MIDL: A SCRIPTING LANGUAGE FOR SYNTHESIZED MUSIC

Romit Das

Problem report submitted to the
Lane Department of Computer Science at
West Virginia University
in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science

Dr. Donald Adjeroh
Dr. Elaine Eschen
Dr. Frances Van Scoy (Committee Chair)

Lane Department of Computer Science

Morgantown, WV
2005

Keywords: MIDL, MIDI, ANTLR, Digital Music, Synthesized Music, Parsers

ABSTRACT

MIDL: A Scripting Language for Synthesized Music

Romit Das

MIDL serves as a framework for creating synthesized music files, and is geared towards individuals with only a rudimentary knowledge of modern musical notation. The heart of the system lies in a very simple scripting language developed using the ANTLR parser generator, and is syntactically reminiscent of BASIC. Abstracting unfamiliar MIDI file constructs such as *ticks* and *channels*, MIDL paves a common ground such that playing a musical note is as simple as typing `{ PLAY C }`.

In addition, I have developed an integrated development environment (IDE) named MIDLMan in Java that provides error reporting based on line numbers, provisions for previewing and exporting to MIDI files, and loading pre-existing MIDL listings.

While numerous projects have tackled the problem of digital music representation, MIDL breaks new ground as it is not a library; it is a self-contained programming language, and a simple one at that. Informal testing has shown that provided a sample script, most users are able to grasp the major facets of the language in just a few minutes.

TABLE OF CONTENTS

I'd Like to Thank the Academy

This work would have been impossible without the constant feedback and encouragement from my advisor, Dr. Frances Van Scoy. Her patience, straightforwardness, and endless enthusiasm even through topics known to put the best sleeping pills to shame made it possible for me to make it through my 2 years at WVU. For that, I thank her.

Not to be outdone, I am very grateful to Dr. Elaine Eschen, who has taught me humility and something or other about sorting. To Dr. Donald Adjeroh, who was the first professor here to truly challenge me with a seemingly impossible project, I am also in debt.

As I never got a chance to thank the two cool cats responsible for me staying on this path, let me do so now; Ms. Meg Geroch and Mr. Patrick Plunkett, a simple thanks isn't nearly enough. To my friend and partner in crime, Mr. Ryan Scotka, I give the traditional omghi2u2, and to the Morgan family, who made my stay at WVU a little bit more bearable, I offer my profound gratitude.

Last, but definitely not least, I'd like to thank my family. In particular, my mother, brother, and Philip and Evelyn Kirby for being there when I needed it the most. Who knows where I would have ended up without them. And yes mom, I'm sure I'd rather not be a doctor. No mom, I can't fix the toaster just because I have a degree in computers. No mom, don't open that attachmen -- argggh. Just kidding (she'd never say or do any of those things).

Last saved by:	Romit
Revision number:	575
Total editing time:	2757 Minutes

575 (give or take) saves later, here I am. Stochastic. Ad-hoc. There, I said it.

Two strings walk into a bar. One string says, "Barkeep, I'd like a rum and coke&@31-*23&y3&!&)2#(&fjlOVnUer_#($..."

The other string shakes his head and says, "You'll have to excuse my friend; he's not null terminated."

// Academic schmacademic, it's one page and it's all mine
// runs

INTRODUCTION

An appreciation of music is perhaps one of the fundamental traits of being human. Whether coming from a mother's voice or a grand piano, music is as prevalent as the air around us. I myself have always had a deep appreciation for music, but unfortunately, any attempts to produce it have always resulted in annoyed relatives and exasperated music teachers. Guitars, drums, you name it, I can't play it.

Modern software tools are not much help either; many of the tools out there now require an in-depth knowledge of music to produce even the simplest tunes. Other alternatives use pre-recorded beats and loops to create melodies. While both approaches can result in satisfactory results, I believe the problem of creating harmonious sounds on a computer has a much simpler solution – a programming language. And why not? Languages such as C were created as a layer of abstraction over assembly language, which in turn hid the ugly details of machine code. Why should music be any different?

The final goal of this project is to create a high-level scripting language to hide the often confusing details of musical notation. Reminiscent of QBASIC's simplicity [1] (think PLAY statements), but using modern MIDI for sound output, the language places various digital instruments at the fingertips of anyone willing to grasp a few basic coding concepts. To ease the transition for programmers and non-programmers alike, an integrated development environment is included to help create error-free scripts.

READ ONCE, WRITE ANYWHERE?

Initial work on the project called for a mixture of C++ for the front-end and Flex/Bison for handling syntax and semantics. The shift to the Java platform came after learning about the javax.sound.midi classes introduced in recent versions of the Java runtime [9]. Instead of reinventing the wheel and writing functions for MIDI messages, the Java developers have provided a very robust set of methods geared for low-level MIDI message passing. Java's "Write once, run anywhere" motto made it an even more attractive candidate for MIDL and its accompanying IDE, MIDLMan. While various libraries including GTK, Qt, and Tk exist for cross-platform GUI development, only Java has these built-in. Thanks to Java's platform independence, MIDLMan has been successfully tested on Windows, Linux, and Solaris machines.

Of course, shifting to Java on the front-end creates a dilemma for the back-end; Flex and Bison only output C and C++. Writing a recursive-descent parser by hand, while not out of the question, detracts from the actual task at hand. Enter Another Tool for Language Recognition, more commonly known as ANTLR [12]. Described at greater length in subsequent sections, ANTLR is a combination lexer and parser which accepts grammars in EBNF notation and outputs equivalent Java, C++, C#, and even Python code. One of the greatest strengths I found in ANTLR was its seamless integration and ability to call my own Java classes with ease. Since the syntax is modeled after Lex/Yacc, the learning curve was fairly gentle.

SYNTHESIZED MUSIC

THE GOOD OLD DAYS™

A little more than 25 years ago, the musical composer was the poster-child of patience [6]. With MIDI standards not even a gleam in the eye of their developers, a large portion of a composer's time was devoted to making sure that each musician, responsible for a key instrument, was at a given place at a given time. Keeping in mind that musicians are not generally renowned for their punctuality, this was no trivial task.

Somehow having gathered said musicians in a cramped room -- not a recording studio as they are far too expensive -- there still remains the issue of hardware; not tracks and sequencers, but tapes, microphones, and a never-ending bed of wires. But hard part is over, right? Correcting notes and rearranging pieces of music should be no harder than flicking a switch or turning a knob. Surely.

In reality, all editing was performed with a trusty tape recorder. Missing a note meant rewinding to the exact spot of the error and recording the piece over. Rearranging the final composition was unheard of, as it meant re-playing each piece from the start.

The early 80's brought a paradigm shift in the way people wrote music. At the time, IBM had just introduced its now-famous IBM PC and a Japanese company named Roland Corporation saw its potential as a music sequencer. Teaming up with a company known for manufacturing analog sequencers, the two created an interface to mesh input from electronic keyboards to an ISA card. Using a simple protocol to recognize digital keyboard inputs, the so-called MIDI interface became the de-facto standard for creating and editing music with ease.

A MINOR IN MIDI

The Musical Instrument Digital Interface, more commonly known as MIDI, is a hardware and software protocol developed to allow sound synthesis equipment (synthesizers) to communicate with one another. The original protocol was sometimes called the MIDI wire protocol, as it consisted of real-time data being transmitted through connecting wires. The basis of this protocol was a series of messages consisting of status bytes indicating the event (Note On, Note Off, etc.), followed by a stream of data bytes.

On the hardware side of MIDI lie sequencers and synthesizers (although nowadays, both can be implemented in software). A typical MIDI synthesizer contains instrument sound samples, also known as *patches*, which may be activated by issuing a *Program Change* event. Patches make it possible to play a middle-C on a Grand Piano or a Harmonica with just one device. With 128 instruments defined in the General MIDI Standard, one synthesizer can replace an entire ensemble of instruments.

Modern hardware synthesizers have an In, Out, and Thru port. While the first two are self-explanatory, the Thru port can be thought of as a repeater; any data received from the In port is immediately transmitted via the Thru port to the next synthesizer down the MIDI chain. This makes it possible for a musician to play a note on one of several devices with different patches. However, what if one wishes to play different notes on each synthesizer at the same time? This is where channels come into play.

Each chained MIDI device receives data on a single channel. In order to allow selective transmission, the MIDI protocol was modified to allow the use of 16 distinct channels. This makes it possible to play an F# on a Grand Piano listening on channel 1 while playing a C on a Flute, which may be monitoring channel 7. The 16 channel limit is a result of a special MIDI construct known as a channel message which includes a status byte, 4 bits to specify a channel, and a stream of data bytes [17].

Putting all this together, we can simulate playing middle-C on a piano as follows:

1. A *Program Change* event is sent to a synthesizer to set the instrument type to 1, more commonly known as a Grand Piano as defined in the General MIDI Standard.
2. When the user depresses middle-C on the synthesizer input, the following byte stream is sent: 144 60 64, where 144 is the event code for *Note On* for Channel 1, 60 signifies the key pitch C, and 64 is the velocity or volume.
3. The note plays and is transmitted to any devices chained on the Out port.
4. A corresponding *Note Off* event must also be sent at a later time to make sure the key is released. Failure to do so is akin to holding down a key until the sound tapers off.

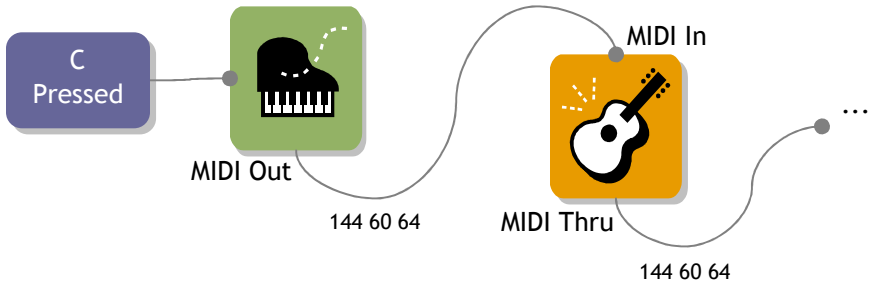

Figure 1

Synthesizers help solve the issue of space as one device replaces a roomful of instruments, but the responsibility of editing falls to sequencers. Usually a piece of computer software, a sequencer is analogous to a multi-track tape recorder. Each track holds one of the 16 MIDI channels and stores a series of MIDI events for future playback. Previously time-consuming tasks such as editing become straightforward as it is simply a matter of removing the offending event(s).

Simple MIDI Files

MIDI was originally developed to help musicians control multiple devices simultaneously at a live performance. In this case, there was no need to store when a note was pressed or released since it was all real-time data. Of course, this poses a problem if one wishes to save the sequence for later playback. To overcome this limitation, Simple MIDI Files (SMF) were developed to append timing data known as ticks to MIDI events. Along with the tempo of a MIDI sequence, ticks effectively assign clock time to events. In the sample sequence shown below, we see pairs of Note On and Off events occurring at different ticks to produce sustained notes:

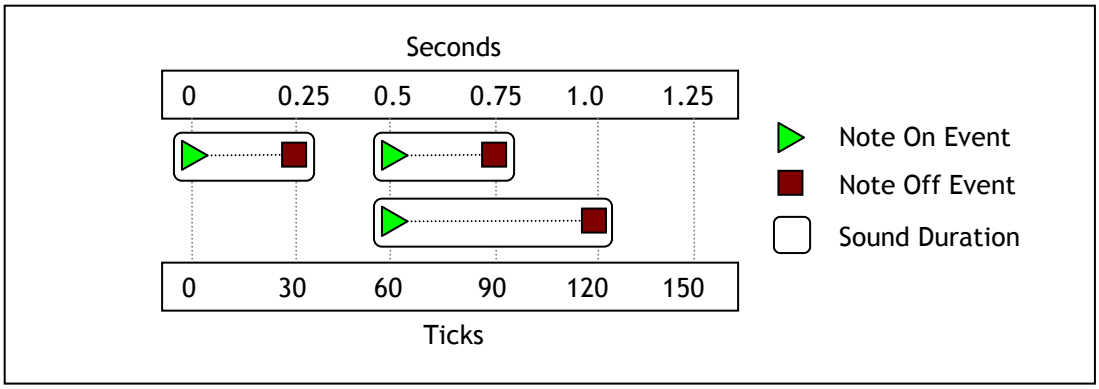

Figure 2

Ticks and MIDI events form a one-to-many relationship that makes it possible for events to share one tick, thereby allowing multiple notes to be played simultaneously [7].

BREWING MUSIC

HOW JAVA HANDLES MIDI

As stated earlier, one of the main reasons for switching to Java was due to its excellent MIDI library functions, which nearly mirror real-world synthesis. The essential classes for MIDI manipulation in Java can be placed in two groups: devices (Sequencer and Synthesizer) and data (Sequence, Track, MidiEvent, and MidiMessage). Both are necessary in order to generate music. The data components are arranged in a hierarchical fashion as follows: a Sequence contains a number of Tracks which in turn contain a series of MidiEvents. Recalling that MIDI is a real-time protocol, a MidiEvent is simply a class that ties timing information to a MidiMessage.

Sequencers in Java perform the basic operations of loading a series of MidiEvents from a Sequence or MIDI file and transmitting the data to an output device. Synthesizers are the workhorse of the Java MIDI classes; they receive input from a Sequencer and actually generate sound. Like physical synthesizers, the Java counterpart doesn't always need a Sequencer for input; in fact, it is entirely possible to send a series of MidiMessages (without timing data) to a Synthesizer and generate sound. For example, an application created to model a piano where mouse clicks create sounds has no need to remember when the mouse was clicked. Since the events generated are realtime, the program need only send a Note On MidiMessage to a Synthesizer, which plays the sound immediately.

Delving even deeper, MidiMessage is a parent class for various types of MIDI messages: ShortMessage, SysexMessage, and MetaMessage. ShortMessage has several constructors for both channel-specific and global messages and is used for most MIDI messages including Note On and Note Off. SysexMessages are not as prevalent as the other two message types as they generally contain manufacturer specific instructions. MetaMessages are a little different from the other two as they are not part of the MIDI wire protocol. They usually contain Sequencer-specific information like copyright notices, lyrics, and tempo information [9].

OF TICKS AND TIME

As revealed earlier, a MidiEvent is a MidiMessage tied to a timestamp. In the current context, a MidiEvent timestamp is not in seconds but an abstract measurement known as a tick. The duration of a tick is dependent on two attributes – tempo and resolution. The resolution of a Sequence is declared upon initialization:

```
Sequence mySeq = new Sequence(Sequence.PPQ, 120);
```

PPQ stands for *pulses per quarter note* and denotes the number of ticks given to a quarter note. The last piece of the puzzle which links ticks to seconds is shown below:

Number of ticks in a second = Resolution (in PPQ) x Tempo (in beats per minute) / 60

The default tempo is set to 120 beats per minute (BPM) and can be changed either via a MetaMessage [2] or a call to one of the various tempo methods provided by the Java Synthesizer.

Then, given a default tempo of 120 BPM and a resolution of 120 PPQ, there are 120 x 120 / 60 = 240 ticks per second. Since a quarter note is 120 ticks (set by PPQ), it lasts exactly 240 / 120 = 0.5 seconds.

Adding events to a Sequence is now simply a matter of creating a MidiMessage and placing it on a track at a particular tick. As the reader has probably guessed, an event at tick 0 occurs before an event at tick 1, while two events with the same tick occur simultaneously.

JUST CODE IT

With the above in mind, the listing for playing a middle C lasting 0.5 seconds is as follows (for the sake of brevity, required exception handling code and import statements have been omitted) [5]:

```
1.  public class MiddleC {
2.    public static void main(String [] args) {
3.          Sequence mySong;
4.          Track myTrack;
5.          final Sequencer sequencer;
6.                Synthesizer synthesizer;
7.                final Object lock;

8.          mySong = new Sequence(Sequence.PPQ, 120);
9.          myTrack = mySong.createTrack();
10.         lock = new Object();

11.         ShortMessage noteOnMsg = new ShortMessage();
12.         ShortMessage noteOffMsg = new ShortMessage();
13.         noteOnMsg.setMessage(ShortMessage.NOTE_ON, 0, 60, 64);
14.         noteOffMsg.setMessage(ShortMessage.NOTE_OFF, 0, 60, 0);
15.         myTrack.add(new MidiEvent(noteOnMsg, 0));
16.         myTrack.add(new MidiEvent(noteOffMsg, 120));

17.         sequencer = MidiSystem.getSequencer();
18.         sequencer.setSequence(mySong);
19.         sequencer.open();

20.         synthesizer = MidiSystem.getSynthesizer();
21.         synthesizer.open();

22.         sequencer.getTransmitter().setReceiver(
            synthesizer.getReceiver());

23.         sequencer.addMetaEventListener() { new MetaEventListener() {
24.               public void meta(MetaEvent mEvent) {
25.                     if ( mEvent.getType() == 47 ) {
26.                           synchronized(lock) {
27.                                 lock.notify();
28.                           }
29.                     }
30.               }
31.         });

32.         sequencer.start();

33.         while ( sequencer.isRunning() ) {
34.               lock.wait();
35.         }

36.         if ( sequencer != null ) sequencer.close();
37.         if ( synthesizer != null ) synthesizer().close();
38.   }
39. }
```

Lines 1 through 6 are generally uninteresting; they simply declare some required variables for preparing a MIDI Sequence. Line 8 actually creates the Sequence and assigns 120 *ticks* to a quarter note (hence, playing a half note would consist of a Note On event followed by a Note Off event 240 ticks later). As discussed previously, since the default tempo is 120 BPM, 120 ticks is a duration of 0.5 seconds.

