COLUMBIA UNIVERSITY

PROGRAMMING LANGUAGES AND TRANSLATORS

COMS W4115

DEVELOPERS

Andrew Goldin
Project Manager
adg2160@columbia.edu

Cindy Long
System Architect
xl2259@columbia.edu

Matt Kim
System Integrator
mjk2189@columbia.edu

Kevin Walters
Language Guru
kmw2168@columbia.edu

SUPERVISOR

Professor Alfred Aho
aho@cs.columbia.edu

DATE

May 13, 2013

TABLE OF CONTENTS

1 An Introduction to *SouL* *1*

2 *SouL* Language Tutorial *5*
- **2.1** Introduction *5*
- **2.2** Getting Started *5*
- **2.3** Control Flow Statements and Loops *7*
- **2.4** Writing MIDI Files *9*
- **2.5** Playing MIDI Files *11*
- **2.6** Editing MIDI Files *11*
- **2.7** Using Built-in Functions *13*

3 *SouL* Language Reference Manual *15*
- **3.1** Lexical Definitions *15*
  - **3.1.1** Tokens *15*
  - **3.1.2** Comments *15*
  - **3.1.3** Identifiers *15*
  - **3.1.4** Keywords *16*
  - **3.1.5** Constants *16*
    - **3.1.5.1** Number Constants *16*
    - **3.1.5.2** Boolean Constants *17*
    - **3.1.5.3** Pitch Constants *17*
  - **3.1.6** String Literals *17*
- **3.2** Expressions *18*
  - **3.2.1** Operators *18*
    - **3.2.1.1** Binary Operations *18*
      - **3.2.1.1.1** Addition and Subtraction *18*
      - **3.2.1.1.2** Multiply, Divide and Mod *19*
      - **3.2.1.1.3** Exponentiation *19*
      - **3.2.1.1.4** Comparative Operators *19*
      - **3.2.1.1.5** Equality Operators *20*
      - **3.2.1.1.6** Logical Operators *20*
      - **3.2.1.1.7** Assignment Operators *20*
      - **3.2.1.1.8** Commas *21*
    - **3.2.1.2** Unary Operations *21*
      - **3.2.1.2.1** Prefix Expressions *21*
      - **3.2.1.2.2** Postfix Expressions *22*
- **3.3** Declarations *22*
  - **3.3.1** Declarators *22*
  - **3.3.2** Type Declarations *23*
  - **3.3.3** Object Declarations *23*
  - **3.3.4** Type and Object Specifiers *24*
  - **3.3.5** Initialization *24*
- **3.4** Statements *25*
  - **3.4.1** Expression Statements *25*

3.4.1.1 Built-in Functions *26*
3.4.1.1.1 Play Statement *26*
3.4.1.1.2 Print Statement *26*
3.4.1.1.3 Write Statement *26*
3.4.1.1.4 Append Statement *26*
3.4.1.1.5 Clear Statement *27*
3.4.1.1.6 Addition Statement *27*
3.4.1.1.7 Get Sequence Statement *27*
3.4.1.1.8 Set Instrument Statement *27*
3.4.1.1.9 Set Tempo Statement *28*
3.4.2 Compound Statements *28*
3.4.3 Selection Statements *28*
3.4.4 Iteration Statements *29*
3.5 Scope *29*
3.6 Full Grammar *30*
3.7 *SouL* Style Guide *35*

4 Project Plan *38*
4.1 Overall Process *38*
4.2 Team Roles and Responsibilities *38*
4.3 Implementation Style Guide *39*
4.4 Project Timeline *40*
4.5 Meetings Log *40*

5 Language Evolution *46*

6 Translator Architecture *49*
6.1 Block Diagram and Module Description *49*
6.2 Authors of Each Module *49*

7 Development and Runtime Environment *50*

8 Test Plan *54*
8.1 Test Methodology *54*
8.2 Test Suite *56*

9 Conclusions *66*
9.1 Lessons Learned as a Team *66*
9.2 Lessons Learned by Each Team Member *66*
9.3 Advice for Future Teams *69*
9.4 Suggestions for the Course *69*

APPENDIX A: Full *SouL* Source Code *70*
Author Credits *70*
Lexer Code *71*
Parser Code *72*
Semantic Analyzer Code *79*
Java Back-End Code *105*
Makefiles and Shell Script *121*

1. AN INTRODUCTION TO *SouL*

Written by the entire SouL Team

The programming language SouL (an abbreviation of **Sou**nd **L**anguage) allows users to easily program musical compositions. The targeted users are musicians, but the versatility of SouL makes it appealing to several other groups as well. Users can generate and edit Musical Instrument Digital Interface (MIDI) files using SouL. The primitives of SouL are pitch, velocity, duration, instrument, boolean, int, decimal, and string. The ability of SouL to properly access and use any system's MIDI library makes it a portable, architecture neutral programming language that can run on multiple systems identically. SouL is a scripting-like language with some built-in objects, that is compiled rather than interpreted. It contains the predefined standard objects Midi, Note, Chord, Track, and Sequence. SouL source code is translated into Java and compiled by the Java compiler. Since Java is architecture neutral, SouL is as well.

DESIGN GOALS

The purpose of this language is to make it easier for musicians who have minimal programming experience to be able to design, build, and edit musical sequences and passages programmatically. In doing so, they can better automate the construction of their music and save time in the process. Hence, one of the main goals of SouL is concerned with the creation and editing of MIDI files. As the process of music production becomes more digitizes, MIDI data is being used more and more by all sort of people, ranging from producers to hobbyist performers.

One purpose of SouL is the creation of MIDI files. MIDI sequence creation in SouL follows this hierarchy: A sequence contains tracks, and each track contains a series of notes and chords. Once a sequence is created, it can be played, edited, or written out to a file. This MIDI file can then be saved into either a default directory or one defined by the user. Musicians may use these MIDI files to transmit musical data across different digital interfaces.

SouL can also be used to edit existing MIDI files. Using a proprietary tokenizer, SouL is able to tokenize the individual tracks, chords, and notes that make up a given MIDI file and edit them to the user's liking. Through this process, a user can write scripts to mass-edit certain portions of a MIDI file all in the same way. For example, suppose the MIDI file is written in C Major. By transposing every note up one whole step using SouL's built-in transpose functionality, a user can create the same file but transposed into D Major.

SouL is able to play the files and sequences it creates. Through this process, SouL draws upon the built-in MIDI libraries provided on the user's system. Because of this, a programmer need only interact with SouL files and be able to use a much simpler interface than MIDI to transmit musical data.

SouL IS SIMPLE

A major goal of SouL is to provide a simple and easy-to-understand tool for performing music-related tasks, to be both created and read by musicians alike. To help facilitate this, SouL uses an intuitive set of primitive types that are closely related to musical terms that any average user would be familiar with. These types include: pitch (e.g. C4, F6, G#3), velocity (a value representing input volume), duration (e.g. QUARTER, HALF, WHOLE), instrument (MIDI instrument type), as well as standard operating types such as boolean, int, decimal and string to allow for more standard programming paradigms and control flow logic. These primitive values are universally important to the creation and manipulation of music and sound, and can be applied to various objects in order to represent certain parts of a musical score or musical application.

SouL IS A SCRIPTING-LIKE LANGUAGE

Since many aspects of musical notation and creation are thought of in groups, be it measures, phrases, or chords, so SouL incorporate some object oriented approach to programming. However, SouL is not an object-oriented language. Instead, SouL has characteristics of scripting languages and a bit of object-oriented design.

Certain native objects will contain primarily primitive data, such as a Note (a piece of data containing a pitch, velocity, and duration), a Chord (a set of concurrent Notes), or a Track (a set of asynchronous Notes/Chords). A Track object will be able to have additional tracks concatenated onto it, as well as various subtracks extracted from it to suit the programmer's needs. Other objects exist to facilitate the creation and manipulation of external data, such as a MIDI File object, which will store sequence and instrument data in order to easily create and modify MIDI file information.

SouL IS SECURE

Because the SouL compiler is written in Java, it provides a secure programming platform. Java is a programming language that is more secure than most other languages because the Java compiler

catches more compile-time errors. In turn, a SouL program cannot be compiled if it could produce unpredictable results. Further, Java's garbage collection feature will be utilized by SouL so that objects that are eligible for garbage collection will be reclaimed to heap space. Essentially, SouL will run efficiently because it will not have leftover objects that are unimportant and use extra space.

Because of the large amounts of musical sounds available for MIDI files, including them within SouL would take too much space and affect its performance. Since nearly all computer systems have a built-in MIDI synthesizer that already contains these musical sounds, SouL will take advantage of these files. This way, memory and space are saved, so that SouL programs can run as efficiently as possible.

SouL WORKS ON ANY UNIX-BASED MACHINE

SouL is a semi-portable programming language that can run successfully on any machine with a UNIX-based architecture. The SouL compiler is made for the programmer's convenience, and is designed to be as consistent as possible across these systems. Every system has a unique location where the MIDI synthesizer exists. Luckily, because SouL's back end is written using Java, which can automatically detect and read from the MIDI system of nearly any machine with MIDI support, SouL programs should run reliably on multiple platforms. This degree of independence makes SouL a very useful programming language. It is also important to note that, as discussed in the following section, SouL is translated into Java, meaning that all code will be run through the Java Virtual Machine, and memory management is taken care of automatically.

SouL IS TRANSLATED AND COMPILED

A program written in SouL is both translated and compiled. Take for example a simple Hello World program coded in SouL. The programmer creates a HelloWorld.soul file, which is then translated by the SouL compiler into the equivalent Java file, HelloWorld.java. The SouL compiler will then run the command javac HelloWorld.java and java HelloWorld.class to run the program. After the completion of the program, these files will be deleted by the compiler, leaving only the .soul file. This implementation hides the translation of the .soul files into .java and .class files and makes the interface much simpler for the user. Assuming the programmer used SouL to engineer a MIDI file, the result of running the file would be the creation of a MIDI output file, which can then be played using a music player, or another SouL program!

SouL IS STATICALLY TYPED

SouL is a static programming language. This means that it uses static typing, determining the types of variables when it compiles. This is comparable to programming languages such as C, C++, C#, and Java. Rather than using dynamic typing, which determines the validity of types at runtime like in Perl, Python, and Ruby, SouL's compiler will ensure that all types are compatible when it compiles. This does potentially remove some level of versatility from the programmer, but it also makes debugging for the user much easier. Since SouL is a simple programming language to help users easily create and edit MIDI files without much programming knowledge, static typing was chosen to make debugging appropriately easier for the programmer.

2. *SouL* LANGUAGE TUTORIAL

Written by the entire SouL Team

2.1 INTRODUCTION

This tutorial teaches the basics to programming MIDI in SouL. This tutorial outlines four SouL programs: how to write MIDI to a file, how to transpose a MIDI file, how to play a MIDI file, and how to play MIDI without writing to a file. These programs can run on any operating system, thereby providing ease and convenience for the users to operate on any computer platform.

An advantage of SouL programming is that users who do not have much programming experience can easily create their MIDI files. In particular, even though the SouL compiler is written in Java, users without Java programming knowledge can program in SouL. However, SouL assumes that users have prior musical knowledge, such as the use of pitch, velocity, and duration of musical notes.

After reading this tutorial, users will be able to write useful programs without extensive knowledge about SouL. Because users use the basics of SouL, these programs may not be as concise as they could be, but will serve their purpose.

2.2 GETTING STARTED

For most languages, the most basic program is a Hello, World! program. In SouL, however, since the purpose of the language is for music playback and writing, we will construct a program that plays C4, or more commonly, Middle C.

The code for this program is listed below:

```
play(Note(‘C4’, 127, WHOLE));
```

If this is stored in playC.soul, the command used to execute this program is:

```
./soul playC.soul
```

It is important to use the Make command before you do this. Running this command takes the .souL file, translate it into a .java file (with the appropriate commands), compile this java program, and execute it. After this has been completed, the intermediate .java file and .class file are deleted.

Looking at the program itself, the first thing that is executed is the library function play(). This function, which is defined by the language, takes a number of different musical constructions (Note, Sequence, Track, Chord, Midi, or a filename) inside its parentheses and plays it back to the user.

Inside the play() method, we have constructed an object, called a Note, giving it the appropriate parameters of a pitch, velocity, and duration. These parameters can be a combinations of integers, decimals, or string constants, depending on the parameter. This Note object which we have made holds three different parameters.

Object-oriented programming is a part of the SouL programming language. It is used to represent musical constructs that we have defined, in order to easily manipulate them and understand them from a high-level standpoint. In addition to these properties that are bound to the objects, we also allow for each of these objects to invoke methods. Methods in SouL are commands that an object can perform to change the internal state of the object. We will see an instance of a method being performed in the next program.

The first argument, which we have called 'C4', specifies the pitch of the note. In Midi construction, all note values are represented by a letter, ranging from A to G, possibly followed by a modifier # or b, which raises the note by a half step up or down respectively, followed by a number. Pitch values are arranged by higher number first, then by letter value (i.e. C4 > D3). The pitches that are within a chromatic scale (i.e. all pitches between a two pitches with the same letter value but with number values that differ by 1) are listed below:

A	Bb	B	B#	Db	D	D#	Fb	F	F#		Ab	A
	A#	Cb	C	C#		Eb	E	E#	Gb	G	G#	

These twelve values (symbols in the same column represent the same pitch, so Bb = A#, B = Cb, etc.) are all the pitches within a chromatic scale. We say that the difference between two adjacent pitches is a half step, and the difference between a pitch two half steps apart is a whole step.

For the purpose of SouL, these pitches range from C-1 to G9, are all constants that are recognized by the compiler, and are always closed within single quotation marks.

The second argument represents the velocity, or loudness, of the note. This value ranges from 0-127 and is an integer value. Higher values have higher velocities.

The final argument represents the length of the note. It can be represented by one of the following constants: WHOLE, HALF, QUARTER, EIGHTH, SIXTEENTH, THIRTYSECOND, SIXTYFOURTH. These constants have been listed in decreasing length, with each subsequent value being half of the previous one.

After passing these three values to the Note object, it has been constructed and stored in memory. The play() function plays it back to the user and the object is then deleted from memory when the program terminates.

The line terminates with a semicolon. All lines of code in SouL are delimited by semicolons and all other whitespace is ignored. Theoretically, we could write all multiline programs on one line, but that is often very difficult to read, so it is not done in practice.

2.3 CONTROL FLOW STATEMENTS AND LOOPS

The next feature of SouL that will be covered is the use of control flow statements, loops, and other SouL constructs. To demonstrate this functionality, we will write a program similar to the one above, except play the notes of a whole tone scale (a scale consisting of 7 notes, all spaced apart by whole steps) except for G4. The code for this program is listed below:

```
Sequence s = Sequence();
Track t = Track();
pitch temp = ‘C4’;
pitch end = ‘C5’;
while (temp <= end) {
  Note n = Note(temp, 127, QUARTER);
  if (temp != ‘G#4’ and temp != ‘F#4’)
     t.add(n);
  temp += 2;
}
s.add(t);
play(s);
```

In the first line of the program, we have constructed a Sequence object. This object holds a Track, which holds a collection of Note objects and can be implemented for reusability.

The next line of the program is the declaration of a Track object. This is similar to as above, but this object is able to hold singular Note objects.

The next four lines define pitch variables for use in relational expressions. The next line of the program is a construct called a while loop. The curly braces {} here denote all code within the while loop (if these curly braces are omitted, only the the next block of code before the first semicolon is within the loop). Essentially, the code within the loop continues to execute as long as the condition within the parentheses is true. Furthermore, all lines of code within the while loop start with a tab character, in order for it to be easy to understand how the code is separated (again, because whitespace is ignored, this is not necessary, but it is implemented for code readability). Inside the parentheses, we have the condition temp <= end, which means it will run as long as the pitch of the Note n is less than C5 (we also have other comparators, >=, ==, !=, denoting greater than or equal to, equal to, and not equal to, respectively).

Inside the loop, we have our first if statement. This control flow statement will only execute the code within it (since there are no curly braces, it only refers to the next line of code) if the statement is true.

In this condition, we have also used the and operator for the first time. In this case, the statement will only execute if both of the statements to the left and right of it are true. Conversely, we also have the or operator, which reads `x or y`, and will not only execute when both statements are false (will execute when at least one is true). Furthermore, the the and operation always takes precedence over the or operation, unless specified with parentheses (i.e. `x or y and z` translates to `x or (y and z)`).

Inside the if statement, we invoke a method for the first time (this line is prefixed with two tabs for code readability). In this method, we add the Note to the Track, which will automatically put the Note at the end. The next line of the program uses the assignment-update operator (+=) to set temp to temp + 2. In this case, temp + 2 denotes the note two half steps above n (i.e. C4 + 2 = D4). This line of the program executes with each execution of the loop because it is outside of the if statement.

Looking at the entire loop, we have initialized the value of n to be at the pitch C4, added it to the track as long as it is not G4, and have added 2 (one whole step) at the end of each run through of it. What this will do is produce a whole tone scale from C4 to C5 with the omission of G4.

Outside of the loop, we call the play() function again, but this time on the sequence, which will play all of the notes in the sequence. Note that the play function could have also been played on the track itself, but this is a good example of how to add a track to a sequence. This is a paradigm of object oriented programming called method overloading, in which the same method (in this case play()), can take different types of arguments (in this case it takes an object of type Sequence; before it took an

instance of type Note), and still produce meaningful results (in this case, SouL plays the musical construction regardless of whether it is a single Note or a Sequence).

We can shorten this code by using a different type of loop, called a for loop. The code for this construction is shown below:

```
Sequence s = Sequence();
Track t = Track();
for (pitch p = 'C4'; p <= 'C5'; p += 2)
  if (p != 'G#4' and p != 'F#4')
     t.add(Note(p, 127, QUARTER));
s.add(t);
play(s);
```

The first line, last line, and code within the loop are the same as before. The first statement inside the parentheses, the initialization, is done before loop executes at all.

The next statement within the for loop is the condition which controls the loop. It is the same as the statement within the while parentheses above.

The last statement inside the for loop is the increment step, which gets executed at the end of the loop (like in the while loop). We have again used the += construction, which in this case translates to the same as above (n += 2 is the same as n = n + 2). There is also the construction -=, which lowers the pitch by the specified amount (n -= 2 is the same as n = n -2).

This is also a useful time to discuss the ++ and the -- constructions. These work similarly to += and -=, but are used to increment by a half step and decrement by a half step, respectively. However, if they are placed before a variable, such as in ++n, it is performed before the line is executed (prefix expression), and if it is placed after the variable, such as in n++, it is performed after the line is executed (postfix expression).

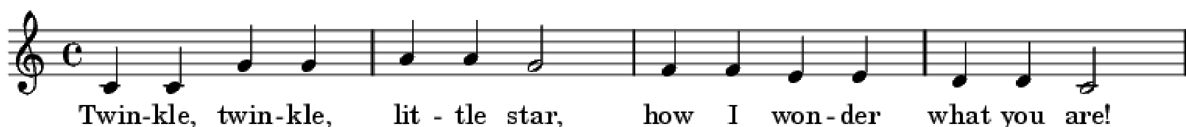
2.4 WRITING MIDI FILES

The next functionality of SouL that will be covered is using the grammar defined by the language to write MIDI files. MIDI, standing for Musical Instrument Digital Interface, refers to a standard that allows different musical instruments to communicate with one another. Basically, it is a way for computers to store different pitches and sounds digitally.

The next program, shown below, writes the famous song Twinkle, Twinkle, Little Star into a MIDI file called test.mid (MIDI files either have the extension .mid or .midi):

```
Sequence s = Sequence();
Track t = Track();
t.add(Note('C4', 127, QUARTER));
t.add(Note('C4', 127, QUARTER));
t.add(Note('G4', 127, QUARTER));
t.add(Note('G4', 127, QUARTER));
t.add(Note('A5', 127, QUARTER));
t.add(Note('A5', 127, QUARTER));
t.add(Note('G4', 127, HALF));
t.add(Note('F4', 127, QUARTER));
t.add(Note('F4', 127, QUARTER));
t.add(Note('E4', 127, QUARTER));
t.add(Note('E4', 127, QUARTER));
t.add(Note('D4', 127, QUARTER));
t.add(Note('D4', 127, QUARTER));
t.add(Note('C4', 127, HALF));
s.add(t);
Midi m = Midi("test.mid");
m.write(s);
```

This program uses the same Sequence object that was implemented in Section 1.2, and a Track object. We add notes to the sequence anonymously, similar to the implementation seen in Section 1.1. The notes that we have added are in the following order:



The construction of the Track t and Sequence s is followed by the construction of a new Midi object, called m. As an argument to the constructor, we have passed "test.mid". This is a construction within SouL called a string literal, which represents a group of characters. We are allowed to use any characters that are allowed within the ASCII standard. Here, it represents the name of the file. Notes are added to the Track, and the Track is then added to the Sequence.

Lastly, we have called a method called write on the Midi object m. This method writes the Sequence s to the Midi file. If the Midi file does not exist, it SouL will create a new file. If the Midi file already exists, then it will clear all data from the file and write from the beginning (users must be careful not to accidentally overwrite their preexisting files).

2.5 PLAYING MIDI FILES

Once we have created our own Midi files (or have imported ones that others have made) we can also use SouL to play these files back to the user. The program to complete this process is shown below:

```
Midi m = Midi("test.mid");
play(m);
```

Because playback is a very important part of SouL, this program was sought to be made as simple as possible. Here, we have declared a Midi file with the same process as Section 1.3.

In the next line, we use the library function that we have used previously, except on the Midi file m. In this case, everything in m will be played out to the user. This is another instance of method overloading because we have invoked play on yet another object.

If we were to execute this program directly following the execution of the program from Section 1.3, SouL would play back "Twinkle, Twinkle, Little Star." From a practical standpoint, this implementation could be used for testing purposes to ensure that everything got correctly written to the Midi file (much like print statements in other programming languages).

2.6 EDITING MIDI FILES

This section will be dedicated to the different ways a user can edit MIDI files. The first thing that will be covered is transposing MIDI files (transposition is a method by which a note, or more often a group of notes, are all either raised or lowered in pitch by a certain value). The SouL program to complete this process is shown below:

```
Midi m = Midi("test.mid");
Sequence s = m.getSequence();
s.transpose(2);
m.write(s);
```

This process constructs the Midi object m, and then uses overloading once again to raise the pitch of all the notes within the file by a whole step.

Although this program is very small, it is very powerful. Using SouL, artists are able to translate their music into any key using the correct transposition using only two lines of code. This process saves the programmer the hassle of individually editing the notes and sequences himself and frees up time for better music production.

The next part of this section will be devoted to using SouL for code reusability and appending sequences together. Suppose we have the first line of Twinkle, Twinkle, Little Star in the file test.mid, from Section 1.3. We want to modify this file to complete the song as shown in the sheet music below:

Twin-kle, twin-kle, lit - tle star, how I won-der what you are!

5 Up a - bove the world so high, like a dia-mond in the sky.

9 Twin-kle, twin-kle, lit - tle star, how I won-der what you are!

The code to complete this process is shown below:

```
Midi m = Midi("test.mid");
Sequence s1 = m.getSequence();
Sequence s2 = Sequence();
Track t = Track();
t.add(Note('G4', 127, QUARTER));
t.add(Note('G4', 127, QUARTER));
t.add(Note('F4', 127, QUARTER));
t.add(Note('F4', 127, QUARTER));
t.add(Note('E4', 127, QUARTER));
t.add(Note('E4', 127, QUARTER));
t.add(Note('D4', 127, HALF));
t.add(Note('G4', 127, QUARTER));
t.add(Note('G4', 127, QUARTER));
t.add(Note('F4', 127, QUARTER));
t.add(Note('F4', 127, QUARTER));
t.add(Note('D4', 127, QUARTER));
t.add(Note('D4', 127, QUARTER));
t.add(Note('C4', 127, HALF));
s2.add(t);
m.append(s2);
m.append(s1);
```

In the first line of this program, we open the test.mid file as in other programs.

In the second line, however, we invoke a new method on m, called getSequence(). Previously, all methods that we have invoked have been mutator methods (another aspect of object oriented programming), or methods that change the state of the object that they are invoked upon. Additionally, all of these methods have not had a return value, meaning assigning them to something within a program would have no meaning and would return an error.

On the other hand, this method is an accessor method, or one that is used to view the internal state of an object. It returns an object of type Sequence, meaning that we can assign it to a new variable that we have named s1 using the assignment operator (=). This method takes the entire track that is enclosed within the Midi file and represents it within one Sequence object. In this case, it is the first line of Twinkle, Twinkle, Little Star.

In the next line, we declare a new Sequence object called s2. Like in Section 1.3, we will add the appropriate notes to s2 in the following lines by using a Track which is next added to s2.

In the next two lines, we call a new method on m, called append(). This method takes the current Midi file and appends the sequence at the end of it, rather than overwriting with the write() method.

