks, as well as cross-fades of overlapping buffers to smoothen transitions. One interesting way to build breakpoint functions is *Csound*'s GEN7 routine. Its straightforward syntax is illustrated in Listing 6.1.

Listing 6.1. Enveloping an audio buffer in *Csound* with GEN7.

```
<CsoundSynthesizer>                                          1
   <CsInstruments>                                           2
      0dbfs = 1                                              3
      instr 1                                                4
         kIndex phasor 1 / p3                                5
         kEnvelope tablei kIndex, 1, 1                       6
         aSine oscil .5 * kEnvelope, 1220                    7
         out aSine                                           8
      endin                                                  9
   </CsInstruments>                                          10
   <CsScore>                                                 11
      f 1 0 1024 7 0 512 1 512 0                             12
      i 1 0 2                                                13
   </CsScore>                                                14
</CsoundSynthesizer>                                         15
```

3. Line 3 sets the overall amplitude to range from zero to one, since the default behavior in *Csound* is to have amplitudes range from zero to 32768, the 32-bit word length.
4. Lines 4 to 9 form the instrument 1 block.
5. The phasor opcode is used to generate indices, and is given an argument of 1 / p3, that is, it is being asked to provide indices ranging from zero to one over the duration p3, which is the third argument given to instr 1 in the CsScore section.
6. The tablei opcode reads from a table indexed by kIndex, and is given as other arguments the table number (1), and a normalization factor (1).
7. The oscil opcode produces a sine wave with .5∗ kEnvelope amplitude, and a 1220 Hz frequency of oscillation.
8. The out opcode outputs aSine to the speakers.
11. Function table 1 is instantiated at time 0, has size 1024, is computed by the GEN7 routine, and its values range from 0 to 1 over 512 samples, and from 1 to 0 over 512 samples.
12. Instrument 1 starts playing at time 0 and plays for 2 seconds.

Even thought quite verbose, the *Csound* code in Listing 6.1 accomplishes a lot behind the scenes. It is interesting to observe, however, that the implementation hiding in the opcode methods does not release the musician from writing structured text, and in many cases actually blinds the user to implementations that are in fact very easy. Any one willing to learn *Csound* would likely have but little trouble learning how to implement GEN7, but would have immense gains in learning how to do so. Moreover, simple implementations like GEN7 are problematic from the standpoint of the language designer too, since maintaining literally over a thousand opcodes is extremely burdening in all aspects of a compiler's development ecosystem. These observations lie at the core of *Scandal*'s philosophy. Listing 6.2 demonstrates the implementation of GEN7 in *Scandal*. Its use is as similar as possible to that of *Csound*: the first argument provides a length, and

the the second argument is an array that provides a variable number of breakpoints and their lengths. Line 12 of Listing 6.1 would be declared in Scandal as array f1 = GEN7(1024, [0, 512, 1, 512, 0]).

Listing 6.2. A *Scandal* implementation of GEN7.

```
lambda GEN7 = int length -> array args -> {
    array table = new(length)
    float height = 0.0
    float increment = 0.0
    int width = 0
    int i = 0
    int j = 0
    while j < size(args) - 2 {
        height = args[j]
        increment = (args[j + 2] - args[j]) / args[j + 1]
        width = i + args[j + 1]
        while i < width {
            if i < length { table[i] = height }
            height = height + increment
            i = i + 1
        }
        j = j + 2
    }
    return table
}
```

The algorithm in Listing 6.2 is quite self-explanatory. Of course, the user may always *choose* to hide its implementation by putting it in a separate file and using an import statement. Creating a fade-out, for example, becomes the simple task illustrated in Listing 6.3. A fade-in method, as well as a cross-fade method that relies on both a fade-in and a fade-out are may be found in the *lib/Fades.scandal* file, which is bundled with the IDE. Naturally, these routines can be re-factored to accept any other enveloping function other

than GEN7 by simply making them into higher-order functions, and setting the enveloping routine as a parameter. This type of re-factoring shows both the musical creative potential and software re-usability associated to functional programming, corroborating *Scandal*'s choice of paradigm. A *Scandal* implementation of Listing 6.1 will be given at the end of the next section, after covering more ground.