Line 9 is a bit peculiar. In order to add a track, one must call the Sequence's createTrack method, which returns a reference to the new Track, and can then be modified as necessary. Note that there is no function such as Sequence.getTrack(int) which would have seemed more logical. Line 10, while not required, is useful for reasons we shall discuss later.

In lines 11 through 16, we see the actual notes being added. A pair of ShortMessages are created, the status byte set to predefined constants, and the data bytes follow. In the first case (Line 13), the syntax is explained as follows:

```
noteOnMsg.setMessage(ShortMessage.NOTE_ON, 0, 60, 64);
```

`ShortMessage.NOTE_ON` – A predefined status byte (0x90) representing the Note On event.
`0` – Assigns this command to channel 0.
`60` – The pitch of the note, in this case a C in octave 5.
`64` – The volume or *velocity* of the note.

Line 14 raises an interesting point; do we really need a Note Off event? Well, it depends. Without a corresponding Note Off event, the sound generated would be akin to depressing middle-C and not letting go; the sound would gradually taper off on its own. The MIDI standard also allows Note Off events to be replaced with Note On events with a velocity of 0.

Once the Sequence is complete, it must be loaded in the Sequencer and sent to the default Synthesizer for playback as shown in lines 17 to 21. Recalling that sequencers and synthesizers in the real world have MIDI In and Out ports, Java's equivalent is in the form of receivers and transmitters. In line 22, the Sequencer's transmitter (Out port) is linked to the Synthesizer's receiver (In port). Thus, when the sequencer starts playing, its output is redirected and handled by the synthesizer. Recall that only the Synthesizer can generate sounds.

The reader may be wondering; how do we know when the sequencer has finished playing? The answer is through the use of a MetaEvent. Each event in the MIDI sequence has a particular event type such as NOTE_ON or NOTE_OFF. 47 is a special event signifying the end of a track and is added automatically by the Java subsystem after the last user-added event.

Line 32, `sequencer.start()` is a bit problematic as it is non-blocking. If one were to perform cleanup immediately after starting playback by skipping to Line 36, the program would seem to exit without playing any sounds. In order to correct this issue, we use the generic Object we declared in line 10. Immediately after playback, the program performs a check at line 33 to see if the sequencer is still playing, and if not, the lock object blocks. The block is released only when the MetaListener catches an End of Track event, after which the Sequencer and Synthesizer objects are released for future use.

Contrast this listing with MIDL's `{ PLAY C }` and its advantage is self evident.

RELATED WORK

While modern musical notation has a set of standard representation, there remain many intricacies which are cause for debate. Translating notation to digital form raises even more complications as MIDI holds no means of conveying notation such as *ties* and *slurs*. Despite such issues, there have been several attempts to formalize a digital representation for sheet music.

SCORE AND MIDI EDITORS

Arguably the most robust MIDI sequencer on the Linux platform, Rosegarden [16] is a full-featured audio sequencer for experienced musicians. Most sequencer applications mimic its timeline-based working area and receive input directly via MIDI device input, or by dragging and dropping notes onto the workspace.

GOING LOOPY

The other end of the spectrum is the home of the jukebox application. Notable examples include Logic Pro, and its little brother, Garageband. Also known as loop based editors, these applications are generally based on the idea of using pre-recorded music samples. The composer selects a sample, modifies various attributes, and loops it on a track. The process is repeated until several edited samples play in unison to produce a unique melody.

JAVA HELPERS – JFUGUE AND JMUSIC

Application developers will find JFugue [10], a robust Java API for generating music, an excellent tool for designing Java-based programs. Like most modern music creation tools, JFugue abstracts the MIDI file structure thereby allowing the user to concentrate on the composition. A sample Java program implementing the API is listed below:

```
import org.innix.jfugue.Player;
import org.innix.jfugue.Pattern;

public class MiddleC {
      public static void main(String[] args)
      {
            Player player = new Player();
            Pattern pattern = new Pattern("C");
            player.play(song);
      }
}
```

In JFugue, all compositions stem from *patterns*, which in turn may contain notes, harmonies, and tempo and instrument changes.

jMusic is another add-on library for Java which allows on-the-fly music generation [3]. Maintained by the music department at the Queensland University of Technology in Brisbane, Australia, it serves as an excellent tool for beginners and experienced musicians alike. The general paradigm seems to mirror that of JFugue's and as such, the syntax is very similar.

XML AND MIDI

XML has become more visible in the last few years with its ability to seamlessly represent data without any ties to platforms or programs. XMidi's developer, Peter Loeb, took advantage of XML's simple representation and created a Document Type Definition (essentially a grammar) for MIDI events.

Paired with a set of tools to convert from XML to MIDI and vice versa, XMidi's most attractive feature is the ability to represent MIDI data in a human readable form [11].

Other parties have also expressed an interest in meshing XML with music, the most notable example being from a firm specializing in music software named Recordare. Recordare ships several products that use MusicXML as a common data store.

THE MIDL GROUND

It seems we have a problem with each of the solutions presented above. Score editors are too overwhelming for someone new to music, while jukeboxes do not offer enough precision for those already familiar with the notation. MIDI libraries are all well and good, provided one is familiar with their host languages.

MIDL is a self-contained language with a handful of keywords, which allows neophytes to compose simple pieces, but retains the ability to create more precise works by more gifted composers. The only pre-requisite for using MIDL is a rudimentary grasp of scales, octaves, and note lengths.

MIDL IN A NUTSHELL

The MIDL grammar can be separated into various classes of statements: assignments, playback, comments, and global settings. Global settings, which include tempo and title assignments, must precede the starting brace. Conversely, playback and assignments must be inside the braces. A general overview of the MIDL grammar is best exhibited with a simple listing:

```
1.  TEMPO = 140
2.  TITLE = "Frere Jacques"
3.  {
4.    $verseA = CDEC
5.    $verseB = EFGh
6.    $verseC = GeAeGeFeEC
7.    $verseD = CG4Ch

8.    VOICE 0 = "Honky-tonk Piano"
9.    VOICE 1 = "Acoustic Grand Piano"
10.   PLAY (CEG)  // Chord for emphasis
11.   PLAY REPEAT 2 $verseA
12.   PLAY REPEAT 2 $verseB
13.   PLAY REPEAT 2 $verseC
14.   PLAY REPEAT 2 $verseD
15.   PLAY Rq
16. }
```

GLOBAL SETTINGS

Line 1 - TEMPO [optional]: TEMPO followed by an integer representing the beats per minute. Unless otherwise specified, the tempo is set to 120 BPM. This is the direct opposite of how MIDI expresses tempo -- as microseconds per quarter note.

Line 2 - TITLE [optional]: TITLE followed by a string enclosed in quotes. This adds a MetaEvent to the MIDI Sequence containing the title of the song.

ASSIGNMENTS

Lines 4 through 7 - Variable Assignments [optional]: A variable name is preceded by a dollar sign and stores a sequence of notes. Variables may contain alphanumeric characters but must begin with a dollar sign and letter.

NOTE REPRESENTATION

Lines 5 through 7 - Here we have an example of MIDL's advanced note representation syntax. Taking the following snippet of code as an example:

(Cvol32 E3bhvol32)

( ) - Parentheses state that both notes are part of a harmony and must be played simultaneously.
vol32 - Sets the note volume in a range from 0 to 127. Unless otherwise specified, the default volume is set to 64.
C - A note may range from values A through G, or R to signify a rest.
3 - Sets the octave which may range from 0 through 10 and defaults to 5.
b - Example of an accidental, in this case, a flat. Sharps are represented by a pound sign (#).
h - Sets the note length. This may range from whole, half, quarter, eight, sixteenth, or a dot (.). If omitted, all notes are quarter notes. The dot notation extends the note by 150% such that with a PPQ

of 120, a C. will last 120 x 1.5 = 180 ticks.

A note may be represented in a more formal manner as follows:

[A..G | R](b | #)?(0..10)?(w | h | q | e | s | (.)?)?(vol[0..127])?

VOICE ASSIGNMENT

Line 8 - Voice assignment [optional]: VOICE ranging from 0 through 15 = a General MIDI instrument within quotes. MIDL offers 16 voices to choose from. Voice 0 plays a Grand Piano by default, while the others must be explicitly assigned.

COMMENTS

Comments are preceded by a double slash (//) and terminate with a newline as shown on Line 10.

PLAYBACK COMMANDS

Lines 10 through 15 showcase typical playback commands in MIDL. As stated previously, parentheses denote harmonies while dollar signs specify predefined melodies. The optional REPEAT keyword immediately follows the PLAY statement and repeats the trailing note sequence. Note that each variable must have its own PLAY statement. For example, the following code is *not* valid:

```
PLAY $varA $varB  // Invalid!
PLAY $varC (CEG)  // Also invalid!
```

An alternative correct syntax is:

```
PLAY $varA PLAY $varB PLAY $varC
PLAY (CEG)
```

MISCELLANEOUS ISSUES

MIDL is a case-sensitive language meaning play is not equivalent to PLAY. White spaces and newlines are ignored although recommended for readability purposes. All voice assignments other than voice 0 must be assigned to an instrument.

BUILDING MIDL

The MIDL interpreter can be divided into high and low level components; the former consists of five custom classes that result in more concise code, while the latter includes the ANTLR-generated lexer and parser along with a *MIDIComposer* class. MIDIComposer is simply a set of methods which link the ANTLR tools to the low-level MIDI classes. Its primary functions include initializing, playing, and editing Sequences. Each class will be examined in the next section.

Input to the interpreter is broken down into tokens by the lexer, which are then fed through the parser. The parser is composed of a set of rules which match combinations of tokens, and with the help of methods in the manager classes, perform a user-specified action. For example, the now ubiquitous `{PLAY C}` command is interpreted by the lexical analyzer as: BEGIN, PLAY, NOTESEQUENCE, END. This sequence of tokens matches a rule in the parser named *song*, which adds the parsed note, middle-C, to a new MIDI Sequence.

NOTELENGTH

Primarily a helper class, NoteLength contains a `static final doubles` indicating note lengths from 1/16 notes to whole notes. It is used exclusively by the parser to assign note lengths.

NOTE

The building block of a MIDI Sequence, the Note object has several attributes including `value, duration, volume, attack,` and `decay`. It is particularly important as Java's MIDI specifications have no mention of a Note object, since MIDI doesn't represent them anyway. The Note class then serves a container for several variables, which are at some point passed into a Track object. An interesting dilemma arose during the early stages of development and can be demonstrated in the following listing:

```
1.  Note middleC = new Note(60, NoteLength.Q);
2.  Note copyC = middleC;
3.  copyC.setDuration(NoteLength.W);
```

Clearly, the intent of this code is to make a copy of middleC, then modify the copy while leaving the original intact. In actuality, line 2 assigns the memory location of middleC to copyC so any changes to copyC also affects middleC and vice versa. The corrected code is shown below:

```
2.  Note copyC = new Note(middleC);
```

The Note class defines a special constructor which accepts an existing Note object and creates a deep copy of it as a new object. Every instance of note passing is performed in this manner in the MIDL parser.

The astute reader will notice that the attack and decay attributes, while declared, are not used in any member function; this is intentional. Both attributes range from 0 to 127 and represent how hard a note is struck; the attack determines how fast the note is pressed, while the decay describes how fast it is released. By default, MIDI sets both values to 64 [8].

As an example, a violin has a high attack if bowed abruptly and a low attack if bowed gradually [15]. During MIDL development, I had good results without manually tweaking these attributes, and as such, decided to leave them out of the final implementation. Adding either element is a trivial matter; a simple token addition to the ANTLR source file and an accompanying semantic rule are all that is required.

MELODY

Again, since Java holds no provisions for representing notes, the Melody class was created as a container to hold Note objects. The heart of the class lies in the ArrayList data structure which allows Melody to contain any number of notes without defining a size at compile-time. Borrowing the concept of an Iterator from the C++ Standard Template Library, external callers can step through each Note object with ease. The primary writing function named `add()`takes a Note object as input and appends it to the internal ArrayList.

A cursory glance at Melody.java shows it to be nothing more than a container. The only distinguishing feature is the member function named `isHarmony()`, which returns true if the melody contains more than one note. Couldn't I have simply used an ArrayList of Note objects for a data structure and avoided the Melody entirely? The answer is not clear at the moment, but will be discussed when we analyze MIDL's parsing component.

MIDIMSG

The first layer of abstraction after Java's MIDI classes, MIDIMsg is a simple class containing a set of statically accessible methods, which return MidiMessage objects representing Note On, Note Off, and Tempo and Program Change events. There is no error checking performed in this class and any invalid data results in the method throwing an exception and exiting. If one were to extend MIDL to handle additional events such as Sustain or Reverb, this would be a good spot for an auxiliary function. An example of a skeleton function is as follows:

```
1.  public static ShortMessage someEvent(int data, int channel) {
2.    ShortMessage someMsg = new ShortMessage();
3.    try {
4.    someMsg.setMessage(status byte, data byte[s], channel optional]);
5.    } catch (InvalidMidiDataException invdata) { ... }
6.    return someMsg;
7.  }
```

Of course, the return type may be a `SysexMessage` or `MetaMessage` and the `setMessage` function would vary as necessary. If no other error handling is present, the `InvalidMidiDataException` will fire upon receiving out-of-bounds data (for example, attempting to set the volume of a note to a value that is not between 0 and 127).

Before we end our discussion of the MIDIMsg class, the tempo method requires a bit of explanation:

```
1.  public static MetaMessage tempo(int tempoMPQ, int channel)
```

Our prior discussion regarding a note's duration involved the tempo in beats per minute and the resolution of a Sequence in pulses (or ticks) per quarter note. The listing above demonstrates a deviation in the actual MIDI message; instead of receiving a PPQ value, the tempo is passed in as microseconds per quarter, or MPQ. The conversion from BPM to MPQ is as follows:

MPQ = 60000000 / BPM

While the tempo method receives an MPQ value, the user simply sets the tempo in BPM and the conversion is handled in MIDL's syntactic analyzer.

MIDITRACK

MIDITrack is an extension and in a way, a simplified version of the Java Track. A limitation of the MIDI standard is the set number of 16 channels, meaning that there can only be 16 distinct instruments

playing at one time. Note that this does not limit one to 16 instruments in a sequence; it simply means one can control only 16 instruments at a time.

MIDI files have fashioned a workaround by using Tracks, each of which can have 16 distinct channels. Therefore, each track can have up to 16 instruments being controlled simultaneously.

The MIDITrack class rejects this notion of tracks and channels being different and treats both terms as equivalent. It does so by instructing each track to listen on its corresponding channel (track 0 listens on channel 0, track 1 on channel 1, etc.). The MIDL interpreter can then play up to 16 instruments at the same time, or have 16 separate tracks.

The justification of this seemingly crippling feature is threefold. First, the MIDI documentation I found made no clarifications of how MIDI files can play distinct notes on more than 16 instruments *simultaneously*. Assume a MIDI file has two tracks, both of which have 16 channels. Since a Note On message is a channel-wide event, any instrument assigned to a particular channel will play, regardless of the track it lies on. Second, most software synthesizers are unable to play more than 16 instruments at a time. And lastly, as MIDL is geared towards fairly simple musical pieces, composers managing a large number of instruments should probably look at timeline based editing tools.

Some of the more important methods in MIDITrack include:

`MIDITrack(Sequence)` - The constructor takes a parent Sequence as an argument, assigns a channel to the track, and sets the tempo to match that of the sequence.

`noteEvent(Note, long)` - Recalling that playing a note in MIDI is actually a pair of On and Off events, this method takes the note in question as well as a variable named position as arguments. The Note On event is added at the specified position while the corresponding Note Off event is added at (position + the note's duration).