This program is particularly important because it demonstrates another one of the key ways in which SouL makes the musician-programmer's life easier: code reuse. In the second line of the program, we extracted s1 from the file m and then used it again in the final line of the program. Because Twinkle, Twinkle, Little Star has a repeating four-measure sequence, we were able to refactor this sequence into m with a single line, rather than having to rewrite and redeclare each note individually all over again.

2.7 USING BUILT-IN FUNCTIONS

Code reuse is implemented more traditionally in SouL through the implementation of functions. A function is a subprocess in a SouL program that may be called multiple times within the execution of a program that takes a finite number of arguments and may or may not return a value. SouL provides a number of built-in functions that are available for a user to be called directly by passing the appropriate parameters. The following SouL code makes use of the functions add, write, play, append, and clear. It is important to note how these functions can be used on, and accept arguments of, many different types.

```
Sequence s = Sequence();
Track t = Track();
```

```
t.add(Note('C4', 127, EIGHTH));
t.add(Note('D4', 127, EIGHTH));
t.add(Note('E4', 127, EIGHTH));
s.add(t);
Midi m = Midi("test.mid");
m.write(s);
play(m);
m.write(t);
play(m);
m.append(s);
Midi m1 = Midi("test_appended.mid");
play(m1);
m1.clear();
play(m1);
Midi m2 = Midi("test2.mid");
Track t2 = Track();
for (int i = 0; i <= 10; i++) {
      t2.add(Note('C4', 127, WHOLE));
}
Sequence s2 = Sequence();
s2.add(t2);
m2.append(s2);
play(m2);
int j = 30;
t.add(Note('Gb4', j, EIGHTH));
t.setInstrument(30);
m2.write(t);
play(m2);
```

In this program, a variety of functions are called at different points on different objects. The play function is called on three different Midi objects, after each has been either written to, appended, or cleared. The add function's use is also displayed here, where it is shown adding Notes to a Track and a Track to a Sequence. The method setInstrument is also introduced here. It is called on a given Track, and takes a numerical parameter that corresponds to a specific instrument.

3. *SouL* LANGUAGE REFERENCE MANUAL

Written by the entire SouL Team

This manual provides a detailed description of the SouL programming language. It will cover all areas of language use, including lexical definitions, syntax, semantics, and grammar of the language.

3.1 LEXICAL DEFINITIONS

A program consists of a source code - a string of characters - that is run through a series of translators, which convert the string to runnable machine code. A lexical analyzer is first run on this source code, converting it into a sequence of tokens.

3.1.1 TOKENS

The types of tokens defined in SouL are classified as follows: *keywords*, *identifiers*, *constants*, *string literals, operators,* and *separators*. Space, tab, and newline characters are considered "white spaces" and are ignored by the compiler unless they act as separators between tokens that cannot be directly adjacent, such as keywords and identifiers.

Tokens in an input stream extend as far as possible to the right. That is to say, when a stream is separated into tokens up to a certain point, the next token is the longest string of characters in the stream that constitutes a single token.

3.1.2 COMMENTS

Comments are user source code that is ignored by the compiler, effectively making it whitespace. Comments are any sequence of characters that fall between the set of starting characters /* and ending characters */. This allows for both single and multi-line comments.

3.1.3 IDENTIFIERS

An identifier consists of letters and digits. The alphabet and underscore character qualify as letters. This sequence of letters and digits must begin with a letter character or an underscore to be

recognized as an identifier. Identifiers are case sensitive. In other words, upper and lower cases of the same letter are treated as different identifiers. In the case of SouL, an identifier primarily serves as the name of a variable or a function, be it a primitive type or an object. Every identifier in a SouL program needs to be named uniquely. This applies no matter what block a variable is declared in.

3.1.4 KEYWORDS

Keywords are identifiers that are reserved for use by the compiler to denote types, special values, or the names of built-in functions. Because they are a reserved set of words, keywords may not be used in any other context, such as variable names. SouL uses the following case-sensitive terms as keywords:

```
if              boolean         int             decimal
pitch           velocity        duration        instrument
string          true            false           Note
Chord           Track           Sequence        Midi
play            write           append          transpose
clear           add             getSequence     setInstrument
setTempo        print           while           for
else
```

As a convenience, SouL also provides note durations as pseudo-keywords for defining the properties of note objects. These durations presently include:

```
WHOLE           QUARTER         SIXTEENTH       SIXTYFOURTH
HALF            EIGHTH          THIRTYSECOND
```

3.1.5 CONSTANTS

There are three types of constants defined in SouL: number constants, boolean constants and pitch constants.

3.1.5.1 NUMBER CONSTANTS

Number constants can be separated into two subtypes: integer constants and decimal constants. Integer constants represent whole values, whereas decimal constants represent values with

fractional components. Both integers and decimals support the use of scientific notation, exponents, and negative values. For example, the following two examples show a valid int and double in SouL:

```
int n = -10;
decimal m = 1.27^8.9;
```

The integer constant supported by SouL uses the int keyword. Similar to the `int` type in Java, an `int` in SouL is a signed 32-bit (4-byte) integer ranging from $-(2^{31})$ to $2^{31}-1$. The floating point constant supported by SouL uses the `decimal` keyword that represents a signed 64-bit decimal number, similar to the `double` type in Java.

3.1.5.2 BOOLEAN CONSTANTS

The boolean constants are `true` and `false`. These values represent the output of conditional statements, as well as values that can be assigned to variables of type boolean.

3.1.5.3 PITCH CONSTANTS

In musical terms, a pitch is a value corresponding to a particular sound frequency, usually represented in musical notation by a letter and an optional accidental mark. In MIDI, pitches are valued on a limited range from 0 to 127, with the value 60 being "middle C".

A pitch constant in SouL is represented by an uppercase letter from A to G, followed optionally by an accidental mark, then a single digit integer to indicate the octave of the pitch, all enclosed within a set of single quotes. Accidental marks include sharp (#) and flat (b). For example, `'C4'` indicates the fourth C in the MIDI range (otherwise known as "middle C"), and `'D#5'` indicates the second D sharp above middle C. Certain pitch names can also assume the same value. For example, `'G#3'` and `'Ab3'` are the same pitch in MIDI.

Each pitch constant value corresponds to the MIDI value representing that pitch. As such, SouL can only support the pitch range of standard MIDI, which is 0 to 127, or `'C-1'` to `'G9'`.

3.1.6 STRING LITERALS

A string literal is any sequence of characters (including space characters) set between double quotes. Examples of string literals are items such as "`flying purple people eater`" or "`testfile.mid`". String literals have a limited implementation in SouL, primarily functioning as a part of processes involving File I/O. They are used to represent file paths for reading from and writing out sequence data to MIDI files.

3.2 EXPRESSIONS

An expression is a type of statement that can be evaluated logically, mathematically, comparatively, or through other means. An expression can consist of constants, operators, and identifiers of all type specifications. Because of precedence levels and associativity specifications, expressions can be linked together and still evaluate to the correct result. For example:

$x + y - z - 2$ groups as $x + y - (z) - (2)$, not $x + y - (z - 2)$

...in order to be evaluated. In SouL, an expression can consist of a variety of things, including booleans, variables, integers and strings. Therefore, support is given for performing operations on different combinations of objects, such as comparing an integer and a pitch, or two Sequences. Expressions are the basis for a variety of things, from mathematical calculations to control-flow to conditional statements.

3.2.1 OPERATORS

An operator specifies a mathematical, logical, or comparative operation that is to be performed. SouL supports the use of 24 different operators. Each of these operators are either binary or unary, with each unary operator being further subdivided into postfix and prefix operators.

3.2.1.1 BINARY OPERATIONS

3.2.1.1.1 ADDITION AND SUBTRACTION

SouL supports full binary addition and subtraction. For user convenience, SouL allows addition for both numerical values and string literals. When the binary plus operator is assigned to two string literals, they will be concatenated with the leftmost operand beginning the new string.

Any two numerical values can be added or subtracted - for example, an integer can be added to a decimal to produce a decimal. Although object addition is not supported through the binary plus operator, SouL contains built-in functions that can be used to receive the expected result (see the Inherent Functions section). Binary addition and subtraction has the following syntax, each having left associativity:

```
Expression :
        Expression + Expression
        Expression - Expression
```

3.2.1.1.2 MULTIPLY, DIVIDE AND MOD

The multiplication, division and modulus operations only support the use of real numbers, and cannot be used with objects. The modulus operator returns the remainder after dividing the two expressions. The syntax for using these binary operations is a follows, each having left associativity:

```
Expression :
        Expression * Expression
        Expression / Expression
        Expression % Expression
```

3.2.1.1.3 EXPONENTIATION

The exponentiation operator is supported by using the caret character between two expressions. This operator raises the first expression to the power of the second expression. This operator has right associativity and the highest arithmetical precedence.

```
Expression :
        Expression ^ Expression
```

3.2.1.1.4 COMPARATIVE OPERATORS

Four relational comparative operators are available for use on expressions. These operators return a boolean value of `true` if the comparison is true, and `false` otherwise. Both note constants and numerical values are supported by this. For example, 2 < 3 would return `true`, as would 'C5' < 'G3'. Integer or decimal numbers can also be compared to a pitch constant. For example, the comparative expression 10 < 'D7' would return `true`. The syntax is as follows:

```
Expression :
       Expression  < Expression
       Expression  > Expression
       Expression <= Expression
       Expression >= Expression
```

3.2.1.1.5 EQUALITY OPERATORS

SouL supports equality and non-equality comparisons that are functional for most object and primitive types. The equality operator works in the following way based on the type of its operands:

Two numerical operands are equal if their values are equal.
Two strings are equal if they contain the same sequence of characters.
Two objects are equal if they both contain identical components.

The expression returns either `true` or `false` depending if the expression is true or false, respectively. The syntax for equality expressions is as follows:

```
Expression :
       Expression == Expression
       Expression != Expression
```

3.2.1.1.6 LOGICAL OPERATORS

SouL supports the use of the two logical operators `and` and `or`. If both expressions are true, an `and` returns `true`. If not, `false` will be returned. If one or both expressions are true, an `or` returns `true`, or `false` if otherwise. The syntax for logical operator expressions is as follows:

```
Expression and Expression
Expression or  Expression
```

3.2.1.1.7 ASSIGNMENT OPERATORS

SouL contains two types of assignment operators, each of which assign some modified form of the value on the right to the identifier on the left. The pure assignment operator, =, simply assigns the value on the right to the identifier on the left. The "assign-update" operators are also supported, which assign the identifier a value of the original identifier's value updated by the right operand with the appropriate operator. For example, `x += 3` is the shorthand equivalent of `x = x + 3`. The pure

assignment operator is right-associative, while the assign-update operators are left-associative. The following expressions are supported assignment operator expressions in their correct syntax:

```
Identifier  = Expression
Identifier += Expression
Identifier -= Expression
Identifier *= Expression
Identifier /= Expression
Identifier %= Expression
```

3.2.1.1.8 COMMAS

Commas are used as separators for multiple-parameter lists, such as function parameters, object arguments, or lists of declarations. Expressions separated by commas are evaluated from left-to-right. The following are examples of the syntax usage of commas in SouL:

```
Note n = Note('C4', 98, EIGHTH);
int x, y, z = 3;
```

3.2.1.2 UNARY OPERATIONS

SouL also supports a variety of unary operators. Unary operations are operations performed on one operand only, and are either *prefix* (operator before the operand) or *postfix* (operator after the operand).

3.2.1.2.1 PREFIX EXPRESSIONS

SouL supports the use of five prefix operators, all of which come before a given expression. The unary minus and unary plus operations simply negate or maintain the value of the expression, respectively. The double-plus operator increases the operand by one, while the double-minus operator decreases the operand by one. The final prefix operation involves the negation operator. This is a very versatile operator that can be used on any boolean value to negate it. For example, `!true` equates to `false`. The negation operator can be used on any expression that has a boolean value. Therefore, it can be used in a variety of contexts, such as on an expression or function that returns a boolean value. For example, `!(3 <= 1)` equates to `true`. The syntax for prefix expressions is shown below.

```
PrefixExpression :
       - Expression
       + Expression
```

```
        ! Expression
        ++ Identifier
        -- Identifier
```

3.2.1.2.2 POSTFIX EXPRESSIONS

SouL allows the use of two postfix operations. The same double-plus and double-minus operators that are available for use in prefix expressions are also available in postfix expressions, and perform the same operation. The syntax for postfix expressions is shown below.

```
PostfixExpression :
        IDENTIFIER ++
        IDENTIFIER --
```

3.3 DECLARATIONS

A declaration associates an identifier with a certain type. Declarations can also use an assignment operator to assign a value or action to the identifier, be it a variable or a function definition, but they do not necessarily have to do this. A declaration can take the following forms:

```
Declaration :
        Type DeclaratorList
```

This specifies a type, followed by a list of declarators. Type refers to the type specifier, or the form that the data will take, such as an int or boolean. The declarator list can contain a single or multiple declarations or assignments, separated by commas.

```
DeclaratorList :
        Declarator
        DeclaratorList , Declarator

Declarator :
        IDENTIFIER
        IDENTIFIER = Expression
```

3.3.1 DECLARATORS

A declaration is defined by a type specifier followed by a declarator. The type specifier indicates the data type that the declarator will represent its value. A declarator, in turn, can be either an identifier or an assignment to an expression, as discussed in the expressions section.

```
Declarator :
        IDENTIFIER
        IDENTIFIER '=' Expression
```

This allows for a declaration to be composed of a type specifier followed by a variable, or a type specifier followed by an assignment operation. These are both shown in examples below.

```
duration d;
velocity v = 30;
boolean b = !(v > 50);
```

3.3.2 TYPE DECLARATIONS

Primitive types, such as int, decimal, boolean, and duration, can be declared or instantiated using a list of declarators. This allows for declarations and assignments such as the following examples:

```
int x, y, z = 3, n;
boolean t = true, f = false;
decimal d;
```

In the last example, the identifier can later be given a value by using an assignment operator, as explained previously:

```
d = 1.34;
```

3.3.3 OBJECT DECLARATIONS

SouL has five object types: Note, Chord, Sequence, Track, and Midi. Because each of these has their own specifications, they cannot be declared in the same manner as primitive types. These objects are the same as primitives, however, in that they can be declared without being initialized and in lists, as shown below. It cannot, however, be initialized if it is part of a list.

```
Midi m, n;
Sequence s;
```

When being assigned to a value, each object requires different parameters. A Note object requires a pitch, velocity, and duration. A Chord object requires a list of pitches, a velocity, and a duration. A Sequence object requires no arguments, and begins empty. A Track object requires no arguments, and begins empty. A Midi object takes a MIDI file name as a parameter. An example of a full declaration and assignment for each object type is shown below.

```
Note n = Note('C5', 100, QUARTER);
Chord c = Chord(('C5', 'G3', 'D6'), 48, EIGHTH);
Midi m = Midi("myMidiFile.mid");
Track t = Track();
Sequence s1 = Sequence();
Sequence s2 = m.getSequence();
```

3.3.4 TYPE AND OBJECT SPECIFIERS

SouL allows for the following types and objects in declarations. They are all part of the Type production in the grammar:

```
Type :
       INT
       DECIMAL
       STRING
       PITCH
       VELOCITY
       DURATION
       INSTRUMENT
       BOOLEAN
       NOTE
       CHORD
       TRACK
       SEQUENCE
       MIDI
```

In any given declaration, at most one type can be specified. Omitting type specifiers is presently not permitted in SouL.

3.3.5 INITIALIZATION

In a declaration, the declarator can contain an assignment statement, as discussed in section 4.1. An assignment for a variable stores the value of an expression into the memory location represented by the variable's name using the assignment operator, =. Initializing type identifiers requires either other identifiers of the same type, or expressions that evaluate to a value of the proper type for the declaration. Initializing object identifiers requires either another object of the same type, or the creation of a new object via the object's name followed by any required parameters in parentheses. Examples of type and object initialization can be found in the sections describing their declarations, 4.2 and 4.3, respectively. Initialization of objects can also be used to pass anonymous parameters to

built-in functions. For example, the following function-call uses a Note initialization as an anonymous parameter.

```
print(Note('C4', 80, WHOLE));
```

3.4 STATEMENTS

A statement describes commands to be executed. They are executed primarily for a specific action they produce.

```
FullStatement :
        Statement ;
        IfStatement
        CompoundStatement
        WhileStatement
        ForStatement
```

There are three main types of statements in SouL. *Expression Statements* are those that can go to a specific expression or function call, and are followed by a semicolon. *Multi-line Statements* contain all multi-line blocks of code, including control flow statements.. Both of these are expanded upon below.

3.4.1 EXPRESSION STATEMENTS

Most statements can be categorized as expression statements. Expression statements are ones that can go to an expression or end in a semicolon, as follows:

```
Statement :
        PlayStatement
        DeclarationInitialization
        Declaration
        WriteStatement
        ClearStatement
        AppendStatement
        Addition
        PrintStatement
        GetSequence
        Assignment
        Expression
        Transpose
        SetInstrument
        SetTempo
```

Most of these statements are either expressions, declarations, or function calls. All of these statements are required to end in a semicolon.

3.4.1.1 BUILT-IN FUNCTIONS

SouL contains ten built-in functions that allow the user to manipulate Midi, Sequence, Note, Track and Chord objects, as well as perform other simple tasks.

3.4.1.1.1 PLAY STATEMENT

The `play` function is built-in so Midi, Note, Chord, Track, or Sequence objects can easily be played with SouL. To play an object, the desired object (either a declared variable or anonymously instantiated) is passed to the `play` function as an argument, as follows:

```
play(x);
play(Note('C4', 80, WHOLE));
```

3.4.1.1.2 PRINT STATEMENT

The `print` function allows information to be displayed on the console. The function accepts as input integer or decimal numbers, booleans, and string literals, amongst all other objects and primitives. An example of using the `print` function is the following code:

```
print("hello, world");
print(x);
```

3.4.1.1.3 WRITE STATEMENT

The `write` function acts on a Midi object and overwrites the object's associated file with a given combination of Note, Chord, Sequence, or Track objects. For a Midi object with identifier *x*, this is done as follows:

```
x.write(y);
x.write(Note('C4', 80, WHOLE));
```

3.4.1.1.4 APPEND STATEMENT

The `append` function is similar to the `write` function in that it alters a Midi object's file, but it appends the given argument to the end of the file, rather than overwriting it. The syntax is analogous to that of the `write` function, and is shown below. Only Sequence objects can be appended to Midi

objects. Because a sequence initialization creates an empty sequence, content must be added to a sequence before it can be appended to a file. For a Sequence object with identifier *s*:

```
x.append(s);
```

3.4.1.1.5 CLEAR STATEMENT

The `clear` function is used on a Midi, Sequence, or Track object to clear it of all associated data. It takes no parameters, and leaves the object's file or object blank. The syntax is shown below.

```
x.clear();
```

3.4.1.1.6 SEQUENCE AND TRACK ADDITION STATEMENT

Sequence and Track addition is supported by the add function. add is performed on a given Sequence to add a Track object to it. add can also be performed on a given Track to add a Note, Chord or Track to it. By default, this adds the given object to the end of the Sequence. Below are three examples of how to use the function. Variable *s* is a Sequence, *t* is a Track, and *c* is a Chord.

```
t.add(c);
t.add(Note('C4', 80, WHOLE));
s.add(t);
```

3.4.1.1.7 GET SEQUENCE STATEMENT

SouL offers a `getsequence` function that allows you to assign a Sequence object a sequence of Notes and Chords from the Midi object the function is acting on. For example, in the code below, the identifier *s* is assigned a new Sequence which consists of every note from the Midi file *m*.

```
Sequence s = m.getSequence();
```

3.4.1.1.8 SET INSTRUMENT STATEMENT

SouL allows its users to set the instrument of a track. This is done by supplying the number, from 0 to 127, of the instrument to the `setInstrument` function. This can only be done on a Track object.

```
Track t = Track();
```

```
t.setInstrument(80);
```

3.4.1.1.9 SET TEMPO STATEMENT

SouL also allows its users to set the tempo of a Sequence by supplying a tempo as a parameter to the `setTempo` function. The tempo can be any positive integer or decimal value. An example of this is shown below.

```
Sequence s = Sequence();
s.setTempo(120.5);
```

3.4.2 COMPOUND STATEMENTS

Compound statements refer to productions that allow for multiple statements to be executed where one would normally be expected. This takes the form of the following productions, where each has a use in different locations:

```
StatementBlock :
        StatementBlock FullStatement
        empty

CompoundStatement :
        { CompoundStatementBlock }

CompoundStatementBlock :
        FullStatement CompoundStatementBlock
        empty
```

StatementBlock is reserved as being used only for the start-production of the language, while CompoundStatement and CompoundStatementBlock are used within control-flow statements.

3.4.3 SELECTION STATEMENTS

Selection statements are used for control flow. The productions used for this are as follows:

```
IfStatement :
        IF ( Expression ) FullStatement ELSE FullStatement
        IF ( Expression ) FullStatement
```

In all forms that have the `if` statement, the expression *Expression* is evaluated. If it evaluates to `true`, then the statement following it is executed. If it evaluates to `false`, then SouL looks at the following `else` (if there is one) and executes anything after that. Note that this can be another if statement, creating an "`else if`" functionality.

3.4.4 ITERATION STATEMENTS

Iteration statements are used for loops.

```
FullStatement:
        WhileStatement
        ForStatement

WhileStatement:
        while ( Expression ) FullStatement

ForStatement:
        for ( OptStatement ; OptExpression ; OptStatement ) FullStatement
```

In the construction for `while`, the *Statement* will continue to execute as long as the *Expression* within parentheses is `true`. `for` statements continue to execute as long as *OptExpression* is `true`. If *OptExpression* evaluates to empty, the loop runs infinitely. OptStatement and OptExpression can both be empty.

3.5 SCOPE

Because SouL does not support the linking and inclusion of different files, it does not concern itself with external variables that can span across multiple files. SouL is thus primarily concerned with lexical scope only, which defines where in a single program a certain variable can be accessed.

The lexical scope of an identifier within the body of the program begins at the end of its declarator and persists until the end of the program. The scope of an identifier within a block persists from the end of its declarator until the end of the innermost block it appears in.

Because SouL does not support the creation of user-defined functions, it does not concern itself with scoping due to function declarations or with namespaces.

