Cb

Programming for musicians.

Final Report

Mehmet Erkilic (me2419)
Marcellin Nshimiyimana (mn2587)
Kyle Rego (kar2150)
Cole Diamond (cid2105)
Matthew Cowan (mpc2145)

Contents

1. Introduction

Cb is a language designed for rapid, concise, extensive music creation. It leverages the algorithmic nature of musical compositions to allow anyone to write code for music generation.

Music is inherently an algorithmic construct. It's filled with repeating portions, chords composed of notes with predetermined intervals etc. (COLE ADD SOME MUSIC SHIT IN HERE). Although music encompasses these traits it is still composed by one note after the other. Cb aims to make the lives of composers easier by providing a way to easily define methods, loops and various other constructs to create music. The defining quality of Cb lies in its simplicity. The concise nature of variable declarations, and defining methods allows the programmer to concentrate on writing music and not deal with unnecessarily long lines of code to understand what is going on.

The output of compiling a Cb program is a Midi file, which is a widely used industry standard, which is supported by all major music players and has hundreds, if not thousands, of professional grade software written to edit and manipulate such files. Without dealing with the clunky user interfaces of such software the programmer can rapidly compose music using Cb and then, if needed, further fine tune the midi file(s) via other software.

2. Language Tutorial

2.1 Getting Started

```
note a = (C, 0, whole);
chord b = ([a], whole) ;
stanza p = [b];
score m = [p];
compose(m);
```

This program simply creates a note that represent a whole middle C. This note is used to form a chord, which is used to form a stanza, which in turn is used to produce the score. The built-in function compose(), is called to initiate the composing process. This program, when compiled, will produce a midi file, which plays a whole middle C on the piano, the default instrument.

2.1.1 Compiling and Running

To compile HelloWorld.cb written above, run the following command:

```
python compile.py HelloWorld.cb
```

This will produce a java file called Cb.class file. Compile.py will run translate.ml on the file, which will generate a HelloWorld.java file and then runs the command javac HelloWorld.java to produce the Cb.class file.
This file can be executed by the command:

```
java Cb
```

This will result in the creation of a MIDI file named out.mid if the compose() function was called in the HelloWorld.cb program.
This file can be played with any modern music player that will play a middle C for a whole duration on a piano.

2.2 Selection Statement

```
if(true)
    print(0);
end
```

```
if(true)
   if(false and true)
      print(1);
   else
      print(0);
   end
else
   print(2);
end
```

The If/Else statements are used to express decisions. These statements can be nested. The conditional expressions within if have to evaluate to type bool. It can be the keywords “true” or ”false” or a relational operational, a is 0 as in the examples above.

2.3 Iteration

```
scale s = [(A,0,1),(B,0,2),(C,0,3),(D,0,4),(E,0,5)];
int check = 0;
foreach (note n in s)
   check += 1;
   if((n.duration) is check)
      print(0);
   else
      print(1);
   end
end
```

Cb supports two types of iterative control structures. The foreach loop and the while loop.

Foreach runs the statement block on each element of whatever was passed in. Foreach loops support chords, stanzas, scales and scores.
The while loop executes the statement block until the conditional expression evaluates to false. While fairly simple these examples above demonstrate the nesting and list iteration capabilities of Cb. List iteration is crucial to the language as chords, scales, stanzas and scores are all essentially lists of different variables.

2.4 Arithmetic Operations and Computation

```
int a = 10;
print(a+2);
print(a-2);
print(a*2);
print(a/3);
print(a%2);
```

```
a += 1;
print(a);
a -= 1;
print(a);
a++;
a--;
print(a);
```

The above examples demonstrate the arithmetic operations of Cb. As Cb only supports integer types, all numerical operations are designed for integers. The operation "a/3" above evaluates to 3 as it is integer division. The result 3.33 is truncated to 3 as the return value of the operation must be an integer. The precedence of operations follows the PMDAS (parentheses, multiplication, division, addition, subtraction).

2.5 Methods

Cb defines thirteen standard library methods:

print, major, minor, sharp, flat, randint, chordOfNote, repeat, rest, append, prepend, concat, and compose

Cb also allows the ability to create user-defined methods. The naming conventions for defining a method are the same as with variables; however, note that user-defined methods may not override the seven standard library methods and therefore may not use their names.

```
<-The Recursive GCD Algorithm->
meth int gcd(int a, int b)
   if(b is 0) return a;
   else return gcd(b,a%b); end
end

if(gcd(20,15) is 5)
   print(0);
else
   print(4111);
end
```

Above is an example of a user-defined method called gcd. User-defined methods are defined by prefacing the method with the keyword meth and a return type before stating the name of the method. In this example, the return type is an int.

The arguments for a method are defined in the parentheses following the name of the method. This example takes two int arguments a and b. Within the gcd method, a tail-end recursive definition is defined, running gcd on new arguments a and b until the arguments meet the criteria for the base case (in which case we return a, which is also an int).

The gcd method is called in a conditional statement, which runs gcd(20,15) will call the user defined method with arguments a and b.

The above recursive gcd function example demonstrates the computational power of Cb. While very simple the above code perfectly demonstrates the capabilities of Cb. Writing a simple gcd method in 4 lines is an example of the conciseness of the language. There are no unnecessary syntactic constructs getting in the way of the programmer. It also demonstrates recursion, a property of music as demonstrated by minor scales. Without declaring each note in 10's of lines of code, a programmer can simply form a scale recursively in 4-5 lines.

3. Language Reference Manual

3.1 Introduction

The Cb language is designed to be the most intuitive language for a musician to not only write basic music quickly with a focus on chord creation and manipulation, but include more algorithmic music compilation as naturally as possible. This manual describes the syntax for the Cb language.

3.2 Lexical Conventions

3.2.1 Whitespace

Spaces, tabs, and newlines (collectively, "white space") are ignored except when used as separators. Separators are white space that is needed to separate otherwise adjacent identifiers, keywords, and constants.

3.2.2 Comments

A comment, whether single or multiline, goes between <- characters, which indicates the start of it and ->, which indicate the end. The comment can be placed anywhere in the program as long as it is between these two characters. Comments do not nest and are ignored.
Example:

```
<- create a chord with three notes with a duration of 1/8
   note that the duration of the chord overrides that of the notes->
Note c = (C, 0, HALF);
Note g = (G, 0, HALF);
Note e = (E, 0, HALF);
Chord cr = ([c, g, e], EIGHTH)
```

3.2.3 Identifiers

In Cb language, an identifier is a sequence of letters, digits, and underscores "_". An identifier must start with a letter or an underscore and may not start with a number. There is no limit on how long identifiers can be. Below is the list of characters allowed in creating an identifier:

```
a b c d e f g h i j k l m
n o p q r s t u v w x y z
A B C D E F G H I J K L M
N O P Q R S T U V W X Y Z _
0 1 2 3 4 5 6 7 8 9
```

Ex: nice_note, NICE_note, and _NOTE2 are acceptable identifiers. However, 1nicenote and 2nicenote are not acceptable identifiers

3.2.4 Keywords

Keywords are identifiers used for specifying the types of expressions and for including methods from an external packages. These keywords listed below are reserved for Cb, which means that they cannot be used as normal identifiers.

Int	is	isnt
Note	if	meth
Chord	else	return
Scale	while	compose
Stanza	foreach	end
Score	in	elsif
and	or	use

3.2.5 Literals

Cb uses Integer literals that consist of a sequence one or more digits as well as Boolean literals that go to true or false.

3.2.6 Constants

3.2.6.1 Integer Constants

Cb has a set of Integer constants that are used to represent basic notes and known durations of notes. These can also note be used as normal identifiers. Below is a list of Integer constants:

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

A rest pitch constant is "R"
Note rest = (R, 0, HALF); <-Create a half note rest ->
<- the octave here doesn't matter ->

SIXTEENTH	EIGHTH	QUARTER	HALF	WHOLE

3.2.7 Operators

An operator specifies an operation to be performed. Operators are described in depth in the Expressions section.

3.2.8 Punctuators

A punctuator is a symbol that adds semantic value to the expression or statement that it belongs to, but does not perform an actual operation. These punctuators are used in declaration and assignment of variables. Below is a list of Punctuators:

[ ] ( ) . ;

Example:

```
Note asharp = (A#, 0, QUARTER);
<- do re mi song ->
meth Stanza doremi(Int duration)
        Note do = (C, 0, duration);
        Note re = (D, 0, duration);
        Note mi = (E, 0, duration);
        Stanza s = [do, re, mi];
        return s;
end
```

3.3 Meaning of Identifiers

3.3.1 Disambiguating Names

A Cb identifier is disambiguated mainly by the following characteristics: lexical uniqueness and function scope

3.3.2 Lexical Uniqueness

Cb identifiers are created with a combination of Latin characters and the underscore character as specified (and constrained) in the section, *Lexical Conventions*. All identifiers are first disambiguated by its lexical name being different from all other identifiers in the file.

3.3.3 Method Scope

Scope is the block of the program in which a variable exist. It can be accessed and used.

Cb indentifiers have two kinds of scope, global and local. Global variables are defined outside of any block. Local variables are variables defined within a block,

where a block is defined as any segment of code starting with a control statement and ending with the keyword “end”.
Also any variables declared within a method are local within the scope of that method. For example:

```
meth Note test()
        Note n = (D, 0, whole*2);
        return n;
end
Note n = (C, 0, 1);
test();
```

Here, a note is declared twice; once while defining a method and once after defining the method. Moreover, the method is subsequently called after the second declaration. In this scenario, having method scope is important since it is likely that the user does not want to let the method call alter the declaration/initialization of identifiers he/she makes beforehand. In this situation, ‘n’ will still have a value (C, 0, 1).