Similar to the normal Track object, a MIDITrack has provisions to add notes, although they must be in the form of Melodies. The function `addNote` requires further inspection:

```
1.  public void addNote(Melody m) {

2.    if ( m.isHarmony() ) {
3.          int harmonySize = m.size();
4.          double longestDuration = 0;

5.          while ( harmonySize-- > 0 ) {
6.                Note note = m.next();
7.                noteEvent(note, position);
8.                if ( longestDuration < note.getDuration() )
9.                      longestDuration = note.getDuration();
10.         }
11.         movePosition((long) (longestDuration * ppq));

12.   } else {
13.         Note note = m.next();
14.         noteEvent(note, position);
15.         movePosition((long) (note.getDuration() * ppq));
16.   }

17.   m.rewind();
18.  }
```

The reader may recall that a Melody can be a single note or a chord containing multiple notes (in practice, only 2 or 3). The algorithm for adding a note event to a Track is as follows:

```
Is the Melody a single Note?
  If so, add the note and increment the current tick to the note's duration
  Else
    Set longest note = 0
    For each note N in the melody
      Add note N to the track
      If N's length is longer than the longest note, set the longest note to
      N's length
    Next
    Increment the current tick by the longest note
  End If
Rewind the Melody's iterator
```

Why are there so many extra steps for adding a harmony? What not use the following algorithm for adding harmonies?:

```
Set last note length = 0
For each note N in the melody
  Add note N to the track
  Set last note length = N
Next
Increment the current tick by the last note length
```

Imagine the composer wishes to play the notes C E G together followed by an A. Assuming that a quarter note spans 120 ticks, C, G, and A are all quarter notes, and the E is a half note, we can see the timeline below:

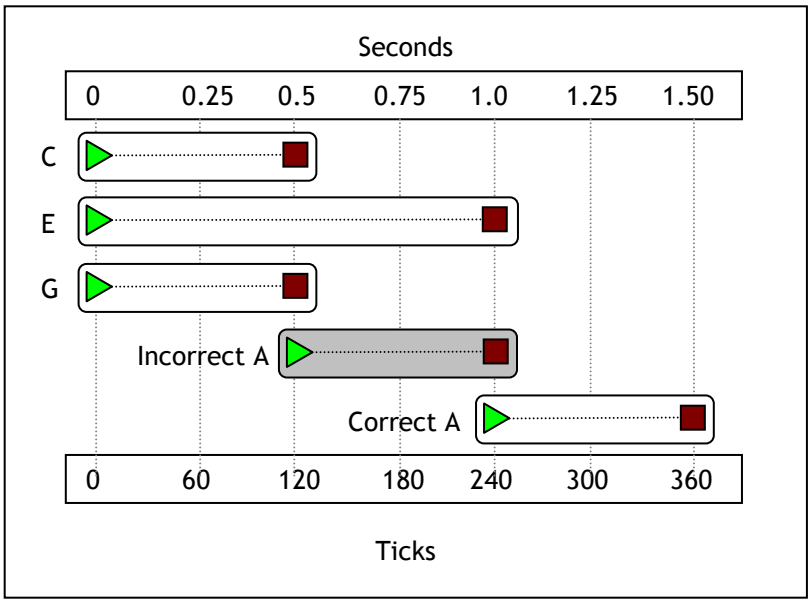

Figure 3

The notes C, E, and G play as they should; the problem lies with the starting position of the A note. Using the correct algorithm, A should play only after E completes playing. In the errant example, A begins to play while E still has 120 ticks till completion and finishes at the same time as E.

Early versions of MIDL naively assumed that a harmony would always be composed of notes of identical lengths. With the new code in place, harmonies with different length notes are possible.

In addition to the methods above, the MIDITrack class also contains a few auxiliary methods including those for changing a track's tempo, adding title information, moving the current tick, and applying a patch to change a specified channel's instrument.

MIDIComposer

Even though we have yet to discuss the heart of MIDL, that being the lexer and parser, let us peek into MIDIComposer, the class that links the grammar tools with the MIDI generators. For such an important class, its composition is deceptively simple. Upon initialization, MIDIComposer creates a Sequencer, Synthesizer, 14 MIDITracks, and a Sequence. One may go so far as to think of MIDIComposer as a container class for the low level classes.

LEXER WALKTHROUGH

The ANTLR lexer code is fairly uninteresting string matching based on EBNF constructs. However, there are a few portions of code worth exploring further:

```
options {
      k=2;
}
```

k = 2 is a special ANTLR directive which sets the lookahead to two characters. This is necessary to parse tokens such as TEMPO and TITLE or newline characters. While we may left-factor to form an LL(1) grammar, it comes at the cost of readability.

```
BEGIN options { paraphrase = "opening brace ({)"; }: '{';
```

The previous line is a simple example of ANTLR's error-handling capabilities. If the interpreter is expecting an opening brace and receives something else, the paraphrase keyword chimes in with "Expecting opening brace ({)" as opposed to the more cryptic default of "Expecting Token BEGIN."

```
NEWLINE: ("\r\n" | '\r' | '\n') { newline(); $setType(Token.SKIP); };
COMMENT: "//" (~('\n'|'\r'))* ('\n'|'\r') { $setType(Token.SKIP); };
```

The NEWLINE token matches three different newline sequences, each representing a newline on Windows, Macintosh, and Unix systems respectively and takes two actions; the first is calling `newline`, an ANTLR method which increments its internal line counter (this is useful later on for error reporting), while the second is instructing ANTLR not to send the token to the parser.

Regular expressions allow the COMMENT token to function properly. The rule states: find two sets of forward slashes (//), followed by any number of characters which are NOT newlines, and finish off by matching a newline.

Another point of interest is the uniform structure of ANTLR notation whether it is in the lexer or parser. Note that most token rules follow the general syntax of

```
TOKEN_NAME: Regular Expression or Literal String to match { action }
```

PARSER WALKTHROUGH

Once the lexer component classifies input as a token that is not a whitespace or comment, an object denoting the token type, value, and various other attributes is passed to the parser. The parser generated by ANTLR is your typical recursive descent parser and matches streams of tokens to rules. The basic idea of a recursive descent parser is simple; each function matches a specific rule, which in turn matches subrules. Take the following trivial language as an example:

S -> Dog | Cat | Cow
Dog -> ARFF
Cat -> MEOW
Cow -> MOO

The Barnyard language denoted by the above grammar matches one of three strings, ARFF, MEOW, or MOO. Psuedocode for a parser is as follows [14]:

```
function match_S
  switch(currentCharacter)
    case TOKEN_A:
      match_Dog()
    case TOKEN_M:
      if (currentCharacter + 1) == TOKEN_E then match_Cat()
      else if (currentCharacter + 1) == TOKEN_O then match_Cow()
  break
end function

function match_Dog()
  consume(TOKEN_A) consume(TOKEN_R) consume(TOKEN_F) consume(TOKEN_F)
end function

function match_Cat()
  consume(TOKEN_M) consume(TOKEN_E) consume(TOKEN_O) consume(TOKEN_W)
end function

function match_Cow()
  consume(TOKEN_M) consume(TOKEN_O) consume(TOKEN_O)
end function

function consume(TOKEN token)
  if ( currentCharacter == token.value ) then currentCharacter++
  else Display error and exit or attempt recovery
end function
```

Attempting to validate input with the language begins by calling the top level rule, in this case, S. The parser uses a one token lookahead to read the first character, and calls a corresponding subrule. With an input of ARFF, the match_S method immediately recognizes A as a possible start of a Dog rule and as a result, calls the match_Dog method. Assuming A is followed by R, F, and F, the match_Dog method terminates without any errors. If however we have ARFA, the program terminates.

Things change a bit with MOO as an input since both the Cow and Cat rules begin with an M. In this case, the parser has to look at one more character to differentiate between a MOO and MEOW. The language as defined is thus LL(2). To avoid a two character lookahead, we could have utilized the following left-factored grammar.

```
S -> Dog | CowOrCat
```

```
Dog -> ARFF
CowOrCat -> M (Cow | Cat)
Cow -> OO
Cat -> EOW
```

Although the parser for this LL(1) grammar performs faster, it comes at the cost of comprehension. It takes more than a quick glance to know the three strings recognized by the language.

It is fairly evident from the pseudocode presented above that writing a recursive descent parser by hand for anything other than a tiny language is far too time consuming. Note that the above listing is simply a barebones parser; it has no provisions for error checking or executing external code. Fortunately, with compiler construction tools like ANTLR, the designer can concentrate on the structure of the language instead of having to deal with the nitty-gritty details.

MIDL GRAMMAR

```
Song -> (TempoAssignment)? (TitleAssignment)? { (VoiceAssignment |
VariableAssignment | PlayCommand)+ } EOF
TempoAssignment -> TEMPO = INT
TitleAssignment -> TITLE = STRING
VoiceAssignment -> VOICE INT = STRING
VariableAssignment -> VARIABLE = (NoteSequence)+
NoteSequence -> Note
Note -> (A|B|C|D|E|F|G|R) (#|b)? (INT)? (LENGTH)? (INT)?
PlayCommand -> PLAY (WITH VOICE INT)? (REPEAT INT)? (NoteSequence |
VariableSequence)
VariableSequence -> VARIABLE
```

An ANTLR input file is composed of several portions: header, options, and finally, the lexer and parser class. MIDL's header consists of the following:

```
1.  {
2.    import java.util.*;
3.    import midl.*;
4.    import javax.sound.midi.*;
5.  }
```

The Java utility namespace is included because it is used by MIDL's symbol table component to store variables. midl.* contains the lower level components discussed in a previous section. The Java MIDI libraries are required to recognize the Sequence objects. Next we have our parser class, which begins with a header component of its own:

```
1.  class MidlParser extends Parser;
2.  options { defaultErrorHandler=false; }
3.  {
4.    MIDIComposer composer = new MIDIComposer();
5.    Map<String, ArrayList<Melody>> symbolTable = new HashMap<String,
      ArrayList<Melody>>();
6.  }
```

Line 1 defines the name of our parser class, which inherits its attributes and methods from ANTLR's Parser class. Among the inherited methods are provisions for error recovery by skipping ahead in the token stream until valid input is found. While this is fine for inputting data files, MIDL syntax needs to be strictly enforced in order to protect the integrity of the Sequence created. If there is an error of any kind in the MIDL input, we must halt compilation and throw an exception. Line 2 ensures this by disabling the ANTLR recovery methods.

Line 5 defines a symbol table based on a associative array data type. Variables in MIDL are stored as a string denoting the variable name, and an ArrayList of Melody objects. This allows a variable to store anything from a single note to multiple notes mixed in with harmonies.

SONG

```
1.  song returns [Sequence output = null]:
2.    (tempoAssignment)?
3.    (titleAssignment)?
4.    BEGIN
5.    (voiceAssignment | varAssignment | playCommand)+
6.    END EOF{
7.          output = composer.getSequence();
8.    }
9.  ;
```

Already, Line 1 throws us deep into ANTLR territory. Let's break it down piece by piece starting with `song`, which is defines the name of the rule. In ANTLR, rules can return variables, and in this case, `returns [Sequence output = null]` states that the song rule returns a Sequence object. Return variables must be declared and instantiated in square brackets.

Lines 2 through 6 proceed in recursive descent fashion, calling additional rule methods as necessary and end in the special EOF token. Omitting the EOF token allows input even after a valid recognized sequence.

The reader will notice that line 7 is enclosed in curly brackets. This is not to be confused with normal Java or C++ syntax; the brackets are ANTLR specific and denote external code which should be executed after the preceding token, in this case, END. Note that a precondition for executing line 7 is that all the rules prior to it have executed properly; that is, there are no errors in the input. Once this is assured, we set output (declared previously) to point to the Sequence created by the MIDIComposer object and it returns to the calling function.

The following psuedocode listing demonstrates how the lexer/parser code could be called from a command-line based MIDL interpreter:

```
1.  MidlLexer lexerObj
2.  MidlParser parserObj
3.  lexerObj receive input from standard input
4.  parserObj receive input from lexerObj
5.  try
6.    Sequence output = parserObj.song()
7.  catch TokenStreamException, RecognitionException
8.  If Exceptions exist, display an error and halt
9.  Otherwise, do something with Sequence output (i.e.: play or write
    to file)
```

While fairly straightforward, one point worth mentioning is the error handling exceptions found on line 7. By default, the ANTLR Parser throws TokenStream and Recognition exceptions to handle errors while reading the input stream and invalid token sequences respectively. Semantic errors such as attempting to set a note's volume to a value greater than 127 result in a SemanticException, which is derived from a RecognitionException. Each error must be caught and dealt with by the driver program; MIDL only recognizes errors, and it is up to the user to correct them.

TEMPO ASSIGNMENT

```
1.  tempoAssignment:
2.    TEMPO EQUALS tokenTempo:INT {
3.      int bpm = Integer.parseInt(tokenTempo.getText());
4.      int mpq = 60000000 / bpm;
5.      composer.changeTempo(mpq);
6.    }
7.  ;
```

Recall that the lexer does not pass tokens, it passes token objects. As a result, it is possible to determine both the type and value of an input as shown in line 2. Here, the tempoAssignment is looking for the words `TEMPO` = followed by some integer. `tokenTempo:INT` assigns whatever was passed in as an INT to the variable tokenTempo, which is used in the following line. Noting from our previous discussion of the discrepancy of tempo units in Java and MIDI, lines 3 and 4 handle the conversion from beats per minute (user supplied) to microseconds per quarter (MIDI event friendly). Line 5 calls a MIDIComposer method to write the event to all the tracks in the Sequence.

TITLE ASSIGNMENT

```
1.  titleAssignment:
2.    TITLE EQUALS tokenTitle:STRING {
3.      String parsedTitle = tokenTitle.getText().substring(1,
        tokenTitle.getText().length() - 1);
4.      composer.addTitle(parsedTitle);
5.  }
6.  ;
```

Line 3 strips the quotation marks from the title and passes the resulting string to line 4. The addTitle method creates a Sequence/Track Name event (0x03) and inserts it as the first event of the first track. An interesting issue with this keyword is that none of the major software MIDI players seem to acknowlege the event. As a result, future revisions of MIDL may replace this with a Copyright Event (0x02); an event recognized by most media players.

VOICEASSIGNMENT

```
1.  voiceAssignment
2.  { int voice, instrument = 0; }
3.  :
4.  (VOICE
5.  tokenVoice:INT {
6.    voice = Integer.parseInt(tokenVoice.getText());
7.    if ( voice > 14 )
8.      throw new SemanticException("Voices must be between 0 and 14",
        tokenVoice.getFilename(), tokenVoice.getLine(),
        tokenVoice.getColumn());
9.  }

10. EQUALS

11. (tokenInstrumentNum:INT {
12.   instrument = Integer.parseInt(tokenInstrumentNum.getText()) - 1;
13.   if ( instrument < 0 || instrument > 127 )
14.     throw new SemanticException("Instruments must be between 0 and
        127", tokenInstrumentNum.getFilename(),
```

```
        tokenInstrumentNum.getLine(), tokenInstrumentNum.getColumn());
15. }

16. | tokenInstrumentString:STRING {
17.  String parsedInstrument = tokenInstrumentString.getText
     ().substring(1, tokenInstrumentString.getText().length() - 1);
18. instrument = composer.getInstrument(parsedInstrument);
19. if ( instrument == -1 )
20.   throw new SemanticException("No such instrument",
      tokenInstrumentString.getFilename(), tokenInstrumentString.getLine(),
      tokenInstrumentString.getColumn());
21. } )

22. )
23. { composer.changeInstrument(instrument, voice); }
24. ;
```

A bit of a departure from the comparatively simple rules discussed previously, voiceAssignment assigns instruments to voices either by accepting the instrument number as defined in the General MIDI Standard, or by the name of the instrument itself. This binary operation is signified by the presence of the | operator on line 16.

The first token the rule accepts is the literal string VOICE followed by an integer denoting the voice itself. Lines 7 and 8 make certain that the voice assigned is within the 14 limit imposed by MIDL.

The next token after the equals sign on line 11 decides whether the code should branch to line 11 or 16. If the user assigns an instrument by name, the sample input is as follows: `VOICE 0 = "Piano"`. Line 17 simply strips the quotation marks around Piano and calls the `getInstrument` helper function in the ever-useful MIDIComposer class. This method in turn scans through the first 128 instruments in the GM Standard and returns an array index corresponding to a match. If the user inputs an invalid instrument, `getInstrument` returns a -1 and the rule exits by throwing a SemanticException.