3.6 FULL GRAMMAR

The following operators have left associativity:

```
+       -       *       /       %
++      --      ==      !=      <
>       <=      >=      or      and
+=      -=      *=      /=      %=
```

The following four operators have right associativity:

```
^       =       !       -  (unary minus)
```

The following are terminals and do not contain any productions:

```
BOOLEAN, TRUE, FALSE, INT, DECIMAL, PITCH, VELOCITY, DURATION, INSTRUMENT, STRING,
NOTE, CHORD, TRACK, SEQUENCE, MIDI, FILENAME, IDENTIFIER, NOTENAME, NUMBER_INT,
NUMBER_DECIMAL, STRINGLITERAL, '.', ',', '(', ')', '{', '}', ';', '+', '-', '*', '/',
'%', '^', '=', '!', PLAY, WRITE, APPEND, TRANSPOSE, CLEAR, ADD, GETSEQUENCE,
SETINSTRUMENT, SETTEMPO, PRINT, WHOLE, HALF, QUARTER, EIGHTH, SIXTEENTH, THIRTYSECOND,
SIXTYFOURTH, PLUSEQ, MINUSEQ, MULTEQ, DIVEQ, MODEQ, EXPEQ, PLUSPLUS, MINUSMINUS, EQ,
NOTEQ, LTEQ, GTEQ, OR, AND, WHILE, FOR, IF, ELSE, LOWER_THAN_ELSE
```

```
StatementBlock :
       StatementBlock FullStatement
       empty


CompoundStatement :
       '{' CompoundStatementBlock '}'


CompoundStatementBlock :
       FullStatement CompoundStatementBlock
       empty


FullStatement :
       Statement ';'
       IfStatement
       CompoundStatement
       WhileStatement
       ForStatement
```

```
Statement :
        PlayStatement
        DeclarationInitialization
        Declaration
        WriteStatement
        ClearStatement
        AppendStatement
        Addition
        PrintStatement
        GetSequence
        Assignment
        Expression
        Transpose
        SetInstrument
        SetTempo


Expression :
        '(' Expression ')'
        Expression '+' Expression
        Expression '-' Expression
        Expression '*' Expression
        Expression '/' Expression
        Expression '%' Expression
        Expression '^' Expression
        Expression AND Expression
        Expression OR Expression
        Expression '<' Expression
        Expression '>' Expression
        Expression LTEQ Expression
        Expression GTEQ Expression
        Expression EQ Expression
        Expression NOTEQ Expression
        PrefixExpression
        PostfixExpression
        Duration
        Pitch
        NUMBER_INT
        NUMBER_DECIMAL
        IDENTIFIER
        TRUE
        FALSE
        STRINGLITERAL


PostfixExpression :
        IDENTIFIER PLUSPLUS
        IDENTIFIER MINUSMINUS


PrefixExpression :
        '-' Expression
        '+' Expression
        '!' Expression
        PLUSPLUS IDENTIFIER
        MINUSMINUS IDENTIFIER
```

```
IfStatement :
        IF '(' Expression ')' FullStatement ELSE FullStatement
        IF '(' Expression ')' FullStatement


WhileStatement :
        WHILE '(' Expression ')' FullStatement


ForStatement :
        FOR '(' OptStatement ';' OptExpression ';' OptStatement ')' FullStatement


OptExpression :
        Expression
        empty


OptStatement :
        Statement
        empty


PlayStatement :
        PLAY '(' Initialization ')'
        PLAY '(' FILENAME ')'
        PLAY '(' IDENTIFIER ')'


Initialization :
        '(' Initialization ')'
        MIDI '(' FILENAME ')'
        NOTE '(' Expression ',' Expression ',' Expression ')'
        CHORD '(' '(' PitchList ')' ',' Expression ',' Expression ')'
        SEQUENCE '(' ')'
        GetSequence
        TRACK '(' ')'


PitchList :
        NOTENAME ',' PitchList
        NOTENAME


Pitch :
        NOTENAME


Duration :
        WHOLE
        HALF
        QUARTER
        EIGHTH
        SIXTEENTH
```

```
        THIRTYSECOND
        SIXTYFOURTH


DeclarationInitialization :
        Type IDENTIFIER '=' Initialization


Declaration :
        Type DeclaratorList


DeclaratorList :
        Declarator
        DeclaratorList ',' Declarator


Declarator :
        IDENTIFIER
        IDENTIFIER '=' Expression


Type :
        INT
        DECIMAL
        STRING
        PITCH
        VELOCITY
        DURATION
        INSTRUMENT
        BOOLEAN
        NOTE
        CHORD
        TRACK
        SEQUENCE
        MIDI


WriteStatement :
        IDENTIFIER '.' WRITE '(' IDENTIFIER ')'
        IDENTIFIER '.' WRITE '(' Initialization ')'


AppendStatement :
        IDENTIFIER '.' APPEND '(' IDENTIFIER ')'
        IDENTIFIER '.' APPEND '(' Initialization ')'


ClearStatement :
        IDENTIFIER '.' CLEAR '(' ')'

Addition :
        IDENTIFIER '.' ADD '(' IDENTIFIER ')'
        IDENTIFIER '.' ADD '(' Initialization ')'
```

```
PrintStatement :
        PRINT '(' Expression ')'
        PRINT '(' Initialization ')'


GetSequence :
        IDENTIFIER '.' GETSEQUENCE '(' ')'
        Initialization '.' GETSEQUENCE '(' ')'


Assignment :
        IDENTIFIER '=' Expression
        IDENTIFIER AssignUpdate Expression
        IDENTIFIER '=' Initialization


AssignUpdate :
        PLUSEQ
        MINUSEQ
        MULTEQ
        DIVEQ
        MODEQ
        EXPEQ


Transpose :
        IDENTIFIER '.' TRANSPOSE '(' Expression ')'


SetInstrument :
        IDENTIFIER '.' SETINSTRUMENT '(' Expression ')'


SetTempo :
        IDENTIFIER '.' SETTEMPO '(' Expression ')'
```

3.7 *SouL* STYLE GUIDE

While SouL is not a particularly complicated language, there are a few conventions that we followed as a team that would be generally good practice. The following is a simple style guide for SouL itself.

File Names

- All SouL source file names should end with **.soul** in order to be recognized by the compiler.
- Make sure file names are concise but informative, and easy to read. For example, instead of naming a file **x.soul**, use something like **arpeggioCmaj.soul**.

Indentation

- Try not to make any given line in a program too long, as it makes files harder to read and edit.
- Break very long expressions into multiple lines by breaking before an operator and indenting the next line to show visually that is is part of the same expression.
- Indent the bodies of if-statements, while-statements, and for-statements.

Comments

- Use `/* … */` to indicate a comment in SouL. Comments are ignored by the compiler. Use comments as a way to annotate your code and provide meaningful information about the code that is not immediately obvious.

```
/* here is a single line comment */
/*
here is a
multi-line comment
*/
```

Declarations

- Although variables of the same type can be declared in a single line (e.g. `int x, y, z;`) , it is preferable to perform one declaration per line for clarity, regardless of types:

```
int x;
int y;
Note n;
```

- As often as possible, try to initialize a variable in the same line that it is declared. The only reason not to do this would be if its initial value depends on a later computation. Example:

```
Note n = Note('D5', 100, HALF);
```

Statements

- For simple statements, try to limit each line to one statement. For example:

```
x += 2;         /* good */
x += 4; y--;    /* BAD! */
```

- A compound statement contains a list of statements inside curly braces, i.e. "{ statements }". The opening brace should be at the end of the first line of the compound statement and the

closing brace should be at the beginning of the last line and be lined up with the beginning of the compound statement. The statements between should be indented. Examples follow.

- Following from compound statements, `if`, `if/else`, and `if/else if/else` statements should use the following form to minimize ambiguities:

```
 if (condition) {
   statements;
}

if (condition) {
   statements;
} else {
   statements;
}

if (condition) {
   statements;
} else if (condition) {
   statements;
} else {
   statements;
}
```

- Additionally, `while` and `for` statements should have the following form:

```
while (condition) {
   statements;
}

for (initialization; condition; update) {
   statements;
}
```

White Space

- Make use of blank lines to set apart sections of code that are logically related.
- Use blank spaces in the following ways:

  Put a space between a keyword and set of parentheses to distinguish it from a function call:

```
if (true) {
   . . .
}
play(x);
```

  Put spaces between commas in argument lists, and between expressions in a `for` statement:

```
Chord c = Chord(('B3', 'B4', 'B5'), 85, QUARTER);
for (expression1; expression2; expression3) . . .
```

  Separate binary operators with spaces. Do NOT separate unary operators. Example:

```
a = b + c;
a++;
```

Variable Names

- All variables names should begin with a lower case letter. If a variable name contains multiple words, the first letter of each new word should begin with a capital letter:

```
int counter;
string nameOfFile;
```

- Try to keep variable names short but meaningful. The name itself should ideally indicate its intent in the program. Try to only use single-character names as throwaways, such as temporary variables or loop counters.

General Programming Practices

- When possible, try to avoid using "magic numbers". In other words, don't hardcode any numerical constants that could be considered a parameter in the program or may be used multiple times.
- When dealing with more complicated expressions, make liberal use of parentheses. This will help clear up any operator precedence confusion, as well as make the code easier to look at:

```
if (a == b and c == d)       /* not so good */
if ((a == b) and (c == d))   /* good */
```

4. PROJECT PLAN

Written by Andrew Goldin (Project Manager)

4.1 OVERALL PROCESS

In our first couple of meetings we were able to introduce ourselves, assign team roles, and throw around a few language ideas. We quickly narrowed our list down to a music programming language, which eventually came to be known as SouL. We created a Facebook group and a WhatsApp chat for our primary means of non-physical communication. We then set up a rudimentary meeting schedule using when2meet.com, and made a point to meet a minimum of once per week for a few hours. Of course, there were a few weeks in which we had two or more meetings, especially when approaching deadlines for project deliverables. Additionally, we kept a meetings log in which we would record our main accomplishments for each meeting as well as a list of things to work on for the next one.

For this project, our team utilized an iterative and incremental development process. Most of our initial planning work was done as a group, but as development became increasingly modular, we were able to break up the busywork on an individual basis, while utilizing group time to work on design and testing. Generally speaking, we followed a cycle of planning, implementation, testing, evaluation, repeat. We started with an overall goal, creating a full grammar and some test programs to go with it. We then implemented a very small portion of the translator in order to figure out how to get simple programs to compile, and get them up and running quickly. We were able to use this knowledge to continue implementation in small portions, which allowed us to catch and treat errors quickly and before they became disastrous.

4.2 TEAM ROLES AND RESPONSIBILITIES

The team roles are as follows:

- Project Manager - Andrew Goldin
- System Architect - Cindy Long
- System Integrator - Matt Kim
- Language Guru - Kevin Walters

Even though these are the official team roles, they are not distinct, as every team member spent some amount of time working on various parts of the compiler. Kevin and Matt worked together to implement the lexer and parser, Andrew wrote the entire Java API (called JSouL) to perform all of the

required MIDI operations, Cindy and Kevin worked together to implement the abstract syntax tree walker to convert SouL code to JSouL code, and due to the absence of a system tester, Matt wrote the entire test suite and everyone helped out with testing.

4.3 IMPLEMENTATION STYLE GUIDE

For the JFlex lexer file, our team used a very simple style for formatting. Each line in the "transition rules" section is of the form...

```
pattern    { action }
```

...where the pattern is a regular expression for a specific kind of token in our language and the action states what token is to be returned to the parser, if any (for example, white space and comment patterns are ignored).

For the BYacc/J grammar file, our team followed many of the stylistic suggestions of Stephen C. Johnson in his Yacc paper, here: http://www.cs.utexas.edu/users/novak/yaccpaper.htm. In the "declarations" section, lists of token declarations are grouped by their similarity in function. Each grammar production in the "transition rules" section is of the form...

```
Nonterminal :
        Item1           { parse action }
        | Item2         { parse action }
        | Item3         { parse action }
        ;
```

...where the first line is the name of a nonterminal in the grammar, and items 1, 2, and 3 are any sentential form of terminals and nonterminals. Each parse action creates an AST node containing the relevant information from the production.

For the Java back-end, we took style cues from Oracle's style guide, which can be found here: http://www.oracle.com/technetwork/java/javase/documentation/codeconvtoc-136057.html. All of the Java code was written using Eclipse, and so mostly uses styles of formatting suggested by Eclipse. Here is an example of a method in JSouL that uses the suggested style:

```
private MetaMessage sequenceTempo() {
      // convert tempo to a byte array to be recognized by Midi system
      String hexString = Integer.toHexString(60000000 / (int) tempo);
      if (hexString.length() % 2 != 0) {
            hexString = "0" + hexString;
      }
```

```
        byte[] data = new java.math.BigInteger(hexString, 16).toByteArray();

        // create and return message with tempo change
        MetaMessage m = new MetaMessage();
        try {
              // tempo change MetaMessage has command 81
              m.setMessage(81, data, data.length);
        } catch (InvalidMidiDataException e) {
              System.err.println("Error: (Sequence) cannot set tempo for file");
        }
        return m;
    }
```

4.4 PROJECT TIMELINE

The following is a timeline of dates of major completion points in project development:

2/7	Team formed, introductions, initial brainstorming
2/16	Roles decided, language type determined (music language)
2/26	Language white paper completed and submitted
3/11	First draft of grammar completed, sample programs written
3/24	Created first parts of Java back end
3/26	Language tutorial and reference manual completed and submitted
4/21	Hello World program is parsed and executed properly, Java back-end completed
5/10	Full grammar implemented with type checking, test suite fully written
5/13	Final report completed and submitted
5/16	Final presentation

4.5 MEETINGS LOG

The following is an informal log we kept of things we discussed and worked on at each meeting. It does not take into account the work done individually.

Meeting 1 - 2/7/2013

- Introductions, some ideas, nothing concrete

Meeting 2 - 2/16/2013

- Team roles:
    - Project Manager: Andrew
    - Language Guru: Kevin
    - System Architect: Matt
    - System Integrator: Cindy

- Decent ideas: image editor, audio/midi editor, etc

Meeting 3 - Aho Meeting 1 - 2/20/2013

- Most likely language: language for sound or MIDI data
- Suggested language names:
  - **S**ound **L**anguage → **SL**
  - **Sou**nd **L**anguage → **SouL**
  - **M**usic **L**angage → **ML**
  - **Mu**sic **L**anguag**e** → **MuLe**
- Primary IDE: Eclipse?
- Most likely runtime system: Mac OS X
  - OS X has leading MIDI sequencing and playback software
  - Possibly try to develop for optimal use with OS X first
- Typical users:
  - Hobbyist/amateur musicians
  - Audio engineers
  - Composers
  - Music Teachers
  - etc.
- Typical "Hello world" style program:

```
main {
        create and open new MIDI file M
        M.addNote(60, 32, 5000) // add middle C with velocity 32 for 5 seconds
        close M
}
```

- More complicated operations
  - Generate patterns with minimal effort
  - Scales/loops/
  - Read-in/manipulate existing files & data
  - MAYBE try to work with real audio samples...?
- Language type:
  - Functional or imperative?
  - Modular/Object-oriented?
- Primitive data types?
  - No idea
- Tips for white paper
  - Descriptors/buzzwords should describe eventual goals
  - Back up descriptors with how you plan to approach each goal
  - Keep in mind that particulars on the specification are subject to change

Meeting 4 - 2/23/2013

- Language name: SouL
- Whitepaper Ideas
  - Introduction - Cindy
    - Names, super-high level goals, purpose
    - What, why

- Design Goals - Matt
    - Easy for musicians to use
    - Create MIDI files
    - Edit MIDI files
    - Play sounds
    - Make programs to do so
- Simple - Andrew
    - Primitive data types - how they are intuitive and correspond to music terminology
- Object-oriented - Andrew
    - Note
    - Chord
    - Sequence
- Secure - Cindy
    - Java is being used (garbage collection)
    - Sound files are internal to compiler
- Portable, architecture neutral (possibly?) - Kevin
    - Use architecture of system for instruments
    - No matter what OS you have, you can play the same sounds
- Translated and Compiled - Kevin
    - Translated into Java
    - Compiled by Java compiler
- Static - Kevin?
    - Typing of things is static
    - Nothing changes at runtime

Meeting 5 - 2/26/2013

- Finish writing/formatting white paper

Meeting 6 - 3/9/2013

- Began writing grammar

Meeting 7 - 3/11/2013

- Meeting with Prof. Aho
- Perhaps each write sample programs and compare?
- Why use language?
    - Model pieces to learn
    - Make a tuner
    - Practice parts with other parts
    - Music minus one - Mask out
    - MIDI to sheet music - software (Finale, Sibelius) - to go from sheet music to MIDi
- PIece - Twinkle, Twinkle, Little Star
    - Tutorial - Driving Examples -> File generation, file editing, audio playback
    - File generation -> generate MIDI file for song
    - File editing -> Speed up, transpose song
    - Audio playback -> Play song
    - Language Reference Manual - needs syntax
    - Advice - get something rudimentary up and running

- Two weeks (3/25) - next meeting with Aho

Meeting 8 - 3/24/13

- MIDI Player: http://web.nmsu.edu/~tintaton/cs/music_player/MidiPlayer.java
- Playing, Recording and Editing Sequences : http://download.java.net/jdk8/docs/technotes/guides/sound/programmer_guide/chapter11.html
- Created java files to write, edit, and play files

Meeting 9 - 3/25/13

- More grammar/parser work

Meeting 10 - 3/26/13

- Things to fix later:
    - Argument lists in grammar
    - AssignmentLists in grammar, possibly contain object declarations
    - Unify declarations and assignments in the grammar, then update the manual to reflect the changes (in the TYPES section)
        - declaration: type declarator-list
        - assignment can include object initialization
    - Make a separate section in the manual on function definitions
- Things to implement later:
    - Consolidate the grammar
    - Notes: compare by velocity or duration, not just pitch

Meeting 11 - 4/07/13

- Started implementing the translator
- Makefile is bug-free
- Can't debug to play chords
- Grammar still needs to be edited

Meeting 12 - 4/14/13

- Still working on grammar
- Added files: ConstructTree.java, Node.java, PrintNode.java, SoulLexer.lex, SoulTestLexer.lex
- Can play chords now

Meeting 13 - 4/21/2013

- JSouL completed
- Hello World program works (play single random note)
- Can play from file now

Meeting 14 - 4/25/13

- More translator work, nothing significant

Meeting 15 - 4/28/13

- Creating more node classes for each grammar production
- Setting up lexer in JFlex
- Optimizing jsoul

Meeting 16 - 5/2/2013

- Enclosing initializations in parentheses
- Play statement for an anonymous Note
- Play statement for an anonymous Midi
- Play statement for a filename

- Play statement for an anonymous chord with an arbitrarily long list
- Statement blocks supported
- Statement blocks enclosed in curled braces
- Planning: Want to make SouL be installed so it can be used as a command

Meeting 17 - 5/4/2013

- Declarations with and without initializations
- Comments added to the language with the form /*...*/
- Lists of declarations (i.e. int x,y,z;)
- Declaring a new Note, Chord and Midi and playing them through calls on their identifiers
- Applying clear method to a Midi object
- Applying write method to a Midi object
- Applying append method to a Midi object
- Planning:
    - Make a symbol table, associating an identifier with a given type
    - Make a "clear" method for the Midi jsoul class
    - Make Sequence Initialization
    - Arithmetic/boolean expressions
    - Control flow

Meeting 18 - 5/7/2013

- Print statements working
- Clear statements working
- Sequence declarations
- Sequence additions
- Limited Expressions:
    - Adding, subtracting, multiplication, division, modulo, exponentiation
    - Grouping of terms
    - Correct translation exponentiation to Math.pow (and type casting if necessary)

  i.e.
- Assignments with expressions
- Planning: Assignment -> ID ASSIGN Expression | ID ASSIGN Initialization
    - MAKE A SYMBOL TABLE (ADD IT INTO EXPRESSIONNODE) - a class
    - ADD BOOLEANS ETC. INTO "TYPE"
- Next Meeting: Conditional expressions, logical expressions, postfix/prefix expressions, equality expression
    - Control Flow
    - Type checking (symbol table)

Meeting 19 - 5/8/2013

- Logical expressions (and, or) and booleans (true, false)
- Relational expressions (<, >, <=, >=)
- Equality expressions (==, !=)
- Postfix expressions (x++, x--)
- Prefix expressions (++x, --x, !x, -x, +x)

Meeting 20 - 5/9/2013-5/10/2013

- Grammar functionality finished
    - a bit different from original grammar design

- If, While, and For statements added
- Transpose added
- Support of Tracks implemented
- Type and error checking began
- Add for sequence and tracks

Meeting 21 - 5/11/2013

- Ordered a pizza
- All parse actions completed
- Test suite fully functional
    - All test programs written and passed
- Final report almost complete

Meeting 22 - 5/12/2013

- Finalize report for submission
- Pass out from exhaustion

5. LANGUAGE EVOLUTION

Written by Kevin Walters (Language Guru)

SouL has remained reasonably consistent in incorporating its original goals and expected implementation into its final product. The impetus for creating SouL was to provide a language centered around MIDI files that is useful as well as easy to use and understand. While MIDI creating and editing is supported through languages like Java, the SouL team felt that drastic simplification could take place. By allowing the same functionality to be achieved with a much simpler, more intuitive syntax, a wider audience could be reached, allowing people with little programming knowledge to successfully create and edit MIDI files. In this way our target users, musical composers, teachers, and students, were able to be reached.

The language and syntax of SouL was implemented in order to keep this idea of simplicity and ease of use intact. Something that was proposed early in the process of the language development was the use of special functions provided by SouL to perform specific actions for the user. These functions are the key to SouL's compact syntax. For example, suppose you are a musical programmer and have a MIDI file that you want to edit. In Java, doing this may require 70 lines of code - 5 lines to create an object to store this file, and 65 lines to store this into a Sequence, with can then be edited by transposing it, adding more notes, etc. In SouL, this same action could be done in only two lines of code:

```
Midi m = Midi("filename.mid");
Sequence s = m.getSequence();
```

The language was developed with these functions as its basis. Despite having a complete grammar for our language at the time the Language Reference Manual was submitted, clear changes needed to be made to ensure SouL would have as little errors, ambiguity and complexity as possible. To do this, our grammar was implemented in steps, with each step being tested by a variety of syntaxes to ensure it functions as it is expected to. The first functionality that was supported by SouL was the Hello World program - playing an anonymous note. This was then tested by ensuring the note could take appropriate parameters. The grammar was then expanded by allowing Chords, Sequences, and MIDI files to be played, which were also tested similarly. In the same manner the grammar continued to expand on itself, until a complete context-free grammar specifying the SouL syntax had been created.

Because SouL's expected user does not necessarily have a background in programming, we decided to allow some user freedom where it would be useful, and restrictions elsewhere. Early proposals for

SouL included allowing user freedom in passed parameters for initializations and built-in functions. This freedom received high priority when implementing our grammar, because of the benefits it would provide the user. For example, a standard print function can print things ranging from numbers, to strings, user-generated objects such as Notes. This provides flexibility to the user, letting them do as much as possible without changing syntax. This can also be seen in initializing a new Note , which has parameters that can accept numbers (i.e. `1` to `127`), expression (i.e. `i+3`) note names (i.e. `'C4'`), or duration names (i.e. `WHOLE`).

SouL's language has been changed slightly since previous versions, either to add more functionality or to help guide the user. The largest change is that the completed version of SouL includes the added object Track. The addition of Tracks allows a Sequence to contain multiple Tracks, which in turn contains multiple notes or chords. The addition of Tracks also allowed the the built-in functions `setInstrument` and `setTempo`, which allow you to set a specific instrument or tempo to a track. These additions allow for a SouL user to generate and edit a complete MIDI file, with all the "bells and whistles". We did, however, strive to keep the language as close to that in the Language Reference Manual as possible. A huge help in maintaining this consistency was creating the backend Java functions for each built-in SouL function very early. Because the backend was completed early in the process, it was relatively simple to add into the language everything that was included in the reference manual.

As SouL's developers, we made the decision to translate into Java. While C was considered so that SouL could be as fast as possible, Java's ease of use and built-in MIDI packages made it a much more reasonable choice. After much consideration, we determined that because SouL is a musical editing language, there are very little instances where a program would be computationally intensive, and therefore trading in speed for pre-made functionality was a reasonable sacrifice. Because of this minimal need for computing, SouL only incorporates numerical values of int and double as the basic numerical data types.

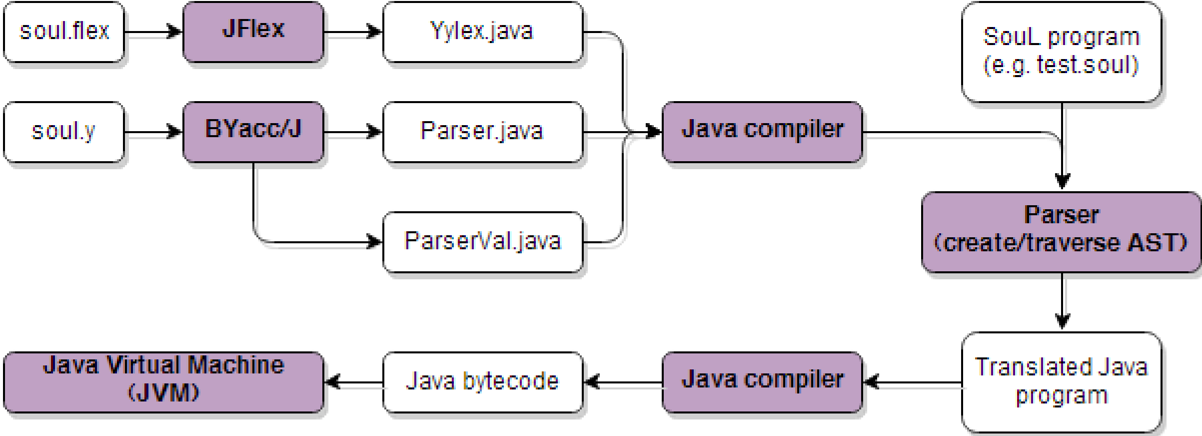
The compiler front-end for SouL consists of a lexical analyzer and a parser. The Lexer separates the input into a token stream, and the Parser uses context-free grammar productions to specify the syntax of the language. The programs used to compile the Lexer and Parser are JFlex and BYacc/J, respectively. JFlex is a lexical analyzer written in and designed for Java use. Similarly, BYacc/J is a version of YACC that allows for Java implementation in the syntax-directed actions. The use of these versions of the lexical analyzer and parser allowed for the simple creation and use of Node objects, which were used to create the Abstract Syntax Tree. A tree-walk was then performed on the AST, which produced the appropriate Java output code.

No unusual external libraries were used to implement SouL. The backend for our compiler is completely proprietary, written using Java version 1.6.0_45. Although it was initially used for its extra functionality, Java version 1.7 was not implemented for the final version of SouL for compatibility reasons. The only Java package that had to be imported to complete the backend of the compiler was `javax.sound.Midi.*`, which contains all important Java MIDI libraries.

6. TRANSLATOR ARCHITECTURE

Written by Cindy Long (System Architect)

6.1 BLOCK DIAGRAM AND MODULE DESCRIPTION



Block diagram of the SouL translator and compiler

Since SouL translates into Java, JFlex and BYacc/J are used to generate the parser for SouL. The block diagram above shows that JFlex takes in as input the lex file, soul.flex, and outputs the Java file Yylex.java. BYacc/J takes soul.y as input and outputs Parser.java and ParserVal.java. Then, the Java compiler takes in all three of these output files and creates the primary parser that translates SouL source programs into Java programs. With a .soul file as input, it creates an abstract syntax tree (AST) using Node classes written in Java for each grammar production. It then traverses the AST to generate the corresponding .java file. These .java files are subsequently compiled by the Java compiler to produce the corresponding .class files which finally are interpreted by the Java Virtual Machine.