3.3.4 Types

There is 1 basic type: integers, and there are 5 derived types: a *Note*, C*hord*, S*cale*, *Stanza*, and *Score*. Their identifiers are listed below:

Types:
int, note, chord, scale, stanza, score

3.3.4.1 Basic Types

An integer specifies a whole, signed integer denoted by the keyword “Int”
Ex. *int x = 5;*

3.3.4.2 Derived Types

A note is defined by a string representing a note constant, an integer ([-5, 5]) representing octave displacement, and a positive integer representing duration.

The duration value refers to a multiple of 1/64 (the 64th note). So a duration of 2 is equivalent to a 32nd note and a value of 4 is equivalent to a 16th note. You may also use the duration constants (sixteenth, eighth, …) to help you with defining the duration of a note. You can also use the * operator to help ease the 1/64th base multiples.

```
note n = (A, -3, 72);
note n = (C, 2, half*2);
```

A chord is defined by a list of notes and a positive integer representing duration.

```
chord c = ([n1, n2, n3], 70);
```

A scale is defined by a list of notes.

```
scale s = [n1, n2, n3];
```

A stanza is defined by a list of both chords.

```
stanza sz = [c1, c2, c3, c4];
```

A score is defined by a list of stanzas.

```
score sc = [sz1, sz2, sz3];
```

3.4 Declarations

3.4.1 Declaration Syntax

Function definitions have the form:

function-definition:

type identifier(parameter-listopt) compound-statement

parameter-list:

type-specifier identifier

parameter-list, type-specifier identifier

Type is one of the following keywords: int, bool, note, chord, stanza, scale, score.

Identifier is a non-reserved alpha-numeric sequence.

Compound-statement is any legal code that returns a value of agreeable type with the declaration.

3.4.2 Blocks

A block is a section of code enclosed by one of *meth, while, foreach, if,* and the *end* keywords. Blocks can be nested within other blocks. Identifiers visible in an outer block are visible in the inner block, but identifiers declared in the inner block will not be visible in the outer block when the inner block ends.

3.4.3 Scope

The scope of an identifier is the subsequent statements within the block of code where it is declared including blocks nested in that block. Declarations can appear after certain keywords that open a block of code. These keywords are *meth, while*, *foreach*, and *if.* When identifiers are declared in these expressions, the scope of the identifiers is the block opened by the keyword. At the beginning of a function's execution, its parameters will be the only local identifiers in scope.

3.4.4 Identifier Naming

All identifiers within a block of code must be unique and a nested block's identifiers must not conflict with the identifier names in its parent block. This means that an identifier is visible over its entire scope and cannot be hidden by a subsequent re-declaration of the identifier.

3.5 Expressions

In Cb, expressions consist of one or more operators in tandem with operands. Associativity rules determine precedence, but parentheses can override the default orderings. The two most pervasive expressions in Cb are assignment expressions and operation expressions. The table below outlines the associativity rules of the Cb's built in functions.

Tokens (Descending Priority)	Operators	Class	Associativity
Identifiers, constants, parenthesized expression	Primary expression	Primary	
() [] .	Function calls, subscripting, direct selection	Postfix	L-R
+ - * / % ^	Arithmetic and augmentation	Binary	L-R
is isnt	Equality comparisons	Binary	L-R
< <= >= >	Relational	Binary	L-R
and	Logical And	Binary	L-R
or	Logical Or	Binary	L-R
= += -= *= /=	Assigment	Binary	R-L

3.5.1 Primary Expressions

3.5.1.1 Identifier

An identifier typifies a primary expression. Its declaration calls for the specification of a type of the identifier followed by the value of the identifier. It can refer to a function designator.

3.5.1.2 Constant

An integer, decimal, character, or floating constant is a primary expression of constant value. The capitalized letters C, B#, C#, Db, D, Eb, D#, E, Fb, F, E#, F#, Gb, G, Ab, G#, A, Bb, A#, B, Cb are constant expressions that each represent Notes of default duration having pre-defined values associated with their respective notes. Naturally, *Note* constants are the most frequent example of constants in Cb.

3.5.1.3 Parenthesized Expression

A parenthesized expression is a primary expression of the form (expression). It can be used to override precedence. For example, consider the two expressions below.

Expression 1: (1 > 2) and (3 < 2 or 3 < 1)

Expression 2: (1 > 2 and 3 < 2) or (3 < 1)

While the former will "and" the two subexpressions together, the latter will instead apply the "or" operator to the result.

3.5.2 Postfix

Postfix calls are direct selection. Examples are:

```
Note1.pitch;
Note1.duration;
Note1.octave;
Chord1.duration
Score1.instrument
```

3.5.2.1 Direct Selection

Pitch and duration in objects of type Note and Chord, and instrument in object score can be changed through directly accessing the objects. For example, A.pitch += 2 will result in C. The same paradigm applies to duration objects as well.

3.5.3 Unary Operations

3.5.3.1 Whole-Step Increment/Decrement Operations

Plus-plus (++) and minus-minus (--) operations of the form (expression)++ or (expression)-- can be used for the purpose of incrementing and decrementing, respectively, and integer.

3.5.3.2 Octave Increment and Decrement Operations

Carrot-plus (^+) and Carrot-minus (^-) operations of the form (expression)^+ or (expression)^- will shift a single Note or all constituent Notes of a Chord up or down an octave. Specifically, carrot-plus will shift up an octave while carrot-minus will shift down an octave.

```
Ex:
Note n = (G, 0, 1);
n = n^+;
<- Now n is the note with pitch of G transposed up one octave ->
```

3.5.4 Binary Operations

3.5.4.1 Add and Subtract

Add and subtract binary operations can be applied to integers only.
integer + integer
integer - integer
Example:

```
5 + 3;
6 - 2;
```

3.5.4.2 Multiply, Divide and Modulus

Multiply can applied to integer:

Multiply-expression: integer * integer
Divide-expression: integer / integer
Modulus-expression: integer % integer

3.5.4.3 Relational Comparison

Yields a Number result (1 if true, 0 if false) that uses the following syntax:
Relational-expression:

relational-expression < relational-expression
relational-expression > relational-expression
relational-expression >= relational-expression
relational-expression <= relational-expression

3.5.4.4 Equality Comparison

Determines if two values are equal. Cb uses 1 to denote true and 0 to denote false.
The token “is” denotes equality while “isnt” denotes inequality.

The following rules govern equality relations:
Two Number objects are equal if they have the same value.
Equality Comparisons take the following form:

Equality-expression is equality-expression
Equality-expression is not equality-expression

3.5.4.5 Logical Operators

The symbols “and” and “or” perform a logical and, or operation on two expressions, respectively. If the expression evaluates to false, then a zero is returned. Otherwise, 1 is returned.

Logical-expression:

logical-and-expression and logical-and-expression
logical-or-expression or logical-or-expression

3.5.4.6 Assignment

Assignment is a right associative operation – the expression on the right is evaluated and then used to set the lvalue. The rvalue must have the same type as the lvalue since no casting is implicitly done.

3.5.4.7 Commas

Commas are used to separate list elements like parameters in a function or Notes in a Chord. Consider, for example, *chord c = ([noteA, noteB], dur)*. Moreover, a pair of expressions separated by a comma is evaluated left-to-right and that the type and value of the result are identical to the type and value of the right operand.

3.5.4.7 Expressions of the form [Operation]-Equals

The tokens “+=”, “-=”, “/=”, “*=” can be used to modify the state of a variable by a given amount. For example, 3 += 2 will return 5. Each of the operators uses the pre-defined operations of addition, subtraction, division and multiplication to compute the result.

3.6 Statements

Except as indicated, statements are executed in sequence. Statements are executed for their effect, and do not have values. They fall into the following categories:

statement:
expression;
return expression;
conditional-statement;
while-statement
foreach-statement

3.6.1 Expression Statement

expression ;

Most statements take this form, as assignments or function calls. All side effects from the expression are completed before the next statement is executed.

3.6.1.1 Compound Statement

statement-list:
statement
statement-list statement

Inside methods and other structures there is the concept of multiple statements.

3.6.1.2 Conditional Statement

if (expr) statement-list else-statement END

In both cases the expression is evaluated and if it is nonzero or the bool value of true, the first substatement is executed. If an else clause is included its code will be executed if the prior if condition was not accepted.

3.6.1.3 While Statement

while (expr) statement-list END

The while statement allows for looping over the statement-list as long as the expr evaluates down to true. This means the expr evaluates to either a nonzero integer or the bool value true.

3.6.1.4 Foreach Statement

param-decl:
    DATATYPE ID

foreach (param-decl IN ID) statement-list END

The foreach statement allows for looping over all elements of the specified datatype in the specified item.

3.6.1.5 Return Statement

return expression ;

A function returns to its caller by means of the *return* statement, which must be of the form expressed above. In Cb a value must be returned by all methods.

4. Project Plan

4.1 Planning

Initial brainstorming, planning and design of the Cb language stemmed from the group members interest and experience with music. Music is nothing more than a list of notes, which is why Cb focuses on forming note, chord, stanza, and other lists easily, along with simple note declarations. Along these lines, the language syntax was designed to minimize the programmer's distractions and provide him with a rapid and concise way to generate music. This paradigm helped us achieve short, user-friendly note declarations and powerful lists. These put together further reduced the number of lines of code necessary to compose most music.

Starting from the time that the group was formed, the group held weekly meetings on Sunday afternoons at 5pm to discuss the following week's agenda, assigning responsibilities to each member of the group and ensuring that each team member could properly schedule their task given their other obligations. Depending on necessity, every two to three weeks our meetings extended to include coding parts of our language together and answering each other's questions regarding code.