Listing 6.3. Implementation of a fade-out effect.

```
lambda fadeOut = array x -> int samples -> {
    array fade = GEN7([samples, 1, 0, samples])
    array buffer = new(size(x))
    int i = 0
    int j = 0
    while i < size(x) {
        buffer[i] = x[i]
        if i >= size(x) - samples {
            buffer[i] = buffer[i] * fade[j]
            j = j + 1
        }
        i = i + 1
    }
    return buffer
}
```

6.2 Oscillators

This section discusses *Scandal* methods intended to synthesize sound. Results are again compared with similar *Csound* routines, but also with an object-oriented way of accomplishing the same task. Given the ubiquity of the OOP paradigm, and the fact that *Scandal*'s underlying foundation is object-oriented, a decision of making *Scandal* a simple scripting language with a functional flavor was made in order to facilitate restricting its domain. It is the language's philosophy to remain focused, with a clear intent to abstract away some of *Java*'s more verbose syntax. There are many obvious sacrifices

involved with this restriction, especially when it comes to handling user-defined types and data structures. A foreseeable roadblock is that of frequency-domain signal processing, which involves complex numbers, and arrays thereof, and whose implementation would require *Scandal* to step up from its naivety. That said, the functional paradigm is capable of handling elegantly and concisely most of the implementation challenges involved in audio signal processing. An OOP way of defining an oscillator would involve defining a general waveform type, then sub-classing it into specific waveforms. Common to every waveform would be a routine that returned a sample value, given a phase argument in radians. An oscillator class would then hold the phase value as a property, as well as a pointer to a waveform generator. It is easy to see how such a design translates into the functional paradigm. Instead of subclasses of a parent class that override a certain method, a function itself defines a method to compute a waveform, given a phase argument. And instead of an object, an oscillator is a higher-order function which takes a specific waveform-producing function as an argument. Listing 6.4 illustrates how an oscillator may be defined in *Scandal*.

Listing 6.4. Defining an oscillator.

```
lambda oscillator = float dur -> float amp -> float freq -> lambda shape -> {
    array buffer = new(dur * 44100)
    float phase = 0.0
    int i = 0
    while i < dur * 44100 {
        buffer[i] = shape(phase)
        buffer[i] = buffer[i] * amp
        phase = phase + freq * 2 * pi / 44100
        if phase >= 2 * pi { phase = phase - 2 * pi }
        i = i + 1
    }
    return buffer
}
```

Listing 6.4 creates a buffer of audio samples containing a waveform specified by the function shape. It is important to observe that the parameter shape has its own parameters and return types inferred, hence needs to be applied in line 9 before used in the binary expression of line 10. In order to make use of oscillator , all that is needed is some function that returns a waveform given a phase in radians. The cosine function comes to mind as the most straightforward, but in fact any waveform is easy to implement. Listing 6.5 provides a few examples. In particular, line 1 is just a wrapping around the built-in cos expression, meant to work with oscillator .

Listing 6.5. Waveform-generating lambdas.

```
lambda cosine = float phase -> cos(phase)                                    1
lambda sawtooth = float phase -> 2 * (1 - phase / (2 * pi)) - 1              2
                                                                             3
lambda square = float phase -> {                                             4
    float val = 1.0                                                          5
    if phase >= pi { val = -1.0 }                                            6
    return val                                                               7
}                                                                            8
                                                                             9
lambda triangle = float phase -> {                                           10
    float val = 1 - phase / pi                                               11
    if val < 0.0 { val = -val }                                              12
    return 2 * val - 1                                                       13
}                                                                            14
```

Putting all the above together, this section concludes with a *Scandal* implementation of Listing 6.1. What the *Csound* program basically does is modulate the amplitude of the aSine by kEnvelope. Instead of using a phasor to get indices, *Scandal*'s instr1 simply combines both arrays using a while loop. Of course, a phasor lambda could equivalently be defined, or even any lambda that took two arrays of different sizes and computed their point-wise product, which is exactly what applying an envelope does. In line 6 of Listing

6.6, the envelope array is indexed in terms of the size of the waveform array. The simple statement play(instr1 (2.0, GEN7(1024, [0, 512, 1, 512, 0])), 1) causes the *Scandal* program to have the same sound output as the *Csound* program.