6.2 WHO WROTE EACH MODULE?

Despite each member having an “official” role, there was a great deal of inter-collaboration on the modules of the compiler. Kevin and Matt wrote the majority of the lexer source (soul.flex), the parser source (soul.y), and the semantic parse actions. Cindy, Matt and Kevin contributed the majority to the creation of all of the relevant Node classes for each production in the grammar, while Andrew handled the Java back end in order to get the translated Java programs to compile and execute correctly.

7. DEVELOPMENT AND RUNTIME ENVIRONMENT

Written by Matt Kim (System Integrator)

The first decision made by the group was to make the target language Java, for two reasons. Firstly, all of the members of the group were familiar with Java, so there was no need to learn the syntax of a new language. Secondly, Java was chosen over C because of the handling of memory errors.

Originally, we began to implement the syntax-directed definitions using lex and yacc. However, we ran into difficulty in expressing the abstract syntax tree produced by yacc in a meaningful intermediate form to be parsed and recreated in Java. Hence, we began to look for alternatives to lex and yacc that could be implemented in Java so the objects did not have to be translated from C. We eventually settled on JFlex and Byacc/J. JFlex is an implementation of lex in Java. The tokens are similarly defined as they are in lex, but the rules inside each token are written using Java code. Running jflex on a .flex file creates a Yylex.java file, which is used as a lexer. Byacc/J is an implementation of yacc for Java. The layout of the file is essentially the same, with the rules inside each of the productions as well as the code on the bottom and top written in Java instead of C. When running BYacc/J, a file named Parser.java is created which serves as the Parser, and then once compiled and run, can act as a parser. Furthermore, a ParserVal class was also created which can store primitives and arbitrary objects within nodes. JFlex is invoked with the command:

```
jflex filename.flex
```

BYacc/J is invoked in the same way as yacc but with a `-J` flag, as below:

```
yacc -J filename.y
```

JFlex can be downloaded at http://jflex.de/download.html and Byacc/J can be downloaded at http://byaccj.sourceforge.net/#download. Once downloaded, these tools need to be installed within the system, which is detailed in each of the above websites. Once this was established, we set up a repository on Github. Github was chosen because some of the members were already familiar with git and it was easy to develop in. The repository is located at https://github.com/mkim823/SouL. In addition to GitHub, we also used CollabEdit to all view and share code at the same time and Google Drive to share written documents and reading material. For communication, we made a private Facebook group to coordinate meetings and the cross-mobile messaging app WhatsApp for more instant messaging. From here, we developed the code for each of the different types of Node objects that were to built as part of the abstract syntax tree. In this development phase, each of us used

different tools to write the Java files. Some of us used Eclipse, others used the editor available on the Github website, and others used text editors.

When building our project we created two different Makefiles: one in the AST directory and one in the home directory. The one in the AST directory built Parser.class and made the parser ready for development. A clean target was also made to clean the AST directory from non-necessary files. This Makefile is shown below:

```
all: jflex yacc java

clean:
	rm *.class *~ Parser.java ParserVal.java Yylex.java

yacc: SoulGrammarSimple.y
	yacc -J SoulGrammarSimple.y && javac Parser.java

jflex: SoulLexerSimple.flex
	jflex SoulLexerSimple.flex

java:
	javac *.java
```

In addition to this Makefile, another one was written in the SouL directory to build the entire project and run tests. This Makefile is shown below:

```
all: compiler jsoul tests

compiler:
	@(cd AST && make)

build: clean Parser.class

clean:
	rm -f *~ *.class *.java AST/Parser.java AST/Yylex.java tests/output.txt

jsoul:
	@javac jsoul/*.java

tests: AST/Parser.class
	./test_suite.sh
```

Here, another clean target was written to remove unnecessary files. Furthermore, the output of the commands were not printed to the user.

When a user wants to use SouL, he first has to execute the make command from the SouL directory. This command will then build the Parser for the user. From here, if he places his program named `filename.soul` into the SouL directory, he can execute the following command to execute his program:

```
./soul filename.soul
```

The `soul` shell script is listed below:

```
#!/bin/sh
if [[ $1 != *.soul ]]
then
    echo "Error: files must end with .soul"
    exit 0
fi
rm -f errors.txt
rm -f scope_errors.txt
cat java_wrapper.txt > jsoul/Soul.java
(cd AST && java Parser < '../'$1) 1>> jsoul/Soul.java 2> errors.txt
if [ -s errors.txt ]; then
    cat errors.txt
    rm jsoul/Soul.java
    rm errors.txt
    exit 0
fi
echo }catch \(RuntimeException e\) {System.err.println\(\"Error: \" +
e.getMessage\(\)\)\;}}} >> jsoul/Soul.java
(cd jsoul && javac Soul.java) 2> errors.txt
if [ -s errors.txt ]; then
        awk '/^symbol.*$/' errors.txt > scope_errors.txt
        awk '{ print $0 " out of scope" }' < scope_errors.txt
        rm scope_errors.txt
        exit 0
fi
(cd jsoul && java Soul) 2> errors.txt
if [ -s errors.txt ]; then
    cat errors.txt
    rm errors.txt
fi
```

This script runs in a few parts. First, it makes sure that the filename ends with ".soul". Next, it removes the temporary text files errors.txt and scope_errors.txt if they exist. Then, it writes the file java_wrapper.txt to Soul.java. This piece of text represents the beginning of any translated Java program and is shown below:

```
import javax.sound.midi.*;
public class Soul {

        public static void main(String[] args) {
        try {
```

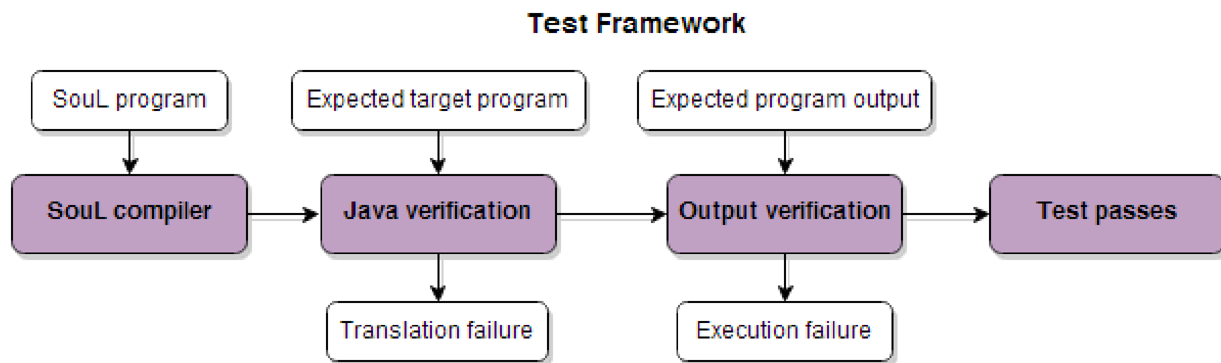
Next it passes the file as input to Parser.class and outputs the translated Java code into Soul.java. This code represents the body of the main method to be executed from Soul.java. However, if any errors are found during parsing (such as syntax errors or type checking errors), they are written to the file errors.txt. The script then checks if there are any errors within errors.txt and prints them if they are there. It then removes this file and exits. If there are no errors, it then appends the end of Soul.java to the file, completing the entire class. Next, this program, Soul.java, is passed into the Java compiler. Again, any errors that are taken from the compilation process are passed into errors.txt. If errors.txt is nonempty, it then finds all instances of lines starting with "symbol" and passes them to a file called scope_errors.txt using awk. Next, it appends the text " out of scope" to each line and prints them. This is used to use the Java compiler to catch out of scope errors, which should be the only compile time errors left. However, if there are no errors, it runs Soul.class and passes any errors to errors.txt. Again, if there are any errors they are printed to the user.

8. TEST PLAN

Written by Matt Kim (System Integrator, but also sort of the Primary Tester)

8.1 TEST METHODOLOGY

The testing framework underwent two phases: checking if the SouL program produced the correct translated Java code, and then checking if the SouL program produced the correct output afterwards. A diagram of this process is shown below:



However, for programs that required that notes be played from SouL, the output step was skipped, as it was not possible to compare files this way. Rather, the expected output (in sounds) was outputted to the user and the user, in running tests, would have to verify that the sounds are the same as the expected output.

In order to improve testing, automated testing was completed under the Makefile. Basically, whenever the SouL grammar was constructed from the home directory, the test suite was executed on the tests that were already written. The code for this script is below:

```
#!/bin/sh
java_count=0
output_count=0
total_java=0
total_output=0
for D in `find tests -type d`
do
   if [ "${D}" != "tests" ]
   then
       name=$(basename ${D})
       if [[ $name == "play" || $name == "durations" || $name == "tracks_sequences" ||
$name == "write_append_clear" ]]
       then
```

```
            echo
            cat "tests/"$name"/expected_output.txt"
        fi
        ./soul "tests/"$name"/"$name".soul" 1> tests/output.txt
        java_test=$(./same_java_test.sh $name)
        if [[ "$java_test" == *passed* ]]
        then
            java_count=`expr $java_count + 1`
        fi
        echo $java_test
        if [[ $name != "play" && $name != "durations" && $name != "tracks_sequences"
&& $name != "write_append_clear" ]]
        then
            output_test=$(./same_output_test.sh $name)
            if [[ "$output_test" == *passed* ]]
            then
                output_count=`expr $output_count + 1`
            fi
            echo $output_test
            total_output=`expr $total_output + 1`
        fi
        total_java=`expr $total_java + 1`
    fi
done
echo "$java_count of $total_java Java tests passed"
echo "$output_count of $total_output output tests passed"
```

For each test that has been placed in the test directory, this script runs the two tests detailed above (or one if the program executes play()) and prints whether or not they passed. At the end of the script, the number of successful tests for each type are outputted.

This script also made use of two other scripts, same_output_test.sh, and same_java_test.sh, which checks if the output is the same as a file called expected_output.txt and checks if the Java code is the same as a file called ExpectedJava.java, respectively. The code for these two scripts is listed below:

```
#!/bin/sh
if diff tests/output.txt "tests/$1/expected_output.txt" >/dev/null ; then
    echo $1 output test passed!
else
    echo $1 output test failed!
fi
```

```
#!/bin/sh
if diff jsoul/Soul.java "tests/$1/ExpectedJava.java" >/dev/null ; then
    echo $1 Java test passed!
else
    echo $1 Java test failed!
fi
```

The directory structure of the tests was very important for these tests to run. Basically, if a user needed to add a new test named testname, he would create a new directory called testname within the tests directory. In this directory, he would create a file called testname.soul with the SouL code to be tested, expected_output.txt with the expected output, and ExpectedJava.java with the expected Java code. If this were a file that only tests if the Java code is the same because it implements the play() method, he would also have to modify the test_suite.sh file.

The tests that were run follow in the next section.

8.2 TEST SUITE

COMMENTS

SouL Code

```
/* Comment */

/* Comm
ent */
```

Expected Java Code

```
import javax.sound.midi.*;
public class Soul {

       public static void main(String[] args) {
       try {


}catch (RuntimeException e) {System.err.println("Error: " + e.getMessage());}}}
```

Expected Output

no output

VARIABLE DECLARATIONS

SouL Code

```
Midi m = Midi("test.mid");
int x = 0;
int y, z;
int f = 0, g, h = 1;
int a = 1, b = 2, c = 3;
decimal d = 1.0;
pitch p = 1;
velocity v = 1;
instrument i = 1;
boolean e = true;
Chord ch = Chord(('C4', 'E4', 'G4'), 127, WHOLE);
```

```
Sequence s = Sequence();
Note n = Note('C4', 127, WHOLE);
Track t = Track();
```

Expected Java Code

```
import javax.sound.midi.*;
public class Soul {

        public static void main(String[] args) {
        try {


Midi m = new Midi("../test.mid");
int x = 0;
int y, z;
int f = 0, g, h = 1;
int a = 1, b = 2, c = 3;
double d = 1.0;
int p = 1;
int v = 1;
int i = 1;
boolean e = true;
Chord ch = new Chord("C4 E4 G4", 127, Note.WHOLE);
Sequence s = new Sequence();
Note n = new Note(Note.stringToPitch("C4"), 127, Note.WHOLE);
Track t = new Track();
}catch (RuntimeException e) {System.err.println("Error: " + e.getMessage());}}}
```

Expected Output

no output

DURATIONS

SouL Code

```
duration x = 128;
play(Note('C4', 127, x));
play(Note('C4', 127, 96));
play(Note('C4', 127, WHOLE));
play(Note('C4', 127, HALF));
play(Note('C4', 127, QUARTER));
play(Note('C4', 127, EIGHTH));
play(Note('C4', 127, SIXTEENTH));
play(Note('C4', 127, THIRTYSECOND));
play(Note('C4', 127, SIXTYFOURTH));
```

Expected Java Code

```
import javax.sound.midi.*;
public class Soul {

        public static void main(String[] args) {
        try {
```

```
int x = 128;
Player.play(new Note(Note.stringToPitch("C4"), 127, x));
Player.play(new Note(Note.stringToPitch("C4"), 127, 96));
Player.play(new Note(Note.stringToPitch("C4"), 127, Note.WHOLE));
Player.play(new Note(Note.stringToPitch("C4"), 127, Note.HALF));
Player.play(new Note(Note.stringToPitch("C4"), 127, Note.QUARTER));
Player.play(new Note(Note.stringToPitch("C4"), 127, Note.EIGHTH));
Player.play(new Note(Note.stringToPitch("C4"), 127, Note.SIXTEENTH));
Player.play(new Note(Note.stringToPitch("C4"), 127, Note.THIRTYSECOND));
Player.play(new Note(Note.stringToPitch("C4"), 127, Note.SIXTYFOURTH));
}catch (RuntimeException e) {System.err.println("Error: " + e.getMessage());}}}
```

Expected Output (sounds, not console output)

Double whole note
Dotted half note
Whole note
Half note
Quarter note
Eighth note
Sixteenth note
Thirty second note
Sixty fourth note

EXPRESSIONS

SouL Code

```
print((1));
print(1 + 1);
print(4 - 1);
print(2 * 2);
print(10 / 2);
print(13 % 7);
print(7^1);
print(false and false);
print(false and true);
print(true and false);
print(true and true);
print(false or false);
print(false or true);
print(true or false);
print(true or true);
print(1 < 2);
print(2 < 2);
print(3 < 2);
print(1 > 2);
print(2 > 2);
print(3 > 2);
print(1 <= 2);
print(2 <= 2);
print(3 <= 2);
print(1 >= 2);
print(2 >= 2);
print(3 >= 2);
print(1 == 2);
print(2 == 2);
```

```
print(3 == 2);
print(1 != 2);
print(2 != 2);
print(3 != 2);
int x = 0;
print(++x);
print(x++);
print(x);
print(1);
print(1.0);
print(true);
print(false);
print("test");
x += 2;
print(x);
x -= 3;
print(x);
x -= 2;
print(x);
x += 3;
print(x);
```

Expected Java Code

```
import javax.sound.midi.*;
public class Soul {

       public static void main(String[] args) {
       try {



System.out.println((1));
System.out.println(1+1);
System.out.println(4-1);
System.out.println(2*2);
System.out.println(10/2);
System.out.println(13%7);
System.out.println((int)Math.pow(7, 1));
System.out.println(false && false);
System.out.println(false && true);
System.out.println(true && false);
System.out.println(true && true);
System.out.println(false || false);
System.out.println(false || true);
System.out.println(true || false);
System.out.println(true || true);
System.out.println(1<2);
System.out.println(2<2);
System.out.println(3<2);
System.out.println(1>2);
System.out.println(2>2);
System.out.println(3>2);
System.out.println(1<=2);
System.out.println(2<=2);
System.out.println(3<=2);
System.out.println(1>=2);
System.out.println(2>=2);
System.out.println(3>=2);
System.out.println(1==2);
System.out.println(2==2);
System.out.println(3==2);
```

```
System.out.println(1!=2);
System.out.println(2!=2);
System.out.println(3!=2);
int x = 0;
System.out.println(++x);
System.out.println(x++);
System.out.println(x);
System.out.println(1);
System.out.println(1.0);
System.out.println(true);
System.out.println(false);
System.out.println("test");
x += 2;
System.out.println(x);
x -= 3;
System.out.println(x);
x -= 2;
System.out.println(x);
x += 3;
System.out.println(x);
}catch (RuntimeException e) {System.err.println("Error: " + e.getMessage());}}}
```

Expected Output

```
1
2
3
4
5
6
7
false
false
false
true
false
true
true
true
true
false
false
false
false
true
true
true
false
false
true
true
false
true
false
true
false
true
1
1
2
1
```

```
1.0
true
false
test
4
1
-1
2
```

IF / ELSE

SouL Code

```
boolean x = true;
boolean y = false;
if (x)
  print("test1");
else
  print("test2");
if (!x) {
  print("test3");
}
else if (y) {
  print("test4");
}
```

Expected Java Code

```
import javax.sound.midi.*;
public class Soul {

       public static void main(String[] args) {
       try {



boolean x = true;
boolean y = false;
if (x) System.out.println("test1"); else System.out.println("test2");
if (!x) {System.out.println("test3");} else if (y) {System.out.println("test4");}
}catch (RuntimeException e) {System.err.println("Error: " + e.getMessage());}}}
```

Expected Output

```
test1
```

LOOPS

SouL Code

```
int j = 0;
while (j < 2) {
  print("while");
  j++;
}
for (int i = 0; i < 3; i++)
   print("for");
```

Expected Java Code

```
import javax.sound.midi.*;
public class Soul {

       public static void main(String[] args) {
       try {



int j = 0;
while (j<2) {System.out.println("while");
j++;}
for (int i = 0; i<3; i++) System.out.println("for");
}catch (RuntimeException e) {System.err.println("Error: " + e.getMessage());}}}
```

Expected Output

```
while
while
for
for
for
```

PLAY STATEMENTS

SouL Code

```
play(Note('C4', 127, WHOLE));
play(Chord(('C4', 'E4', 'G4'), 127, WHOLE));
play(Midi("midi_files/WholeToneScale.mid"));
```

Expected Java Code

```
import javax.sound.midi.*;
public class Soul {

        public static void main(String[] args) {
        try {



Player.play(new Note(Note.stringToPitch("C4"), 127, Note.WHOLE));
Player.play(new Chord("C4 E4 G4", 127, Note.WHOLE));
Player.play(new Midi("../midi_files/WholeToneScale.mid"));
}catch (RuntimeException e) {System.err.println("Error: " + e.getMessage());}}}
```

Expected Output (sounds, not console output)

C4 whole note

C4 major triad

A prewritten whole tone scale

PRINT STATEMENTS

SouL Code

```
print("Hello, World!");
```

Expected Java Code

```
import javax.sound.midi.*;
public class Soul {

       public static void main(String[] args) {
       try {


System.out.println("Hello, World!");
}catch (RuntimeException e) {System.err.println("Error: " + e.getMessage());}}}
```

Expected Output

```
Hello, World!
```

SCOPE

SouL Code

```
{ int x = 0; int y = 0;}
x = 4;
y = 4;
```

Expected Java Code

```
import javax.sound.midi.*;
public class Soul {

        public static void main(String[] args) {
        try {


{int x = 0;
int y = 0;}
x = 4;
y = 4;
}catch (RuntimeException e) {System.err.println("Error: " + e.getMessage());}}}
```

Expected Output

```
symbol  : variable x out of scope
symbol  : variable y out of scope
```

TRACKS / SEQUENCES

SouL Code

```
Track t1 = Track();
t1.add(Note('C4', 127, WHOLE));
t1.add(Chord(('C4', 'E4', 'G4'), 127, WHOLE));
t1.setInstrument(40);
Track t2 = Track();
t2.add(Note('D4', 127, WHOLE));
```

```
s.add(t1);
s.add(t2);
play(s);
play(t2);
s.setTempo(240);
play(s);
```

Expected Java Code

```
import javax.sound.midi.*;
public class Soul {

        public static void main(String[] args) {
        try {



Sequence s = new Sequence();
Track t1 = new Track();
t1.add(new Note(Note.stringToPitch("C4"), 127, Note.WHOLE));
t1.add(new Chord("C4 E4 G4", 127, Note.WHOLE));
t1.setInstrument(40);
Track t2 = new Track();
t2.add(new Note(Note.stringToPitch("D4"), 127, Note.WHOLE));
s.add(t1);
s.add(t2);
Player.play(s);
Player.play(t2);
s.setTempoInBPM(240);
Player.play(s);
}catch (RuntimeException e) {System.err.println("Error: " + e.getMessage());}}}
```

Expected Output (sounds, not console output)

C4 whole note (violin), D4 whole note (piano)

C4 major chord whole note (violin)

D4 whole note (piano)

C4 whole note (violin), D4 whole note (piano) at double speed

C4 major chord whole note (violin) at double speed

WRITE / APPEND / CLEAR

SouL Code

```
Sequence s = Sequence();
Track t = Track();
t.add(Note('C4', 127, EIGHTH));
t.add(Note('D4', 127, EIGHTH));
t.add(Note('E4', 127, EIGHTH));
s.add(t);
Midi m = Midi("test.mid");
m.write(s);
play(m);
Sequence s2 = m.getSequence();
play(s2);
m.write(t);
play(m);
m.append(s);
```

```
Midi m1 = Midi("test_appended.mid");
play(m1);
m1.clear();
play(m1);
Midi m2 = Midi("test2.mid");
m2.write(Note('C4', 127, WHOLE));
play(m2);
```

Expected Java Code

```
import javax.sound.midi.*;
public class Soul {

        public static void main(String[] args) {
        try {



Sequence s = new Sequence();
Track t = new Track();
t.add(new Note(Note.stringToPitch("C4"), 127, Note.EIGHTH));
t.add(new Note(Note.stringToPitch("D4"), 127, Note.EIGHTH));
t.add(new Note(Note.stringToPitch("E4"), 127, Note.EIGHTH));
s.add(t);
Midi m = new Midi("../test.mid");
m.writeToFile(s);
Player.play(m);
Sequence s2 = m.getSequence();
Player.play(s2);
m.writeToFile(t);
Player.play(m);
m.append(s);
Midi m1 = new Midi("../test_appended.mid");
Player.play(m1);
m1.clear();
Player.play(m1);
Midi m2 = new Midi("../test2.mid");
m2.writeToFile(new Note(Note.stringToPitch("C4"), 127, Note.WHOLE));
Player.play(m2);
}catch (RuntimeException e) {System.err.println("Error: " + e.getMessage());}}}
```

Expected Output (sounds, not console output)

Repeated 4 times:

C4 eighth note

D4 eighth note

E4 eighth note

C4 whole note

9. CONCLUSIONS

Written by the entire SouL Team

9.1 LESSONS LEARNED AS A TEAM

Our lessons learned as a team are very similar to our own personal lessons learned. We found that if we had planned our goals better from the start, we would have been heading towards the right direction from the very beginning. This way, any wasted time and effort would have been put to better use and our finalized SouL program may be able to support more functions.

9.2 LESSONS LEARNED BY EACH TEAM MEMBER

KEVIN - Language Guru

Needless to say, the process of creating our own programming language brought many difficulties, both expected and unexpected. As generic as this sounds, the most difficult part of the project was working successfully and efficiently with a group. Working with a group can be difficult in general, but it becomes increasingly complex when coding is involved. Things like agreeing on meeting times, ideas, or implementations can be barriers to creating a successful language on time; as a result, compromises had to be made often. Nonetheless, working well with a group is a huge benefit and in the end allowed us to complete the project with great speed and ease.

The biggest lesson learned is to not delay things. There are many ways that things can go wrong, and in programming such a huge project it can take longer than normal to solve these issues. We wandered down incorrect paths on multiple occasions, which ate up a lot of our time. So, it is very important to go over what you plan to do and make sure it is the correct path. It is far better to spend an hour planning things out than to spend an entire meeting implementing things that can't actually be used.

Finally, it became clear after a few meetings that the project roles are definitely not restrictive, especially when your team only consists of four people. Although it is good for one person to focus on one specific aspect of the language, it helps out immensely when everyone is collaborating on different parts of the project. Participating in every part of the language, no matter how small it may be, helps to expand your understanding of the project and can even help you perform your own role better. In all, this was a very demanding project and although it was a lot of work, I am very pleased with our results and have learned a lot about both programming and working as a team.

I would suggest future teams to get started early, meet often, assign goals and deadlines for each meeting, and make sure every member has a clear understanding of what is happening to let things flow much more easily.