Towards the end of the semester, we held a four-day, caffeine fueled, non-stop hackathon to implement our language collectively. This period of time was extremely productive albeit exhausting. The proximity of the group members for extended periods of time increased our performance and helped each member know what was going on in each portion of the code.

4.2 Team Responsibility

Although all team members were responsible for designing, coding, testing, and integrating their part of the project, everyone worked on each file in Cb. Each member concentrated on different methods, but having every member actively participate on each files creation helped deepen each members understanding of the language as a whole.

Everyone was collectively responsible for deciding features to be implemented, overall architectural design, the grammar and scanner, and the final report. Testing was also done by every member, which helped us identify bugs and different situations that 1 or 2 debuggers might not have though of.

Below is the representation of who was responsible for which parts of the code.

Team Member	Responsibility
Matt Cowan	Parser, AST, Translator, compile.py, preprocessor.py
Cole Diamond	Parser, Translator
Mehmet Erkilic	Parser, Translator, LRM
Macellin Nshimiyimana	LRM, Translator (creation of MIDI)
Kyle Rego	AST, Translator

4.3 Project Timeline

September 26, 2012	Project Proposal and core language features defined
October 21, 2012	Project roles defined; development Git repository initialized
October 31, 2012	LRM, grammar, and scanner are complete
November 18, 2012	Parser, complete
December 15, 2012	Interpreter and semantic check complete
December 18, 2012	Transformation to translator and test suite complete

4.4 Software Development Environment

The project was primarily developed on Windows and Mac machines using Ocaml and SublimeText2 as an IDE with the following major components:

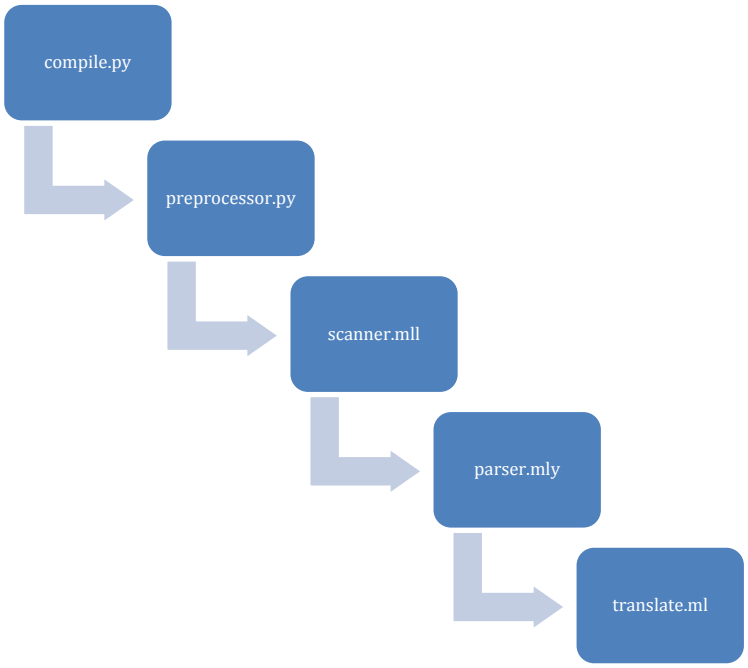
OCaml v4.00.1:
Ocaml was used as the primary development language for the project.
Ocamllex:
Utility for lexical analysis
Ocamlyacc:
Utility for parsing
Java v1.6:
Java was used for compilation and midi file generation
Git:
Version control for source code and documentation
SublimeText2:
IDE used to edit files
Makefile:
Compilation script
Trello:
Project organization webapp

5. Architectural Design

5.1 Overview

There are 5 component code files that run the show:

- compile.py (the director script that runs the following scripts in the correct order, takes just an argument of file to compile)
- preprocessor.py (finds use directives and replaces them with the code in the file specified)
- scanner.mll (tokenizes program, removes whitespace, removes comments)
- parser.mly (with help of scanner, generates AST)
- translate.ml (given an AST, prints to file Cb.java the .cb program as java code)



5.2 translate.ml

Since its job is to traverse the AST and print out for the user a valid java version of the program the translate.ml file tries to take on am much error checking as possible. For example, type errors are almost universally handled within the translator and would never lead to java creations. One type of check that was left out was checking that all code paths return a value due to the complexity of that operation.

It traverses the tree by calling an eval function of expressions to update the variable maps as well as to return the java string for the expression. There is also a base translate method that goes through all of the program in a list view and translate its thing it sees separately. There is a call method which is used for looking inside blocks to get the java code as well as an exec function which when given a statement would give back the java string for the statement and any body after all checking. After coming out of translate.ml the idea is the .java file should be much more likely to compile.

6. Test Plan

6.1 Representative Programs

See appendix for representative programs written in Cb.

6.2 Test Suites

Test suites were essential to the development of the language. Aggressive testing performed by each member of the team helped reveal bugs and inconsistencies in the language. After these inconsistencies were ironed out, more testing was performed, and the whole process was repeated until no one could identify any apparent errors.

Testing occurs at different stages of the language. The correctness of the AST, the operations performed by the translator, the output of the translator and the final output midi final are all tested individually. This approached meant more testing but in the long run it helped us progress faster as errors in lower stages were detected and eliminated before they started causing problems at higher levels.

6.3 Tests

To view the full suite of tests for each phase it is advised to crack open the zipped folder and inspect all of the test folders:

- Parser_Tests (tests created and used while debugging the parsing, scanning, and creation of the AST)
- Interpreter_Tests (tests that were used while we were accidentally created an interpreter, plus side was that they did greatly help eliminate bus)
- Suite (tests that are fully fleshed out and capable of being used in an automated way)
- Tests (testing folder for saving tests that don't exact fall into one of the above categories)

7. Lessons Learned

7.1 Team Dynamics

Working in a team of five led to long discussions over which features should be implemented in the language. Having a team leader greatly increased our productivity as these long discussions were cut short by one person calling the shots. Having the team members regularly update each other via an email thread and Trello (project organization software) was essential to our progress.

Cb's team leader, according to each member's strengths, performed work delegation. Efficient division of labor helped us progress quickly.

7.2 Version Control

Git was extremely essential to our productivity, keeping each member up to date on what was being completed by every other member. Every team member working on the same branch sometimes caused problems regarding merge conflicts, which could have been avoided. One way the project may have avoided this problem was by creating separate branches.

7.3 Test Suite

The test suite was essential to uncovering bugs and helping us improve the functionality of our program. Aggressive testing performed by each member at differing points of the language helped us move more rapidly, as we identified errors quicker. Testing was way more important than we originally thought

7.4 Scheduling

Originally, our meetings were delegated and scheduled at the end of each previous meeting. However, we found it more beneficial to rely on predictability and decided to instead meet at a regular time each weekend.

Scheduling regular weekly meetings helped us stay on track better as a result; however, pacing ourselves with regards to deadlines was not efficient and the group ended up doing most of the work a few weeks before the deadline.

7.5 Communication with Instructor and TA's

Our first implementation of our project was an AST to CSV file interpreter. We did not realize that we had written an interpreter until we had a meeting with our TA. We learned that communication with the instructor and TAs should have

been regularly made at each step of the project instead of when needed.

Appendix A

A.1 Testing Files

Append.cb

```
chord c = ([n],whole);
c = append(n, c);

foreach(note n in c)
    if(n.pitch is 0 and n.octave is 2 and n.duration is 16)
        print(0);
    else
        print(1352);
    end
end

stanza s = [c];
s = append(c, s);

int counter = 0;
foreach(chord c_temp in s)
    counter += 1;
end

if(counter is 2)
    print(0);
else
    print(1353);
end

s = append(c, s);
s = append(c, s);

foreach(chord c_temp in s)
    counter += 1;
end

if(counter is 6)
    print(0);
else
    print(1354);
end

score sc;
sc = append(s, sc);

counter = 0;
foreach(stanza s_temp in sc)
    counter++;
end

if(counter is 1)
    print(0);
else
    print(1355);
end
```

ChordOfNote.cb

```
chord rest = rest(1);
if(rest.chord_duration is 1)
    print(0);
else
    print(12345);
end

note n = (C,0,half);
chord n_chord = chordOfNote(n);

if(n_chord.chord_duration is 32)
    print(0);
else
    print(12346);
end

chord n_chord2 = chordOfNote((C,0,quarter));

if(n_chord2.chord_duration is 16)
    print(0);
else
    print(12347);
end
```

Concat.cb

```
stanza s1;
stanza s2;
stanza test = concat(s1,s2);

foreach(chord c in test)
    print(8111);
end

note n = (C,0,quarter);
chord c1 = ([n],whole);
chord c2 = ([n],half);
s1 = [c1];
s2 = [c2];

stanza s3 = concat(s1,s2);
bool first = true;
foreach(chord c in s3)
    if(first and c.chord_duration is 64)
        first = false;
        print(0);
    else
        if(first)
            first = false;
            print(8112);
        end
        if(c.chord_duration is 32)
            print(0);
        else
            print(8113);
        end
    end
end
```