Listing 6.6. Implementing a *Csound* program in *Scandal*.

```
lambda instr1 = float dur -> array kEnvelope -> {
    int samples = dur * 44100
    array aSine = oscillator(dur, 1.0, 1220.0, cosine)
    int kIndex = 0
    while i < samples {
        aSine[i] = aSine[i] * kEnvelope[kIndex * size(kEnvelope) / samples]
        kIndex = kIndex + 1
    }
    return aSine
}
```

6.3 Composing Music With Loops

This section demonstrates how an entire musical composition may be coded in *Scandal*. The technique of choice is that of composing with audio loops, a common technique in the music industry, particularly with DJ's. An audio loop is a pre-composed snippet of music containing properties, namely style, beats-per-minute, number of bars, and harmonic key, if applicable. Loops are usually single-instrument recordings, and composing with them often involves orchestrating these instruments. Most loops are rather short, and a common technique is to *stretch* them. One stretches a loop by, as the name suggests, repeating them as desired. Since loops are cut to encompass a certain number of bars exactly, the next repetition will always be rhythmically synchronized with the previous. By orchestrating loops with the same BPM, the entire composition becomes easily synchronized, as well. Listing 6.7 gives a *Scandal* implementation of a simple stretch routine.

Listing 6.7. Stretching a loop in *Scandal*.

```
lambda stretch = float tail -> float bars -> lambda b2s -> array start -> {      1
    int samples = b2s(bars)                                                     2
    samples = samples + size(start)                                             3
    array buffer = new(samples)                                                 4
    int offset = b2s(tail)                                                      5
    int i = 0                                                                   6
    int j = size(start)                                                         7
    while i < j {                                                               8
        buffer[i] = start[i]                                                    9
        i = i + 1                                                               10
    }                                                                           11
    while i < samples {                                                         12
        buffer[i] = buffer[j - offset]                                          13
        i = i + 1                                                               14
        j = j + 1                                                               15
    }                                                                           16
    return buffer                                                               17
}                                                                               18
```

1. Line 1 declares a stretch lambda with the following parameters: a tail parameter that indicates the portion in bar numbers of the loop, from right to left, that should be stretched. A loop may contain multiple bars, in which case only the last few may stretched, say. A bars parameter, used to indicate how long in bars the loop should be stretched. A lambda parameter that converts from bars to samples given a particular sampling rate and BPM. Finally, the array to be looped.
2. The local variable samples is used to apply bars to b2s, so its parameterized type can be inferred. In line 3, the total number of samples to be returned is computed, and an array of that size is initialized in line 4.
5. Line 5 applies tail to b2s so that stretching the loop may be restricted to that many samples, from right to left. Lines 6 and 7 declare iterators for the subsequent while-loops.

8. The while-loop in lines 8 to 11 simply copies the contents of the given loop to the beginning of the array that is going to be returned. The while-loop in lines 12 to 16 copies the tail of the loop to the return array for however many bars were given. Finally, line 17 returns the array containing the stretched loop.

Having built a mechanism to stretch loops, the next step is to have a way of reading from audio files and positioning these loops in time. Being that the routine that stretches loops is a lambda expression, it would be interesting to make it composable with itself, and with other lambdas that take an array and return another, such as the lambda that reads from files and positions that buffer in time which is about to be constructed. It is no coincidence that the *last* parameter of stretch is the array. It was constructed that way so that the array parameter could be left as the last to be fixed, hence composable to other array-to-array lambdas. In Listing 6.8, the main idea is to create a lambda that holds a path to an audio file and is capable of splicing that audio file to the right of a given buffer.