CINDY - System Architect

This project was a great learning process both in developing technical skills and in team working. I found that it was efficient for us to communicate status updates through group texting and group messaging on Facebook, and communicate lines of code on CollabEdit. This way, we were able to respond quickly to one another to resolve issues and confusion.

GitHub was a great tool for us but it required some communication that we lacked at first. To be precise, GitHub requires users to git pull the latest file edits before new versions can be pushed, otherwise there would be merge conflicts. However, since we were working simultaneously, there were git conflicts when we do not communicate well about what and when we were each git pushing. We learned from these conflicts and quickly adapted to keep each other informed on our actions on GitHub.

Even though we have been working on SouL throughout the semester, we headed in the wrong direction at first, which resulted in wasted time and effort. In retrospect, we should have created a timeline of goals we want to accomplish during the semester. This would have helped us maintain a regular schedule and easily identify the correct tasks that needed to be accomplished.

Overall, I had a positive experience creating a programming language. Despite the many highs and lows, it was great to work in a team at length. Teamwork is critical in all aspects of professional and personal life, so having worked in team SouL, I feel more prepared for future projects that require extensive group tasks.

MATT - System Integrator

Building our own language came with many unexpected challenges. Many of these challenges, in my opinion, could have been avoided if we had planned more carefully.

First off, I think that we as a group underestimated the scope of the project and the number of roadblocks that we would run into along the way. Looking at our meeting schedule and Github commits, they are very heavily weighted close to the project deadline. This is because in the beginning of the semester, we had very sporadic meetings and this had us start very slowly and be forced to rush at the end. However, I think that this meeting schedule would have been satisfactory if not for the challenges that we encountered while building our language. During many times in the completion of the project, we thought we had found a viable solution, began to pursue that solution, only to realize that there were too many problems with that solution, and then

ultimately abandon it. I think that this pitfall could have been avoided if we had done more thorough research before attempting every first solution that came to us.

I also found working in a group difficult because the four of us all came from different backgrounds. While we were all proficient in Java, all of us had to learn new technologies to ultimately complete the project. While this was very educational, I would have liked to spend more time actually working on the project rather than spending time learning new programming constructs.

Despite all this, I still found it very rewarding to complete a project on such a large scale. At the end of the last day of coding, we were talking, almost in surprise, about how we had actually built this language from scratch. The idea that we, as a group, had created something that no one had done before was a gratifying feeling. Furthermore, because so much of programming in the real world is done in groups, learning about the ups and downs of working in one is something that I really feel prepared me for the future.

In summary, I would advise teams to definitely start early and plan for obstacles along the way. Moreover, before starting to implement anything in the language, I think that teams should always complete the necessary research before starting to code it. However, like I said, in the end I enjoyed working on this project and definitely would like to complete something similar in the future.

ANDREW - Project Manager

There isn't much I can say that the other members of this team haven't already nailed. However, one of the biggest things that made managing this project difficult is the fact that our team of four people had to complete a project designed to be done by a team of five within the *same* amount of time. Whenever even a single member wasn't available for a meeting, development time would sometimes slow to a crawl. The absence of a System Tester made it so that each of us had to take extra time to contribute to testing the various stages of our implementation. Even putting together this final report became a more tedious process as a result (props to Matt for writing the entire Testing chapter).

As a result of this, the biggest lesson I learned as a manager was that the logistics of the project need to be concrete and totally sound at the onset. There needs to be a fixed meeting schedule, with a very clear set of both short-term and long term goals made in advance. Don't take any breaks after deliverable deadlines that can't be afforded. Also important: constant communication. We were able to get a lot of work done at meetings because we were able to communicate face to face. Although communication lessened outside of meetings (which is to be expected), often it would become too sparse and development progress on an individual basis suffered as a result. This would sometimes lead to having multiple meetings in a row leading up to deliverable deadlines in which the team would stay up late and rush to get everything finished. Not a great idea.

9.3 ADVICE FOR FUTURE TEAMS

We would advise future teams to devise a plan of action with a timeline of goals to be accomplished throughout the semester. This plan will help them immensely to keep on the correct track. This plan combined with a regular schedule to meet once or twice a week will be tremendously helpful for teams to both communicate and track their progress.

Furthermore, it would be smart to make use of the resources available, namely the professor and the TAs. They are, indisputably and thoroughly, knowledgeable on the subject and so they can clarify any confusions teams have. This way, teams can save time and effort, which would be more productively used towards their progress.

9.4 SUGGESTIONS FOR THE COURSE

It's somewhat problematic that the majority of the course material focuses on theory rather than implementation. Not because the information presented in the course isn't interesting or engaging, but because this semester-long project that is worth 40% of our grade is a very large and complex implementation task.

Much of the help given by the professor (who was our mentor on this project) and TAs both in class and during office hours was done so in abstract terms that made it a bit more difficult to understand the concrete processes required to implement each component of our final compiler. Because of this, there was a great deal of indecision and initial confusions as to whether our group was taking the proper approach to the project, given our development and runtime environments.

APPENDIX A: FULL *SouL* SOURCE CODE

Below is the complete source code listing for the SouL compiler. Some parts of the code may look messy in this document. This is because many of the lines of code are longer than the margins of the document will allow, thus some line wrapping has occurred. The code can be viewed with a cleaner format at: http://github.com/mkim823/SouL.

AUTHOR CREDITS

All members of the team contributed to almost every module of the compiler. Each component is listed below, along with its primary author and assisting members, if any.

MODULE	FILE(S)	PRIMARY AUTHOR	ASSISTING MEMBERS
Lexer	SouL/AST/soul.flex	Matt Kim	Kevin Walters
Parser	SouL/AST/soul.y	Kevin Walters	Matt Kim
Semantic analyzer	Node classes for grammar parse actions, and symbol table *i.e. SouL/AST/Node.java, all classes that extend Node, and SouL/AST/SymbolTable.java*	Cindy Long	ALL *author names are written in comments at the beginning of each class declaration*
Java code generation	SouL/jsoul/*.java *i.e. all java source files in the jsoul directory*	Andrew Goldin	none
Misc.	SouL/Makefile, SouL/AST/Makefile *makefiles for building the compiler* SouL/soul *shell script for running SouL programs* SouL/java_wrapper.txt *Header for all translated Java programs*	Matt Kim	none

The JFlex lexer file **SouL/AST/soul.flex**:

```
%%

%byaccj

%{
        private Parser yyparser;

        public Yylex(java.io.Reader r, Parser yyparser) {
                this(r);
                this.yyparser = yyparser;
        }
%}
%%
[\ |\t|\n]+                     { }
\/\*[^(\/\*)]*?\*\/              { /* comments - do nothing */ }
[0-9]+                          { yyparser.yylval = new ParserVal(new
IntegerNode(Integer.parseInt(yytext()))); return Parser.NUMBER_INT; }
[0-9]+(\.[0-9]+)?       { yyparser.yylval = new ParserVal(new DecimalNode(Double.parseDouble(yytext())));
return Parser.NUMBER_DECIMAL; }
\.                      { yyparser.yylval = new ParserVal(new StringNode(yytext())); return '.'; }
\(                      { yyparser.yylval = new ParserVal(new StringNode(yytext())); return '('; }
\)                      { yyparser.yylval = new ParserVal(new StringNode(yytext())); return ')'; }
\{                      { yyparser.yylval = new ParserVal(new StringNode(yytext())); return '{'; }
\}                      { yyparser.yylval = new ParserVal(new StringNode(yytext())); return '}'; }
;                       { yyparser.yylval = new ParserVal(new StringNode(yytext())); return ';'; }
,                       { yyparser.yylval = new ParserVal(new StringNode(yytext())); return ','; }
'[A-G](#|b)?(-1|[0-9])'         { yyparser.yylval = new ParserVal(new PitchNameNode(yytext())); return
Parser.NOTENAME; }
if              { yyparser.yylval = new ParserVal(new StringNode(yytext())); return Parser.IF; }
else            { yyparser.yylval = new ParserVal(new StringNode(yytext())); return Parser.ELSE; }
while           { yyparser.yylval = new ParserVal(new StringNode(yytext())); return Parser.WHILE; }
for             { yyparser.yylval = new ParserVal(new StringNode(yytext())); return Parser.FOR; }
write           { yyparser.yylval = new ParserVal(new StringNode(yytext())); return Parser.WRITE; }
clear           { yyparser.yylval = new ParserVal(new StringNode(yytext())); return Parser.CLEAR; }
append          { yyparser.yylval = new ParserVal(new StringNode(yytext())); return Parser.APPEND; }
Note            { yyparser.yylval = new ParserVal(new TypeNode(yytext())); return Parser.NOTE; }
Midi            { yyparser.yylval = new ParserVal(new TypeNode(yytext())); return Parser.MIDI; }
Chord           { yyparser.yylval = new ParserVal(new TypeNode(yytext())); return Parser.CHORD; }
int             { yyparser.yylval = new ParserVal(new TypeNode(yytext())); return Parser.INT; }
decimal         { yyparser.yylval = new ParserVal(new TypeNode(yytext())); return Parser.DECIMAL; }
Sequence{ yyparser.yylval = new ParserVal(new TypeNode(yytext())); return Parser.SEQUENCE; }
Track           { yyparser.yylval = new ParserVal(new TypeNode(yytext())); return Parser.TRACK; }
string          { yyparser.yylval = new ParserVal(new TypeNode(yytext())); return Parser.STRING; }
pitch           { yyparser.yylval = new ParserVal(new TypeNode(yytext())); return Parser.PITCH; }
velocity{ yyparser.yylval = new ParserVal(new TypeNode(yytext())); return Parser.VELOCITY; }
duration{ yyparser.yylval = new ParserVal(new TypeNode(yytext())); return Parser.DURATION; }
instrument      { yyparser.yylval = new ParserVal(new TypeNode(yytext())); return Parser.INSTRUMENT; }
boolean         { yyparser.yylval = new ParserVal(new TypeNode(yytext())); return Parser.BOOLEAN; }
\".+\.mid(i)?\"   { yyparser.yylval = new ParserVal(new StringNode(yytext())); return Parser.FILENAME; }
play            { yyparser.yylval = new ParserVal(new StringNode(yytext())); return Parser.PLAY; }
print           { yyparser.yylval = new ParserVal(new StringNode(yytext())); return Parser.PRINT; }
clear           { yyparser.yylval = new ParserVal(new StringNode(yytext())); return Parser.CLEAR; }
add             { yyparser.yylval = new ParserVal(new StringNode(yytext())); return Parser.ADD; }
getSequence    { yyparser.yylval = new ParserVal(new StringNode(yytext())); return Parser.GETSEQUENCE; }
transpose       { yyparser.yylval = new ParserVal(new StringNode(yytext())); return Parser.TRANSPOSE; }
```

```
setInstrument    { yyparser.yylval = new ParserVal(new StringNode(yytext())); return Parser.SETINSTRUMENT;
}
setTempo{ yyparser.yylval = new ParserVal(new StringNode(yytext())); return Parser.SETTEMPO; }
true             { yyparser.yylval = new ParserVal(new BooleanNode(Boolean.parseBoolean(yytext()))); return
Parser.TRUE; }
false            { yyparser.yylval = new ParserVal(new BooleanNode(Boolean.parseBoolean(yytext()))); return
Parser.FALSE; }
and              { yyparser.yylval = new ParserVal(new StringNode(yytext())); return Parser.AND; }
or               { yyparser.yylval = new ParserVal(new StringNode(yytext())); return Parser.OR; }
\<=              { yyparser.yylval = new ParserVal(new StringNode(yytext())); return Parser.LTEQ; }
\>=              { yyparser.yylval = new ParserVal(new StringNode(yytext())); return Parser.GTEQ; }
\<               { yyparser.yylval = new ParserVal(new StringNode(yytext())); return '<'; }
\>               { yyparser.yylval = new ParserVal(new StringNode(yytext())); return '>'; }
\+\+             { yyparser.yylval = new ParserVal(new StringNode(yytext())); return Parser.PLUSPLUS; }
--               { yyparser.yylval = new ParserVal(new StringNode(yytext())); return Parser.MINUSMINUS; }
\+               { yyparser.yylval = new ParserVal(new StringNode(yytext())); return '+'; }
-                { yyparser.yylval = new ParserVal(new StringNode(yytext())); return '-'; }
\/               { yyparser.yylval = new ParserVal(new StringNode(yytext())); return '/'; }
\*               { yyparser.yylval = new ParserVal(new StringNode(yytext())); return '*'; }
\^               { yyparser.yylval = new ParserVal(new StringNode(yytext())); return '^'; }
%                { yyparser.yylval = new ParserVal(new StringNode(yytext())); return '%'; }
\+=              { yyparser.yylval = new ParserVal(new StringNode(yytext())); return Parser.PLUSEQ; }
-=               { yyparser.yylval = new ParserVal(new StringNode(yytext())); return Parser.MINUSEQ; }
\/=              { yyparser.yylval = new ParserVal(new StringNode(yytext())); return Parser.DIVEQ; }
\*=              { yyparser.yylval = new ParserVal(new StringNode(yytext())); return Parser.MULTEQ; }
%=               { yyparser.yylval = new ParserVal(new StringNode(yytext())); return Parser.MODEQ; }
\^=              { yyparser.yylval = new ParserVal(new StringNode(yytext())); return Parser.EXPEQ; }
==               { yyparser.yylval = new ParserVal(new StringNode(yytext())); return Parser.EQ; }
\!=              { yyparser.yylval = new ParserVal(new StringNode(yytext())); return Parser.NOTEQ; }
\!               { yyparser.yylval = new ParserVal(new StringNode(yytext())); return '!'; }
=                { yyparser.yylval = new ParserVal(new StringNode(yytext())); return '='; }
WHOLE            { yyparser.yylval = new ParserVal(new StringNode(yytext())); return Parser.WHOLE; }
HALF             { yyparser.yylval = new ParserVal(new StringNode(yytext())); return Parser.HALF; }
QUARTER          { yyparser.yylval = new ParserVal(new StringNode(yytext())); return Parser.QUARTER; }
EIGHTH           { yyparser.yylval = new ParserVal(new StringNode(yytext())); return Parser.EIGHTH; }
SIXTEENTH        { yyparser.yylval = new ParserVal(new StringNode(yytext())); return Parser.SIXTEENTH; }
THIRTYSECOND  { yyparser.yylval = new ParserVal(new StringNode(yytext())); return Parser.THIRTYSECOND; }
SIXTYFOURTH    { yyparser.yylval = new ParserVal(new StringNode(yytext())); return Parser.SIXTYFOURTH; }
([a-zA-Z]|_)([a-zA-Z0-9]|_)*     { yyparser.yylval = new ParserVal(new IdentifierNode(yytext())); return
Parser.IDENTIFIER; }
\".+\"       { yyparser.yylval = new ParserVal(new StringNode(yytext())); return Parser.STRINGLITERAL; }
```