Foreach.cb

```
scale s = [(A,0,1),(B,0,2),(C,0,3),(D,0,4),(E,0,5)];
chord c = ([(A,0,1),(B,0,2),(C,0,3),(D,0,4),(E,0,5),(E,0,5)], whole);
stanza st = [c, c];
score p = [st, st, st];
int check = 0;

foreach (note n in s)
    check++;
end

if(check is 5)
    print(0);
else
    print(3112);
end

check = 0;
foreach (note n in c)
    check += 1;
end

if(check is 6)
    print(0);
end
if(check isnt 6)
    print(3113);
end

check = 0;
foreach (chord n in st)
    check++;
    if(n.chord_duration is 64)
        print(0);
    else
        print(3114);
    end
end

if(check is 2)
    print(0);
else
    print(3115);
end

check = 0;

foreach (stanza n in p)
    foreach (chord c2 in n)
        check++;
    end
end

if(check is 6)
    print(0);
else
    print(3116);
end
```

ifElse.cb

```
if(true)
    print(0);
end

if(true)
    print(0);
else
    print(5543);
end

if(false)
    print(5544);
else
    print(0);
end

if(true)
    if(true and true)
        print(0);
    else
        print(5545);
    end
else
    print(5546);
end

if(false or true)
    print(0);
else
    print(5547);
end

if(5 isnt 4)
    print(0);
else
    print(5548);
end
```

ints_arithmatic.cb

```
<-Testing Integers and Arithmetic->

int a;
a=5;
if(a is 5)
    print(0);
else
    print(2111);
end

int b = 10;
if(b is 10)
    print(0);
else
    print(2112);
end

int c = a + b;
if(c is 15)
    print(0);
else
    print(2113);
end

int d = 1;
d += 1;

if(d is 2)
    print(0);
else
    print(2114);
end

int e = 20;
e -= 1;

if(e is 19)
    print(0);
else
    print(2115);
end


int f = 50;
f++;

if(f is 51)
    print(0);
else
    print(2116);
end

int g = 40;
g--;

if(g is 39)
    print(0);
else
    print(2117);
end
```

IterativeGCD.cb

```
meth int gcd(int d, int e)
    while (d isnt e)
        if(d > e)
            d = d-e;
        else
            e = e-d;
        end
    end
    return d;
end

if(gcd(20,15) is 5)
    print(0);
else
    print(4112);
end
```

RecursiveGCD.cb

```
meth int gcd(int a, int b)
    if(b is 0)
        return a;
    else
        return gcd(b,a%b);
    end
end

if(gcd(20,15) is 5)
    print(0);
else
    print(4111);
end
```

MethodTest.cb

```
meth void test1(bool b)
    bool c = true;
    if(c)
        print(0);
    end
end

meth bool test2(bool b)
    bool c = true;
    if(c)
        print(0);
    end
    return false;
end

test1(true);

test2(false);
```

Prepend.cb

```
note a = (A,0,half);
note b = (B,0,quarter);

chord c;
chord d;

c = append(a, c); <-c is now (9,0,32)->
c = append(b, c); <-c is now (9,0,32),(11,0,16)->

d = prepend(a, d); <-d is now (9,0,32)->
d = prepend(b, d); <-d is now (11,0,16),(9,0,32)->

bool first = true;
foreach(note e in c)
    if(first and e.pitch is 9)
        first = false;
        print(0);
    else
        if(first and e.pitch isnt 9)
            first = false;
            print(7123);
        else
            if(e.pitch is 11)
                print(0);
            else
                print(7124);
            end
        end
    end
end

first = true;
foreach(note f in d)
    if(first and f.pitch is 11)
        first = false;
        print(0);
    else
        if(first and f.pitch isnt 11)
            first = false;
            print(7125);
        else
            if(f.pitch is 9)
                print(0);
            else
                print(7126);
            end
        end
    end
end
```

Repeat.cb

```
note n = (F,0,half);
scale s1 = repeat(n, 3);

foreach(note n1 in s1)
    print(n1);
end

scale s2;

n.pitch += 2;

s2 = repeat(n, 7);

foreach(note n1 in s2)
    print(n1);
end

chord c = ([n],quarter);

stanza st;

st = repeat(c, 5);

foreach(chord c1 in st)
    print(c1.chord_duration);
end

score sc;

sc = repeat(st, 2);

foreach(stanza st1 in sc)
    print(0);
end

sc = repeat(sc, 2);

foreach(stanza st1 in sc)
    print(1);
end
```

TwinkleTwinkleLittleStar.cb

```
chord c1 = chordOfNote((C,0,quarter/2));
chord c2 = chordOfNote((G,0,8));
chord c3 = chordOfNote((A,0,8));
chord c4 = chordOfNote((F,0,8));
chord c5 = chordOfNote((E,0,8));
chord c6 = chordOfNote((D,0,8));
chord c7 = chordOfNote((G,0,quarter));
chord c8 = chordOfNote((C,0,quarter));
chord c9 = chordOfNote((D,0,quarter));

stanza p1 = [c1, c1, c2, c2, c3, c3, c7];
stanza p2 = [c4, c4, c5, c5, c6, c6, c8];
stanza p3 = [c2, c2, c4, c4, c5, c5, c9];

score m = [p1, p2, p3, p3, p1, p2];

compose(m);
```

Expected output / result of java Cb:

play the resulting midi file and it should sound like twinkle twinkle

Appendix B

B.1 Complete Code Reference

compile.py

```
#! /usr/bin/python
#Cb language compiler, by Matthew Cowan

import sys
import os
import subprocess

verbose = False

def compile(fName):
  if verbose: print "=====Running The PreProcessor=====\n"
  proc = subprocess.Popen(['python','preprocessor.py', fName],stdout=subprocess.PIPE)
  output = ""
  while True:
    line = proc.stdout.readline()
    if line != '':
      output += line.rstrip() + "\n"
    else:
      break
  if verbose: print "Program Was:\n" + output
  if verbose: print "=====Calling OCaml Cb=====\n"
  ocaml = subprocess.Popen(['./Cb'],stdin=subprocess.PIPE,stdout=subprocess.PIPE)
  ocaml.stdin.write(output)
  ocaml.stdin.close()
  while True:
    line = ocaml.stdout.readline()
    if line != '':
      print line.rstrip()
    else:
      break
  if verbose: print "\n=====Ocaml COMPILER COMPLETE=====\n"
  if os.path.isfile("Cb.java"):
    if verbose: print "\n=====Finding & Compiling Output\n"
    toJava = "Cb.java"
    for f in toJava:
      java = subprocess.Popen(['javac', 'Cb.java'],stdout=subprocess.PIPE)
      while True:
        line = java.stdout.readline()
        if line != '':
          if verbose: print line.rstrip()
        else:
          break
      if verbose: print "\n"
    if verbose: print "\n=====Java File Compiled=====\n"
    if verbose: print "\n=====Compilation Complete=====\n"
  else:
    print "\n=====Failed to find Cb.java, Translator likely failed=====\n"

def ast(fName):
```

```
  print "=====Running The PreProcessor=====\n"
  proc = subprocess.Popen(['python','preprocessor.py', fName],stdout=subprocess.PIPE)
  output = ""
  while True:
    line = proc.stdout.readline()
    if line != '':
      output += line.rstrip() + "\n"
    else:
      break
  print "Program Was:\n" + output
  print "=====Calling OCaml Cb=====\n"
  ocaml = subprocess.Popen(['./Cb', '-a'],stdin=subprocess.PIPE,stdout=subprocess.PIPE)
  ocaml.stdin.write(output)
  ocaml.stdin.close()
  while True:
    line = ocaml.stdout.readline()
    if line != '':
      print line.strip()
    else:
      break
  print "\n=====Ocaml COMPILER COMPLETE=====\n"
  print "\n=====Compilation Complete=====\n"

if len(sys.argv) == 1:
  print "Error: please provide the compiler with a file to compile"
  sys.exit()
elif len(sys.argv) == 2:
  if os.path.isfile(sys.argv[1]):
    compile(sys.argv[1])
  else:
    print "Error: " + sys.argv[1] + " not found"
elif len(sys.argv) == 3:
  if sys.argv[1].lower() == "-a":
    ast(sys.argv[2])
  elif sys.argv[1].lower() == "-i":
    compile(sys.argv[2])
  elif sys.argv[1].lower() == "-v":
    verbose = True
    compile(sys.argv[2])
  elif sys.argv[2].lower() == "-a":
    ast(sys.argv[1])
  elif sys.argv[2].lower() == "-i":
    compile(sys.argv[1])
  elif sys.argv[2].lower() == "-v":
    verbose = True
    compile(sys.argv[1])
  else:
    print "Error: usage python compile.py [compiler flag] <cb file>"
    sys.exit()
else:
  print "Error: usage python compile.py [compiler flag] <cb file>"
  sys.exit()
```

preprocessor.py

```
#!/usr/bin/python
#Cb language preprocessor, by Matthew Cowan

import sys
import os
import re

comments = re.compile(r'<-.*?->')
uses = re.compile(r'use (.*?);')

def proc(fName):
    try:
        with open(fName) as f:
            asArray = f.readlines()
            progString = "".join(asArray)
            without_comments = comments.sub('', progString)
            for match in uses.findall(without_comments):
                if os.path.isfile(match):
                    try:
                        with open(match) as f2:
                            toArr = f2.readlines()
                            toStr = "".join(toArr)
                            progString = re.sub("use " + match + ";",toStr,progString)
                    except IOError:
                        print "Preprocessor Error: use statement file: " + match + "
not found"
                else:
                    print "Preprocessor Error: use statement file: " + match + " not
found"
                    sys.exit()
            print progString
    except IOError:
        print "Preprocessor Error: file provided not found"
        sys.exit()


if len(sys.argv) != 2:
    print "Preprocessor Error: needs to take name of file"
    sys.exit()

if os.path.isfile(sys.argv[1]):
    proc(sys.argv[1])
else:
    print "Preprocessor Error: file provided not found"
```