Assuming there were no errors, line 23 writes a program change event with the correct instrument patch to the Sequence.

VARIABLEASSIGNMENT

```
1.  varAssignment
2.  {
3.    String lValue;
4.    ArrayList<Melody> rValue = new ArrayList<Melody>();
5.    Melody melody;
6.  }
7.  :
8.  tokenVarname:VARNAME { lValue = tokenVarname.getText(); }
9.  EQUALS
10. (melody = noteSequence { rValue.add(melody); })+
11. { symbolTable.put(lValue, rValue); }
12. ;
```

Since variable must be capable of holding more than one note or harmony for it to be useful, the procedure above has two main components. The first shown on line 10 listens for a token stream matching a `noteSequence`, which is discussed later on. Each time the subrule fires, it returns a Melody object, which is subsequently queued onto the `rValue` array list. Once there are no more

noteSequence tokens, the second component on line 11 binds the array of melodies to the variable name.

NoteSequence

```
1.  noteSequence returns [Melody sequence = new Melody()]
2.  { Note single; }
3.  :
4.  single = singleNote { sequence.add(single); }
5.  | LPAREN
6.  (single = singleNote { sequence.add(single); })+
7.  RPAREN
8. ;
```

NoteSequence is another branching rule that returns a Melody containing a single note or a harmony consisting of multiple notes. Recalling that a Melody is basically a container object for a Note, the only syntactic difference between a single note and harmony is the inclusion of parentheses around the latter (the LPAREN and RPAREN tokens on lines 5 and 7, respectively). In either case, the Note object returned by the singleNote subrule is added to the Melody and returned.

SingleNote

The SingleNote is the foundation of the parser as it is responsible for parsing and returning a Note object to all the other major rules. Let us analyze this lengthy rule piece by piece.

```
1.  singleNote returns [Note single = null]
2.  {
3.    Note note = new Note(0);
4.    note.setVolume(64);
5.  }
6.  :
```

The standard declarations include creating a new Note object and setting its volume to the default value of 64.

```
7.  ( tokenNote:NOTE {
9.  switch(tokenNote.getText().charAt(0)) {
10.    case 'C':  note.setValue(0); break;
11.    case 'D':  note.setValue(2); break;
12.    case 'E':  note.setValue(4); break;
13.    case 'F':  note.setValue(5); break;
14.    case 'G':  note.setValue(7); break;
15.    case 'A':  note.setValue(9); break;
16.    case 'B':  note.setValue(11); break;
17.    case 'R':  note.setVolume(0);
18.  }
19.  note.setValue(note.getValue() + 60);
20.  note.setDuration(NoteLength.Q);
21.  }
```

The MIDI note scale ranges from a C in octave 0 (Pitch 0) all the way to a G in octave 10 (Pitch 127). Unless otherwise specified, a note is assumed to be a quarter note in the 5th octave. Since each octave holds 12 notes, converting a base note (lines 10 through 16) to the 5th octave is achieved by adding 12 x 5 = 60 as shown in line 19. As a side note, a rest in MIDL is represented as a note with no volume. Making this assumption cuts down on extraneous code that would otherwise be required.

```
22.  (SHARP { note.setValue(note.getValue() + 1); }
23.  | FLAT { note.setValue(note.getValue() - 1); })?
```

Sharps and flats, also known as accidentals, simply increment or decrement the pitch, respectively. The parentheses enclosing lines 22 through 23 are used by the question mark at the end of line 23 to denote that accidentals are optional.

```
24.  (tokenOctave:INT {
25.    int octave = Integer.parseInt(tokenOctave.getText());
26.    if ( octave > 10 )
27.      throw new SemanticException("Octaves must be between 0 and
         10", tokenOctave.getFilename(), tokenOctave.getLine(),
         tokenOctave.getColumn());
28.    note.setValue((note.getValue() - 60) + 12 * octave);
29.  } )?
```

Since the pitch of a note is augmented by 60 in order to move it to the 5th octave by default, we must perform the reverse operation when a different octave is specified. Line 25 simply subtracts 60 (the offset for octave 5) and adds the correct octave (12 x the specified octave).

```
30.  (tokenLength:LENGTH {
31.    switch(tokenLength.getText().charAt(0)) {
32.      case 'w':      note.setDuration(NoteLength.W); break;
33.      case 'h':      note.setDuration(NoteLength.H); break;
34.      case 'q':      note.setDuration(NoteLength.Q); break;
35.      case 'e':      note.setDuration(NoteLength.E); break;
36.      case 's':      note.setDuration(NoteLength.S); break;
37.      case '.':      note.setDuration(NoteLength.Q * 1.5); break;
38.    }
39.    if ( tokenLength.getText().length() == 2 && tokenLength.getText
       ().charAt(1) == '.' ) {
40.      note.setDuration(note.getDuration()*1.5);
41.    }
42.  })?
```

Notes can vary in length from a sixteenth (30 ticks) to a whole (240 ticks). Finer variations can be achieved by using the dot (.) operator, which increases the note length by 150% as shown in lines 37 to 39. Since notes are quarter notes by default, applying a dot note length results in 1.5 x a quarter note length.

The NoteLength class is simply a set of static variables ranging from 0.25 to 4.0 for sixteenth to whole notes, respectively. When a Note On event is added to a Sequence, the corresponding Note Off tick position is determined by multiplying the NoteLength constant by 120, the default resolution in PPQ defined by MIDL.

```
43.  ( VOLUME tokenVolume:INT {
44.    int volume = Integer.parseInt(tokenVolume.getText());
45.    if ( volume > 127 )
46.      throw new SemanticException("Volume must be between 0 and
         127");
47.    note.setVolume(volume);
48.  } )? )
```

Similar to the note octave section, a note's volume is the last optional attribute and must fall between 0 and 127. Once each attribute has been parsed, the note is finalized and returned to the calling rule.

VarSequence

```
1.  varSequence returns [ArrayList<Melody> rValue = null]
2.  :
3.  tokenVarname:VARNAME {
4.    String lValue = tokenVarname.getText();
5.    if ( symbolTable.containsKey(lValue) )
6.      rValue = symbolTable.get(lValue);
7.    else
8.      throw new SemanticException("Variable " + lValue + "
        undefined");
9.  }
10. ;
```

VarSequence holds nothing special syntax-wise that we have not discussed previously. It reads the VARNAME token, determines if the variable is defined and if so, returns the associated array of Melody objects to the calling rule. If the variable has not been previously defined, the rule exits gracefully by throwing a SemanticException.

PlayCommand

The workhorse of the MIDL parser, PlayCommand calls upon the previously defined subrules and writes the required events to the Sequence.

```
1.  playCommand
2.  {
3.    ArrayList<Melody> sequence = new ArrayList<Melody>();
4.    Melody melody;
5.    int repeat = 1;
6.    int voice = 0;
7.  }
8.  :
```

Each PLAY command must contain either a sequence of notes or a variable. In addition, the user may choose to play using a different voice or more than once. Unless otherwise specified, each sequence plays once with Voice 0 as defined in lines 5 to 6.

```
9.  PLAY
10. (WITH VOICE tokenVoice:INT {
11.   voice = Integer.parseInt(tokenVoice.getText());
12.   if ( voice > 14 )
13.     throw new SemanticException("Voices must be between 0 and 14");
14. })?
15. (REPEAT tokenRepeat:INT {
16.   repeat = Integer.parseInt(tokenRepeat.getText());
17. })?
```

Prior to parsing any notes or variables, the parser records any optional voice or repeat commands.

```
18. (
19.   (melody = noteSequence {sequence.add(melody); })+
20.   | sequence = varSequence
21. )
```

MIDL's design allows it to play multiple notes, but only one variable per PLAY command. Inputs such as PLAY $varA $varB are illegal and should be replaced with an alternative such as PLAY $varA PLAY $varB. This is reflected in lines 19 and 20.

```
22. {
23. while ( repeat-- > 0 ) {
24.   for ( Melody m : sequence ) composer.addNote(m, voice);
25. }
26. ;
```

Lines 23 and 24 execute at the end of the rule and repeat as specified.

THE MIDLMAN IDE

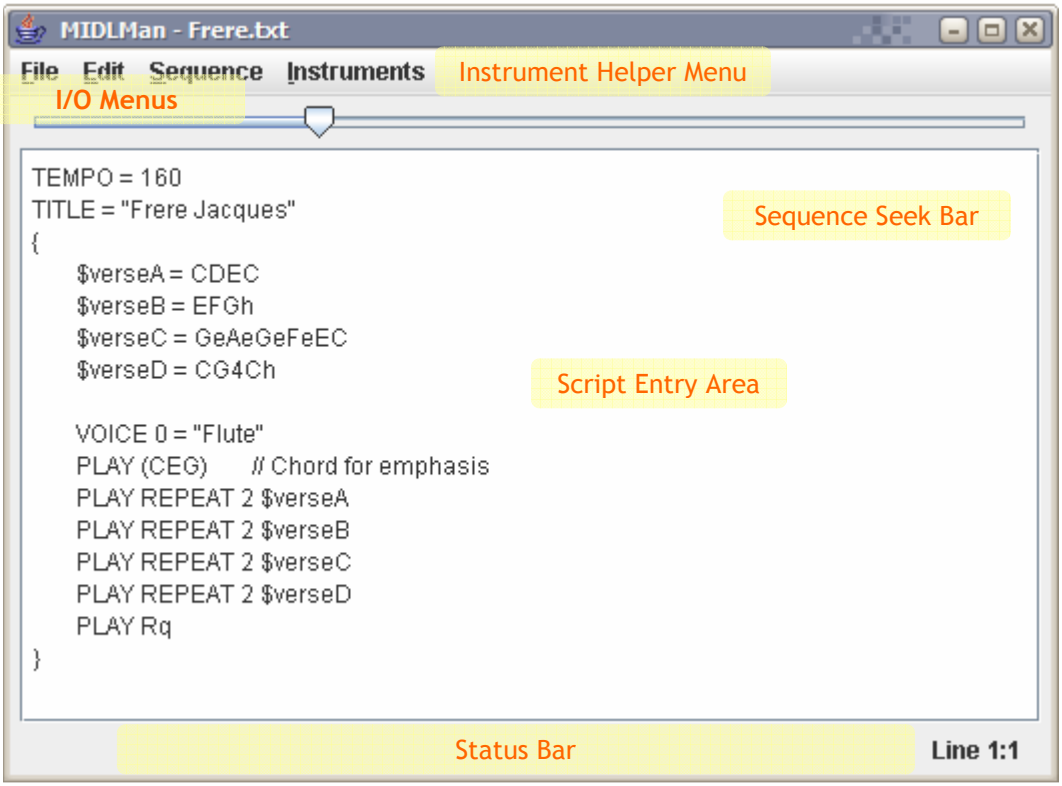

Figure 4

The above screenshot shows the MIDLMan development environment. Each of the six labeled portions of the window are interconnected with the Script Entry area which serves as the main document model.

I/O MENUS

File and Edit contain the usual assortment of reading and writing methods to modify the script entry area. Opening or saving a script appends the script name to the title bar. The Sequence menu allows the user to preview his/her composition as well as export the data to a Standard MIDI File. Both entries on the Sequence menu call the lexer and parser methods defined by the MIDL grammar.

INSTRUMENT HELPER MENU

With 128 instruments defined in the General MIDI Standard, any IDE worth its bits needs an effective way to present each one. The Instrument menu contains 16 submenus denoting instrument groups, each of which in turn contain 8 instruments. Upon selecting an instrument, its name is inserted at the caret position in the script entry area.

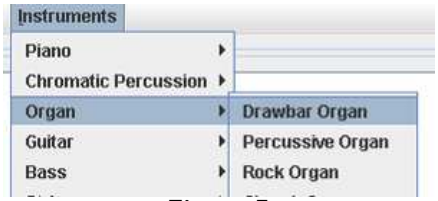

Figure 5

SEEK BAR

Initially disabled, the sequence bar activates only upon selecting Sequence>Preview. Upon activation, its maximum length is set to the length of the sequence in ticks, and a Timer object is called every 10ms to update its position. The user may drag the slider during playback to seek to different portions of the sequence. Note that once the sequence ends, the slider position is reset to 0 and becomes disabled.

STATUS BAR

A JPanel composed of two JTextInput boxes, the right-hand box always displays the current line and column, while the left-hand box (not visible in this screenshot) only shows up during fatal errors, which may include I/O reading and writing problems as well as ANTLR specific errors. In the erroneous script below, calling one of the Sequence commands results in the lexer throwing a RecognitionException, which propagates up as a RuntimeException. Upon catching the error, the operation halts until script is corrected.

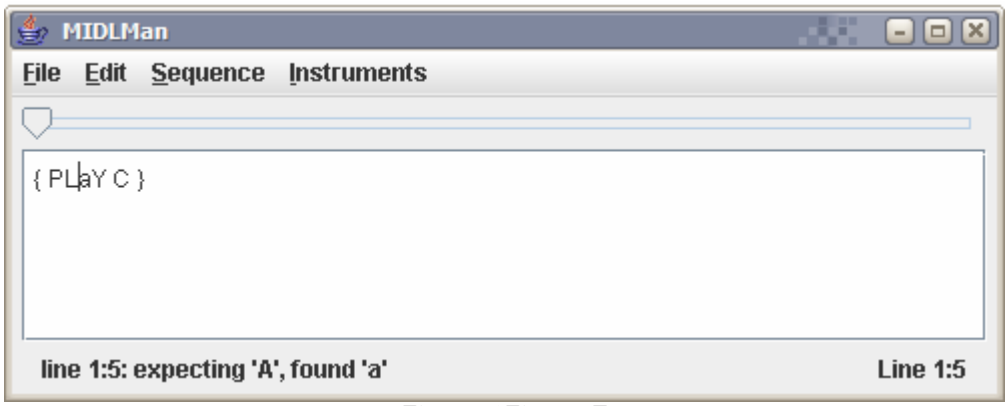

Figure 6Figure 7

MIDLMAN CLASS OVERVIEW

MIDLMan consists of five major upper-level classes to perform its main functions: MidlAppUI, MidlAppIO, MidiIO, MidlManApp, FileMask. The last two are simply helper classes and not worth expounding upon.

MIDIIO

MidiIO is instantiated once by MidlAppUI and is thus accessible to all of MidlAppUI's member functions. Note that it has no ties to the GUI and therefore may be reused in other applications. Its main duties include initializing and closing the Sequencer and Synthesizer objects, exporting MIDI sequences, controlling playback of the sequence, and reporting the sequence position.

Several odd problems while developing MIDLMan such as multiple sequences playing simultaneously were rectified by this class. It is crucial to remember to properly close both the Sequencer and Synthesizer upon completion. Earlier versions of MIDLMan failed to close the latter and worked seemingly well until multiple Preview commands were invoked.

MIDLAPPIO

This performs open and save functions and includes a crucial method that accepts a String, calls the lexer and parser, and returns a resulting Sequence. A helper class of sorts, MidlAppIO is not coupled to the GUI.

MIDLAPPUI

The 800 lb gorilla of the MIDLMan program, MidlAppUI is the Presentation portion of the well-known Presentation-Model GUI design pattern. It contains all the event handler classes corresponding to each menu Item, an instance of MidiIO and MidlAppIO, and three main portions of the GUI: textArea, statusbar, and slider.

The textArea is an inherited form of a normal JTextArea object, but has provisions for reporting caret position changes, and includes a special "dirty bit" flag to signify a modified script. If the user changes the contents and performs a potentially dangerous action such as exiting or creating a new document, MidlAppUI checks the flag and confirms the action if necessary.

Perhaps the most challenging aspect of developing the GUI was the inclusion of threads. At any given time during playback, the MIDLMan is performing no less than three actions: playing a sequence, updating the seek bar position, and monitoring keyboard events. If the app was single-threaded, playing back a sequence would halt all other tasks until the playback was finished. The multi-threaded components of MIDLMan comes into play once the user elects to preview a sequence:

1. The main GUI thread, which is always running, captures a Sequence Preview event.
2. A new thread is created to play a sequence. Playback is a blocking operation and only halts execution of its parent thread, not the GUI.
3. The seek bar is enabled and playback starts.
4. The seek bar's enable function is overloaded to start a new Timer, which is a thread itself.
5. Ever so often, the seek bar calls a function using the Timer object to see if its position matches the sequence's position. If not, the sequence position is updated.
6. Once playback completes, the thread created in step 2 proceeds to the next statement and disables the seek bar.
7. The seek bar calls its overloaded disable method to halt its timer before disabling itself.