The BYacc/J grammar with parse actions **SouL/AST/soul.y**:

```
%{
        import java.io.*;
        import java.util.*;
%}

%token BOOLEAN TRUE FALSE INT DECIMAL PITCH VELOCITY DURATION INSTRUMENT STRING
%token NOTE CHORD TRACK SEQUENCE MIDI FILENAME
%token IDENTIFIER NOTENAME NUMBER_INT NUMBER_DECIMAL STRINGLITERAL
%token '.' ',' '(' ')' '{' '}' ';' '+' '-' '*' '/' '%' '^' '=' '!'
%token PLAY WRITE APPEND TRANSPOSE CLEAR ADD GETSEQUENCE SETINSTRUMENT SETTEMPO PRINT
```

```
%token WHOLE HALF QUARTER EIGHTH SIXTEENTH THIRTYSECOND SIXTYFOURTH
%token PLUSEQ MINUSEQ MULTEQ DIVEQ MODEQ EXPEQ PLUSPLUS MINUSMINUS EQ NOTEQ LTEQ GTEQ OR AND
%token WHILE FOR IF ELSE LOWER_THAN_ELSE

%nonassoc LOWER_THAN_ELSE
%nonassoc ELSE
%left PLUSEQ MINUSEQ
%left TIMESEQ DIVIDEEQ MODEQ
%left EXPEQ
%right '='
%left OR
%left AND
%left EQ NOTEQ
%left '<' '>' LTEQ GTEQ
%left '+' '-'
%left '*' '/' '%'
%right '!'
%right '^'
%left PLUSPLUS MINUSMINUS

%%

        StatementBlock :
        StatementBlock FullStatement     { $$ = new ParserVal(new StatementBlockNode((FullStatementNode)
$2.obj, (StatementBlockNode) $1.obj));
                                                         if ($2.obj != null)
System.out.println($2.obj.toString()); }
                |
                ;

        CompoundStatement :
                '{' CompoundStatementBlock '}'   { $$ = new ParserVal(new
CompoundStatementNode((StatementBlockNode) $2.obj)); }
                ;

        CompoundStatementBlock :
                FullStatement CompoundStatementBlock     { if
($1.obj.toString().equals($2.obj.toString()))
                                                                 $$ = new ParserVal(new
StatementBlockNode((FullStatementNode) $1.obj));
                                                           else
                                                         $$ = new ParserVal(new
StatementBlockNode((FullStatementNode) $1.obj, (StatementBlockNode) $2.obj)); }
                |
                ;

        FullStatement :
                Statement ';'                            { $$ = new ParserVal(new
FullStatementNode((StatementNode) $1.obj)); }
                | IfStatement                            { $$ = new ParserVal(new
FullStatementNode((IfStatementNode) $1.obj)); }
                | CompoundStatement                      { $$ = new ParserVal(new
FullStatementNode((CompoundStatementNode) $1.obj)); }
                | WhileStatement                 { $$ = new ParserVal(new
FullStatementNode((WhileStatementNode) $1.obj)); }
                | ForStatement                           { $$ = new ParserVal(new
FullStatementNode((ForStatementNode) $1.obj)); }
                ;
```

```
        Statement :
                PlayStatement                           { $$ = new ParserVal(new
StatementNode((PlayStatementNode) $1.obj)); }
                | DeclarationInitialization             { $$ = new ParserVal(new
StatementNode((DeclarationInitializationNode) $1.obj)); }
                | Declaration                           { $$ = new ParserVal(new
StatementNode((DeclarationNode) $1.obj)); }
                | WriteStatement                        { $$ = new ParserVal(new
StatementNode((WriteStatementNode) $1.obj)); }
                | ClearStatement                        { $$ = new ParserVal(new
StatementNode((ClearStatementNode) $1.obj)); }
                | AppendStatement                       { $$ = new ParserVal(new
StatementNode((AppendStatementNode) $1.obj)); }
                | Addition                              { $$ = new ParserVal(new
StatementNode((AdditionNode) $1.obj)); }
                | PrintStatement                        { $$ = new ParserVal(new
StatementNode((PrintStatementNode) $1.obj)); }
                | GetSequence                           { $$ = new ParserVal(new
StatementNode((GetSequenceNode) $1.obj)); }
                | Assignment                            { $$ = new ParserVal(new
StatementNode((AssignmentNode) $1.obj)); }
                | Expression                            { $$ = new ParserVal(new
StatementNode((ExpressionNode) $1.obj)); }
                | Transpose                             { $$ = new ParserVal(new
StatementNode((TransposeStatementNode) $1.obj)); }
                | SetInstrument                         { $$ = new ParserVal(new
StatementNode((SetInstrumentNode) $1.obj)); }
                | SetTempo                              { $$ = new ParserVal(new
StatementNode((SetTempoNode) $1.obj)); }
                ;

        Expression :
                '(' Expression ')'                      { $$ = new ParserVal(new
ExpressionNode((ExpressionNode) $2.obj)); }
        | Expression '+' Expression             { $$ = new ParserVal(new ExpressionNode((ExpressionNode)
$1.obj, "+", (ExpressionNode) $3.obj)); }
        | Expression '-' Expression             { $$ = new ParserVal(new ExpressionNode((ExpressionNode)
$1.obj, "-", (ExpressionNode) $3.obj)); }
        | Expression '*' Expression             { $$ = new ParserVal(new ExpressionNode((ExpressionNode)
$1.obj, "*", (ExpressionNode) $3.obj)); }
        | Expression '/' Expression             { $$ = new ParserVal(new ExpressionNode((ExpressionNode)
$1.obj, "/", (ExpressionNode) $3.obj)); }
        | Expression '%' Expression             { $$ = new ParserVal(new ExpressionNode((ExpressionNode)
$1.obj, "%", (ExpressionNode) $3.obj)); }
        | Expression '^' Expression             { $$ = new ParserVal(new ExpressionNode((ExpressionNode)
$1.obj, "^", (ExpressionNode) $3.obj)); }
        | Expression AND Expression             { $$ = new ParserVal(new ExpressionNode((ExpressionNode)
$1.obj, "and", (ExpressionNode) $3.obj)); }
        | Expression OR Expression              { $$ = new ParserVal(new ExpressionNode((ExpressionNode)
$1.obj, "or", (ExpressionNode) $3.obj)); }
        | Expression '<' Expression             { $$ = new ParserVal(new ExpressionNode((ExpressionNode)
$1.obj, "<", (ExpressionNode) $3.obj)); }
        | Expression '>' Expression             { $$ = new ParserVal(new ExpressionNode((ExpressionNode)
$1.obj, ">", (ExpressionNode) $3.obj)); }
        | Expression LTEQ Expression            { $$ = new ParserVal(new ExpressionNode((ExpressionNode)
$1.obj, "<=", (ExpressionNode) $3.obj)); }
        | Expression GTEQ Expression            { $$ = new ParserVal(new ExpressionNode((ExpressionNode)
$1.obj, ">=", (ExpressionNode) $3.obj)); }
```

```
        | Expression EQ Expression              { $$ = new ParserVal(new ExpressionNode((ExpressionNode)
$1.obj, "==", (ExpressionNode) $3.obj)); }
        | Expression NOTEQ Expression           { $$ = new ParserVal(new ExpressionNode((ExpressionNode)
$1.obj, "!=", (ExpressionNode) $3.obj)); }
                | PrefixExpression                      { $$ = new ParserVal(new
ExpressionNode((PrefixExpressionNode) $1.obj)); }
                | PostfixExpression                     { $$ = new ParserVal(new
ExpressionNode((PostfixExpressionNode) $1.obj)); }
                | NUMBER_INT                            { $$ = new ParserVal(new
ExpressionNode((IntegerNode) $1.obj)); }
                | NUMBER_DECIMAL                { $$ = new ParserVal(new ExpressionNode((DecimalNode)
$1.obj)); }
                | IDENTIFIER                            { $$ = new ParserVal(new
ExpressionNode((IdentifierNode) $1.obj)); }
                | TRUE                                  { $$ = new ParserVal(new
ExpressionNode((BooleanNode) $1.obj)); }
                | FALSE                                 { $$ = new ParserVal(new
ExpressionNode((BooleanNode) $1.obj)); }
                | STRINGLITERAL                         { $$ = new ParserVal(new
ExpressionNode((StringNode) $1.obj)); }
                | Duration                              { $$ = new ParserVal(new
ExpressionNode((DurationNode) $1.obj)); }
                | Pitch                                 { $$ = new ParserVal(new
ExpressionNode((PitchNode) $1.obj)); }
                ;

        PostfixExpression :
                IDENTIFIER PLUSPLUS             { $$ = new ParserVal(new
PostfixExpressionNode((IdentifierNode) $1.obj, "++")); }
                | IDENTIFIER MINUSMINUS         { $$ = new ParserVal(new
PostfixExpressionNode((IdentifierNode) $1.obj, "--")); }
                ;

        PrefixExpression :
                '-' Expression                  { $$ = new ParserVal(new PrefixExpressionNode("-",
(ExpressionNode) $2.obj)); }
                | '+' Expression        { $$ = new ParserVal(new PrefixExpressionNode("+",
(ExpressionNode) $2.obj)); }
                | '!' Expression        { $$ = new ParserVal(new PrefixExpressionNode("!",
(ExpressionNode) $2.obj)); }
                | PLUSPLUS IDENTIFIER           { $$ = new ParserVal(new PrefixExpressionNode("++",
(IdentifierNode) $2.obj)); }
                | MINUSMINUS IDENTIFIER         { $$ = new ParserVal(new PrefixExpressionNode("--",
(IdentifierNode) $2.obj)); }
                ;

        IfStatement :
        IF '(' Expression ')' FullStatement ELSE FullStatement           { $$ = new ParserVal(new
IfStatementNode((ExpressionNode) $3.obj, (FullStatementNode) $5.obj, (FullStatementNode) $7.obj)); }
        | IF '(' Expression ')' FullStatement %prec LOWER_THAN_ELSE      { $$ = new ParserVal(new
IfStatementNode((ExpressionNode)$3.obj, (FullStatementNode) $5.obj)); }
                ;

        WhileStatement :
        WHILE '(' Expression ')' FullStatement          { $$ = new ParserVal(new
WhileStatementNode((ExpressionNode) $3.obj, (FullStatementNode) $5.obj)); }
                ;
```

```
        ForStatement :
        FOR '(' OptStatement ';' OptExpression ';' OptStatement ')' FullStatement { $$ = new ParserVal(new
ForStatementNode((OptStatementNode) $3.obj, (OptExpressionNode) $5.obj, (OptStatementNode) $7.obj,
(FullStatementNode) $9.obj)); }
                ;

        OptExpression :
                Expression      { $$ = new ParserVal(new OptExpressionNode((ExpressionNode) $1.obj)); }
                |               { $$ = new ParserVal(new OptExpressionNode()); }
                ;

        OptStatement :
                Statement       { $$ = new ParserVal(new OptStatementNode((StatementNode) $1.obj)); }
                |               { $$ = new ParserVal(new OptStatementNode()); }
                ;

        PlayStatement :
                PLAY '(' Initialization ')'     { $$ = new ParserVal(new
PlayStatementNode((InitializationNode) $3.obj)); }
                | PLAY '(' FILENAME ')'         { $$ = new ParserVal(new PlayStatementNode((StringNode)
$3.obj)); }
                | PLAY '(' IDENTIFIER ')'       { $$ = new ParserVal(new
PlayStatementNode((IdentifierNode) $3.obj)); }
                ;

        Initialization :
        '(' Initialization ')'                                          { $$ = new ParserVal(new
InitializationNode((InitializationNode) $2.obj)); }
        | MIDI '(' FILENAME ')'                                 { $$ = new ParserVal(new
InitializationNode((TypeNode) $1.obj, (StringNode) $3.obj)); }
        | NOTE '(' Expression ',' Expression ',' Expression ')'         { $$ = new ParserVal(new
InitializationNode((TypeNode) $1.obj, (ExpressionNode) $3.obj, (ExpressionNode) $5.obj, (ExpressionNode)
$7.obj)); }
        | CHORD '(' '(' PitchList ')' ',' Expression ',' Expression ')'  { $$ = new ParserVal(new
InitializationNode((TypeNode) $1.obj, (PitchListNode) $4.obj, (ExpressionNode) $7.obj, (ExpressionNode)
$9.obj)); }
                | SEQUENCE '(' ')'                                              { $$ = new
ParserVal(new InitializationNode((TypeNode) $1.obj)); }
                | GetSequence                                                   { $$ = new
ParserVal(new InitializationNode((GetSequenceNode) $1.obj));}
                | TRACK '(' ')'                                         { $$ = new ParserVal(new
InitializationNode((TypeNode) $1.obj)); }
                ;

        PitchList :
        NOTENAME ',' PitchList          { $$ = new ParserVal (new PitchListNode((PitchNameNode) $1.obj,
(PitchListNode) $3.obj)); }
                | NOTENAME                      { $$ = new ParserVal (new PitchListNode((PitchNameNode)
$1.obj)); }
                ;

        Pitch :
                NOTENAME                { $$ = new ParserVal(new PitchNode((PitchNameNode) $1.obj)); }
                ;

        Duration :
                WHOLE                   { $$ = new ParserVal(new DurationNode((StringNode) $1.obj)); }
```

```
		| HALF			{ $$ = new ParserVal(new DurationNode((StringNode) $1.obj)); }
		| QUARTER		{ $$ = new ParserVal(new DurationNode((StringNode) $1.obj)); }
		| EIGHTH	{ $$ = new ParserVal(new DurationNode((StringNode) $1.obj)); }
		| SIXTEENTH		{ $$ = new ParserVal(new DurationNode((StringNode) $1.obj)); }
		| THIRTYSECOND		{ $$ = new ParserVal(new DurationNode((StringNode) $1.obj)); }
		| SIXTYFOURTH		{ $$ = new ParserVal(new DurationNode((StringNode) $1.obj)); }
		;

	DeclarationInitialization :
	Type IDENTIFIER '=' Initialization	{ $$ = new ParserVal (new
DeclarationInitializationNode((TypeNode) $1.obj, (IdentifierNode) $2.obj, (InitializationNode) $4.obj)); }
		;

	Declaration :
		Type DeclaratorList	{ $$ = new ParserVal (new DeclarationNode((TypeNode) $1.obj,
(DeclaratorListNode) $2.obj)); }
		;

	DeclaratorList :
		Declarator				{ $$ = new ParserVal (new
DeclaratorListNode((DeclaratorNode) $1.obj)); }
	| DeclaratorList ',' Declarator	{ $$ = new ParserVal (new DeclaratorListNode((DeclaratorListNode)
$1.obj, (DeclaratorNode) $3.obj)); }
		;

	Declarator :
		IDENTIFIER				{ $$ = new ParserVal (new
DeclaratorNode((IdentifierNode)$1.obj)); }
	| IDENTIFIER '=' Expression		{ $$ = new ParserVal(new DeclaratorNode((IdentifierNode)
$1.obj, (ExpressionNode) $3.obj)); }
		;

	Type :
		INT		{ $$ = new ParserVal($1.obj); }
		| DECIMAL	{ $$ = new ParserVal($1.obj); }
		| STRING	{ $$ = new ParserVal($1.obj); }
		| PITCH { $$ = new ParserVal($1.obj); }
		| VELOCITY	{ $$ = new ParserVal($1.obj); }
		| DURATION	{ $$ = new ParserVal($1.obj); }
		| INSTRUMENT	{ $$ = new ParserVal($1.obj); }
		| BOOLEAN	{ $$ = new ParserVal($1.obj); }
		| NOTE		{ $$ = new ParserVal($1.obj); }
		| CHORD { $$ = new ParserVal($1.obj); }
		| TRACK		{ $$ = new ParserVal($1.obj); }
		| SEQUENCE	{ $$ = new ParserVal($1.obj); }
		| MIDI		{ $$ = new ParserVal($1.obj); }
		;


	WriteStatement :
	IDENTIFIER '.' WRITE '(' IDENTIFIER ')'  { $$ = new ParserVal(new
WriteStatementNode((IdentifierNode) $1.obj, (IdentifierNode) $5.obj)); }
	| IDENTIFIER '.' WRITE '(' Initialization ')'	{ $$ = new ParserVal(new
WriteStatementNode((IdentifierNode) $1.obj, (InitializationNode) $5.obj)); }
		;

	AppendStatement :
```

```
        IDENTIFIER '.' APPEND '(' IDENTIFIER ')'         { $$ = new ParserVal(new
AppendStatementNode((IdentifierNode) $1.obj, (IdentifierNode) $5.obj)); }
        | IDENTIFIER '.' APPEND '(' Initialization ')'           { $$ = new ParserVal(new
AppendStatementNode((IdentifierNode) $1.obj, (InitializationNode) $5.obj)); }
                ;

        ClearStatement :
                IDENTIFIER '.' CLEAR '(' ')'     { $$ = new ParserVal(new
ClearStatementNode((IdentifierNode) $1.obj)); }
                ;

        Addition :
        IDENTIFIER '.' ADD '(' IDENTIFIER ')'            { $$ = new ParserVal(new
AdditionNode((IdentifierNode) $1.obj, (IdentifierNode) $5.obj)); }
        | IDENTIFIER '.' ADD '(' Initialization ')'      { $$ = new ParserVal(new
AdditionNode((IdentifierNode) $1.obj, (InitializationNode) $5.obj)); }
                ;

        PrintStatement :
                PRINT '(' Expression ')'         { $$ = new ParserVal(new
PrintStatementNode((ExpressionNode) $3.obj)); }
                | PRINT '(' Initialization ')'   { $$ = new ParserVal(new
PrintStatementNode((InitializationNode) $3.obj)); }
                ;

        GetSequence :
                IDENTIFIER '.' GETSEQUENCE '(' ')'               { $$ = new ParserVal(new
GetSequenceNode((IdentifierNode) $1.obj)); }
                | Initialization '.' GETSEQUENCE '(' ')' { $$ = new ParserVal(new
GetSequenceNode((InitializationNode) $1.obj)); }
                ;

        Assignment :
        IDENTIFIER '=' Expression                { $$ = new ParserVal(new AssignmentNode((IdentifierNode)
$1.obj, (ExpressionNode) $3.obj)); }
        | IDENTIFIER AssignUpdate Expression     { $$ = new ParserVal(new AssignmentNode((IdentifierNode)
$1.obj, (AssignUpdateNode) $2.obj, (ExpressionNode) $3.obj)); }
        | IDENTIFIER '=' Initialization  { $$ = new ParserVal(new AssignmentNode((IdentifierNode) $1.obj,
(InitializationNode) $3.obj)); }
                ;

        AssignUpdate :
                PLUSEQ          { $$ = new ParserVal(new AssignUpdateNode((StringNode) $1.obj)); }
                | MINUSEQ       { $$ = new ParserVal(new AssignUpdateNode((StringNode) $1.obj)); }
                | MULTEQ        { $$ = new ParserVal(new AssignUpdateNode((StringNode) $1.obj)); }
                | DIVEQ { $$ = new ParserVal(new AssignUpdateNode((StringNode) $1.obj)); }
                | MODEQ { $$ = new ParserVal(new AssignUpdateNode((StringNode) $1.obj)); }
                | EXPEQ { $$ = new ParserVal(new AssignUpdateNode((StringNode) $1.obj)); }
                ;

        Transpose :
        IDENTIFIER '.' TRANSPOSE '(' Expression ')'      {$$ = new ParserVal(new
TransposeStatementNode((IdentifierNode)$1.obj, (ExpressionNode)$5.obj));}
                ;

        SetInstrument :
```

```
        IDENTIFIER '.' SETINSTRUMENT '(' Expression ')'          { $$ = new ParserVal(new
SetInstrumentNode((IdentifierNode) $1.obj, (ExpressionNode) $5.obj)); }
                ;

        SetTempo :
        IDENTIFIER '.' SETTEMPO '(' Expression ')'               { $$ = new ParserVal(new
SetTempoNode((IdentifierNode) $1.obj, (ExpressionNode) $5.obj)); }
                ;

%%

private Yylex lexer;

private int yylex () {
        int yyl_return = -1;
        try {
                yylval = new ParserVal(0);
                yyl_return = lexer.yylex();

        } catch (IOException e) {
                System.err.println("IO error :"+e);
        }
        return yyl_return;
}

public void yyerror (String error) {
        System.err.println ("Error: " + error);
}

public Parser(Reader r) {
        lexer = new Yylex(r, this);
}

static boolean interactive;

public static void main(String args[]) throws IOException {
        Parser yyparser;
        if ( args.length > 0 ) {
                // parse a file
                yyparser = new Parser(new FileReader(args[0]));
        }
        else {
                // interactive mode
                interactive = true;
                yyparser = new Parser(new InputStreamReader(System.in));
        }
        yyparser.yyparse();
}
```

Abstract Node class **SouL/AST/Node.java**:

```
/* Cindy */
```

```
import java.util.ArrayList;

/*
* Abstract class for node
*/

public abstract class Node {

        public ArrayList<Node> children = new ArrayList<Node>();

        public abstract String toString();

}
```