scanner.mll

```
{ open Parser } (* Get the token types *)

rule token = parse
                        [' ' '\t' '\r' '\n'] { token lexbuf } (* Whitespace *)
                        | "<-" { comment lexbuf } (* Comments *)
                        | '(' { LEFTPAREN }
                        | ')' { RIGHTPAREN } (* punctuation *)
                        | '[' { LBRAC } (* punctuation *)
                        | ']' { RBRAC } (* punctuation *)
                        | ';' { SEMICOLON }
                        | ',' { COMMA }
                        | '.' { DOT }
                        | '+' { PLUS } (* started here *)
                        | '-' { MINUS }
                        | '*' { TIMES }
                        | '/' { DIVIDE }
                        | '%' { MOD }
                        | '=' { ASSIGN }
                        | "end" { END }
                        | "+=" { PLUSEQ }
                        | "-=" { MINUSEQ }
                        | "*=" { TIMESEQ }
                        | "/=" { DIVIDEEQ }
                        | "%=" { MODEQ }
                        | "++" { PLUSPLUS }
                        | "--" { MINUSMINUS }
                        | "^+" { RAISE }
                        | "^-" { LOWER }
                        | '<' { LT }
                        | "<=" { LEQ }
                        | ">" { GT }
                        | ">=" { GEQ }
                        | "and" { AND }
                        | "or" { OR }
                        | "if" { IF } (* keywords *)
                        | "else" { ELSE }
                        (*| "elseif" { ELSIF } *)
                        | "foreach" { FOREACH }
                        | "in" { IN }
                        | "is" { IS }
                        | "isnt" { ISNT }
                        | "while" { WHILE }
                        | "return" { RETURN }
                        (* | "void" | "int" | "bool" | "note" | "chord" | "scale" |
"stanza" | "score" { SCORE } *)
                        | "void" { VOID}
                        | "int" { INT }
                        | "bool" { BOOL }
                        | "note" { NOTE }
                        | "chord" { CHORD }
                        | "scale" { SCALE }
                        | "stanza" { STANZA }
                        | "score" { SCORE }
                        | "meth" { METH }
                        | "return" { RETURN }
                        | "end" { END }
                        | "true"|"false" as boollit { BOOLLITERAL(bool_of_string
boollit) }
```

```
                        (*| '-'? ['0' - '5'] as octave { OCTAVE(int_of_string octave) }
*)        (*mn always between -5 and 5 *)
                        (*| ['1'-'9'](['0'-'9']) as durInt { DURATIONINT(int_of_string
durInt) }*)             (*mn only positive int *)
                        | ((['A'-'G'](['b' '#']?))|'R') as noteconst {
NOTECONST(noteconst) }
                        | ("whole" | "half" | "quarter") as durConst {
DURATIONCONST(durConst) }
                        | eof { EOF } (* Endoffile *)
                        | '-'?['0'-'9']+ as lxm { INTLITERAL(int_of_string lxm) } (*
integers *)
                        | ['a'-'z' 'A'-'Z' '_']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm {
ID(lxm) }
                        | _ as char { raise (Failure("illegal character: " ^
Char.escaped char)) }

                        and comment = parse
                        "->" { token lexbuf } (* Endofcomment *)
                        | _ { comment lexbuf } (* Eat everything else *)
```

parser.mly

```
%{ open Ast %}

%token <int> INTLITERAL
/*%token <int> OCTAVE */
/*%token <int> DURATIONINT */

%token <string> DURATIONCONST /* whole half etc. */
%token <string> NOTECONST  /* Goes to string A or B or any note*/
%token <bool> BOOLLITERAL
%token <string> ID

%token LEFTPAREN RIGHTPAREN LBRAC RBRAC EOF
%token INT NOTE CHORD SCALE STANZA SCORE VOID

%token BOOL /* Added bool */
%token IN
%token IF
%token ELSE NOELSE
%token WHILE FOREACH
%token ASSIGN
%token PLUSEQ
%token MINUSEQ
%token TIMESEQ
%token DIVIDEEQ
%token MOD
%token AND
%token OR
%token MODEQ
%token PLUS
%token MINUS
%token TIMES
%token DIVIDE
%token IS
%token ISNT
%token LT
%token LEQ
%token GT
%token GEQ
%token PLUSPLUS
%token MINUSMINUS
%token RAISE
%token LOWER

%token METH RETURN END
%token PLUS MINUS TIMES DIVIDE

%token ASSIGN          /* Variable Assign  only used for variable decleration */
%token SEMICOLON
%token COMMA DOT

%nonassoc NOELSE
%nonassoc ELSE
%left PLUSEQ MINUSEQ
%left TIMESEQ DIVIDEEQ MODEQ
%right ASSIGN
%left OR
%left AND   /* ex: a > b && c is 3 or isnt A  => ((a > b) && (c is 3)) or (isnt A) */
%left IS ISNT
%left LT GT LEQ GEQ
```

```
%left PLUS MINUS
%left TIMES DIVIDE MOD
%left PLUSPLUS MINUSMINUS RAISE LOWER
%left SHARP FLAT

%start program
%type <Ast.program> program         /* ocamlyacc: e - no type has been declared for
the start symbol `program'*/

%%

program:
  /* nothing */  { [] }
  | program generic { $2 :: $1 }

 abstraction:
                       /* nothing */  { [] }
  | abstraction innerblock { $2 :: $1 }

innerblock:
                       vdecl { VDecl2($1) }
                       | fullvdecl { FullDecl2($1) }
                       | statement { Stmt2($1) }

generic:
    vdecl { VDecl($1) }
  | fullvdecl { FullDecl($1) }
  | methdecl { MDecl($1) }
  | statement { Stmt($1) }

vdecl:
                       cb_type ID SEMICOLON
                             {{ vartype = $1;
                                    varname = $2 }}

fullvdecl:
                       cb_type ID ASSIGN expr SEMICOLON {{ fvtype = $1;

                       fvname = $2;

                       fvexpr = $4 }}

methdecl:
                       METH cb_type ID LEFTPAREN meth_params RIGHTPAREN abstraction END
/* m stuff */
                             { {
                                    rettype = $2;
                                    fname = $3;
                                    formals = $5;
                                    body = List.rev $7 } }

cb_type:
      INT                                 { Int }
    | NOTE                                { Note }
    | CHORD                               { Chord }
    | SCALE                               { Scale }
    | BOOL                                { Bool }
    | STANZA                              { Stanza }
    | SCORE                               { Score }
                       | VOID                                      { Void }
```

```
meth_params:
                        { [] }
                        | param_list { List.rev($1) }

param_list:
                        param_decl { [$1] }
                        | param_list COMMA param_decl { $3 :: $1 }

param_decl:
                        cb_type ID
                              { {     paramname = $2;
                                      paramtype = $1 } }

/*
statement_list:
                        { [] }
                        | statement_list statement { $2 :: $1 }
*/

statement:
                        expr SEMICOLON { Expr($1) }
                        | RETURN expr_opt SEMICOLON { Return($2) }   /*return; or return
7;*/
                        | RIGHTPAREN abstraction END { Block(List.rev $2) }
                        | IF LEFTPAREN expr RIGHTPAREN abstraction %prec NOELSE END {
If($3, $5, Block([])) }
                        | IF LEFTPAREN expr RIGHTPAREN abstraction ELSE abstraction END
{ If($3, $5, Block(List.rev $7)) }
                        | WHILE LEFTPAREN expr RIGHTPAREN abstraction END { While($3,
$5) }
                        | FOREACH LEFTPAREN param_decl IN ID RIGHTPAREN abstraction END
{ Foreach($3, $5, $7)}

/*duration_expr:
                        INTLITERAL { IntLiteral($1) }
                        | ID { Id($1) }
                        | DURATIONCONST { DurConst($1) }
                        */           /* 5                  */
                        /* | duration_expr PLUS duration_expr { Binop($1, Add, $3)  }
                        | duration_expr MINUS duration_expr { Binop($1, Sub, $3)  }
                        | duration_expr TIMES duration_expr { Binop($1, Mult, $3)  }
                        | duration_expr DIVIDE duration_expr { Binop($1, Div, $3)  }*/

generic_list:
                         /*mn cannot have empty ???*/
                         expr { [$1] }
                        | generic_list COMMA expr { $3 :: $1 }

                        /*mn | generic_list COMMA ID TIMES INTLITERAL { BinOp(Id($1),
IDTimes, IntLiteral($3))} confusing  a, b, c*5, b  */

expr_opt:
                        /* nothing */ { NoExpr }
                        | expr { $1 }
                                                    /*  */

expr:
                        ID { Id($1) }
                        | ID DOT ID { MemberAccess($1, $3) }
                                          /* score.put   */
                        | INTLITERAL { IntLiteral($1) }
```

```
                      | NOTECONST { NoteConst($1) }
                      | DURATIONCONST { DurConst($1) }
                      | BOOLLITERAL { BoolLiteral($1) }
                      /*| ID LBRAC expr RBRAC { ElemOp($1, $3) }*/
                      /*| cb_type ID ASSIGN expr { IntTypeAssign($2, $4) } */
                      /*| INT expr { IntTypeAssign($2) }*/
                      /*| duration_expr { $1 } */
                      | LEFTPAREN NOTECONST COMMA expr COMMA expr RIGHTPAREN  {
NoteExpr($2, $4, $6) }  /* x = (A#, 4, 34)         */
                      | LEFTPAREN LBRAC generic_list RBRAC COMMA expr RIGHTPAREN   {
ChordExpr($3, $6) }
                      | LBRAC generic_list RBRAC { ListExpr($2) }
                      /* | generic_list COMMA ID TIMES INTLITERAL { BinOp(Id($1),
IDTimes, IntLiteral($3))}   a, b, c*5, b  */
    | expr ASSIGN expr       { Assign($1, $3) }
                      /* x = y            */
    | expr PLUSEQ expr { Assign($1, BinOp($1, Add, $3)) }                      /* x +=
y                             */
                      | expr MINUSEQ expr { Assign($1, BinOp($1, Sub, $3)) }
                          /* x -= y                          */
                      | expr TIMESEQ expr { Assign($1, BinOp($1, Mult, $3)) }
                          /* x *= y                          */
                      | expr DIVIDEEQ expr { Assign($1, BinOp($1, Div, $3)) }
                          /* x /=y                           */
                      | expr MODEQ expr { Assign($1, BinOp($1, Mod, $3)) }
                          /* x %= y                          */
                      | expr PLUS expr { BinOp($1, Add, $3) }
                                  /* x + y                               */
                      | expr MINUS expr { BinOp($1, Sub, $3) }
                                /* x - y                             */
                      | expr TIMES expr { BinOp($1, Mult, $3) }
                                /* x * y                             */
                      | expr DIVIDE expr { BinOp($1, Div, $3) }
                                /* x / y                             */
                      | expr MOD expr { BinOp($1, Mod, $3) }
                                  /* x % 5                               */
                      | expr AND expr { BinOp($1, And, $3) }
                                  /*(a is 4 and b is 2)        */
                      | expr OR expr { BinOp($1, Or, $3) }
                                  /*(a is 4 or b is 2)  */
                      | expr IS expr { BinOp($1, Eq, $3) }
                                  /* x is y                              */
                      | expr ISNT expr { BinOp($1, NEq, $3) }
                                  /* x isnt y                            */
                      | expr LT expr { BinOp($1, Less, $3) }
                                  /* x < y                               */
                      | expr LEQ expr { BinOp($1, LEq, $3) }
                                  /* x <= y                              */
                      | expr GT expr { BinOp($1, Greater, $3) }
                                /* x > y                             */
                      | expr GEQ expr { BinOp($1, GEq, $3) }
                                  /* x >= y                              */
                      | expr PLUSPLUS { Assign($1, BinOp($1, Add, IntLiteral(1))) }
                      /* x++                             */
                      | expr MINUSMINUS { Assign($1, BinOp($1, Sub, IntLiteral(1))) }
                      /* x--                             */
                      | expr RAISE { UnaryOp(Raise, $1) }
                                  /* x^+                                 */
                      | expr LOWER { UnaryOp(Lower, $1) }
                                  /* x^-                                 */
```

```
                      | LEFTPAREN expr RIGHTPAREN { $2 }
                                          /* (x)                                  */
                      | ID LEFTPAREN actuals_opt RIGHTPAREN { MethodCall($1, $3) }
                      /* x(...)                          */

actuals_opt:
                      { [] }
                      | actuals_list { List.rev $1 }

actuals_list:
                      expr { [$1] }
                      | actuals_list COMMA expr { $3 :: $1 }
```