WHO'S WHO IN MIDLMAN

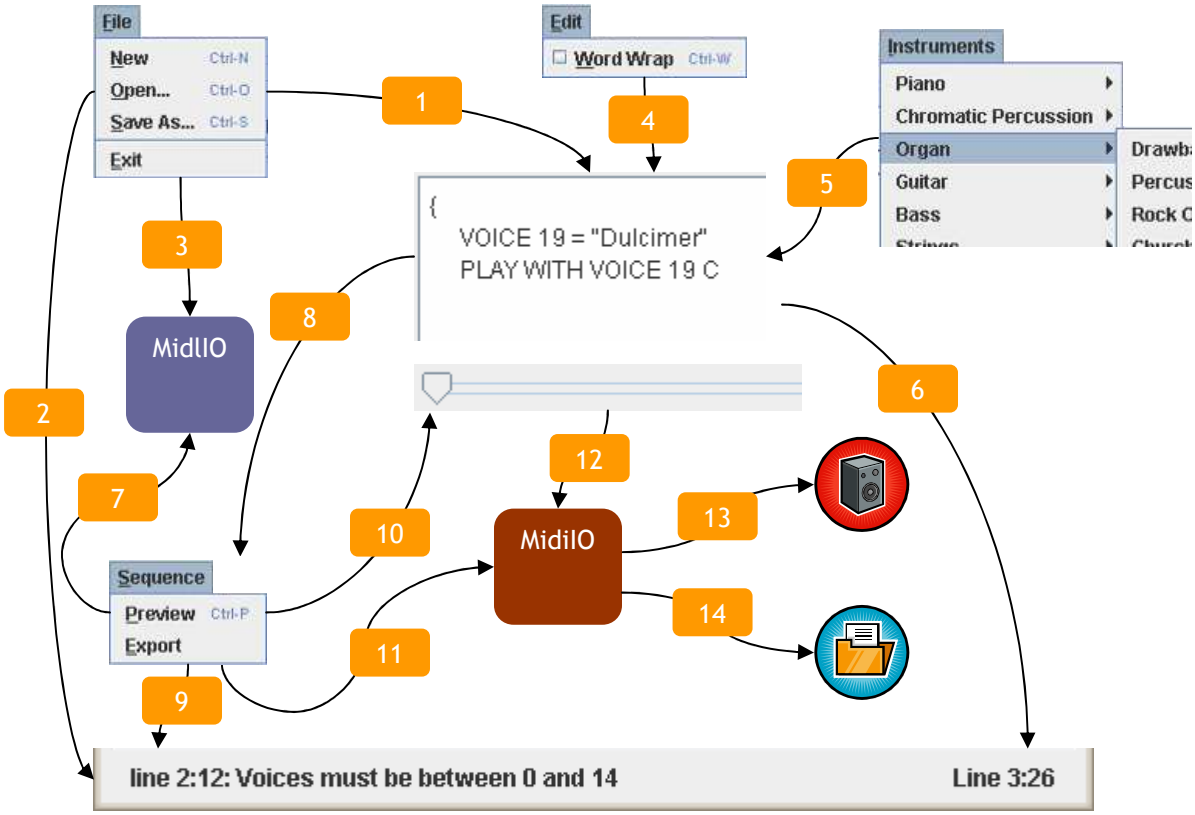

Figure 8

Item	Component	Description
1, 2	File Menu	New clears the script area. Open and Save As both use the MidlIO routines to write and read to the script area. If MidlIO generates an error, it is displayed on the status bar.
3	File Menu	Passes and receives String objects to MidlIO. Propagates IOExceptions up as RuntimeExceptions and writes them to the status bar.
4	Word Wrap	Reads the current line wrap status of the script area object and flips it.
5	Instrument Menu	Reads the current caret position of the script area and inserts (or replaces, if there is selected text) the submenu name into the script area.
6	Script Area	Fires an event when the caret is moved to update the status bar.
7, 8, 9	Sequence Menu	Reads text from the script area and passes it to MidlIO to

		receive an appropriate Sequence object. Language errors at this state are reported to the status bar.
10	Preview Submenu	The slider is enabled.
11	Preview/Export Submenu	A Sequence is passed to MidiIO.
12	Slider Bar	Every 10ms, the slider checks the position of the currently playing Sequence and updates its own position. If the sequence has stopped playing, the slider resets its position to 0 and disables itself.
13, 14	MidiIO	The currently assigned sequence is output to a Synthesizer or General MIDI File.

PARTING THOUGHTS

During informal testing, first-time users were able to deduce the syntax of the MIDL language given a sample script. Within minutes, participants were creating their own melodies with little effort, and as such, MIDL's role as a simple language to generate music has been fulfilled. Easily extensible by editing the ANTLR script, suggestions for future revisions are as follows:

1. Global property changes. Implement grouping tokens which apply a specified property (volume, octave, etc.) to a range of notes. Thus, instead of PLAY C6vol32G6vol32E6vol32, we could have PLAY [CGE]6vol32.

2. Sequence transformers. A transformer is a feature found in the jFugue package which allows actions such as reversing or gradually increasing the volume of a note sequence.

3. AFTER keyword. All PLAY statements in MIDL are executed simultaneously. In order to play notes on a different instrument after a certain sequence requires the inclusion of rests. This is best illustrated by an example:

   ```
   PLAY WITH VOICE 0 ABC
   PLAY WITH VOICE 1 RRRDEF
   PLAY WITH VOICE 2 RRRRRRG     // matey
   ```

   The inclusion of an AFTER statement would instruct MIDL to play a sequence immediately after the preceding one completes:

   ```
   PLAY WITH VOICE 0 ABC
   PLAY AFTER WITH VOICE 1 DEF
   PLAY AFTER WITH VOICE 2 G
   ```

WORKS CITED

[1] Berg, Peter R. *Sound Tutorials*. Pete's QBASIC/QuickBasic Site. <http://www.petesqbsite.com/sections/tutorials/sound.shtml>.

[2] Bomers, Florian, and Pfisterer, Matthias. *FAQ: MIDI Programming*. 13 April, 2005. Java Sound Resources. <http://www.jsresources.org/faq_midi.html>.

[3] Brown, Andrew, and Sorensen, Andrew. *jMusic – Music composition in Java*. <http://jmusic.ci.qut.edu.au/>.

[4] DeBenedetti, Gilbert. Home page. <http://www.pitt.edu/~deben/>.

[5] Flanagan, David. *Java Examples in a Nutshell, 3rd Edition*. 1 January 2004. O'Reilly. Ex. 17-4. <http://www.onjava.com/onjava/excerpt/jenut3_ch17/examples/PlaySoundStream.java>.

[6] Glatt, Jeff. *The Historical Significance of Computers/MIDI*. <http://www.borg.com/~jglatt/tutr/olddays.htm>.

[7] Glatt, Jeff. *MIDI File Format*. <http://www.borg.com/~jglatt/tech/midifile.htm>.

[8] Heckroth, Jim. *Tutorial on MIDI and Music Synthesis*. 1995. The MIDI Manufacturers Association. <http://www.harmony-central.com/MIDI/Doc/tutorial.html>.

[9] *JavaSound API Programmer's Guide*. 24 October, 2001. Sun Microsystems. <http://java.sun.com/j2se/1.5.0/docs/guide/sound/programmer_guide/>.

[10] Koelle, David. *JFugue – Java API for Music Programming*. <http://jfugue.org/>.

[11] Loeb, Peter. *XMidi Home Page*. 22 March, 2005. <http://www.palserv.com/XMidi/>.

[12] Parr, Terence. *ANTLR Parser Generator*. <http://antlr.org/>.

[13] Parr, Terrence. *ANTLR Specification: Meta Language*. <http://www.antlr.org/doc/metalang.html#SyntacticPredicates>.

[14] Parr, Terrence. *Building Recognizers by Hand*. 1999. <http://www.antlr.org/book/byhand.pdf>.

[15] Pihl, Kai. *Shaping the Sound*. Computers and Music: Using SB soundcards, computers, and E-mu SoundFonts. <http://www.hammersound.net/sd_computers_and_music/shaping.html>.

[16] Phillips, Dave. "Linux MIDI: A Brief Survey, Part 2." *Linux Journal* 23 November, 2004. <http://www.linuxjournal.com/article/7912>.

[17] White, Paul. "MIDI Basics: Part 1." *Sound on Sound* August 1995. <http://www.soundonsound.com/sos/1995_articles/aug95/midibasics1.html>.

APPENDIX A – COMPILING AND RUNNING MIDLMAN

COMPILING MIDLMAN

The MIDLMan source code folder has the following structure:

🗁 midl
🗁 resources
MidlManApp.java
midl.g

The two folders contain necessary MIDL libraries and resources respectively, while the midl.g lists the actual grammar for the MIDL language. MidlManApp.java of course contains the source code for the GUI application.

Compilation consists of two steps:

```
{path to java.exe}\java.exe –jar Antlr.tool {path to antlr.jar} midl.g
```

This invokes the functions included in the ANTLR tool set to create the Java source code for the MIDL lexer and parser. Note that we are invoking the Java runtime, not the Java compiler.

```
{path to javac.exe}\javac.exe –cp .,{path to antlr.jar} MidlManApp.java
```

Once the source code is in place, we call the Java compiler to create the necessary class files. On some systems, the –cp directive (.,{path to antlr.jar} must be enclosed in quotes.

RUNNING MIDLMAN

There are several ways to run MIDLMan, the simplest of which involves simply double-clicking the included MidlManApp.jar file (this assumes that antlr.jar is in the same directory as MidlManApp.jar). On supported systems, JAR files are associated with the Java runtime and will execute properly. If the JAR file association is not present, issue the following command:

```
{path to java.exe}\java.exe –jar MidlManApp.jar
```

Another option is to run MIDLMan directly from the web using the Java Web Start option. Simply edit the included MidlManApp.jnlp and replace the codebase value with your URL. In order for this method to work properly, both MidlManApp.jar and antlr.jar must be present in the same folder.

If MIDLMan is being run after compilation from source, the process is slightly more involved:

```
{path to java.exe}\java.exe –cp .;{path to antlr.jar} MidlManApp
```

APPENDIX B – A HEADS UP ON ANTLR

ANTLR (ANother Tool for Language Recognition) is a tool for generating LL(k) compilers, recognizers, and translators in a variety of languages including C++, Python, et al. ANTLR accepts grammars in EBNF notation along with corresponding syntactic actions, and generates a lexer/parser to recognize the language, thus supplanting both Lex and Yacc.

Writing a parser by hand usually entails creating a recursive descent parser where each token is validated by another function; the return values of subsequent function calls determine the validity of the input string. So, what's the problem? Recursive descent techniques generate LL(k) parsers, which require an immediate decision in regards to the validity of the input. This often results in more restrictive grammars and the use of left-factoring to create unnatural looking rules. Yacc in contrast, builds parsers using LALR, thus allowing more freedom during grammar design.

ANTLR retains the simplicity of LL while avoiding its pitfalls by using predicated-LL parsing. In essence, this allows the recognizer to peek ahead when necessary, but not necessarily consume the input stream to determine an appropriate action. Take the following example from the ANTLR documentation [13]:

```
stat =>     (list "=") => list "=" list
      |     list
;
```

The above semantic rule is able to distinguish between a list (another subrule) of items or an assignment. In this case, if a list token is followed by an equals sign, the assignment rule is applied. Consider the alternative pure LL(k) method; the parser would always look k characters ahead instead of only when required.

CHEESE PLEASE

While creating an ANTLR tutorial is outside the scope of this document (the ANTLR home page has countless tutorials [12]), perhaps a simple overview is in order. ANTLR input consists of a myLanguage.g file consisting of the following components:

```
1.  header {
2.    package myClasses;
3.  }
4.  {
5.    import otherClass.Function;
6.  }

7.  class MyParser extends Parser {
8.    sentence: subject predicate;
9.    subject: pronoun;
10.   predicate: verb noun;
11. }

12. class MyLexer extends Lexer {
13. options {
14.   k = 1;
15. }

16.   pronoun: "I";
17.   verb: "love" | "hate";
18.   noun: "cheese";
```

```
19. }
```

Listed above is a simple grammar which accepts one of two inputs: “I love cheese” and “I hate cheese”.

Lines 1-3: Any statement included here will be at the top of the resulting lexer and parser; as a result, common libraries or include files should be placed here.

Lines 4-6: Similar to the header section, any lines placed here are copied verbatim above the class definition.

Line 7-11: MyParser is the name of the user-created parser, which inherits properties from ANTLR’s Parser class. Lines 8 through 10 describe semantic rules.

Line 12: MyLexer is a user-defined lexical analysis class which inherits from ANTLR’s Lexer class.

Lines 13-15: A common option placed here is the amount of character lookahead. Note that a Parser may also have an options section which may include entries to disable default error handling, among other things.

Lines 16-19: The final component of the lexer class, these are token constructs. ANTLR allows regular expressions to denote tokens as well.

MORE CHEESE, PLEASE

As a simple example of what ANTLR is capable of, let us modify our source file as follows:

```
7.  class MyParser extends Parser {
8.    sentence
9.    { boolean cheesePreference; }
10.   :
11.     subject
12.     cheesePreference = predicate
13.     {
14.       if (cheesePreference)
15.         System.out.println(“please”);
16.       else
17.         System.out.println(“who doesn’t like cheese?!”);
18.     }
19.  ;

20.  subject: pronoun;

21.  predicate returns [boolean loveCheese = null]
22.  :  verb
23.     token: noun
24.     {
25.       loveCheese = (token.getText()).equals(“love”);
26.     }
27.  ;
28. }
```

The modified listing serves a more useful purpose by providing feedback based on user input. Let us work backwards from the predicate until we finally reduce to a sentence rule.

Line 21: Predicate is defined as a rule consisting verb and noun tokens, and returns a boolean named loveCheese.

Line 23: In order to determine the value of the noun token, we assign it to a temporary ANTLR.Token object, aptly named token.

Line 25: We call the Token object's getText() method and assign a value to loveCheese.

Let us now examine the top-level rule named sentence.

Line 9: Recall that statements inside curly brackets are copied verbatim into the resulting code, thus making it a perfect place to declare any auxiliary variables we may use.

Line 12: Since the predicate rule returns a boolean, we need a variable in which to store it. In this case, the return value is stored in cheesePreference.

Lines 13-18: Our variable, cheesePreference is now in scope. Since cheesePreference contains the value returned by predicate, we can use it to display some output.

EMBEDDING ANTLR INTO JAVA

With our grammar complete, a sample driver program named Driver.java is as follows:

```
1.  import antlr.*;
2.  import java.io.*;

3.  public class Driver {

4.    MyLexer lexer;
5.    MyParser parser;

6.    try {
7.      lexer = new MyLexer(new DataInputStream(System.in));
8.      parser = new MyParser(lexer);

9.      System.out.println("Enter a statement: ");
10.     parser.sentence();
11.   } catch (Exception e) {
12.     System.err.println("Caught exception: " + e);
13.   }
14. }
```

Line 1: MyLexer and MyParser derive from ANTLR objects, so we must include the ANTLR namespace.

Line 6: We need the try clause because the ANTLR objects may return various exceptions, including MismatchedToken or Recognition errors.

Lines 7-8: The lexer needs to receive input from a source. In this case, it is linked to the input stream, but may easily be attached to virtually any type of data stream. The parser in turn is linked to the lexer.

Line 10: The grammar file developed in the previous section comes into play here as we call the top-level rule, sentence.

Lines 11-13: Any errors passed by the lexer or parser bubble up and finally emerge at this point.

COMPILING AND RUNNING

Creating an ANTLR solution is a two-step process. We must first convert the ANTLR grammar file to our target language code, and then compile all the source files together:

```
java –cp “c:\antlr\275\lib\antlr.jar” antlr.Tool myLanguage.g
javac –cp “.;c:\antlr\275\lib\antlr.jar” *.java
```

Finally, we run our ANTLR solution by issuing the following command:

```
java –cp “.;c:\antlr\275\lib\antlr.jar” Driver
```

Note that the –cp flag sets the classpath to include the necessary ANTLR definitions, and therefore must be included during both compilation and runtime.

Appendix C – Sample Scripts

The first three samples were converted from Gilbert DeBenedetti's very helpful piano music archive [4]. Crab Canon is a modified version of the one included in the jFugue source files.