Listing 6.8. Appending the contents of a file to a given buffer.

```
lambda appendFile = string path -> array start -> {                1
    array loop = read(path, 1)                                     2
    int samples = size(start) + size(loop)                         3
    array buffer = new(samples)                                    4
    int i = 0                                                      5
    while i < size(start) {                                        6
        buffer[i] = start[i]                                       7
        i = i + 1                                                  8
    }                                                              9
    while i < samples {                                            10
        buffer[i] = loop[i - size(start)]                          11
        i = i + 1                                                  12
    }                                                              13
    return buffer                                                  14
}                                                                  15
```

The algorithm in Listing 6.8 is easy to follow: given a start buffer, the routine reads from a file and appends the file's contents to the given buffer. What is not necessarily obvious at first is that appendFile and stretch are highly modularized, reusable, and composable methods to deal with loops. Positioning a loop to start playing at, say, the eighth bar is easily accomplished by giving it an empty array of b2s(8.0) samples. At the moment, there is a mechanism to begin playback of a loop at a certain position in time, and to stretch it arbitrarily, but a way of intercalating silence between occurrences of a loop is still missing. A composable solution is to do basically the same appendFile does, but giving it, instead of a path to an audio file, an integer number of samples, all of whose values would be zero, to append to a start buffer. The corresponding parameter to this start buffer would naturally be left as the last, in order to make this function composable with appendFile and stretch. The code for such appendSilence routine would be very similar to appendFile, hence omitted. With the tools at hand, a loop can be positioned in multiple places within an audio track, and each occurrence of the loop could be stretched. Naturally, a method to combine audio tracks is still missing.

Mixing two audio signals is mathematically equivalent to adding two vectors. An obvious way to implement a mix routine would be to have two nested while-loops. The outer loop iterates over a total-duration number of samples, and the inner loop iterates over all audio tracks, adding their samples point-wise. In the spirit of writing code that is both more concise and reusable, it would be more interesting to write a lambda that took two arrays and mixed them together. Naturally, fixing the first array and leaving the second free makes this lambda composable with the other lambdas declared above. The implementation of such mix lambda is also straightforward and thus omitted. The very last bit of structure needed to compose a musical piece with loops is to implement the b2s routine needed by stretch. A simple implementation is given in Listing 6.9. The parameters are bpm, which is specific to the set of loops being considered, number of beats per bar, that is contextual, and a number of bars, which is the last parameter because its

context is given by the each occurrence of a loop, and not by a characteristic of all loops in a set. The formula reads samples per second (44100) times bars, times beats per bar, times seconds per minute (60), over beats per minute, and returns a value in samples per second.

Listing 6.9. Converting bars to samples.

```
lambda barsToSamples = float bpm -> float beats -> float bars -> {
    int val = 44100.0 * bars * beats * 60.0 / bpm
    return val
}
```

Having defined a b2s function, writing a piece of music with loops is merely a task of applying arguments to the aforementioned lambdas. Listing 6.10 gives a broad overview of *Scandalous*, a musical composition with loops whose complete code listing is bundled with *Scandal*'s IDE.

Listing 6.10. The high-level structure of a composition with loops.

```
lambda barsToSamples = b2s(130.0, 4.0)
lambda stretch8 = stretch(8.0, 8.0, barsToSamples)
lambda mute4 = appendSilence(barsToSamples(4.0))

lambda bass8 = appendFile("wav/bass8.wav")
lambda kick8 = appendFile("wav/kick8.wav")
lambda snare8 = appendFile("wav/snare8.wav")
lambda pad8 = appendFile("wav/pad8.wav")
lambda bass = mix(mute64.bass8.stretch72.mute76(new(0)))
lambda kick = mix(mute80.kick8.stretch40.mute92(new(0)))
lambda snare = mix(mute64.snare8.stretch56.mute92(new(0)))
lambda pad = mix(pad8.stretch56.pulse8.stretch56.pad8.stretch80.mute4(new(0)))

array master = bass.kick.snare.pad.normalize(new(barsToSamples(220.0)))
play(master, 1)
```

1. The barsToSamples lambda is a version of b2s that is specific to the loops chosen for the piece, which all are in 130 beats per minute, with four beats per bar. In line 2, stretch8 is a version of stretch that takes the last eight bars of a buffer and stretches them for eight more bars, hence loops those bars altogether once. Similarly, mute4 appends four bars, with the given tempo and metric, to a given buffer.