ast.ml

```
type op =
    Add | Sub | Mult | Div | Mod
    | And | Or | Eq | NEq | Less | LEq | Greater | GEq (* | IDTimes *)

type uop =
    Raise | Lower

type cb_type = Void | Int | Note | Bool | Chord | Scale | Stanza | Score | Part

type expr = (* Expressions *)
    Id of string (* foo *)
    (*| Cbtype of string mn stand for Datatype*)
    | MemberAccess of string * string (* foo.intensity *)
    | IntLiteral of int (* 42 *)
    | NoteConst of string (*mn A, B#, C, ...*)
    | BoolLiteral of bool (* true *)
    | DurConst of string (*mn whole, half, ... *)
    (*| ElemOp of string * expr (*a[5]*)*)
    | Assign of expr * expr (* x = y *)
    | NoteExpr of string * expr * expr (*mn x = (A#, 5>octave>-5, 4 + 1 ) *)
    | ChordExpr of expr list * expr (* chord =  *)
    | ListExpr of expr list (*mn x = [a, b*6, c] ???*)
    | BinOp of expr * op * expr (* x + y *)
    | UnaryOp of uop * expr
    | MethodCall of string * expr list (*mn foo(x, y) *)
    | NoExpr (* for (;;) *)

type par_decl = {
    paramname : string; (* Name of the variable *)
    paramtype : cb_type; (* Name of variable type *)
}

type var_decl = {
    varname : string; (* Name of the variable *)
    vartype : cb_type; (* Name of variable type *)
}

type fullvdecl = {
    fvtype : cb_type;
    fvname : string;
    fvexpr : expr;
}

type meth_decl = {
    fname : string; (* Name of the function *)
    rettype : cb_type; (* Name of return type *)
    formals : par_decl list; (* Formal argument names *)
    body : innerblock list; }
and
stmt = (* Statements *)
    Expr of expr (* foo = bar + 3; *)
    | Return of expr (* return 42; *)
    | Block of innerblock list (* ) ... end *)
    | If of expr * innerblock list * stmt (*mn if (foo isnt 42) ... else ... end *)
    | Foreach of par_decl * string * innerblock list (*mn foreach (x in nots) ... end
*)
    | While of expr * innerblock list(*mn while (i<10) ... end *)
and
innerblock =
```

```
    Stmt2 of stmt
  | FullDecl2 of fullvdecl
  | VDecl2 of var_decl
and
generic =
    Stmt of stmt
  | FullDecl of fullvdecl
  | VDecl of var_decl
  | MDecl of meth_decl

type program =  generic list

let string_of_cbtype cbt =
    match cbt with
        Note -> "note"
        | Int -> "int"
        | Void -> "void"
        | Chord -> "chord"
        | Bool -> "bool" | Scale -> "scale" | Stanza -> "stanza" | Score -> "score";
        | _ -> "";;

let string_of_uop uop =
    match uop with
    | Raise -> "raise" | Lower -> "lower"

let string_of_pdecl var =
    string_of_cbtype var.paramtype ^ " " ^ var.paramname

let string_of_vdecl var =
    string_of_cbtype var.vartype ^ " " ^ var.varname ^ ";\n"

let rec string_of_expr = function
    Id(s) -> s
    | MemberAccess(id, mt) -> id ^ "." ^  mt
    | IntLiteral(l) -> string_of_int l
    (*| DurInt(c) -> string_of_int c*)
    | NoteConst(c) -> c
    | BoolLiteral(b) -> string_of_bool b;
    | DurConst(c) -> c
    | Assign(v, e) -> string_of_expr v ^ " = " ^ string_of_expr e (* expr = expr ??????
*)
    | NoteExpr(id, e1, e2) -> " ( " ^ id ^ " , " ^ string_of_expr e1 ^ " , " ^
string_of_expr e2 ^ " ) "
    | ChordExpr(l, e) -> "( [ " ^ String.concat " , " (List.map string_of_expr
(List.rev l)) ^ "] , " ^ string_of_expr e ^ " ) ";
    (*| TypeAssign(v, e) -> v ^ " = " ^ string_of_expr e (* Chord a = ... *) *)
  (*| ElementOp(s, e1) -> s ^ "[" ^ string_of_expr e1 ^ "]";*)
    | ListExpr(e) -> " [ " ^ String.concat " , " (List.map string_of_expr (List.rev e))
^ " ] "

(*type op =
    Add | Sub | Mult | Div | Mod
    | And | Or | Eq | NEq | Less | LEq | Greater | GEq | IDTimes*)

    | BinOp(e1, o, e2) -> begin
        string_of_expr e1 ^ " " ^
        (match o with
        Add -> "+" | Sub -> "-"
            (*| DotAdd -> ".+" | DotSub -> ".-" *)
            | Eq -> "==" | NEq -> "!="
            | Less -> "<" | LEq -> "<=" | Greater -> ">" | GEq -> ">="
```

```
            | And -> "&&" | Or -> "||" | Mod -> "%" | Mult -> "*" | Div -> "/") ^

            " " ^ string_of_expr e2
        end

    | UnaryOp(up, e) -> string_of_uop up ^ " " ^ string_of_expr e
    | MethodCall(f, el) ->
      f ^ "(" ^ String.concat " , " (List.map string_of_expr (List.rev el)) ^ " ) "
(*?????*)
    | NoExpr -> "";;

let string_of_op op =
    match op with
        Add -> "+" | Sub -> "-"
            (*| DotAdd -> ".+" | DotSub -> ".-" *)
            | Eq -> "==" | NEq -> "!="
            | Less -> "<" | LEq -> "<=" | Greater -> ">" | GEq -> ">="
            | And -> "&&" | Or -> "||" | Mod -> "%" | Mult -> "*" | Div -> "/"

let string_of_fvdl fvdel =
    string_of_cbtype fvdel.fvtype ^ " " ^ fvdel.fvname ^ " = " ^ string_of_expr
fvdel.fvexpr ^ ";\n"


let rec string_of_stmt = function
    Block(stmts) -> begin
        let rec string_of_abst = function (* inner block list *)
            [] -> ""
            | innerb::abstr ->
                match innerb with
                    Stmt2(st) -> string_of_stmt st ^ "\n" ^ string_of_abst abstr;
                    | VDecl2(v) -> string_of_vdecl v ^ "\n" ^ string_of_abst abstr;
                    | FullDecl2(fv) -> string_of_fvdl fv ^ "\n" ^ string_of_abst abstr;
        in
        "\n" ^  string_of_abst stmts ^ "\nend\n" (* ) Block end *)
        end
    | Expr(expr) -> string_of_expr expr ^ ";\n";
    | Return(expr) -> begin
        match expr with
            NoExpr -> "\n" (* print nothing user has to input Return ; though *)
            | _ -> "return " ^ string_of_expr expr ^ ";\n"; end
    (*| Break -> "break;\n";
        | Continue -> "continue;\n";*)
    | If(e, s, Block([])) ->

        "if (" ^ string_of_expr e ^ ")\n" ^ String.concat "\n" (List.map
                (fun innerb ->
                    match innerb with
                    Stmt2(st) -> string_of_stmt st ^ "\n";
                    | VDecl2(v) -> string_of_vdecl v ^ "\n" ;
                    | FullDecl2(fv) -> string_of_fvdl fv ^ "\n" ;
                )
                (List.rev s)) ^ "\n";

    | If(e, s1, s2) ->
        "if (" ^ string_of_expr e ^ ")\n" ^ String.concat "\n" (List.map
             (fun innerb ->
                match innerb with
                Stmt2(st) -> string_of_stmt st ^ "\n";
                | VDecl2(v) -> string_of_vdecl v ^ "\n" ;
                | FullDecl2(fv) -> string_of_fvdl fv ^ "\n" ;
```

```
            )
            (List.rev s1)) ^ "\n" ^ "else\n" ^ string_of_stmt s2;
    | Foreach(p, id, s) ->
        "foreach (" ^ string_of_pdecl p  ^ " in " ^ id ^ " )\n" ^ String.concat "\n"
(List.map
             (fun innerb ->
                    match innerb with
                    Stmt2(st) -> string_of_stmt st ^ "\n";
                    | VDecl2(v) -> string_of_vdecl v ^ "\n" ;
                    | FullDecl2(fv) -> string_of_fvdl fv ^ "\n" ;
                )
         (List.rev s));

    | While(e, s) -> "while (" ^ string_of_expr e ^ ")\n" ^ String.concat "\n"
(List.map
         (fun innerb ->
                    match innerb with
                    Stmt2(st) -> string_of_stmt st ^ "\n";
                    | VDecl2(v) -> string_of_vdecl v ^ "\n" ;
                    | FullDecl2(fv) -> string_of_fvdl fv ^ "\n" ;
                )
     (List.rev s));;

let string_of_mdecl mdecl =
    mdecl.fname ^ "(" ^ String.concat ", " (List.map string_of_pdecl mdecl.formals) ^
")\n" ^
   (* String.concat "" (List.map string_of_vdecl mdecl.locals) ^*)
    String.concat "" (List.map
         (fun innerb ->
                    match innerb with
                    Stmt2(st) -> string_of_stmt st ^ "\n";
                    | VDecl2(v) -> string_of_vdecl v ^ "\n" ;
                    | FullDecl2(fv) -> string_of_fvdl fv ^ "\n" ;
                )
     mdecl.body) ^ "\nend\n"

let rec string_of_program = function
   [] -> "";
    | generic::gen_lst ->
        match generic with
            VDecl(arg_vd) -> string_of_vdecl arg_vd ^ "" ^ string_of_program gen_lst ;
            | MDecl(arg_mt) -> string_of_mdecl arg_mt ^ "" ^ string_of_program gen_lst
;
            | FullDecl(arg_vd) -> string_of_fvdl arg_vd ^ "" ^ string_of_program
gen_lst ;
            | Stmt(arg_st) -> string_of_stmt arg_st ^ "" ^string_of_program gen_lst ;;
```