Row, Row, Row Your Boat

```
TEMPO = 260
{
      PLAY Ch. Ch. Ch D Eh.               // Row, row, row your boat,
      PLAY Eh D Eh F Gw                   // Gently down the stream,
      PLAY C6C6C6 GGG EEE CCC             // Merrily, merrily, etc.
      PLAY Gh F Eh D Cw                   // Life is but a dream.
}
```

Clementine

```
{
      // Oh, my darling, Oh my darling, Oh my
      PLAY CeCe C G4 EeEe E C CeEe

      // darling, Clementine, Thou art lost and gone for
      PLAY G G FeEe Dh DeEe F F EeDe

      // ever, dreadful sorry, Clementine.
      PLAY E C CeEe D G4 B4eDe Ch
}
```

Oh, When the Saints

```
TEMPO = 180
{
      $saintsA = C E F G (FG) (CE) (CD) (CEG4)
      PLAY REPEAT 2 $saintsA

      PLAY C E F Gh Eh Ch Eh D (FG) (FG) (CE) (CD)
      PLAY E E D (ChEhG4h) Ch (EhG4hCh) G G G Fh.
      PLAY E F Gh Eh Dh Dh Ch (FG) (CE) (CD) (CEG4)
}
```

Crab Canon (Bach)

```
{
$sequence = D4h E4h A4h Bb4h C#4h R A4 A4 Ab4h G4 G4 F#4h F4 F4 E4 Eb4 D4 A4e
G4e A4e D5e A4e F4e E4e F4e G4e A4e B4e C#5e D5e F4e G4e A4e Bb4e E4e F4e G4e
A4e G4e F4e E4e F4e G4e A4e Bb4e C5e Bb4e A4e G4e A4e Bb4e C5e D5e Eb5e C5e
Bb4e A4e B4e C#5e D5e E5e F5e D5e C#5e B4e C#5e D5e E5e F5e G5e E5e A4e E5e
D5e E5e F5e G5e F5e E5e D5e C#5e D5 A4 F4 D4

$reverse = D4 F4 A4 D5 C#5e D5e E5e F5e G5e F5e E5e D5e E5e A4e E5e G5e F5e
E5e D5e C#5e B4e C#5e D5e F5e E5e D5e C#5e B4e A4e Bb4e C5e Eb5e D5e C5e Bb4e
A4e G4e A4e Bb4e C5e Bb4e A4e G4e F4e E4e F4e G4e A4e G4e F4e E4e Bb4e A4e
G4e F4e D5e C#5e B4e A4e G4e F4e E4e F4e A4e D5e A4e G4e A4e D4 Eb4 E4 F4 F4
F#4h G4 G4 Ab4h A4 A4 R C#4h Bb4h A4h E4h D4h

PLAY WITH VOICE 0 $sequence
PLAY WITH VOICE 1 $reverse
}
```

APPENDIX D – SOURCE CODE

MIDL.G

```
{
	import javax.sound.midi.*;
	import java.util.*;
	import midl.*;
}

class MidlParser extends Parser;
options { defaultErrorHandler=false; }

{
	MIDIComposer composer = new MIDIComposer();
	Map<String, ArrayList<Melody>> symbolTable = new HashMap<String,
ArrayList<Melody>>();
}

song returns [Sequence output = null]:
	(tempoAssignment)?
	(titleAssignment)?
	BEGIN
	(voiceAssignment | varAssignment | playCommand)+
	END {
		output = composer.getSequence();
	}
;

tempoAssignment:
	TEMPO EQUALS tokenTempo:INT {
		int bpm = Integer.parseInt(tokenTempo.getText());
		int mpq = 60000000 / bpm;
		composer.changeTempo(mpq);
	}
;

titleAssignment:
	TITLE EQUALS tokenTitle:STRING
	{
		String parsedTitle = tokenTitle.getText().substring(1,
tokenTitle.getText().length() - 1);
		composer.addTitle(parsedTitle);
	}
;

playCommand
{
	ArrayList<Melody> sequence = new ArrayList<Melody>();
	Melody melody;
	int repeat = 1;
	int voice = 0;
}
:
	PLAY
	(WITH VOICE tokenVoice:INT {
		voice = Integer.parseInt(tokenVoice.getText());
		if ( voice > 15 )
			throw new SemanticException("Voices must be between 0 and 15",
tokenVoice.getFilename(), tokenVoice.getLine(), tokenVoice.getColumn());
	})?
    (REPEAT tokenRepeat:INT { repeat = Integer.parseInt(tokenRepeat.getText()); })?
    ( (melody = noteSequence {sequence.add(melody); })+ | sequence = varSequence)
    {
		while ( repeat-- > 0 ) {
```

```
                  for ( Melody m : sequence )
                        composer.addNote(m, voice);
                  }
      }
;

varSequence returns [ArrayList<Melody> rValue = null]
:
      tokenVarname:VARNAME {
            String lValue = tokenVarname.getText();
            if ( symbolTable.containsKey(lValue) )
                  rValue = symbolTable.get(lValue);
            else
                  throw new SemanticException("variable " + lValue + " undefined",
tokenVarname.getFilename(), tokenVarname.getLine(), tokenVarname.getColumn());
      }
;

voiceAssignment
{ int voice, instrument = 0; }
:
      (VOICE
      tokenVoice:INT {
            voice = Integer.parseInt(tokenVoice.getText());
            if ( voice > 15 )
                  throw new SemanticException("Voices must be between 0 and 15",
tokenVoice.getFilename(), tokenVoice.getLine(), tokenVoice.getColumn());
      }
      EQUALS
      (tokenInstrumentNum:INT {
            instrument = Integer.parseInt(tokenInstrumentNum.getText());
            if ( instrument < 0 || instrument > 127 )
                  throw new SemanticException("Instruments must be between 0 and 127",
tokenInstrumentNum.getFilename(), tokenInstrumentNum.getLine(),
tokenInstrumentNum.getColumn());
      }
      | tokenInstrumentString:STRING {
            String parsedInstrument = tokenInstrumentString.getText().substring(1,
tokenInstrumentString.getText().length() - 1);
            instrument = composer.getInstrument(parsedInstrument);
            if ( instrument == -1 )
                  throw new SemanticException("No such instrument",
tokenInstrumentString.getFilename(), tokenInstrumentString.getLine(),
tokenInstrumentString.getColumn());
      } )
      )
      {
            System.out.println("Changing instrument to " + instrument);
            composer.changeInstrument(instrument, voice);
      }
;

varAssignment
{
      String lValue;
      ArrayList<Melody> rValue = new ArrayList<Melody>();
      Melody melody;
}
:
      tokenVarname:VARNAME { lValue = tokenVarname.getText(); }
      EQUALS
      (melody = noteSequence {
```

```
            rValue.add(melody);
      })+
      {
            symbolTable.put(lValue, rValue);
      }
;

noteSequence returns [Melody sequence = new Melody()]
{ Note single; }
:
      single = singleNote { sequence.add(single); }
|     LPAREN
      (single = singleNote {
            sequence.add(single);
      })+
      RPAREN
;

singleNote returns [Note single = null]
{
      Note note = new Note(0);
      note.setVolume(64);
}
:
      ( tokenNote:NOTE {
            switch(tokenNote.getText().charAt(0)) {
                  case 'C':   note.setValue(0); break;
                  case 'D':   note.setValue(2); break;
                  case 'E':   note.setValue(4); break;
                  case 'F':   note.setValue(5); break;
                  case 'G':   note.setValue(7); break;
                  case 'A':   note.setValue(9); break;
                  case 'B':   note.setValue(11); break;
                  case 'R':   note.setVolume(0);
            }
            note.setValue(note.getValue() + 60);
            note.setDuration(NoteLength.Q);
      }
      (SHARP { note.setValue(note.getValue() + 1); } | FLAT {
note.setValue(note.getValue() - 1); })?
      (tokenOctave:INT {
            int octave = Integer.parseInt(tokenOctave.getText());
            if ( octave > 10 )
                  throw new SemanticException("Octaves must be between 0 and 10",
tokenOctave.getFilename(), tokenOctave.getLine(), tokenOctave.getColumn());
            note.setValue((note.getValue() - 60) + 12 * octave);
      } )?
      (tokenLength:LENGTH
            {
                  switch(tokenLength.getText().charAt(0)) {
                        case 'w':   note.setDuration(NoteLength.W); break;
                        case 'h':   note.setDuration(NoteLength.H); break;
                        case 'q':   note.setDuration(NoteLength.Q); break;
                        case 'e':   note.setDuration(NoteLength.E); break;
                        case 's':   note.setDuration(NoteLength.S); break;
                        case '.':   note.setDuration(NoteLength.Q * 1.5); break;
                  }

                  if ( tokenLength.getText().length() == 2 &&
tokenLength.getText().charAt(1) == '.' ) {
                        note.setDuration(note.getDuration()*1.5);
                  }
```

```
            }
      )?
      ( VOLUME tokenVolume:INT {
            int volume = Integer.parseInt(tokenVolume.getText());
            if ( volume > 127 )
                  throw new SemanticException("Volume must be between 0 and 127",
tokenVolume.getFilename(), tokenVolume.getLine(), tokenVolume.getColumn());
            note.setVolume(volume);
      } )? )
      {
            single = note;
      }
;

class MidlLexer extends Lexer;
options {
      k=2;
}

BEGIN options { paraphrase = "Opening brace ({)"; }: "{";
END options { paraphrase = "Closing brace (})"; }:    "}";
TEMPO options { paraphrase = "Tempo keyword"; }: "TEMPO";
TITLE options { paraphrase = "Title keyword"; }: "TITLE";
NOTE options { paraphrase = "Note (A-G) or Rest (R)"; }: ('A'..'G' | 'R');
LENGTH options { paraphrase = "Note length"; }: ('w' | 'h' | 'q' | 'e' | 's' |
'.')('.')?;
VOLUME options { paraphrase = "Note volume"; }: "vol";
SHARP:            '#';
FLAT:       'b';
INT options { paraphrase = "Integer"; }:        ('0'..'9')+;
LPAREN:           '(';
RPAREN:           ')';
QUOTE:            '"';
PLAY options { paraphrase = "Play keyword"; }: "PLAY";
REPEAT options { paraphrase = "Repeat keyword"; }: "REPEAT";
WITH options { paraphrase = "With keyword"; }: "WITH";
VOICE options { paraphrase = "Voice keyword"; }: "VOICE";
STRING options { paraphrase = "Quote enclosed string"; }: '"' (~('"'))* '"';
EQUALS options { paraphrase = "Equals sign"; }: '=';
VARNAME options { paraphrase = "Variable name"; }: '$' ('A'..'Z' | 'a'..'z') ('A'..'Z' |
'a'..'z' | '0'..'9')*;
NEWLINE:    ("\r\n" | '\r' | '\n') { newline(); $setType(Token.SKIP); };
WS:               (' ' | '\t' | '\f') { $setType(Token.SKIP); };
COMMENT:    "//" (~('\n'|'\r'))* NEWLINE { $setType(Token.SKIP); };
```

MIDICOMPOSER.JAVA

```
package midl;

import javax.sound.midi.*;
import java.io.*;
import java.util.ArrayList;

/**
	High level controller class for creating MIDI Sequences.
*/
public class MIDIComposer
{
	Sequence sequence;
	MIDITrack [] tracks;
	Sequencer sequencer;

	int PPQ = 120;
	final int MAX_TRACKS = 16;	// 16 tracks possible in a sequence

	final static int INSTRUMENT_COUNT = 128;
	static String [] INSTRUMENT_LIST = new String[INSTRUMENT_COUNT];

	/**
		Default constructor.  Initializes Sequence with a resolution
			of 120 PPQ and creates all MIDITrack objects.
	*/
	public MIDIComposer() {
		try {
			sequence = new Sequence(Sequence.PPQ, PPQ);
		} catch (InvalidMidiDataException e) {
			System.err.println("Error creating sequence in MIDIComposer()");
			System.exit(1);
		}

		// Create all the tracks in advance
		tracks = new MIDITrack[MAX_TRACKS];

		for ( int counter = 0; counter < MAX_TRACKS; counter++ ) {
			tracks[counter] = new MIDITrack(sequence);
		}

		loadInstrumentList();
	}

	/**
		Info function for debugging purposes.  Displays tick count
			of every MIDITrack.
	*/
	public void info() {
		for ( int counter = 0; counter < MAX_TRACKS; counter++ )
			System.out.println(tracks[counter].getLength());
	}

	/**
		Adds a melody to a specified track.
		@param melody Melody object
		@param channel Track number from 0 to 14
	*/
	public void addNote(Melody melody, int channel) {
		tracks[channel].addNote(melody);
	}
```

```
	/**
		Changes the tempo across all MIDITracks.
		@param tempoMPQ tempo in microsecond per quarter
	*/
	public void changeTempo(int tempoMPQ) {
		for ( MIDITrack t : tracks )
			t.changeTempo(tempoMPQ);
	}

	/**
		Adds a Sequence Name event to the first MIDITrack.
		@param title Title of the sequence
	*/
	public void addTitle(String title) {
		tracks[0].addTitle(title);
	}

	/**
		Changes the instrument assigned to a specified track.
		@param instrument patch number from 0 to 127
		@param channel Track number from 0 to 14
	*/
	public void changeInstrument(int instrument, int channel) {
		tracks[channel].changeInstrument(instrument);
	}

	/**
		Accessor function for private the Sequence object.
		@return Sequence object
	*/
	public Sequence getSequence() {
		return sequence;
	}

	/**
		Loads the set of recognized instruments from a data file.
	*/
	private void loadInstrumentList() {
		final String INSTRUMENT_DATA = "resources/instruments.txt";

		try {
			BufferedReader buffReader = null;
			String line = null;
			int counter = 0;

			InputStream rawIn =
this.getClass().getClassLoader().getResourceAsStream(INSTRUMENT_DATA);
			buffReader = new BufferedReader(new InputStreamReader(rawIn));

			while ( (line = buffReader.readLine()) != null )
				INSTRUMENT_LIST[counter++] = line.split(",")[1];

			buffReader.close();
		} catch (IOException ioe) {
			System.err.println("Error opening instrument data file");
		}
	}

	/**
		Returns the General MIDI Instrument patch number for a
			specified instrument.
		@param instrument case sensitive name of an instrument
```

```
		@return a number from 0 to 127 if the instrument is found, -1 otherwise
	*/
	public int getInstrument(String instrument) {
		for ( int counter = 0; counter < 128; counter++ )
			if ( INSTRUMENT_LIST[counter].equals(instrument) ) return counter;

		return -1;
	}
}
```

MIDITRACK.JAVA

```
package midl;

import javax.sound.midi.*;

/**
      A MIDITrack object is a container object for holding
      tracks, assigning channels and modifying other Track fields.
*/
public class MIDITrack
{
      private Track track;
      private long position = 0;
      private int channel = 0;
      private int ppq = 0;

      /**
            Creates a new MIDITrack
            @param sequence an initialized Sequence object
      */
      public MIDITrack(Sequence sequence) {
            track = sequence.createTrack();
            channel = sequence.getTracks().length;
            ppq = sequence.getResolution();
      }

      /**
            Changes the instrument used on the track
            @param instrument an General MIDI instrument ranging from 0 to 127
      */
      public void changeInstrument(int instrument) {
            ShortMessage patch = MIDIMsg.programChange(instrument, channel);
            track.add(new MidiEvent(patch, 0));
      }

      /**
            Changes the tempo used on the track
            @param tempoMPQ a tempo value in microseconds per quarter
      */
      public void changeTempo(int tempoMPQ) {
            MetaMessage tempo = MIDIMsg.tempo(tempoMPQ, channel);
            track.add(new MidiEvent(tempo, 0));
      }

      /**
            Adds a sequence name event to the track
            @param seqTitle a String containing the sequence name
      */
      public void addTitle(String seqTitle) {
            MetaMessage title = MIDIMsg.title(seqTitle);
            track.add(new MidiEvent(title, 0));
      }

      /**
            Adds a single note or harmony to the track
            @param m a Melody object containing a note or harmony
      */
      public void addNote(Melody m) {
            /*
                  If a Melody contains multiple notes (i.e.: is a
                  harmony, add each note and record the longest one.
```

```
                When done adding, offset the currentTick position
                by the length of the longest note.

                Otherwise, simply add the single note, and offset
                the currentTick by its length.
          */
          if ( m.isHarmony() ) {
                int harmonySize = m.size();
                double longestDuration = 0;

                while ( harmonySize-- > 0 ) {
                      Note note = m.next();
                      noteEvent(note, position);
                      if ( longestDuration < note.getDuration() )
                            longestDuration = note.getDuration();
                }

                movePosition((long) (longestDuration * ppq));
          } else {
                Note note = m.next();
                noteEvent(note, position);
                movePosition((long) (note.getDuration() * ppq));
          }

          // Rewind the Melody object's iterator in case it's used again
          m.rewind();
    }

    /*
          Adds a pair of on + off events to the track
    */
    private void noteEvent(Note note, long notePosition) {
          ShortMessage noteOn = MIDIMsg.noteOn(note, channel);
          ShortMessage noteOff = MIDIMsg.noteOff(note, channel);

          track.add(new MidiEvent(noteOn, notePosition));
          track.add(new MidiEvent(noteOff, (long) (notePosition + note.getDuration() *
ppq)));
    }

    /*
          Moves the currentTick position so we can add new notes
          that don't overlap.
    */
    private void movePosition(long delta) {
          position += delta;
    }

    /**
          Returns the length of the current track in ticks.
          @return the number of ticks in the track
    */
    public long getLength() {
          return track.ticks();
    }
}
```