5. Lines 5 to 8 declare four lambdas that attach a loop to the end of a given buffer. The length of all four loops is eight bars, but the actual piece has many other loops with different lengths.

10. The lambdas in lines 10 to 13 are the audio tracks corresponding to those instruments. The bass track, for example, consists of 64 bars of silence, followed by 8 bars of the bass8 loop, stretched to 72 more bars, hence 80 in total, followed by 76 bars of silence. An empty array containing zero samples is applied to this composition of lambdas, and the result is subsequently applied to mix. The result is a lambda that sums this entire audio track to a given array. Note that all tracks have the same length of 220 bars, so they can be added point-wise.

15. Line 15 declares a master array that consists of all instruments composed (added) to a new array of length 220 bars, which is long enough to accommodate each of them. The very last lambda in the composition of line 15 is normalize, since adding several vectors together may cause the sum at some samples to be greater than one. The normalize lambda is composable with the rest because it too is an array-to-array lambda. Finally, a play statement in line 16 causes the entire piece to be plotted and played back.

6.4 Future Work

One of the biggest challenges encountered so far in *Scandal*'s existence is maintaining a balance between what the language can and cannot do. This balance has deep implications in what the language ultimately is. Given the facility with which *Java* functionality can be ported into *Scandal*, it is not really an issue of *how* the DSL can be extended, but

ultimately an issue of *why* should *Scandal* lose its naivety. The main argument supporting the idea of keeping it simple consists of avoiding that gray area in which a DSL is neither easy to learn anymore, nor as versatile as a general-purpose language. An example would be *Supercollider*, a language that is object-oriented, functional, dynamically typed *and* dynamically interpreted, and built over an incredibly fast and reliable *C++* foundation, that can handle any sort of real-time audio signal processing. However, a language that is difficult to learn, has a frozen-in-time, *Smalltalk*-like syntax and that, ultimately, brings no real advantage over a general-purpose language. Quite the contrary, it lacks too many features to justify its steep learning curve. Learning *Swift* is probably easier, but also liberating in the sense that one can write musical applications for mobile devices, write a http server to compose music collaboratively, and use all sorts of third-party API's, to cite but a few. One can make music and process sounds very easily in *Swift* with a library such as *SoundKit*, which can even perform live coding via *Swift Playgrounds*.

The philosophy that drives the development of *Scandal* is that of keeping it easy to learn, above all things. A good educational tool is better than a super-complete DSL that ends up being difficult to learn. The successful *Scandal* user is that who eventually transcends the language and ventures into learning as many other languages as possible. Learning is the motivation, and *Scandal* is but a doorway. And a flawed one, which is a fact that should be constantly emphasized. Other areas than music face a similar paradigm. Domain-specific languages such as *Matlab* and *R* are good examples of how engineers and statisticians can become oblivious to even knowing how to implement algorithms that are the bread and butter of those professions.

In the spirit of promoting learning, future development of *Scandal* shall emphasize more IDE integration, with better error reporting and code highlighting. Also of vital importance is better visualization tools. In what regards the language alone, a lot of effort has been put into making it as declarative as possible, with built-in statements reduced to a minimum. In many cases, however, 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 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 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. When it comes to import statements, future versions of Scandal will check for backward edges in the DAG, and throw a more informative compilation error instead of a runtime error. Also, 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 space-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 current *Scandal* 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.

Switches, repeat-while-loops, and for-loops are not yet included, but certainly intended. A more complete implementation of the two currently supported types of control statement would also include handling the cases where, instead of a block, a single statement is allowed to be executed, should the condition succeed. Adding this functionality is trivial, and would improve readability since the surrounding brackets could be dropped. 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, the surrounding brackets may be dropped, allowing the entire lambda to be expressed in a single line.

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. Play statements do not overload the format property to accept floats, 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. The same applies to the channel count in write statements.