All subclasses that extend Node (each of these classes is in its own file):

```
/* Cindy */
public class AdditionNode extends Node {

        public AdditionNode (IdentifierNode n, InitializationNode n2) {
                if (SymbolTable.ST.get(n.id) != null) {
                        if (SymbolTable.ST.get(n.id).equals("Sequence")){
                                if (!n2.type.equals("Track"))
                                        System.err.println("Error: Cannot add type "+n2.type+" to a
Sequence");
                        }
                        else if (SymbolTable.ST.get(n.id).equals("Track")) {
                                if
(!(n2.type.equals("Note")||n2.type.equals("Chord")||n2.type.equals("Track")))
                                        System.err.println("Error: Cannot add type "+n2.type+" to a
Track");
                        }
                        else
                                System.err.println("Error: Cannot call add on type
"+SymbolTable.ST.get(n.id)+". Can only add to a Sequence or Track");

                }
                else
                        System.err.println("Symbol variable "+n.id+" is not defined");
                children.add(n);
                children.add(n2);
        }

        public AdditionNode (IdentifierNode n, IdentifierNode n2) {
                if ((SymbolTable.ST.get(n.id) != null)&&(SymbolTable.ST.get(n2.id) != null)) {
                        if (SymbolTable.ST.get(n.id).equals("Sequence")){
                                if (!SymbolTable.ST.get(n2.id).equals("Track"))
                                        System.err.println("Error: Cannot add type
"+SymbolTable.ST.get(n2.id)+" to a Sequence");
                        }
                        else if (SymbolTable.ST.get(n.id).equals("Track")) {
                                if
(!(SymbolTable.ST.get(n2.id).equals("Note")||SymbolTable.ST.get(n2.id).equals("Chord")||SymbolTable.ST.get
(n2.id).equals("Track")))
```

```
                                        System.err.println("Error: Cannot add type
"+SymbolTable.ST.get(n2.id) +" to a Track");
                        }
                        else
                                System.err.println("Error: Cannot call add on type
"+SymbolTable.ST.get(n.id)+". Can only add to a Sequence or Track");

                }
                else {
                        if (SymbolTable.ST.get(n.id)==null)
                                System.err.println("Symbol variable "+n+" is not defined");
                        else
                                System.err.println("Symbol variable "+n2+" is not defined");
                }
                children.add(n);
                children.add(n2);
        }

        public String toString() {
                return ("" + children.get(0) + ".add(" + children.get(1) + ")");
        }
}

/* Matt Kim */
public class AppendStatementNode extends Node{

        public AppendStatementNode(IdentifierNode id, IdentifierNode id2) {
                if (SymbolTable.ST.get(id.id) == null)
                        System.err.println("Symbol variable "+id+" is not defined");
                else if (SymbolTable.ST.get(id2.id) == null)
                        System.err.println("Symbol variable "+id2+" is not defined");
                else {
                        if (!SymbolTable.ST.get(id.id).equals("Midi"))
                                System.err.println("Error: Can only append Midi objects");
                        else {
                                if (!(SymbolTable.ST.get(id2.id).equals("Sequence")))
                                        System.err.println("Error: Can only append Sequences to Midi
objects");
                        }
                }
                children.add(id);
                children.add(id2);
        }

        public AppendStatementNode(IdentifierNode id, InitializationNode init) {
                if (SymbolTable.ST.get(id.id) == null)
                        System.err.println("Symbol variable "+id+" is not defined");
                else {
                        if (!SymbolTable.ST.get(id.id).equals("Midi"))
                                System.err.println("Error: Can only append to Midi objects");
                        else {
                                if (!(init.type.equals("Sequence")))
                                        System.err.println("Error: Can only append Sequences to Midi
objects");
                        }
                }
                children.add(id);
                children.add(init);
```

```
        }



        public String toString() {
                return (""+children.get(0)+".append("+children.get(1)+")");
        }

}
/* Matt Kim */
public class AssignUpdateNode extends Node{

 public AssignUpdateNode(StringNode n) {
                children.add(n);
        }

        public String toString() {

                return ""+children.get(0);
        }

}

/*Author Kevin Walters*/

public class AssignmentNode extends Node{

        public String id;

        public AssignmentNode (IdentifierNode iden, ExpressionNode exp) {
                children.add(iden);
                children.add(exp);
                this.id=iden.id;
                if (SymbolTable.ST.get(iden.id) != null) {
                        if (!SymbolTable.ST.get(iden.id).equals(exp.type) &&
!(SymbolTable.ST.get(iden.id).equals("decimal")&&exp.type.equals("int")))
                                System.err.println("Error: cannot assign type " + exp.type  + " to type "
+ SymbolTable.ST.get(iden.id));
                }
                else
                        System.err.println("Symbol variable "+iden+"is not defined");

        }

        public AssignmentNode (IdentifierNode iden, AssignUpdateNode au, ExpressionNode exp) {
                children.add(iden);
                children.add(au);
                children.add(exp);
                this.id=iden.id;
                if (SymbolTable.ST.get(iden.id) != null) {
                        if (!SymbolTable.ST.get(iden.id).equals(exp.type)) {
                                if
(!(SymbolTable.ST.get(iden.id).equals("decimal")&&exp.type.equals("int")))
                                        if
(!(SymbolTable.ST.get(iden.id).equals("int")&&exp.type.equals("decimal")))
```

```
                                                if
(!(SymbolTable.ST.get(iden.id).equals("String")&&(exp.type.equals("decimal")||exp.type.equals("int"))))
                                                        System.err.println("Error: cannot assign type " +
SymbolTable.ST.get(iden.id) + " to type " + exp.type);
                        }
                }
                else
                        System.err.println("Error: Symbol variable "+iden+" is not defined");
        }

        public AssignmentNode (IdentifierNode iden, InitializationNode init) {
                children.add(iden);
                children.add(init);
                this.id=iden.id;
                if (SymbolTable.ST.get(iden.id) != null) {
                        if (!SymbolTable.ST.get(iden.id).equals(init.type)) {

                                System.err.println("Error: cannot assign type " +
SymbolTable.ST.get(iden.id) + " to type " + init.type);
                        }
                }
                else
                        System.err.println("Error: Symbol variable "+iden+" is not defined");
        }

        public String toString() {

                if (children.size() == 2)
                        return (""+children.get(0)+" = "+children.get(1));
                else
                        return (""+children.get(0)+" "+children.get(1)+" "+children.get(2));
        }

}

/* Matt Kim */
public class BooleanNode extends Node {

        public boolean value;
        public String type;

        public BooleanNode(boolean b) {
                value = b;
                type = "boolean";
        }

        public String toString() {
                return (value+"");
        }

}

/* author Kevin Walters */

public class ClearStatementNode extends Node {
```

```
	public ClearStatementNode (IdentifierNode id) {
		children.add(id);
		if (SymbolTable.ST.get(id.id) != null) {
			if
(!((SymbolTable.ST.get(id.id).equals("Midi"))||(SymbolTable.ST.get(id.id).equals("Sequence"))||(SymbolTabl
e.ST.get(id.id).equals("Track"))))
				System.err.println("Error: Cannot clear a "+SymbolTable.ST.get(id.id)+".
Valid objects are Midi, Sequence and Track");
		}
		else {
			System.err.println("Symbol variable "+id+" is not defined");
		}

	}

	public String toString() {
		return (""+children.get(0)+".clear()");
	}

}

/* Matt Kim */
public class CompoundStatementNode extends Node {

	public CompoundStatementNode(StatementBlockNode n) {
		children.add(n);
	}
	public String toString() {
		return "{" + children.get(0).toString() + "}";
	}

}

/* Cindy */
public class DecimalNode extends Node {

	public double dec;

	public DecimalNode(double d) {
		dec = d;
	}

	public String toString() {
		return (Double.toString(dec));
	}
}

/* author Kevin Walters */


public class DeclarationInitializationNode extends Node{

	String type;
```

```java
	/**
	 * constructor for a type and an initialization
	 */
	public DeclarationInitializationNode(TypeNode type, IdentifierNode id, InitializationNode init) {
		children.add(type);
		children.add(id);
		children.add(init);
		if (SymbolTable.ST.get(id.id) != null)
			System.err.println("Identifier "+id.id+" is already defined");

		else if (!(init.type.equals(type.type))){
			System.err.println("Identifier "+id.id+" is defined for type "+type+", but is
being assigned to "+init.type+".");
		}
		else {
			SymbolTable.ST.put(id.id, type+"");
		}
	}

	public String toString() {
		return (""+children.get(0)+" "+children.get(1)+" = "+children.get(2));
	}

}

/* Kevin Walters*/
import java.util.StringTokenizer;
public class DeclarationNode extends Node {

	public String id;

	public DeclarationNode(TypeNode n1, DeclaratorListNode n2) {
		children.add(n1);
		children.add(n2);
		id = n2.id;
		if (!(n2.type.equals(n1.type)) && !n2.type.equals("dummy")){
			System.err.println("Error: Identifier "+id+" is defined for type "+n1.type+", but
is being assigned to "+n2.type+".");
		}

		StringTokenizer st = new StringTokenizer(id, ",");
		String temp;
		while (st.hasMoreTokens()) {
		temp = st.nextToken();
		if (SymbolTable.ST.get(temp) != null)
				System.err.println("Error: Identifier "+id+" is already defined");
			else
				SymbolTable.ST.put(id, n1+"");
		}
	}

	public DeclarationNode() {

	}
```

```
	public String toString() {
		String s = "";
		s += children.get(0).toString() + " " + children.get(1).toString();
		return s;
	}

}

/* CIndy */

public class DeclaratorListNode extends Node {

	public String id;
	public String type;

	public DeclaratorListNode(DeclaratorListNode n, DeclaratorNode n2) {
		children.add(n);
		children.add(n2);
		if (!n.type.equals(n2.type) && !n2.type.equals("dummy") && !n.type.equals("dummy"))
			System.err.println("Error: Inconsistent types");
		type = n2.type;
		id = n.id+","+n2.id;
	}

	public DeclaratorListNode(DeclaratorNode n) {
		children.add(n);
		id = n.id;
		type = n.type;
	}

	public String toString() {
		if(children.size() == 1)
		    return children.get(0).toString();
		else
			return children.get(0) + ", " +  children.get(1);

	}
}

/* Matt Kim */
public class DeclaratorNode extends Node {

	public String id;
	public String type;

	public DeclaratorNode(IdentifierNode n) {
		children.add(n);
		id = n.id;
		type = "dummy";
	}

	public DeclaratorNode(IdentifierNode n1, ExpressionNode n2) {
		children.add(n1);
		children.add(n2);
```

```
			id = n1.id;
			type = n2.type;
		}

		public String toString() {
			if (children.size() == 2)
				return children.get(0) + " = " + children.get(1).toString();
			return children.get(0).toString();
		}

}

/**Author Kevin Walters*/

public class DurationNode extends Node {

		public String type = "";

		public DurationNode (StringNode d) {
			children.add(d);
			type = "int";
		}

		public String toString() {
			return ("Note."+children.get(0).toString());
		}
}

/* Cindy */

public class ExpressionNode extends Node {

		public String type = "";
		public String id = "";
		public String op = "";

		public ExpressionNode (ExpressionNode n) {
			children.add(n);
			type = n.type;
		}

		public ExpressionNode (IntegerNode n) {
			children.add(n);
			type = "int";
		}

		public ExpressionNode (DecimalNode n) {
			children.add(n);
			type = "decimal";
		}

		public ExpressionNode(IdentifierNode n) {
			children.add(n);
			id = n.id;
			type = SymbolTable.ST.get(n.id);
```

```
	}
	public ExpressionNode(BooleanNode n) {
		children.add(n);
		type = "boolean";
	}

	public ExpressionNode(StringNode n) {
		children.add(n);
		type = "string";
	}

	/*postfix expression*/
	public ExpressionNode(PostfixExpressionNode n) {
		children.add(n);
		type = n.type;
	}

	/*prefix expressions*/
	public ExpressionNode(PrefixExpressionNode n) {
		children.add(n);
		type = n.type;
	}

	public ExpressionNode(DurationNode n) {
		children.add(n);
		type = n.type;
	}

	public ExpressionNode(PitchNode n) {
		children.add(n);
		type = "int";
	}

	public ExpressionNode (ExpressionNode n1, String operator, ExpressionNode n3) {
		children.add(n1);
		op = operator;
		children.add(n3);
		if (!n1.type.equals(n3.type) && (n1.type.equals("int") && n3.type.equals("decimal") ||
n1.type.equals("decmial") && n3.type.equals("int")
											|| n1.type.equals("string") &&
n3.type.equals("int") || n1.type.equals("int") && n3.type.equals("string")
											|| n1.type.equals("string") &&
n3.type.equals("decmial") || n1.type.equals("decimal") && n3.type.equals("string")))
			System.err.println("Error: Cannot perform the operation " + n1.type + " " + op +
" " + n3.type);
		else {
			if (op.equals("+")) {
				if (n1.type.equals("int") && n3.type.equals("int"))
					type = "int";
				else if((n1.type.equals("decimal") && n3.type.equals("int")))
					type = "decimal";
				else if((n1.type.equals("int") && n3.type.equals("decimal")))
					type = "decimal";
				else if
((n1.type.equals("decimal")||n1.type.equals("int"))&&n3.type.equals("string"))
					type = "string";
				else if
((n3.type.equals("decimal")||n3.type.equals("int"))&&n1.type.equals("string"))
```

```
                                        type = "string";
                                else
                                        System.err.println("Error: Cannot perform the operation 
"+n1.type+" "+op+" "+n3.type);
                        }

                        if (op.equals("-
")||op.equals("*")||op.equals("/")||op.equals("^")||op.equals("%")) {
                                if (n1.type.equals("int") && n3.type.equals("int"))
                                        type = "int";
                                else if((n1.type.equals("decimal") && n3.type.equals("int")))
                                        type = "decimal";
                                else if((n1.type.equals("int") && n3.type.equals("decimal")))
                                        type = "decimal";
                                else
                                        System.err.println("Error: Cannot perform the operation 
"+n1.type+" "+op+" "+n3.type);
                        }

                        if (op.equals("<")||op.equals(">")||op.equals("<=")||op.equals(">=")) {
                                if 
(n1.type.equals("boolean")||n1.type.equals("string")||n3.type.equals("boolean")||n3.type.equals("string"))
{
                                        System.out.println("Error: Cannot perform the operation 
"+n1.type+" "+op+" "+n3.type);
                                }
                                else
                                        type = "boolean";
                        }

                        if (op.equals("!=")||op.equals("==")) {
                                if (n1.type.equals("boolean")&& n3.type.equals("boolean"))
                                        type = "boolean";
                                else if 
((n1.type.equals("double")||n1.type.equals("int"))&&(n3.type.equals("double")||n3.type.equals("int")))
                                        type = "boolean";
                                else
                                        System.err.println("Error: Cannot perform the operation 
"+n1.type+" "+op+" "+n3.type);

                        }

                        if (op.equals ("and") || op.equals("or")) {
                                if (n1.type.equals("boolean") && n3.type.equals("boolean"))
                                        type = "boolean";
                        else
                                System.err.println("Error: Cannot perform the operation 
"+n1.type+" "+op+" "+n3.type);

                        }
                }

        }

        public String toString() {
                if(children.size() == 0)
```

```
                        return "";
                else if (children.size() == 1) {
                        if (children.get(0) instanceof ExpressionNode)
                                return ("("+children.get(0)+")");
                        else
                                return ""+children.get(0);
                }
                else {
                        if (op.equals("^")) {
                                if (type.equals("int"))
                                        return ("(int)Math.pow("+children.get(0)+",
"+children.get(1)+")");//for an exponent that is guaranteed to be an integer
                                else
                                        return ("Math.pow("+children.get(0)+",
"+children.get(1)+")");//for an exponent
                        }
                        else if (op.equals("and"))
                                return children.get(0) + " && " + children.get(1);
                        else if (op.equals("or"))
                                return children.get(0) + " || " + children.get(1);
                        else
                                return (""+children.get(0) + op + children.get(1));
                }
        }

}

/* Cindy */

public class ForStatementNode extends Node {

 public ForStatementNode(OptStatementNode n1, OptExpressionNode n2, OptStatementNode n3, FullStatementNode
n4) {
                children.add(n1);
                children.add(n2);
                children.add(n3);
                children.add(n4);
                if (!(n2.type.equals("") || n2.type.equals("boolean")))
                        System.err.println("Error: Second argument of a for statement must be of type
boolean");
        }

        public String toString() {
                return ("for (" + children.get(0) + "; " + children.get(1) + "; " + children.get(2) + ") "
+ children.get(3) +"");
        }
}


/*author Kevin Walters*/

public class FullStatementNode extends Node{

        public FullStatementNode(StatementNode n) {
```

```java
		children.add(n);
	}

	public FullStatementNode(IfStatementNode n) {
		children.add(n);
	}

	public FullStatementNode(WhileStatementNode n) {
		children.add(n);
	}

	public FullStatementNode(ForStatementNode n) {
		children.add(n);
	}

	public FullStatementNode(CompoundStatementNode n) {
		children.add(n);
	}

	public String toString() {
		if (children.get(0) instanceof StatementNode)
			return ""+children.get(0)+";";
		else
			return ""+children.get(0);
	}
}

/*Author Kevin Walters*/

public class GetSequenceNode extends Node{

	public String type = "Sequence";

	GetSequenceNode(IdentifierNode n) {
		if (SymbolTable.ST.get(n.id) != null) {
			if (!SymbolTable.ST.get(n.id).equals("Midi"))
				System.err.println("Error: Can only call getSequence on a Midi object");
		}
		else
			System.err.println("Error: Identifier "+n+" is not defined");
		children.add(n);
	}

	GetSequenceNode(InitializationNode init) {
		if (!init.type.equals("Midi"))
			System.err.println("Error: Can only call getSequence on a Midi object");
		children.add(init);
	}

	public String toString() {
		return (children.get(0)+".getSequence()");
	}

}
```

```
/*Author Kevin Walters*/

public class IdentifierNode extends Node {

   public String id;

   public IdentifierNode(String identifierName) {
       id = identifierName;
   }

   public String toString() {
       return id;
   }

}

/*Author Kevin Walters*/

public class IfStatementNode extends Node{

        public IfStatementNode(ExpressionNode s, FullStatementNode n, FullStatementNode n2) {
                children.add(s);
                children.add(n);
                children.add(n2);
                if (!s.type.equals("boolean"))
                        System.err.println("Error: The condition of an if statement must be of type 
boolean");
        }

        public IfStatementNode(ExpressionNode s, FullStatementNode n) {
                children.add(s);
                children.add(n);
                if (!s.type.equals("boolean"))
                        System.err.println("Error: The condition of an if statement must be of type 
boolean");
        }

        public String toString() {
                if (children.size() == 3)
                        return ("if ("+children.get(0)+") "+children.get(1)+" else "+children.get(2));
                else
                        return ("if ("+children.get(0)+") "+children.get(1));
        }

}

/* Matt Kim, Kevin Walters */
public class InitializationNode extends Node{

        public String type = "";

        // for parentheses
        public InitializationNode (InitializationNode node) {
                children.add(node);
                type = node.type;
```

```
	}

	// for midi file
	public InitializationNode (TypeNode name, StringNode filename) {
		children.add(name);
		StringNode temp = new StringNode("\"../" + filename.toString().substring(1,
filename.toString().length()));
		children.add(temp);
		type = name.type;
	}

	// for a sequence
	public InitializationNode (TypeNode name) {
		children.add(name);
		type = name.type;
	}

	// for a sequence also
	public InitializationNode (TypeNode name, IdentifierNode id) {
		children.add(name);
		children.add(id);
		type = name.type;
	}

	// for a note
	public InitializationNode (TypeNode name, ExpressionNode pitch, ExpressionNode number,
ExpressionNode dur) {
		children.add(name);
		children.add(pitch);
		children.add(number);
		children.add(dur);
		if (!number.type.equals("int"))
			System.err.println("Error: Velocity of a Note must be of type int");
		if (!dur.type.equals("int"))
			System.err.println("Error: Duration of a Note must be of type int, or a duration
name");
		type = name.type;
	}

	// for a chord
	public InitializationNode (TypeNode name, PitchListNode args, ExpressionNode number,
ExpressionNode dur) {
		children.add(name);
		children.add(args);
		children.add(number);
		children.add(dur);
		if (!number.type.equals("int"))
			System.err.println("Error: Velocity of a Chord must be of type int");
		if (!dur.type.equals("int"))
			System.err.println("Error: Duration of a Chord must be of type int, or a duration
name");
		type = name.type;
	}

	public InitializationNode (GetSequenceNode n) {
		children.add(n);
		type = n.type;
	}
```

```
	public String toString() {
		if (children.size() == 1) {
			if (children.get(0) instanceof InitializationNode)
				//initialiaztion is put into parentheses
				return ("("+children.get(0)+")");
			else if (children.get(0) instanceof TypeNode)
				//new sequence() or new track()
				return ("new " + children.get(0)+"()");
			else
				//getsequence()
				return (children.get(0).toString());
		}
		else if (children.size() == 2) {
			//filename or an identifier
			return ("new "+children.get(0)+"("+children.get(1)+")");
		}
		else {
			if (children.get(1).children.get(0) instanceof PitchNameNode)
				//pitch is a pitch name, e.g. 'C4'
				return ("new "+children.get(0)+"(\"" + children.get(1) + "\",
"+children.get(2)+", "+children.get(3)+")");
			else
				//pitch is an integer
				return ("new "+children.get(0)+"(" + children.get(1) + ",
"+children.get(2)+", "+children.get(3)+")");
		}
	}

}

/**Author Kevin Walters*/

public class IntegerNode extends Node {

	public int value;

	public IntegerNode(int v) {
		value = v;
	}

	public String toString() {
		return (value+"");
	}

}

/* Cindy */

public class OptExpressionNode extends Node{

 public String type = "";

 public OptExpressionNode(ExpressionNode n) {
   children.add(n);
   type = n.type;
```

```
 }

 public OptExpressionNode() {
   type = "";

 }

 public String toString() {
   if(children.size() == 1)
     return (""+children.get(0));
   else
     return "";
 }

}

/* Cindy */

public class OptStatementNode extends Node{

 public OptStatementNode(StatementNode n) {
                children.add(n);
        }

        public OptStatementNode() {

        }

        public String toString() {
                if (children.size()==1)
                        return (""+children.get(0));
                else
                        return "";
        }

}

/** Cindy **/

public class PitchListNode extends Node {

        public PitchListNode (PitchNameNode n, PitchListNode n2) {
                children.add(n);
                children.add(n2);
        }

        public PitchListNode (PitchNameNode n) {
                children.add(n);
        }

        public String toString() {
                if (children.size() == 1)
                        return children.get(0).toString();
                else
```

```
                        return children.get(0) + " " + children.get(1);
        }
}

/**Author Kevin Walters*/

public class PitchNameNode extends Node {

        public String value;
        public String type = "";

        public PitchNameNode (String name) {
                value = name;
                type = "int";
        }

        public String toString() {
                return value.substring(1,value.length()-1);
        }

}

/**Author Kevin Walters*/

public class PitchNode extends Node{

        public String type = "";

        public PitchNode (ExpressionNode node) {
                children.add(node);
                if (!node.type.equals("int"))
                        System.err.println("Error: Pitch must be of type int or a note name");
                type = node.type;
        }

        public PitchNode (PitchNameNode node) {
                children.add(node);
                type = node.type;
        }

        public String toString() {
                if (children.get(0) instanceof PitchNameNode)
                        return "Note.stringToPitch(\"" + children.get(0).toString() + "\")";
                return (children.get(0)+"");
        }

}

/* Cindy */

public class PlayStatementNode extends Node {
   public PlayStatementNode(InitializationNode n) {
                children.add(n);
        }
   public PlayStatementNode(IdentifierNode n) {
```

```
        children.add(n);
                if (SymbolTable.ST.get(n.id) != null) {
                        if (!(SymbolTable.ST.get(n.id).equals("Midi")||
SymbolTable.ST.get(n.id).equals("Track") || SymbolTable.ST.get(n.id).equals("Note") ||
SymbolTable.ST.get(n.id).equals("Chord") || SymbolTable.ST.get(n.id).equals("Sequence")))
                                System.err.println("Error: Play statement expected to be performed on a
Midi, Note, Chord, Sequence, or Track object");
                }
                else
                        System.err.println("Symbol variable "+n.id+" is not defined");


        }

   public PlayStatementNode(StringNode n) {
       children.add(n);
        }



        @Override
        public String toString() {
                return ("Player.play(" + children.get(0) + ")");
        }

}

/* Cindy */

public class PostfixExpressionNode extends Node {

        public String op;
        public String type = "";

        public PostfixExpressionNode (IdentifierNode n1, StringNode n2) {
                children.add(n1);
                children.add(n2);
                if
(!(SymbolTable.ST.get(n1.toString()).equals("int")||SymbolTable.ST.get(n1.toString()).equals("decimal")))
                        System.err.println("Error: Can only perform the operation "+ n2 +" on type int or
decimal");
                type = SymbolTable.ST.get(n1.id);
        }

        public PostfixExpressionNode (IdentifierNode n1, String n2) {
                children.add(n1);
                op = n2;
                if
(!(SymbolTable.ST.get(n1.toString()).equals("int")||SymbolTable.ST.get(n1.toString()).equals("decimal")))
                        System.err.println("Error: Can only perform the operation "+ n2 +" on type int or
decimal");
                type = SymbolTable.ST.get(n1.id);
        }

        public String toString() {
                return ""+children.get(0)+op;
        }
}
```

```
/* Cindy */

public class PrefixExpressionNode extends Node {

        public String op;
        public String type = "";

        /*for unary plus, minus, and negate*/
        public PrefixExpressionNode (StringNode n1, ExpressionNode n2) {
                children.add(n1);
                children.add(n2);
                op = n1.toString();
                if (op.equals("+")||op.equals("-")) {
                        if (!(n2.type.equals("int")||n2.type.equals("decimal")))
                                System.err.println("Error: Can only perform the unary operation "+ n1 +"
on type int or decimal");
                        if (op.equals("!")) {
                                if (!n2.type.equals("boolean"))
                                        System.err.println("Error: Can only perform the unary operation
"+ n1 +" on type boolean");
                        }
                }
                type = n2.type;
        }

        /*for ++id and --id*/
        public PrefixExpressionNode (StringNode n1, IdentifierNode n2) {
                children.add(n1);
                children.add(n2);
                if
(!(SymbolTable.ST.get(n2.toString()).equals("int")||SymbolTable.ST.get(n2.toString()).equals("decimal")))
                        System.err.println("Error: Can only perform the operation "+ n1 +" on type int or
decimal");
                type = SymbolTable.ST.get(n2.id);
        }

        /*for unary plus, minus, and negate*/
        public PrefixExpressionNode (String n1, ExpressionNode n2) {
                op = n1;
                children.add(n2);
                if (op.equals("+")||op.equals("-")) {
                        if (!(n2.type.equals("int")||n2.type.equals("decimal")))
                                System.err.println("Error: Can only perform the unary operation "+ n1 +"
on type int or decimal");
                        if (op.equals("!")) {
                                if (!n2.type.equals("boolean"))
                                        System.err.println("Error: Can only perform the unary operation
"+ n1 +" on type boolean");
                        }
                }
                type = n2.type;
        }

        /*for ++id and --id*/
        public PrefixExpressionNode (String n1, IdentifierNode n2) {
                op = n1;
```

```
                children.add(n2);
                if
(!(SymbolTable.ST.get(n2.toString()).equals("int")||SymbolTable.ST.get(n2.toString()).equals("decimal")))
                        System.err.println("Error: Can only perform the operation "+ n1 +" on type int or
decimal");
                type = SymbolTable.ST.get(n2.id);
        }


        public String toString() {
                return ""+op+children.get(0);
        }
}

/* Cindy */

public class PrintStatementNode extends Node {

        //type checking: can't print a Midi
        public PrintStatementNode(ExpressionNode n) {
                children.add(n);
                if (n.type.equals("Midi"))
                        System.err.println("Error: Cannot print a Midi file");
        }

        //anything but a Midi
        public PrintStatementNode(InitializationNode n) {
                children.add(n);
                if (n.type.equals("Midi"))
                        System.err.println("Error: Cannot print a Midi file");
        }

        public PrintStatementNode() {

        }

        public String toString() {
                if (children.get(0)!= null)
                        return ("System.out.println(" + children.get(0) + ")");
                else
                        return ("System.out.println()");
        }
}

/* Matt Kim */
public class SetInstrumentNode extends Node {

/* will need to do type checking for this constructor when we add in option to set instrument name instead
of just instrument number*/
   public SetInstrumentNode(IdentifierNode id, IdentifierNode id2) {
         children.add(id);
         children.add(id2);
         if (SymbolTable.ST.get(id.id)==null)
           System.err.println("Symbol variable "+id+"is not defined");
       else if (SymbolTable.ST.get(id2.id)==null)
           System.err.println("Symbol variable "+id2+"is not defined");
```

```
        else {
            if (!(SymbolTable.ST.get(id.id).equals("Track")))
                  System.err.println("Error: SetInstrument is expected to be performed on a Track object");
            else if (!(SymbolTable.ST.get(id2.id).equals("int")))
                  System.err.println("Error: SetInstrument is expected to take an int");
        }

    }

    public SetInstrumentNode(IdentifierNode id, ExpressionNode ex) {
          children.add(id);
          children.add(ex);

            if (SymbolTable.ST.get(id.id) != null) {
              if (!(SymbolTable.ST.get(id.id).equals("Track")))
                  System.err.println("Error: SetInstrument is expected to be performed on a Track object");
              else if (!ex.type.equals("int"))
                  System.err.println("Error: SetInstrument is expected to take an int");
            }
            else
                System.err.println("Symbol variable "+id+ "is not defined");
    }

    public String toString() {
          return ""+children.get(0)+".setInstrument("+children.get(1)+")";
    }

}

/* Matt Kim */
public class SetTempoNode extends Node{

    public SetTempoNode(IdentifierNode id, ExpressionNode ex) {
          children.add(id);
          children.add(ex);

            if (SymbolTable.ST.get(id.id) != null) {
              if (!(SymbolTable.ST.get(id.id).equals("Sequence")))
                  System.err.println("Error: SetTempo is expected to be performed on a Sequence object");
              else if (!ex.type.equals("int") && !ex.type.equals("decimal"))
                  System.err.println("Error: SetTempo is expected to receive an integer or decimal");
            }
            else
             System.err.println("Symbol variable "+id+ "is not defined");
    }

    public String toString() {
          return ""+children.get(0)+".setTempoInBPM("+children.get(1)+")";
    }
}

/* Cindy */

public class StatementBlockNode extends Node {

 public StatementBlockNode (FullStatementNode n1, StatementBlockNode n2) {
```

```java
    children.add(n1);
    if (n2 != null)
      children.add(n2);
  }

  public StatementBlockNode (FullStatementNode n) {
    children.add(n);
  }

  public String toString() {
    if (children.size() == 1)
        return (children.get(0) + "");
    else
      return (children.get(0) + "\n" + children.get(1));
  }
}

/**Author Kevin Walters*/

public class StatementNode extends Node {

        public StatementNode (GetSequenceNode n) {
                children.add(n);
        }

        public StatementNode (AdditionNode n) {
                children.add(n);
        }

        public StatementNode (PlayStatementNode n) {
                children.add(n);
        }

        public StatementNode (PrintStatementNode n) {
                children.add(n);
        }

        public StatementNode (DeclarationNode n) {
                children.add(n);
        }

        public StatementNode (DeclarationInitializationNode n) {
                children.add(n);
        }
        public StatementNode(WriteStatementNode n) {
                children.add(n);
        }
        public StatementNode(AppendStatementNode n) {
                children.add(n);
        }
        public StatementNode(ClearStatementNode n) {
                children.add(n);
        }

        public StatementNode(ExpressionNode n) {
                children.add(n);
```

```
        }

        public StatementNode(AssignmentNode n) {
                children.add(n);
        }

        public StatementNode (TransposeStatementNode n) {
                children.add(n);
        }

        public StatementNode (SetInstrumentNode n) {
                children.add(n);
        }

        public StatementNode (SetTempoNode n) {
                children.add(n);
        }

        public String toString() {
                return (""+children.get(0));
        }

}

/**Author Kevin Walters*/

public class StringNode extends Node{

        String value;

        public StringNode(String s) {
                value = s;
        }

        public String toString() {
                return value;
        }
}

/* Matt Kim */
public class TransposeStatementNode extends Node{

 public TransposeStatementNode(IdentifierNode id, ExpressionNode exp) {
         children.add(id);
         children.add(exp);

           if (SymbolTable.ST.get(id.id) != null) {
             if
(!((SymbolTable.ST.get(id.id).equals("Track"))||(SymbolTable.ST.get(id.id).equals("Note"))||(SymbolTable.S
T.get(id.id).equals("Chord"))||(SymbolTable.ST.get(id.id).equals("Sequence"))))
                 System.err.println("Error: Transpose expected to be performed on a Note, Chord, Sequence,
or Track object");
             else if (!exp.type.equals("int"))
                 System.err.println("Error: Transpose expected to be performed on integer");
           }
```

```java
           else
             System.err.println("Symbol variable "+id+"is not defined");
 }
 public String toString() {
       return ""+children.get(0)+".transpose("+children.get(1)+")";
 }

}

/*Author Kevin Walters*/

public class TypeNode extends Node {

        String type;

        public TypeNode (String t) {
                type = t;
                setType();
        }

        public void setType() {
                if (type.equals("pitch") || type.equals("velocity") || type.equals("duration") ||
type.equals("int") || type.equals("instrument"))
                        type = "int";

        }

        public String toString() {

                if (type.equals("pitch") || type.equals("velocity") || type.equals("duration") ||
type.equals("int")|| type.equals("instrument"))
                        return "int";
                if (type.equals("decimal"))
                        return "double";
                if (type.equals("boolean"))
                        return "boolean";
                if (type.equals("string"))
                        return "String";
                /* if (type.equals("void"))
                        return "void"; */
                if (type.equals("Note"))
                        return "Note";
                if (type.equals("Sequence"))
                        return "Sequence";
                if (type.equals("Midi"))
                        return "Midi";
                if (type.equals("Track"))
                        return "Track";
                return (type);
        }

}

/* Cindy */

public class WhileStatementNode extends Node {
```

```
 public WhileStatementNode(ExpressionNode n1, FullStatementNode n2) {
		children.add(n1);
		children.add(n2);
	}

	public String toString() {
		return ("while (" + children.get(0) +") " + children.get(1) );
	}

}

/* Author Kevin Walters */
public class WriteStatementNode extends Node{

	public WriteStatementNode(IdentifierNode id, IdentifierNode id2) {
		children.add(id);
		children.add(id2);
		if (SymbolTable.ST.get(id.id) != null) {
			if (!(SymbolTable.ST.get(id.id).equals("Midi")))
				System.err.println("Error: Write statement expected to be performed on a 
Midi object");
			if 
(!((SymbolTable.ST.get(id2.id).equals("Track"))||(SymbolTable.ST.get(id2.id).equals("Note"))||(SymbolTable
.ST.get(id2.id).equals("Chord"))||(SymbolTable.ST.get(id2.id).equals("Sequence"))))
				System.err.println("Error: Can only write a Note, Chord, Sequence or 
Track to a file");
		} else {
			System.err.println("Symbol variable "+id+" is not defined");
		}
	}

	public WriteStatementNode(IdentifierNode id, InitializationNode id2) {
		children.add(id);
		children.add(id2);
		if (SymbolTable.ST.get(id.id) != null) {
			if (!(SymbolTable.ST.get(id.id).equals("Midi")))
				System.err.println("Error: Write statement expected to be performed on a 
Midi object");
			if 
(!((id2.type.equals("Track"))||(id2.type.equals("Note"))||(id2.type.equals("Chord"))||(id2.type.equals("Se
quence"))))
				System.err.println("Error: Can only write a Note, Chord, Sequence or 
Track to a file");
		} else {
			System.err.println("Symbol variable "+id+" is not defined");
		}
	}

	public String toString() {
		return (""+children.get(0)+".writeToFile("+children.get(1)+")");
	}

}
```

The class holding the HashMap for the symbol table **SouL/AST/SymbolTable.java**:

```
/* Andrew Goldin */
import java.util.HashMap;

public class SymbolTable {

        public static HashMap<String, String> ST = new HashMap<String, String>();

        public SymbolTable () {

        }

}
```

All classes for JSouL, the proprietary Java MIDI API written by Andrew:

SouL/jsoul/Playable.java:

```
public interface Playable {

        public void transpose(int steps);

}
```

SouL/jsoul/Note.java:

```
public class Note implements Playable, Comparable<Note> {

        public static final int WHOLE = 64,
                        HALF = 32,
                        QUARTER = 16,
                        EIGHTH = 8,
                        SIXTEENTH = 4,
                        THIRTYSECOND = 2,
                        SIXTYFOURTH = 1;

        public static final String[] KEYNAMES =
                { "C", "C#", "D", "Eb", "E", "F", "F#", "G", "G#", "A", "Bb", "B" };

        private int pitch, velocity, duration;

        // duration is the number of 64th notes
        public Note(int p, int v, int d) {
                setNote(p, v, d);
        }

        public Note(String p, int v, int d) {
                setNote(stringToPitch(p), v, d);
```

```
}

public Note() {
        setNote(60, 85, WHOLE);
}

public void setNote(int p, int v, int d) {
        pitch = p;
        velocity = v;
        duration = d;
}

public void setNote(String p, int v, int d) {
        pitch = stringToPitch(p);
        velocity = v;
        duration = d;
}

public void setPitch(int p) {
        pitch = p;
}

public void setPitch(String p) {
        pitch = stringToPitch(p);
}

public void transpose(int steps) {
        if (pitch + steps >= 0 && pitch + steps <= 127) {
                pitch += steps;
        }
}

public int getPitch() {
        return pitch;
}

public String getPitchString() {
        return pitchToString(pitch);
}

public void setVelocity(int v) {
        velocity = v;
}

public int getVelocity() {
        return velocity;
}

public void setDuration(int d) {
        duration = d;
}

public int getDuration() {
        return duration;
}
```

```
	public static String pitchToString(int pitch) {
		return KEYNAMES[pitch % 12] + ((pitch / 12) - 1);
	}

	public static int stringToPitch(String s) {
		int pitch = 0, base = 0, multiplier = 0;
		String note = "";
		if (s.endsWith("-1")) {
			multiplier = 0;
			note = s.substring(0, s.length() - 2);
		}
		else {
			multiplier = Character.getNumericValue(s.charAt(s.length() - 1)) + 1;
			note = s.substring(0, s.length() - 1);
		}
		if (note.equals("C") || note.equals("B#")) base = 0;
		else if (note.equals("C#") || note.equals("Db")) base = 1;
		else if (note.equals("D")) base = 2;
		else if (note.equals("D#") || note.equals("Eb")) base = 3;
		else if (note.equals("E") || note.equals("Fb")) base = 4;
		else if (note.equals("F") || note.equals("Eb")) base = 5;
		else if (note.equals("F#") || note.equals("Gb")) base = 6;
		else if (note.equals("G")) base = 7;
		else if (note.equals("G#") || note.equals("Ab")) base = 8;
		else if (note.equals("A")) base = 9;
		else if (note.equals("A#") || note.equals("Bb")) base = 10;
		else if (note.equals("B") || note.equals("Cb")) base = 11;
		pitch = base + 12 * multiplier;
		if (pitch > 127) {
			System.err.println("Error: (Note) pitch out of range");
			System.exit(0);
		}
		return pitch;
	}

	public String toString() {
		return "NOTE: Pitch = " + pitchToString(pitch) + ", Velocity = " + velocity + ", Duration
= " + duration;
	}

	public int compareTo(Note other) {
		if (this.getPitch() > other.getPitch())
			return 1;
		else if (this.getPitch() == other.getPitch())
			return 0;
		else
			return -1;
	}

}
```