MELODY.JAVA

```
package midl;

import javax.sound.midi.*;
import java.util.*;

/**
      Container class for holding Note objects.
      Multiple notes denote a harmony (all notes are
      played simultaneously).
*/
public class Melody
{
      private ArrayList<Note> sequence;
      private boolean harmony = false;
      private int index = 0;

      /**
            Default constructor.  Initializes array
                  list of Note objects.
      */
      public Melody() {
            sequence = new ArrayList<Note>();
      }

      /**
            Constructor which adds a single Note object.
            @param note instantiated Note object
      */
      public Melody(Note note) {
            sequence = new ArrayList<Note>(1);
            add(note);
      }

      /**
            Adds a new Note object to the Melody.
            @param note instantiated Note object
      */
      public void add(Note note) {
            sequence.add(new Note(note));
            if ( sequence.size() > 1 ) harmony = true;
      }

      /**
            Gets the next note in the Melody.
            @return Note object containing the next Note
      */
      public Note next() {
            return new Note(sequence.get(index++));
      }

      /**
            Resets the Note list iterator.
      */
      public void rewind() {
            index = 0;
      }

      /**
            Gets the size of the Melody.
            @return integer containing the number of Notes stored
      */
```

```
	public int size() {
		return sequence.size();
	}

	/**
		Checks to see if the Melody is a harmony.
		@return true if the number of notes > 1, false otherwise
	*/
	public boolean isHarmony() {
		return harmony;
	}
}
```

NOTE.JAVA

```
package midl;

/**
	Abstracts a Note object, including fields for value,
	duration, volume, attack, and decay.
*/
public class Note
{
	private int value;
	private double duration;
	private int volume = 64;
	private int attack;
	private int decay;

	/**
		Note copy constructor
		@param n an existing Note object
	*/
	public Note(Note n) {
		this.value = n.value;
		this.duration = n.duration;
		this.volume = n.volume;
		this.attack = n.attack;
		this.decay = n.decay;
	}

	/**
		Note constructor (value and duration)
		@param value an integer ranging from 0 to 127, with 60
			denoting middle-C (C, 5th octave)
		@param duration a Note duration as defined in the NoteLength class
	*/
	public Note(int value, double duration) {
		this.value = value;
		this.duration = duration;
	}

	/**
		Note constructor (value, duration, and volume)
		@param value an integer ranging from 0 to 127, with 60
			denoting middle-C (C, 5th octave)
		@param duration a Note duration as defined in the NoteLength class
		@param volume an integer ranging from 0 to 127 with 0's
			representing rests.
	*/
	public Note(int value, double duration, int volume) {
		this.value = value;
		this.duration = duration;
		this.volume = volume;
	}

	/**
		Note constructor (duration)
		@param duration a Note duration as defined in the NoteLength class
	*/
	public Note(double duration) {
		this.value = 0;
		this.duration = duration;
		this.volume = 0;
	}
```

```
	/**
		Get the note pitch.
		@return integer containing the pitch.
	*/
	public int getValue() {
		return value;
	}

	/**
		Get the note duration.
		@return double containing the duration.
	*/
	public double getDuration() {
		return duration;
	}

	/**
		Get the note volume.
		@return integer containing the volume.
	*/
	public int getVolume() {
		return volume;
	}

	/**
		Set the note volume.
		@param volume ranging from 0 to 127 containing the volume.
	*/
	public void setVolume(int volume) {
		this.volume = volume;
	}

	/**
		Set the note pitch.
		@param value ranging from 0 to 127 containing the pitch.
	*/
	public void setValue(int value) {
		this.value = value;
	}

	/**
		Set the note volume.
		@param duration one of the NoteValue double constants
	*/
	public void setDuration(double duration) {
		this.duration = duration;
	}

	/**
		Converts a Note object to a String.
		@return a String containing the value, duration, and volume
			of a note.
	*/
	public String toString() {
		return "" + value + " " + duration + " (" + volume + ")";
	}
}
```

NoteLength.java

```
package midl;

/**
	Contains various note length constants
*/
public class NoteLength {
	/**	Whole note (480 ticks)  */
	public static final double W = 4;
	/**	Half note (240 ticks)	*/
	public static final double H = 2;
	/**	Quarter note (120 ticks)	*/
	public static final double Q = 1;
	/**	Eighth note (60 ticks)  */
	public static final double E = 0.5;
	/**	Sixteenth note (30 ticks)	*/
	public static final double S = 0.25;
}
```

MIDIMsg.java

```
package midl;

import javax.sound.midi.*;

/**
      A set of static functions used to create Note On, Note Off, Program
      Change, Tempo Change, and Copyright messages (not events).
*/
public class MIDIMsg
{
      /**
            Creates a Note On message.
            @param note a Note object
            @param channel the channel on which to apply the message

            @return a ShortMessage containing a NoteOn command for
                  the specified channel
      */
      public static ShortMessage noteOn(Note note, int channel) {
            ShortMessage on = new ShortMessage();
            try {
                  on.setMessage(ShortMessage.NOTE_ON, channel, note.getValue(),
note.getVolume());
            } catch (InvalidMidiDataException e) {
                  System.err.println("Error while turning note on");
                  System.exit(1);
            }

            return on;
      }

      /**
            Creates a Note Off message.
            @param note a Note object
            @param channel the channel on which to apply the message

            @return a ShortMessage containing a NoteOff command for
                  the specified channel
      */
      public static ShortMessage noteOff(Note note, int channel) {
            ShortMessage off = new ShortMessage();
            try {
                  off.setMessage(ShortMessage.NOTE_OFF, channel, note.getValue(), 0);
            } catch (InvalidMidiDataException e) {
                  System.err.println("Error while turning note off");
                  System.exit(1);
            }

            return off;
      }

      /**
            Creates a Program Change message.
            @param program a GM instrument number from 0 to 127
            @param channel the channel on which to change the instrument

            @return a ShortMessage containing a Program Change command for
                  the specified channel
      */
      public static ShortMessage programChange(int program, int channel) {
            ShortMessage patch = new ShortMessage();
```

```
		try {
			patch.setMessage(ShortMessage.PROGRAM_CHANGE, channel, program, 0);
		} catch (InvalidMidiDataException e) {
			System.err.println("Error while changing instrument");
			System.exit(1);
		}

		return patch;
	}

	/**
		Creates a Tempo Change message
		@param tempoMPQ a number ranging from 0 to (limits unknown)
			recall that MPQ = 60000000 / BPM
		@param channel the channel on which to apply the message

		@return a ShortMessage containing a Tempo Change command for
			the specified channel
	*/
	public static MetaMessage tempo(int tempoMPQ, int channel) {
		MetaMessage tempo = new MetaMessage();
		try {
			byte [] tempoByte = new byte[3];
			tempoByte[0] = (byte)( (tempoMPQ >> 16) & 0xFF );
			tempoByte[1] = (byte)( (tempoMPQ >> 8) & 0xFF );
			tempoByte[2] = (byte)( tempoMPQ & 0xFF );

			tempo.setMessage(0x51, tempoByte, tempoByte.length);
		} catch (InvalidMidiDataException e) {
			System.err.println("Error while changing tempo");
			System.exit(1);
		}

		return tempo;
	}

	/**
		Creates a Title Change message
		@param seqName a String containing the sequence title

		@return a ShortMessage containing a Sequence Name message
	*/
	public static MetaMessage title(String seqName) {
		MetaMessage name = new MetaMessage();
		try {
			name.setMessage(0x03, seqName.getBytes(), seqName.length() );
		} catch (InvalidMidiDataException e) {
			System.err.println("Error while creating sequence name");
			System.exit(1);
		}

		return name;
	}
}
```

MIDLMANAPP.JAVA

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.table.*;
import javax.sound.midi.*;
import java.io.*;
import java.util.StringTokenizer;
import javax.swing.text.BadLocationException;

import antlr.*;
import midl.*;

public class MidlManApp {

	public static void main(String [] args) {
		javax.swing.SwingUtilities.invokeLater(new Runnable() {
			public void run() {
				MidlAppUI ui = new MidlAppUI();
			}
		});
	}
}

class MidlAppUI extends JFrame {
	private static final int SIZE_X = 500;
	private static final int SIZE_Y = 350;
	private static final String WND_TITLE = "MIDLMan";

	// Declaring main UI components + MIDI output class

	private InputArea textArea;
	private StatusBar statusbar;
	private Slider slider;
	private static MidiIO midi;

	public MidlAppUI() {
		super(WND_TITLE);
		midi = new MidiIO();
		createAndShowGUI();
	}

	public void performCleanup() {
		midi.cleanup();
		System.exit(0);
	}

	private void createAndShowGUI() {
		// Center the main window
		Dimension res = Toolkit.getDefaultToolkit().getScreenSize();
		setSize(SIZE_X, SIZE_Y);
		setLocation((int) (res.getWidth() - SIZE_X)/2, (int) (res.getHeight() -
SIZE_Y)/2);

		// Handle the window closing event
		addWindowListener(new WindowAdapter() {
			public void windowClosing(WindowEvent w) {
				if ( textArea.confirmChange() ) performCleanup();
			}
		});
```

```
		/*
			This is a required statement.  If omitted, the program
			continues to run after receiving a windowClosing event.
		*/
			setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);

		// Add menus + main panel
		setJMenuBar(new MidlMenu());
		setContentPane(createMainPanel());
		setVisible(true);
	}

	/*
		Creates a JPanel and adds the three main components:
		sequence slider bar, script area, and status bar.
		Just for the record, I hate layout managers in Java.
	*/
	private Container createMainPanel() {
		final int BORDER = 5;

		JPanel panel = new JPanel(new BorderLayout(BORDER, BORDER));
		textArea = new InputArea();
		statusbar = new StatusBar();
		slider = new Slider();

		// Create a padding area of 5 pixels before adding the components
		panel.setBorder(BorderFactory.createEmptyBorder(BORDER, BORDER, BORDER,
BORDER));

		// Add the components.  Note that a JTextArea must be enclosed
		// in a JScrollPane object.
		panel.add(slider, BorderLayout.NORTH);
		panel.add(new JScrollPane(textArea), BorderLayout.CENTER);
		panel.add(statusbar, BorderLayout.SOUTH);

		return panel;
	}

	/*
		The slider component.  Has various extensions and can
		disable itself by calling the status of the MidiIO object.
	*/
	private class Slider extends JSlider implements ChangeListener {
		private Timer updateThread;
		private static final int INTERVAL = 10;

		public Slider() {
			super(JSlider.HORIZONTAL);

			/*
				When this timer is active, it checks to see if the
				midiIO object has a sequence loaded AND if it's
				currently playing.  If so, the slider synchronizes
				its position with the MIDI file position.

				If the sequence has stopped playing or is not loaded,
				the slider calls its overloaded disable function.
			*/
			updateThread = new Timer(INTERVAL, new ActionListener() {
				public void actionPerformed(ActionEvent e) {
					if ( midi.isLoaded() && midi.isPlaying() )
						setValue(midi.getPosition());
```

```
                }
        });

        // Performs an action when the user manually moves the slider
        addChangeListener(this);

        // Disable the slider on startup
        setEnabled(false);
    }

    private void synchSlider() {
        if ( midi.isLoaded() )
            setValue(midi.getPosition());
    }

    public void setEnabled(boolean flag) {
        super.setEnabled(flag);

        /*
            If flag == true, then set the slider size from 0 to the
            number of ticks in the sequence (normally, a JSlider is
            from 0 to 100).  The call to midi.isLoaded() is required
            to avoid nasty null pointer errors.

            If flag == false, move the slider position to 0 and stop
            the timer thread.
        */
        if (flag) {
            if ( midi.isLoaded() ) {
                setMaximum(midi.getLength());
                updateThread.start();
            }
        } else {
            setValue(0);
            updateThread.stop();
        }
    }

    /*
        Called when the user drags the slider while playing.  The
        first comparison is required because stateChange is also
        called when the slider changes using a setValue() function
        (i.e.: in the Timer).

        Also, just checking if the slider position == sequence position
        is not acceptable.  In several instances, the GUI thread and
        MIDI thread don't match up and some notes are replayed.

        Failure to perform the comparison results in horrible
        skipping output.
    */
    public void stateChanged(ChangeEvent c) {
        final int THRESHOLD = 10;
        /*
            If the sequence position and slider position are more than
            10 ticks apart, move the sequence position to match the
            slider position.
        */
        if ( Math.abs(getValue() - midi.getPosition()) > THRESHOLD)
            midi.setPosition(getValue());
    }
}
```

```
	/*
		The menubar component.  It contains various utility functions
		to create submenus and on occasion, raise the dead.  Is anybody
		reading the comments?
	*/
	private class MidlMenu extends JMenuBar {
		public MidlMenu() {
			super();
			createMenuBar();
		}

		/*
			Creates the top level menus and adds the necessary submenus.
		*/
		private void createMenuBar() {
			JMenu file, edit, seq, instrument;
			JCheckBoxMenuItem editWrap;

			file = createMenu("File", KeyEvent.VK_F);
			edit = createMenu("Edit", KeyEvent.VK_E);
			seq = createMenu("Sequence", KeyEvent.VK_S);

			// The instrument menu is complex enough that it requires
			// its own utility function.
			instrument = createInstrumentMenu();

			file.add(createMenuItem("New", new NewDocumentListener(),
KeyEvent.VK_N));
			file.add(createMenuItem("Open...", new OpenDocumentListener(),
KeyEvent.VK_O));
			file.add(createMenuItem("Save As...", new SaveDocumentListener(),
KeyEvent.VK_S));
			file.addSeparator();
			file.add(createMenuItem("Exit", new ExitListener(), -1));

			// The Edit->Wrap Text menu item requires special handling since
			// it uses a different type of object.
			editWrap = new JCheckBoxMenuItem("Word Wrap");
			editWrap.setMnemonic(KeyEvent.VK_W);
			editWrap.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_W,
ActionEvent.CTRL_MASK));
			editWrap.addItemListener(new WrapListener());
			edit.add(editWrap);

			seq.add(createMenuItem("Preview", new PreviewListener(),
KeyEvent.VK_P));
			seq.add(createMenuItem("Export", new ExportListener(), -1));

			add(file);
			add(edit);
			add(seq);
			add(instrument);
		}

		/*
			createMenuItem accepts various parameters and returns a JMenuItem.
			If accelerator == -1, it doesn't assign a shortcut key.  Examples
			of menus which should not have shortcuts include File->Exit and
			Sequence->Export
		*/
		private JMenuItem createMenuItem(String caption,
```

```
            ActionListener listener, int accelerator) {
                  JMenuItem item = new JMenuItem(caption);
                  if ( accelerator != -1 ) {
                        item.setAccelerator(
                              KeyStroke.getKeyStroke(accelerator,
ActionEvent.CTRL_MASK));
                  }
                  // Underline the first letter of the menu
                  item.setMnemonic(caption.charAt(0));
                  item.addActionListener(listener);

                  return item;
            }

            /*
                  createMenu is a utility function for creating top level JMenu
                  objects.
            */
            private JMenu createMenu(String caption, int mnemonic) {
                  JMenu menu = new JMenu(caption);
                  menu.setMnemonic(mnemonic);

                  return menu;
            }

            /*
                  The instruments menu contains a list of General MIDI Instruments.
                  Rather than hard coding them in, MIDLMan loads them from a data
                  file and forms 16 groups containing 8 instruments each.
            */
            private JMenu createInstrumentMenu() {
                  final String INPUT_FILE = "resources/MidlMenuInstruments.txt";
                  final String [] GROUP_HEADINGS = {
                        "Piano", "Chromatic Percussion", "Organ", "Guitar",
                        "Bass", "Strings", "Ensemble", "Brass", "Reed", "Pipe",
                        "Synth Lead", "Synth Pad", "Synth Effects",     "Ethnic",
                        "Percussive", "Sound Effects"
                  };

                  JMenu menu = new JMenu("Instruments");
                  menu.setMnemonic(KeyEvent.VK_I);

                  // Create the 8 instrument group menus
                  for ( String group : GROUP_HEADINGS )
                        menu.add(new JMenu(group));

                  try {
                        // getClassLoader() is required in order to load files from a JAR
archive
                        InputStream rawIn =
this.getClass().getClassLoader().getResourceAsStream(INPUT_FILE);
                        BufferedReader bReader = new BufferedReader(new
InputStreamReader(rawIn));

                        String line = null;
                        int groupIndex = 0;

                        /*
                              For each instrument loaded, create a new JMenuItem
                              and add it to the appropriate instrument group JMenu.
                        */
                        while ( (line = bReader.readLine()) != null ) {
```

```
                              JMenuItem instrument = new JMenuItem(line);
                              instrument.setActionCommand("instrument");
                              instrument.addActionListener(new InstrumentListener());

                              /*
                                    Since there are 128 lines in the data file, groupMenu
                                    will range from 0 to 127.