Overloading operators can be very beneficial for a DSL. *MATLAB* is perhaps one of the best examples, but also *Csound* operates almost exclusively on overloaded array operations. Expressions that resemble the way they are written mathematically are also fundamental in improving readability, as well as removing the clutter created by words

for which a household symbolic representation exists. The future steps of *Scandal* shall improve even further the functionality of binary expressions. Given the nature of the language's domain, overloaded array operations are the next logical step. As a simple example, one could apply a scalar to an entire array in the same way one would write such an expression mathematically. Let $\alpha = 2$ and $\vec{v} = [1\ 2\ 3]$. Then $\alpha \cdot \vec{v} = [2\ 4\ 6]$. With overloaded array operations, one could write `a * [1, 2, 3]` and have it evaluated to `[2, 4, 6]`. At the moment, one cannot construct arrays of any type other than arrays of floats. Rather than a design choice, this is another incomplete implementation that shall be revised in the future. Cosine is also the only transcendental function built into *Scandal*, since other trigonometric functions can be computed in terms of the cosine. This is, again, not a design choice but a yet incomplete aspect of the language. The syntax for power expression will likely change in the future to that of a caret operator inside a binary expression, which will require one more level of precedence among operators than factors.

Restricting fields to be declared only at the outermost scope is aimed at forcing clarity among declarations. Again here, it would be trivial to allow them to be declared anywhere, but it is arguably confusing to declare a global variable inside an inner scope, so such declarations will likely remain forbidden. 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 their implementation is omitted in this proof of concept. Protecting against bad access remains problematic with the given symbol table configuration. One solution would be to instantiate a `SymbolTable` per each lambda literal, copy all field declarations into it, and not bother with introducing new scopes at all. This feature remains unimplemented, but is planned. The difference at the moment is that a bad access will cause a runtime error, whereas a properly implemented separate

symbol table would fail at finding a the non-field declaration, throwing a much more useful compilation error instead. For the time being, lambda expressions with blocks in *Scandal* are neither allowed to declare lambda literals inside their blocks, nor return expressions of type lambda. This remains one of the most critical shortcomings of the language, and should change in the near future. Lambdas can still be returned from lambdas by partial application, but being able to dynamically generate new functions will represent a major step in making *Scandal* even more of a functional language. Implementing this feature will require that not all lambda literal expressions be fields in the *Java* class, and will thus have a huge impact on how the language manages capturing the environment inside a method body.

The implementation of lambda compositions is at the moment very incomplete, and represents no more than a proof of concept. The main reason for incompleteness is the fact that the LambdaCompExpression class holds an array of identifier expressions, whereas it should hold an array of general expressions that could possibly partial lambda applications. It is impossible at the moment to, say, fix parameters in a composed lambda, except for the very last expression, which is in fact allowed to be a lambda application. This missing functionality does not impair the language, as not being able to return lambdas from lambdas does, it only makes *Scandal* code more verbose than it needs to be. Moreover, holding an array of identifiers alone poses a huge complication when it comes to type-checking. As exemplified by the quite complicated type-checking mechanism of lambda applications, type-checking lambda compositions that are given by name references alone would mean the same effort of a lambda application for *each* of the composed lambdas. What is worse, is that all this work would be a waste, since such implementation would still not provide the language the functionality of fixing partial applications in composed lambdas. For all these reasons, type-checking in lambda compositions is mostly ignored, until the time when such expressions evolve to encompass partial applications of lambdas. It is easy to see that the ability to return lambdas from literal lambdas would

play a crucial role here, since a chain of composed lambdas would then possibly contain *total* applications that would return functions, and those could be further composed to return anything, including other functions. Completing the implementation of lambda composition expressions would also require checking that the argument list given to the last lambda agrees with the expected parameters of the *first* lambda, given that *Scandal* does not compose lambdas in the mathematical sense. Furthermore, a type-checking rule shall be imposed such that the parameterized return type of the first expression equals the input type of the second, and so on until the type of the composition expression is taken to be the return type of the very last 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.