translate.ml

```
open Ast
open Printf
open String

module NameMap = Map.Make(struct
    type t = string
    let compare x y = Pervasives.compare x y
end)

type note = {
    mutable pitch : int;
    mutable octave : int;
    mutable duration : int;
}

type chord = {
    mutable notelist : note list;
    mutable chord_duration : int;
}

type scale = {
    mutable scale_notelist : note list;
}

type stanza = {
    mutable chordlist : chord list;
}

type score = {
    mutable stanzalist : stanza list;
    mutable instrument : int;
}


type cb_type =   Int of int | Bool of bool | Note of note | Chord of chord | Scale of
scale | Stanza of stanza | Score of score

exception ReturnException of cb_type * cb_type NameMap.t

let getType v =
    match v with
        Int(v) -> "int"
        | Bool(v) -> "bool"
        | Note(v) -> "note"
        | Chord(v) -> "chord"
        | Scale(v) -> "scale"
        | Stanza(v) -> "stanza"
        | Score(v) -> "score"

let getInt v =
    match v with
        Int(v) -> v
        | _ -> 0

let getNote v =
    match v with
        Note(v) -> v
        | _ -> {pitch=128; octave=0; duration=0}
```

```
let getChord v =
    match v with
        Chord(v) -> v
        | _ -> {notelist=[]; chord_duration=0}

let getBool v =
    match v with
        Bool(v) -> v
        | _ -> false

let getScale v =
    match v with
        Scale(v) -> v
        | _ -> {scale_notelist=[]}

let getStanza v =
    match v with
        Stanza(v) -> v
        | _ -> {chordlist=[]}

let getScore v =
    match v with
        Score(v) -> v
        | _ -> {stanzalist=[]; instrument=0}

let initIdentifier t =
  match t with
    "int" -> Int(1)
    | "bool" -> Bool(false)
    | "note" -> Note({pitch=128; octave=0; duration=0})
    | "chord" -> Chord({notelist=[]; chord_duration=0})
    | "scale" -> Scale({scale_notelist=[]})
    | "stanza" -> Stanza({chordlist=[]})
    | "score" -> Score({stanzalist=[]; instrument=0})
    | _ -> Bool(false)

 let noteMap =
     NameMap.add "R" (-1) NameMap.empty
    let noteMap = NameMap.add "C" 0 noteMap
    let noteMap = NameMap.add "B#" 0 noteMap
    let noteMap = NameMap.add "C#" 1 noteMap
    let noteMap = NameMap.add "Db" 1 noteMap
    let noteMap = NameMap.add "D" 2 noteMap
    let noteMap = NameMap.add "Eb" 3 noteMap
    let noteMap = NameMap.add "D#" 3 noteMap
    let noteMap = NameMap.add "E" 4 noteMap
    let noteMap = NameMap.add "Fb" 4 noteMap
    let noteMap = NameMap.add "F" 5 noteMap
    let noteMap = NameMap.add "E#" 5 noteMap
    let noteMap = NameMap.add "F#" 6 noteMap
    let noteMap = NameMap.add "Gb" 6 noteMap
    let noteMap = NameMap.add "G" 7 noteMap
    let noteMap = NameMap.add "Ab" 8 noteMap
    let noteMap = NameMap.add "G#" 8 noteMap
    let noteMap = NameMap.add "A" 9 noteMap
    let noteMap = NameMap.add "Bb" 10 noteMap
    let noteMap = NameMap.add "A#" 10 noteMap
    let noteMap = NameMap.add "B" 11 noteMap
    let noteMap = NameMap.add "Cb" 11 noteMap

let defaultInitMap =
```

```
    NameMap.add "int" " = 0;\n" NameMap.empty
    let defaultInitMap = NameMap.add "bool" " = false;\n" defaultInitMap
    let defaultInitMap = NameMap.add "note" " = new note();\n" defaultInitMap
    let defaultInitMap = NameMap.add "chord" " = new chord();\n" defaultInitMap
    let defaultInitMap = NameMap.add "scale" " = new scale();\n" defaultInitMap
    let defaultInitMap = NameMap.add "stanza" " = new stanza();\n" defaultInitMap
    let defaultInitMap = NameMap.add "score" " = new score();\n" defaultInitMap

(*this will need to be passed around*)
let composeJava = ref ""
let methJava = ref ""

let methJava = ref ""
let globalJava = ref ""
let mainJava = ref ""

let import_decl =
"import java.io.File;
import java.io.IOException;
import java.util.ArrayList;
import javax.sound.midi.InvalidMidiDataException;
import javax.sound.midi.MidiEvent;
import javax.sound.midi.MidiSystem;
import javax.sound.midi.Sequence;
import javax.sound.midi.ShortMessage;
import javax.sound.midi.Track;"

let class_start =
"
class note {

    public int pitch;
    public int octave;
    public int duration;

    public note() {
        pitch = 0;
        octave = 0;
        duration = 0;
    }

    public note(int p, int o, int d) {
        pitch = p;
        octave = o;
        duration = d;
    }

    public void inc_oct(){
        this.octave++;
    }
    public void dec_oct(){
        this.octave--;
    }


    public boolean isValid() {
        return -1 <= pitch && 11 >= pitch && -5 <= octave && 5 <= octave;
    }

    public String toString()  {
        return \"(\" + pitch + \",\" + octave + \",\" + duration + \")\";
```

```
    }

    public chord toChord() {
        ArrayList<note> tmp_list = new ArrayList<note>(1);
        tmp_list.add(new note(pitch, octave, duration));
        return new chord(tmp_list, duration);
    }

    public note deepCopy() {
        return new note(this.pitch, this.octave, this.duration);
    }
}

class chord {

    public ArrayList<note> notelist;
    public int chord_duration;

    public chord() {
        notelist = new ArrayList<note>();
        chord_duration = 0;
    }

    public chord(ArrayList<note> nl, int cd) {
        notelist = nl;
        chord_duration = cd;
    }

    public String toString() {
        String s = \"([\";
        for(note n : notelist) {
            s += n.toString();
        }
        return s + \"], \" + chord_duration + \")\";
    }

    private ArrayList<note> nlCopy() {
        ArrayList<note> tmp = new ArrayList<note>(notelist.size());
        for(note n : notelist) {
            tmp.add(n.deepCopy());
        }
        return tmp;
    }

    public chord deepCopy() {
        return new chord(nlCopy(), this.chord_duration);
    }
}

class scale {

    public ArrayList<note> scale_notelist;

    public scale() {
        scale_notelist = new ArrayList<note>();
    }

    public scale(ArrayList<note> snl) {
        scale_notelist = snl;
    }
```

```
    public String toString() {
        String s = \"[\";
        for(note n : scale_notelist) {
            s += n.toString();
        }
        return s + \"]\";
    }

    public scale deepCopy() {
        ArrayList<note> tmp = new ArrayList<note>(scale_notelist.size());
        for(note n : scale_notelist) {
            tmp.add(n.deepCopy());
        }
        return new scale(tmp);
    }
}

class stanza {

    public ArrayList<chord> chordlist;

    public stanza() {
        chordlist = new ArrayList<chord>();
    }

    public stanza(ArrayList<chord> cl) {
        chordlist = cl;
    }

    public String toString() {
        String s = \"[\";
        for(chord c : chordlist) {
            s += c.toString();
        }
        return s + \"]\";
    }

    public stanza deepCopy() {
        ArrayList<chord> tmp = new ArrayList<chord>(chordlist.size());
        for(chord c : chordlist) {
            tmp.add(c.deepCopy());
        }
        return new stanza(tmp);
    }
}

class score {

    public ArrayList<stanza> stanzalist;
    public int instrument;

    public score() {
        stanzalist = new ArrayList<stanza>();
        instrument = 0;
    }
    public score(ArrayList<stanza> sl) {
        stanzalist = sl;
        instrument = 0;
    }

    public score(ArrayList<stanza> sl, int i) {
```

```
        stanzalist = sl;
        instrument = i;
    }

    public String toString() {
        String s = \"([\";
        for(stanza st : stanzalist) {
            s += st.toString();
        }
        return s + \"], instrument=\" + instrument + \")\";
    }

    public score deepCopy() {
        ArrayList<stanza> tmp = new ArrayList<stanza>(stanzalist.size());
        for(stanza s : stanzalist) {
            tmp.add(s.deepCopy());
        }
        return new score(tmp, this.instrument);
    }
}

public class Cb {
    /**
     * ******************compose helper functions**************************
     * source: http://www.penguinpeepshow.com/CSV2MIDI.php
     * helper function for mapping values of pitch and octave to range 0 - 127
     */
    long map(long x, long in_min, long in_max, long out_min, long out_max) {
        return (x - in_min) * (out_max - out_min) / (in_max - in_min) + out_min;
    }
    /*
     * note representation: tick, duration, pitch, volume/loudness/velocity
     * turns note on
     */