SouL/jsoul/Chord.java:

```
import java.util.*;
```

```java
public class Chord implements Playable, Comparable<Chord> {

        private int velocity, duration;
        private int[] pitches;

        public Chord() {
                pitches = new int[3];
                pitches[0] = 60; pitches[1] = 64; pitches[2] = 67;
                Arrays.sort(pitches);
                velocity = 85;
                duration = Note.WHOLE;
        }

        public Chord(int[] p, int v, int d) {
                pitches = p;
                Arrays.sort(pitches);
                velocity = v;
                duration = d;
        }

        public Chord(String pitchList, int v, int d) {
                String[] p = pitchList.split(" ");
                pitches = new int[p.length];
                for (int i = 0; i < pitches.length; i++) {
                        pitches[i] = Note.stringToPitch(p[i]);
                }
                Arrays.sort(pitches);
                velocity = v;
                duration = d;
        }

        public int getSize() {
                return pitches.length;
        }

        public void setPitches(int[] p) {
                pitches = p;
                Arrays.sort(pitches);
        }

        public int[] getPitches() {
                return pitches;
        }

        public int getHighestPitch() {
                return pitches[pitches.length - 1];
        }

        public int getLowestPitch() {
                return pitches[0];
        }

        public void transpose(int steps) {
                if (getLowestPitch() + steps >= 0 && getHighestPitch() + steps <= 127) {
                        for (int i = 0; i < pitches.length; i++) {
                                pitches[i] += steps;
```

```
                }
        }
}

public void setVelocity(int v) {
        velocity = v;
}

public int getVelocity() {
        return velocity;
}

public void setDuration(int d) {
        duration = d;
}

public int getDuration() {
        return duration;
}

public Note[] getNotes() {
        Note[] notes = new Note[pitches.length];
        for (int i = 0; i < notes.length; i++) {
                notes[i] = new Note(pitches[i], velocity, duration);
        }
        return notes;
}

public Note getNote(int i) {
        return getNotes()[i];
}

public String toString() {
        String s = "CHORD: Pitches =";
        for (int i = 0; i < pitches.length; i++) {
                s += " " + Note.pitchToString(pitches[i]);
        }
        s += ", Velocity = " + velocity + ", Duration = " + duration;
        return s;
}

public int compareTo(Chord other) {
        if (this.getSize() > other.getSize())
                return 1;
        else if (this.getSize() == other.getSize())
                return 0;
        else
                return -1;
}
```

```
}
```

SouL/jsoul/Track.java:

```
import javax.sound.midi.*;
```

```java
import java.util.*;

public class Track implements Playable {

	private ArrayList<Playable> elements;
	private Instrument instrument;

	public Track() {
		elements = new ArrayList<Playable>();
		instrument = new Instrument(Instrument.PIANO);
	}

	public Track(Playable... p) {
		elements = new ArrayList<Playable>();
		for (int i = 0; i < p.length; i++) {
			add(p[i]);
		}
		instrument = new Instrument(Instrument.PIANO);
	}

	public void add(Playable p) {
		if (p instanceof Note || p instanceof Chord) {
			elements.add(p);
		}
		else if (p instanceof Track) {
			Track s = (Track) p;
			Playable[] temp = s.getElements();
			for (int i = 0; i < temp.length; i++) {
				elements.add(temp[i]);
			}
		}
	}

	public void set(int index, Playable p) {
		if (p instanceof Note || p instanceof Chord) {
			elements.set(index, p);
		}
	}

	public void remove(int index) {
		elements.remove(index);
	}

	public void clear() {
		elements.clear();
		instrument = new Instrument(Instrument.PIANO);
	}

	public int getNumElements() {
		return elements.size();
	}

	public Playable[] getElements() {
		return elements.toArray(new Playable[elements.size()]);
	}
```

```
	public Playable getElement(int n) {
		return elements.get(n);
	}

	public void setInstrument(Instrument inst) {
		instrument = inst;
	}

	public void setInstrument(int instNum) {
		instrument = new Instrument(instNum);
	}

	public Instrument getInstrument() {
		return instrument;
	}

	public void transpose(int steps) {
		for (int i = 0; i < elements.size(); i++) {
			elements.get(i).transpose(steps);
		}
	}

	public String toString() {
		String s = "TRACK: Elements = " + elements.size() + ", Instrument = " +
instrument.toString() + "\n";
		for (int i = 0; i < elements.size(); i++) {
			s += "    " + elements.get(i).toString() + "\n";
		}
		s += "END TRACK";
		return s;
	}

}
```

SouL/jsoul/Sequence.java:

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

public class Sequence implements Playable {

	private ArrayList<Track> tracks;
	private float tempo;

	public Sequence() {
		tracks = new ArrayList<Track>();
		tempo = 120;
	}

	public Sequence(Track... t) {
		setTracks(t);
		tempo = 120;
	}

	// adds a track to the sequence
```

```java
public void add(Track t) {
        tracks.add(t);
}

public void setTrack(int trackNum, Track t) {
        tracks.set(trackNum, t);
}

public void setTracks(Track... t) {
        tracks = new ArrayList<Track>();
        for (int i = 0; i < t.length; i++) {
                add(t[i]);
        }
}

public void removeTrack(int trackNum) {
        tracks.remove(trackNum);
}

public void clear() {
        tracks.clear();
        tempo = 120;
}

public int getNumTracks() {
        return tracks.size();
}

public Track[] getTracks() {
        return tracks.toArray(new Track[tracks.size()]);
}

public Track getTrack(int n) {
        return tracks.get(n);
}

public void setTempoInBPM(float tempoBPM) {
        tempo = tempoBPM;
}

public float getTempo() {
        return tempo;
}

public void transpose(int steps) {
        for (int i = 0; i < tracks.size(); i++) {
                tracks.get(i).transpose(steps);
        }
}

public javax.sound.midi.Sequence createMidiSequence() {
        javax.sound.midi.Sequence s = null;
        try {
                s = new javax.sound.midi.Sequence(javax.sound.midi.Sequence.PPQ, 16);
                int channelNum = 0;
                for (Track t : tracks) {
```

```
				int tickCount = 0;
				javax.sound.midi.Track javaTrack = s.createTrack();
				javaTrack.add(new MidiEvent(sequenceTempo(), tickCount)); // set tempo
				ShortMessage setInst = new ShortMessage();
				setInst.setMessage(ShortMessage.PROGRAM_CHANGE, channelNum,
t.getInstrument().getInstrumentNumber(), 0);
				javaTrack.add(new MidiEvent(setInst, tickCount)); // set instrument for
track
				for (int i = 0; i < t.getNumElements(); i++) {
					if (t.getElement(i) instanceof Note) {
						Note currentNote = (Note) t.getElement(i);
						ShortMessage onMessage = new ShortMessage();
						onMessage.setMessage(ShortMessage.NOTE_ON, channelNum,
currentNote.getPitch(), currentNote.getVelocity());
						javaTrack.add(new MidiEvent(onMessage, tickCount));
						tickCount += currentNote.getDuration();
						ShortMessage offMessage = new ShortMessage();
						offMessage.setMessage(ShortMessage.NOTE_OFF, channelNum,
currentNote.getPitch(), 0);
						javaTrack.add(new MidiEvent(offMessage, tickCount));
					}
					else if (t.getElement(i) instanceof Chord) {
						Chord currentChord = (Chord) t.getElement(i);
						Note[] noteList = currentChord.getNotes();
						for (int j = 0; j < noteList.length; j++) {
							ShortMessage onMessage = new ShortMessage();
							onMessage.setMessage(ShortMessage.NOTE_ON,
channelNum, noteList[j].getPitch(), noteList[j].getVelocity());
							javaTrack.add(new MidiEvent(onMessage,
tickCount));
						}
						tickCount += currentChord.getDuration();
						for (int j = 0; j < noteList.length; j++) {
							ShortMessage offMessage = new ShortMessage();
							offMessage.setMessage(ShortMessage.NOTE_OFF,
channelNum, noteList[j].getPitch(), 0);
							javaTrack.add(new MidiEvent(offMessage,
tickCount));
						}
					}
				}
				channelNum++;
			}
		} catch (InvalidMidiDataException e) {
			System.err.println("Error: (Sequence) failure to generate Java midi sequence");
			//e.printStackTrace();
		}
		return s;
	}

	private MetaMessage sequenceTempo() {
		// convert tempo to a byte array to be recognized by Midi system
		String hexString = Integer.toHexString(60000000 / (int) tempo);
		if (hexString.length() % 2 != 0) {
			hexString = "0" + hexString;
		}
		byte[] data = new java.math.BigInteger(hexString, 16).toByteArray();

		// create and return message with tempo change
```

```
            MetaMessage m = new MetaMessage();
            try {
                    // tempo change MetaMessage has command 81
                    m.setMessage(81, data, data.length);
            } catch (InvalidMidiDataException e) {
                    System.err.println("Error: (Sequence) cannot set tempo for file");
            }
            return m;
    }

    public String toString() {
            String s = "SEQUENCE: Tracks = " + tracks.size() + ", Tempo = " + tempo + " bpm\n";
            for (int i = 0; i < tracks.size(); i++) {
                    s += tracks.get(i).toString() + "\n";
            }
            s += "END SEQUENCE";
            return s;
    }

}
```

SouL/jsoul/Instrument.java:

```
import javax.sound.midi.*;

public class Instrument {

        public static final int PIANO = 0,
                        BRIGHT_PIANO = 1,
                        ELECTRIC_GRAND = 2,
                        HONKY_TONK_PIANO = 3,
                        ELECTRIC_PIANO_1 = 4,
                        ELECTRIC_PIANO_2 = 5,
                        HARPSICHORD = 6,
                        CLAVINET = 7,
                        CELESTA = 8,
                        GLOCKENSPIEL = 9,
                        MUSIC_BOX = 10,
                        VIBRAPHONE = 11,
                        MARIMBA = 12,
                        XYLOPHONE = 13,
                        TUBULAR_BELL = 14,
                        DULCIMER = 15,
                        HAMMOND_ORGAN = 16,
                        PERC_ORGAN = 17,
                        ROCK_ORGAN = 18,
                        CHURCH_ORGAN = 19,
                        REED_ORGAN = 20,
                        ACCORDION = 21,
                        HARMONICA = 22,
                        TANGO_ACCORDION = 23,
                        NYLON_STR_GUITAR = 24,
                        STEEL_STR_GUITAR = 25,
                        JAZZ_ELECTRIC_GTR = 26,
                        CLEAN_GUITAR = 27,
                        MUTED_GUITAR = 28,
                        OVERDRIVE_GUITAR = 29,
```

```
DISTORTION_GUITAR = 30,
GUITAR_HARMONICS = 31,
ACOUSTIC_BASS = 32,
FINGERED_BASS = 33,
PICKED_BASS = 34,
FRETLESS_BASS = 35,
SLAP_BASS_1 = 36,
SLAP_BASS_2 = 37,
SYN_BASS_1 = 38,
SYN_BASS_2 = 39,
VIOLIN = 40,
VIOLA = 41,
CELLO = 42,
CONTRABASS = 43,
TREMOLO_STRINGS = 44,
PIZZICATO_STRINGS = 45,
ORCHESTRAL_HARP = 46,
TIMPANI = 47,
ENSEMBLE_STRINGS = 48,
SLOW_STRINGS = 49,
SYNTH_STRINGS_1 = 50,
SYNTH_STRINGS_2 = 51,
CHOIR_AAHS = 52,
VOICE_OOHS = 53,
SYN_CHOIR = 54,
ORCHESTRAL_HIT = 55,
TRUMPET = 56,
TROMBONE = 57,
TUBA = 58,
MUTED_TRUMPET = 59,
FRENCH_HORN = 60,
BRASS_ENSEMBLE = 61,
SYN_BRASS_1 = 62,
SYN_BRASS_2 = 63,
SOPRANO_SAX = 64,
ALTO_SAX = 65,
TENOR_SAX = 66,
BARITONE_SAX = 67,
OBOE = 68,
ENGLISH_HORN = 69,
BASSOON = 70,
CLARINET = 71,
PICCOLO = 72,
FLUTE = 73,
RECORDER = 74,
PAN_FLUTE = 75,
BOTTLE_BLOW = 76,
SHAKUHACHI = 77,
WHISTLE = 78,
OCARINA = 79,
SYN_SQUARE_WAVE = 80,
SYN_SAW_WAVE = 81,
SYN_CALLIOPE = 82,
SYN_CHIFF = 83,
SYN_CHARANG = 84,
SYN_VOICE = 85,
SYN_FIFTHS_SAW = 86,
SYN_BRASS_AND_LEAD = 87,
FANTASIA = 88,
WARM_PAD = 89,
```

```
				POLYSYNTH = 90,
				SPACE_VOX = 91,
				BOWED_GLASS = 92,
				METAL_PAD = 93,
				HALO_PAD = 94,
				SWEEP_PAD = 95,
				ICE_RAIN = 96,
				SOUNDTRACK = 97,
				CRYSTAL = 98,
				ATMOSPHERE = 99,
				BRIGHTNESS = 100,
				GOBLINS = 101,
				ECHO_DROPS = 102,
				SCI_FI = 103,
				SITAR = 104,
				BANJO = 105,
				SHAMISEN = 106,
				KOTO = 107,
				KALIMBA = 108,
				BAG_PIPE = 109,
				FIDDLE = 110,
				SHANAI = 111,
				TINKLE_BELL = 112,
				AGOGO = 113,
				STEEL_DRUMS = 114,
				WOODBLOCK = 115,
				TAIKO_DRUM = 116,
				MELODIC_TOM = 117,
				SYN_DRUM = 118,
				REVERSE_CYMBAL = 119,
				GUITAR_FRET_NOISE = 120,
				BREATH_NOISE = 121,
				SEASHORE = 122,
				BIRD = 123,
				TELEPHONE = 124,
				HELICOPTER = 125,
				APPLAUSE = 126,
				GUNSHOT = 127;

	private int instrument;

	public Instrument(int instNum) {
		instrument = instNum;
	}

	public int getInstrumentNumber() {
		return instrument;
	}

	public void setInstrument(int instNum) {
		instrument = instNum;
	}

	public String toString() {
		String s = "";
		try {
			s =
MidiSystem.getSynthesizer().getDefaultSoundbank().getInstruments()[instrument].getName();
```

```
		} catch (MidiUnavailableException e) {
			System.err.println("Error: (instrument) cannot retrieve instrument");
			//e.printStackTrace();
		}
		return s;
	}

}
```

SouL/jsoul/Player.java:

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

// static class for playing midi files or sequences and writing midi information to midi files
public class Player {

	private static Sequencer sequencer = null;

	/* PLAYING MIDI */

	public static void play(Note n) {
		play(new Track(n));
	}

	public static void play(Chord c) {
		play(new Track(c));
	}

	public static void play(Track t) {
		play(new Sequence(t));
	}

	public static void play(Sequence s) {
		initSequencer(s.createMidiSequence(), s.getTempo());
	}

	public static void play(Midi m) {
		play(m.getFileName());
	}

	public static void play(String fileName) {
		try {
			initSequencer(MidiSystem.getSequence(new File(fileName)), 0);
		} catch (InvalidMidiDataException e) {
			System.err.println("Error: (play) failure to retrieve midi data from " +
fileName);
			//e.printStackTrace();
		} catch (IOException e) {
			System.err.println("Error: (play) failure to read file " + fileName);
			//e.printStackTrace();
		}
	}

	private static void initSequencer(javax.sound.midi.Sequence s, float t) {
```

```
		if (sequencer != null) {
			// if the sequencer is already running, wait for it to stop
			while (sequencer.isRunning()) {
				try {
					Thread.sleep(200);
				} catch (InterruptedException e) {
					System.err.println("Error: (play) failure waiting to play next 
sequence");
					//e.printStackTrace();
				}
			}
		}
		try {
			sequencer = MidiSystem.getSequencer(); sequencer.open();
		} catch (MidiUnavailableException e) {
			System.err.println("Error: (play) cannot get MIDI Sequencer");
			//e.printStackTrace();
		}
		try {
			sequencer.setSequence(s);
		} catch (InvalidMidiDataException e) {
			System.err.println("Error: (play) cannot set sequence for MIDI sequencer");
			//e.printStackTrace();
		}
		if (t > 0) sequencer.setTempoInBPM(t); // set tempo if specified
		sequencer.addMetaEventListener(new MetaEventListener() {
			public void meta(MetaMessage m) {
				if (m.getType() == 47) { sequencer.stop(); sequencer.close(); }
			}
		});
		sequencer.start();
	}

}
```

SouL/jsoul/Midi.java:

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

public class Midi {

	private String name;

	public Midi() {
		name = "output.mid";
	}

	public Midi(String fileName) {
		name = fileName;
		if (!name.endsWith(".mid")) name += ".mid";
	}

	public void setFileName(String fileName) {
		name = fileName;
```

```
		if (!name.endsWith(".mid")) name += ".mid";
	}

	public String getFileName() {
		return name;
	}

	// WRITE NOTE TO FILE
	public void writeToFile(Note n) {
		writeToFile(new Sequence(new Track(n)));
	}

	// WRITE CHORD TO FILE
	public void writeToFile(Chord c) {
		writeToFile(new Sequence(new Track(c)));
	}

	// WRITE TRACK TO FILE
	public void writeToFile(Track t) {
		writeToFile(new Sequence(t));
	}

	// WRITE SEQUENCE TO FILE
	public void writeToFile(Sequence s) {
		javax.sound.midi.Sequence seq = s.createMidiSequence();
		try {
			MidiSystem.write(seq, 0, new File(name));
		} catch (IOException e) {
			System.err.println("Error: (Midi) failure to write sequence to file " + name);
			//e.printStackTrace();
		}
	}

	public void clear() {
		Track[] tracks = null;
		try {
			tracks = new Track[MidiSystem.getSequence(new File(name)).getTracks().length];
		} catch (InvalidMidiDataException e) {
			System.err.println("Error: (Midi) cannot clear MIDI file " + name);
			//e.printStackTrace();
		} catch (IOException e) {
			System.err.println("Error: (Midi) cannot clear MIDI file " + name);
			//e.printStackTrace();
		}
		for (int i = 0; i < tracks.length; i++) tracks[i] = new Track();
		Sequence s = new Sequence(tracks);
		writeToFile(s);
	}

	// this method will only work on files created with jsoul
	public Sequence getSequence() {
		javax.sound.midi.Sequence inputSequence = null;
		try {
			inputSequence = MidiSystem.getSequence(new File(name));
		} catch (InvalidMidiDataException e) {
			System.err.println("Error: (Midi) cannot retrieving java sequence from file " +
name);
```

```
                        //e.printStackTrace();
                } catch (IOException e) {
                        System.err.println("Error: (Midi) cannot retrieving java sequence from file " +
name);
                        //e.printStackTrace();
                }
                javax.sound.midi.Track[] inputTracks = inputSequence.getTracks();
                Track[] outputTracks = new Track[inputTracks.length];
                float tempo = 120;
                for (int i = 0; i < inputTracks.length; i++) {
                        javax.sound.midi.Track currentTrack = inputTracks[i];
                        outputTracks[i] = new Track();
                        int velocity = 0, chordSize = 0;
                        ArrayList<Integer> pitchList = new ArrayList<Integer>();
                        boolean chord = false;
                        long currentTick = 0;
                        for (int j = 0; j < currentTrack.size(); j++) {
                                MidiEvent event = currentTrack.get(j);
                                MidiMessage message = event.getMessage();
                                if (message instanceof ShortMessage) {
                                        ShortMessage sMessage = (ShortMessage) message;
                                        if (sMessage.getCommand() == ShortMessage.PROGRAM_CHANGE) {
                                                outputTracks[i].setInstrument(sMessage.getData1());
                                        }
                                        else if (sMessage.getCommand() == ShortMessage.NOTE_ON) {
                                                chord = true;
                                                velocity = sMessage.getData2();
                                                currentTick = event.getTick();
                                                chordSize++;
                                        }
                                        else if (sMessage.getCommand() == ShortMessage.NOTE_OFF) {
                                                pitchList.add(sMessage.getData1());
                                                chordSize--;
                                                if (chordSize == 0 && chord) {
                                                        if (pitchList.size() == 1) {
                                                                outputTracks[i].add(new
Note(pitchList.get(0), velocity, (int) (event.getTick() - currentTick)));
                                                        }
                                                        else {
                                                                int[] list = new int[pitchList.size()];
                                                                for (int k = 0; k < list.length; k++)
list[k] = pitchList.get(k).intValue();

                                                                outputTracks[i].add(new Chord(list,

velocity, (int) (event.getTick() - currentTick)));
                                                        }
                                                        chord = false;
                                                        pitchList.clear();
                                                }
                                        }
                                }
                                else if (message instanceof MetaMessage) {
                                        MetaMessage mMessage = (MetaMessage) message;
                                        if (mMessage.getType() == 81) { // check for set tempo message
                                                String hexString = new
java.math.BigInteger(mMessage.getData()).toString(16);
                                                tempo = 60000000 / Integer.parseInt(hexString, 16);
                                        }
                                }
                        }
                }
```

```
            Sequence s = new Sequence();
            s.setTracks(outputTracks);
            s.setTempoInBPM(tempo);
            return s;
        }

        public void append(Sequence s) {
            Track[] sourceTracks = this.getSequence().getTracks();
            Track[] inputTracks = s.getTracks();
            if (sourceTracks.length == inputTracks.length) {
                for (int i = 0; i < inputTracks.length; i++) {
                    sourceTracks[i].add(inputTracks[i]);
                }
            }
            else {
                System.err.println("Error: (Midi) could not append sequence: mismatching track
numbers");
                return;
            }
            String fname = name.substring(0, name.length() - 4) + "_appended.mid";
            javax.sound.midi.Sequence seq = s.createMidiSequence();
            try {
                MidiSystem.write(seq, 0, new File(fname));
            } catch (IOException e) {
                System.err.println("Error: (Midi) failure to write appended sequence to file");
                //e.printStackTrace();
            }
        }

}
```

Makefiles for building the compiler:

SouL/Makefile

```
all: compiler jsoul tests

compiler:
        @(cd AST && make)

build: clean Parser.class

clean:
        rm -f *~ *.class *.java AST/Parser.java AST/Yylex.java tests/output.txt

jsoul:
        @javac jsoul/*.java

tests: AST/Parser.class
        ./test_suite.sh
```

SouL/AST/Makefile

```
all: jflex yacc java

clean:
	rm *.class *~ Parser.java ParserVal.java Yylex.java

yacc: soul.y
	yacc -J soul.y && javac Parser.java

jflex: soul.flex
	jflex soul.flex

java:
	javac *.java
```

Java wrapper file for SouL programs, **SouL/java_wrapper.txt**:

```
import javax.sound.midi.*;
public class Soul {
          public static void main(String[] args) {
          try {
```

Shell script for running .soul programs and reporting errors, **SouL/soul**:

```
#!/bin/sh
if [[ $1 != *.soul ]]
then
  echo "Error: files must end with .soul"
   exit 0
fi
rm -f errors.txt
rm -f scope_errors.txt
cat java_wrapper.txt > jsoul/Soul.java
(cd AST && java Parser < '../'$1) 1>> jsoul/Soul.java 2> errors.txt
if [ -s errors.txt ]; then
  cat errors.txt
   rm jsoul/Soul.java
   rm errors.txt
   exit 0
fi
echo }catch \(RuntimeException e\) {System.err.println\(\"Error: \" + e.getMessage\(\)\)\;}}} >>
jsoul/Soul.java
(cd jsoul && javac Soul.java) 2> errors.txt
if [ -s errors.txt ]; then
	awk '/^symbol.*$/' errors.txt > scope_errors.txt
	awk '{ print $0 " out of scope" }' < scope_errors.txt
	rm scope_errors.txt
	exit 0
fi
(cd jsoul && java Soul) 2> errors.txt
if [ -s errors.txt ]; then
  cat errors.txt
   rm errors.txt
fi
```