                                    groupMenu/8 index returns a number between 0 and 15
                                    because of the peculiarities of dividing ints.

                                    Since there are exactly 16 top level JMenu instrument
                                    groups, we can use groupMenu/8 to add the JMenuItem
                                    to the correct JMenu.
                              */
                              JMenu group = (JMenu) menu.getMenuComponent(groupIndex/8);
                              group.add(instrument);
                              groupIndex++;
                        }

                              bReader.close();
                  } catch (IOException e) {
                              System.err.println(e);
                              throw new RuntimeException("Error in
MidlMenu:createInstrumentMenu() (Unable to open instrument data file)");
                  }

                  return menu;
            }
      }

      /*
            The script area component below is descended from a JTextArea and
            its primary features include tracking changes to the script
            contents and updating the caret position in the status bar.
      */
      private class InputArea extends JTextArea
      implements DocumentListener, CaretListener {
            private static final int INPUT_ROWS = 5;
            private static final int INPUT_COLS = 15;
            private static final int TAB_SIZE = 2;
            private static final int BORDER = 5;
            private boolean changed = false;

            public InputArea() {
                  super(INPUT_ROWS, INPUT_COLS);
                  setTabSize(TAB_SIZE);
                  setBorder(BorderFactory.createEmptyBorder(BORDER, BORDER, BORDER,
BORDER));
                  getDocument().addDocumentListener(this);
                  addCaretListener(this);
            }

            public void clear() {
                  setText(null);
                  changed = false;
            }

            public void setText(String s) {
                  super.setText(s);
                  changed = false;
            }
```

```
		/*
			confirmChange is called upon by all destructive operations
			including creating new documents, opening documents, and of course,
			exiting the program.  It returns true if the document has not
			changed since the last save (or is a new blank document) and
			false otherwise.
		*/
		public boolean confirmChange() {
			if ( !changed ) return true;

			final String caption = "Are you sure you wish to close this document
without saving?";

			int choice = JOptionPane.showConfirmDialog(null,
				caption, "Confirmation", JOptionPane.YES_NO_OPTION);

			return ( choice == JOptionPane.YES_OPTION );
		}

		public void changedUpdate(DocumentEvent e) { changed = true; }
		public void insertUpdate(DocumentEvent e) { changed = true; }
		public void removeUpdate(DocumentEvent e) { changed = true; }

		public void caretUpdate(CaretEvent c) {
			/*
				getDot() returns a number between 0 and document.length.
				We use the getLineStartOffset() function to get the current
				line, and we subtract the current line's offset from getDot()
				to get the current column.

				Since both lines and columns start from 0, we increment them
				to match ANTLR's token stream.
			*/
			try {
				int caretPosition = c.getDot();
				int lineNumber = getLineOfOffset(caretPosition);
				int column = caretPosition - getLineStartOffset(lineNumber);
				statusbar.displayPosition((lineNumber+1) + ":" + (column+1));
			} catch (BadLocationException b) { System.err.println(b); }
		}
	}

	/*
		The StatusBar component consists of two JLabels packaged in a JPanel.
		It includes member functions for displaying error messages as well
		as caret line + column positions.
	*/
	private class StatusBar extends JPanel {
		private JLabel status, line;

		public StatusBar() {
			super();
			setLayout(new BoxLayout(this, BoxLayout.X_AXIS));

			status = new JLabel();
			line = new JLabel();
			line.setHorizontalTextPosition(JLabel.RIGHT);
			status.setHorizontalTextPosition(JLabel.LEFT);

			add(Box.createHorizontalStrut(10));
			add(status);
```

```
            add(Box.createHorizontalGlue());
            add(line);
            add(Box.createHorizontalStrut(10));

            // Set the line to 1:1 so it doesn't appear blank at first.
            displayPosition("1:1");
      }

      public void displayError(String error) {
            status.setText(error);
      }

      public void clear() {
            status.setText(null);
      }

      public void displayPosition(String position) {
            line.setText("Line " + position);
      }
}

/*
      Handles the File->New action.  The user is prompted to save
      the current script if necessary, the titlebar and status bars
      are both cleared, and any MIDI Sequences currently playing are
      halted.
*/
private class NewDocumentListener implements ActionListener {
      public void actionPerformed(ActionEvent a) {
            if ( textArea.confirmChange() ) {
                  textArea.clear();
                  statusbar.clear();
                  if ( midi.isPlaying() ) midi.stopSequence();
            }
      }
}

/*
      Handles the File->Open action.  Similar to the NewDocumentListener,
      except it appends the name of the opened file to the titlebar.
*/
private class OpenDocumentListener implements ActionListener {
      public void actionPerformed(ActionEvent a) {
            try {
                  String result [] = MidlAppIO.openFile();
                  if ( result != null ) {
                        setTitle(WND_TITLE + " - " + result[0]);
                        textArea.setText(result[1]);
                        statusbar.clear();

                        if ( midi.isPlaying() ) midi.stopSequence();
                  }
            } catch (RuntimeException r) {
                  statusbar.displayError(r.getMessage());
            }
      }
}

/*
      Handles File->Save As and appends the file name to the title bar.
*/
private class SaveDocumentListener implements ActionListener {
```

```
	public void actionPerformed(ActionEvent a) {
		try {
			String buffer = textArea.getText();
			String result = MidlAppIO.saveFile(buffer);
			if ( result != null ) {
				textArea.setText(buffer);
				setTitle(WND_TITLE + " - " + result);
				statusbar.clear();
			}

		} catch (RuntimeException r) {
			statusbar.displayError(r.getMessage());
		}
	}
}

/*
	Called with Sequence->Preview.  This reads the script area contents,
	passes it to a MidlAppIO function to create a Sequence, and begins
	playback.  Any errors in parsing the script are returned as
	RuntimeExceptions and displayed in the status bar.
*/
private class PreviewListener implements ActionListener {
	public void actionPerformed(ActionEvent a) {
		try {
			Sequence seq = MidlAppIO.createSequence(textArea.getText());

			// Assigns returned sequence to the MidiIO instance.
			midi.setSequence(seq);

			/*
				Playback in a new thread is *required*.  Single-threaded
				playback freezes the GUI until the sequence finishes.
			*/
			new Thread() {
				public void run() {
					slider.setEnabled(true);
					midi.playSequence();
					slider.setEnabled(false);
				}
			}.start();

			statusbar.clear();
		} catch (RuntimeException r) {
			statusbar.displayError(r.getMessage());
		}
	}
}

/*
	Similar to the PreviewListener, except it doesn't need a thread
	since the conversion from script->Sequence is nearly instantaneous.
	In future versions, this might be necessary, depending on how people
	are using this program.
*/
private class ExportListener implements ActionListener {
	public void actionPerformed(ActionEvent a) {
		try {
			Sequence seq = MidlAppIO.createSequence(textArea.getText());
			midi.setSequence(seq);
			midi.exportSequence();
			statusbar.clear();
```

```
                } catch (RuntimeException r) {
                        statusbar.displayError(r.getMessage());
                }
        }
    }

    /*
        I'll admit it, this is a bit of a hack.  This event listener
        is attached solely to the Instrument menu and listens for any
        event originating from one of its submenus.  Upon firing, it
        reads the caption from the source and inserts it at the caret
        position.
    */
    private class InstrumentListener implements ActionListener {
        public void actionPerformed(ActionEvent a) {
            JMenuItem source = (JMenuItem) a.getSource();
            String instrument = source.getText();
            textArea.replaceSelection(instrument);
        }
    }

    /*
        Handles File->Exit by confirming to save changes and closing
        down the MIDI devices.
    */
    private class ExitListener implements ActionListener {
        public void actionPerformed(ActionEvent a) {
            if ( textArea.confirmChange() ) performCleanup();
        }
    }

    /*
        This listener for the checkbox menu item toggles text wrapping.
    */
    private class WrapListener implements ItemListener {
        public void itemStateChanged(ItemEvent i) {
            textArea.setLineWrap(!textArea.getLineWrap());
        }
    }
}

/*
    MidlAppIO handles opening and saving, and creates Sequences from Strings.
*/
class MidlAppIO {
    public static String [] openFile() {
        StringBuffer buffer = null;
        JFileChooser fChooser = new JFileChooser();
        int choice = fChooser.showOpenDialog(null);

        // If the user clicked Cancel, exit this function.
        if ( choice != JFileChooser.APPROVE_OPTION ) return null;

        File inputFile = fChooser.getSelectedFile();

        try {
            FileReader fReader = new FileReader(inputFile);
            BufferedReader bReader = new BufferedReader(fReader);

            buffer = new StringBuffer();
            String line = null;
```

```
                while ( (line = bReader.readLine()) != null ) {
                        buffer.append(line); buffer.append("\n");
                }

                bReader.close();
        } catch (IOException e) {
                String error = "Error in MidlAppIO:openFile() (Unable to open file)";
                throw new RuntimeException(error);
        }

        // Return a string array containing the file name and file contents.s
        return new String [] { inputFile.getName(), buffer.toString() };
    }

    public static String saveFile(String contents) {
        JFileChooser fChooser = new JFileChooser();
        int choice = fChooser.showSaveDialog(null);

        if ( choice != JFileChooser.APPROVE_OPTION ) return null;

        File outputFile = fChooser.getSelectedFile();

        try {
                FileWriter fWriter = new FileWriter(outputFile);
                BufferedWriter bWriter = new BufferedWriter(fWriter);

                /*
                        JTextAreas denote newlines with the '\n' character,
                        regardless of platform.  The following lines delimit the
                        input String with '\n' and write the system appropriate
                        newline with the newLine() function.
                */

                StringTokenizer tokenizer = new StringTokenizer(contents, "\n", true);

                while( tokenizer.hasMoreTokens() ) {
                        String line = tokenizer.nextToken();
                        bWriter.write(line); bWriter.newLine();
                }

                bWriter.close();
        } catch (IOException e) {
                String caption = "Error in MidlAppIO:saveFile() (Unable to write
file)";
                throw new RuntimeException(caption);
        }

        // Return a string containing the file name.
        return outputFile.getName();
    }

    /*
        Gets a string, passes it to the lexer + parser duo, and returns
        a Sequence.
    */
    public static Sequence createSequence(String contents) {
        try {
                MidlLexer lexer = new MidlLexer(new StringReader(contents));
                MidlParser parser = new MidlParser(lexer);

                return parser.song();
        } catch (TokenStreamException t) {
```

```
                throw new RuntimeException(t);
            } catch (RecognitionException r) {
                throw new RuntimeException(r);
            }
        }
}

/*
        Manages MIDI playback, device management, etc.
*/
class MidiIO implements MetaEventListener {
        private Sequencer sequencer;
        private Synthesizer synthesizer;
        private Sequence sequence;
        private Object lock;
        private static final int END_OF_SEQUENCE = 47;
        private static final int MIDI_FILE_TYPE = 1;

        public MidiIO() {
                lock = new Object();
                initDevices();
        }

        /*
                Attaches a sequence to the current instance.  This is
                required before any playback/isPlaying member functions
                are called.
        */
        public void setSequence(Sequence sequence) {
                this.sequence = sequence;
                try {
                        sequencer.setSequence(sequence);
                } catch (InvalidMidiDataException i) {
                        System.err.println(i);
                        throw new RuntimeException("MIDI Sequence is invalid");
                }
        }

        // Called automatically upon initialization to init Seq/Synth.
        private void initDevices() {
                try {
                        sequencer = MidiSystem.getSequencer();
                        sequencer.addMetaEventListener(this);
                        sequencer.open();

                        synthesizer = MidiSystem.getSynthesizer();
                        synthesizer.open();

                        sequencer.getTransmitter().setReceiver(synthesizer.getReceiver());

                } catch (Exception e) {
                        System.err.println(e);
                }
        }

        // Cleanup
        public void cleanup() {
                if ( synthesizer != null ) synthesizer.close();
                if ( sequencer != null ) sequencer.close();
        }

        /*
```

```
		Stops the playing sequence, if any, and restarts
		playback.  A sequence MUST be loaded prior to calling
		this function.
	*/
	public void playSequence() {
		if ( sequencer.isRunning() ) sequencer.stop();

		sequencer.start();
		synchronized(lock) {
			while( sequencer.isRunning() ) {
				/*
					The following code halts until the lock
					object receives a notify() message or is
					otherwise interrupted.
				*/
				try {
					lock.wait();
				} catch (InterruptedException e) { }
			}
		}
	}

	/*
		Releases the lock and stops the sequence.
	*/
	public void stopSequence() {
		sequencer.stop();
		synchronized(lock) {
			lock.notify();
		}
	}

	/*
		Creates a MIDI file from the loaded sequence.
	*/
	public void exportSequence() {
		JFileChooser fChooser = new JFileChooser();
		fChooser.setFileFilter(new FileMask("mid", "MIDI Files"));

		int choice = fChooser.showSaveDialog(null);
		if ( choice == JFileChooser.APPROVE_OPTION ) {
			File outputFile = fChooser.getSelectedFile();

			try {
				MidiSystem.write(sequence, MIDI_FILE_TYPE, outputFile);
			} catch (IOException e) {
				System.err.println(e);
				throw new RuntimeException("Error in MidiIO:exportSequence()
(Unable to export file)");
			}
		}
	}

	// Returns the loaded sequence length in ticks.
	public int getLength() {
		return (int) sequencer.getTickLength();
	}

	// Returns the position of the loaded sequence in ticks.
	public int getPosition() {
		return (int) sequencer.getTickPosition();
	}
```

```
	// Moves the current tick position of the loaded sequence.
	public void setPosition(int position) {
		if ( position >= 0 && position <= getLength() )
			sequencer.setTickPosition(position);
	}

	// This returns true if the sequencer is playing something.
	public boolean isPlaying() {
		return sequencer.isRunning();
	}

	// This returns true if there is a valid sequence loaded.
	public boolean isLoaded() {
		return (sequence != null);
	}

	/*
		Another way to release the lock object created with
		playSequence is simply to wait for an event type 47,
		signifying an "end of track" message.
	*/
	public void meta(MetaMessage m) {
		if ( m.getType() == END_OF_SEQUENCE ) {
			synchronized(lock) {
				lock.notify();
			}
		}
	}
}

/*
	A simple helper class for showing only specific files in
	Open/Save dialog boxes.
*/
class FileMask extends javax.swing.filechooser.FileFilter {
	private String extension, description;

	public FileMask(String extension, String description) {
		this.extension = "." + extension;
		this.description = description;
	}

	public boolean accept(File file) {
		return file.isDirectory() ||
			file.getName().toLowerCase().endsWith(extension);
	}

	public String getDescription() {
		return description;
	}
}
```

DRIVER.JAVA

```
import javax.sound.midi.*;
import antlr.*;
import java.io.*;
import midl.*;

public class Driver
{
	/**
		The driver is a console based app which can take input directly
		from the console (terminated by pressing Ctrl+Z), or have a file
		piped to it in the following manner:

		type sampleFile | java -cp .;source-to-antlr.jar.file Driver
	*/
	public static void main(String [] args) throws Exception	{

		try {
			MidlLexer lexer = new MidlLexer(new DataInputStream(System.in));
			MidlParser parser = new MidlParser(lexer);
			Sequence output = parser.song();

			final Sequencer sequencer = MidiSystem.getSequencer();
			sequencer.setSequence(output);

			sequencer.open();
			sequencer.addMetaEventListener(
				new MetaEventListener() {
					public void meta(MetaMessage event) {
						if ( event.getType() == 47 ) {
							sequencer.stop();
							sequencer.close();
						}
					}
				});
			sequencer.start();
		} catch (TokenStreamException tokenE) {
			System.err.println("Caught Token Exception: " + tokenE);
		} catch (RecognitionException recogE) {
			System.err.println("Caught Recognition Exception: " + recogE);
		}
	}
}
```

APPENDIX E – COPYRIGHT