    private MidiEvent createNoteOnEvent(int nKey, long lTick, int channel, int
velocity) {
        return createNoteEvent(ShortMessage.NOTE_ON, nKey, velocity, lTick, channel);
    }

    /*
     * turns note off
     */
    private MidiEvent createNoteOffEvent(int nKey, long lTick, int channel) {
        return createNoteEvent(ShortMessage.NOTE_OFF, nKey, 0, lTick, channel);  //set
note to 0 velocity
    }

    /*
     * turns note on or off
     */
    private MidiEvent createNoteEvent(int nCommand, int nKey, int nVelocity, long
lTick, int channel) {
        ShortMessage message = new ShortMessage();
        try {
            message.setMessage(nCommand, channel, nKey, nVelocity);
        } catch (InvalidMidiDataException e) {
            e.printStackTrace();
            System.exit(1);
        }
        MidiEvent event = new MidiEvent(message, lTick);
```

```
        return event;
    }
    /*
     * Here are all of the built in methods that we have and while we
     * are translating we need to make sure you do not overwrite any of these
     * bad boys unless you give it different args (or maybe we can just let
     * java take care of that, I am not fully certain)
     * Note: this score list has to hold at most 16 stanzas
     */
    public void compose(ArrayList<score> data) throws InvalidMidiDataException {
        int nChannels = data.size();
        Sequence sequence = null;


        for (int i = 0; i < data.size(); i++) {
            System.out.println(data.get(i));
        }


        //***** Read in timing resolution and instruments *****
        int timingRes = 4, instrument[] = new int[nChannels];

        //read in instrument numbers
        for (int inst = 0; inst < data.size(); inst++) {
            //check if this is an integer
            instrument[inst] = data.get(inst).instrument; //this is a number, it has to
be an intrument
            System.out.println(\"Instrument set to \" + instrument[inst] + \" on
channel \" + inst);
        }

        //***** Initialize Sequencer *****
        try {
            sequence = new Sequence(Sequence.PPQ, timingRes);   //initialize sequencer
with timingRes
        } catch (InvalidMidiDataException e) {
            e.printStackTrace();
            System.exit(1);
        }


        //***** Create tracks and notes *****
        /* Track objects cannot be created by invoking their constructor
         directly. Instead, the Sequence object does the job. So we
         obtain the Track there. This links the Track to the Sequence
         automatically.
         */
        Track track[] = new Track[nChannels];
        for (int i = 0; i < nChannels; i++) {
            track[i] = sequence.createTrack();                    //create tracks

            ShortMessage sm = new ShortMessage();
            sm.setMessage(ShortMessage.PROGRAM_CHANGE, i, instrument[i], 0);  //put in
instrument[i] in this track
            track[i].add(new MidiEvent(sm, 0));
        }

        int nt = 0,
                tick = 0,
                duration = 5,
                velocity = 100; // this is the volume of the sound
```

```
        for (int channel = 0; channel < data.size(); channel++) {
            //populating the ith track
            ArrayList<stanza> stan = data.get(channel).stanzalist;
            for (int tr = 0; tr < stan.size(); tr++) {

                ArrayList<chord> chl = stan.get(tr).chordlist;
                for (int cl = 0; cl < chl.size(); cl++) { //chord list

                    ArrayList<note> tnote = chl.get(cl).notelist;

                    for (int nti = 0; nti < tnote.size(); nti++) { //note list

                        duration = (int) ((chl.get(cl).chord_duration / 4) );  //second
number is duration
                        System.out.println(\"dur:-\" + duration + \"> \\n \" );
                        // octave (-5 to 5); pitch (0 to 11)
                        nt = (4 + tnote.get(nti).octave) * 12 +
tnote.get(nti).pitch;//(int) map((long) (tnote.get(nti).pitch + tnote.get(nti).octave),
-5, 16, 0, 127); //this is the pitch representation; middle c = c_4 = 60
                        velocity = 120;  //velocity can not be changed for now

                        if (tnote.get(nti).pitch < 0) { // a rest is received -- any
negative note is rest
                            nt = 0;
                            //track[channel].add(createNoteOffEvent(nt, tick,
channel));              //add note to this track

                        } else {
                            track[channel].add(createNoteOnEvent(nt, tick, channel,
velocity));             //add note to this track
                        }
                        // tick = tick + duration;  //first number is tick
                        // track[channel].add(createNoteOffEvent(nt, tick + duration,
channel));
                    }
                    tick = tick + duration;
                     System.out.println(\"tick:-\" + tick + \"> \\n \" + duration);
                }
            }
            tick = 0;
        }


        // Print track information
        // Print track information

        System.out.println();

        if (track != null) {

            for (int i = 0; i < track.length; i++) {

                System.out.println(\"Track \" + i + \":\");



                for (int j = 0; j < track[i].size(); j++) {

                    MidiEvent event = track[i].get(j);
```

```
                    System.out.println(\" tick \" + event.getTick() + \", \" +
MessageInfo.toString(event.getMessage()));

                }

            }

        }


        /* Now we just save the Sequence to the file we specified.
         The '0' (second parameter) means saving as SMF type 0.
         (type 1 is for multiple tracks).
         */
        try {
            MidiSystem.write(sequence, 1, new File(\"out.mid\"));
        } catch (IOException e) {
            e.printStackTrace();
            System.exit(1);
        }
    }

    // assumes it is passed a minor scale
    public chord major(scale s) throws Exception {
        ArrayList<note> notes = new ArrayList<note>();
        if(s.scale_notelist.size() >= 5){
            notes.add(s.scale_notelist.get(0));
            notes.add(flat(s.scale_notelist.get(2)));
            notes.add(s.scale_notelist.get(4));
        }
        else
            throw new Exception(\"To convert a scale into a major chord, you must pass
a scale of at least five notes\");

        int duration = s.scale_notelist.get(0).duration;

        return new chord(notes, duration);
    }

    public chord major(scale s, int duration) throws Exception {
        ArrayList<note> notes = new ArrayList<note>();
        if(s.scale_notelist.size() >= 5){
            notes.add(s.scale_notelist.get(0));
            notes.add(flat(s.scale_notelist.get(2)));
            notes.add(s.scale_notelist.get(4));
        }
        else
            throw new Exception(\"To convert a scale into a major chord, you must pass
a scale of at least five notes\");
        return new chord(notes, duration);
    }

    public chord minor(scale s) throws Exception {
        ArrayList<note> notes = new ArrayList<note>();
        if(s.scale_notelist.size() >= 5){
            notes.add(s.scale_notelist.get(0));
            notes.add(flat(s.scale_notelist.get(2)));
            notes.add(s.scale_notelist.get(4));
        }
        else
```

```
            throw new Exception(\"To convert a scale into a minor chord, you must pass
a scale of at least five notes\");
        int duration = s.scale_notelist.get(0).duration;
        return new chord(notes, duration);
    }

    public chord minor(scale s, int duration) throws Exception {
        ArrayList<note> notes = new ArrayList<note>();
        if(s.scale_notelist.size() >= 5){
            notes.add(s.scale_notelist.get(0));
            notes.add(flat(s.scale_notelist.get(2)));
            notes.add(s.scale_notelist.get(4));
        }
        else
            throw new Exception(\"To convert a scale into a minor chord, you must pass
a scale of at least five notes\");
        return new chord(notes, duration);
    }


    public note sharp(note n) throws Exception {
        note sharped = new note();
        if(n.pitch == 11)
            sharped = new note(0, n.octave+1, n.duration);
        else
            sharped = new note(n.pitch + 1, n.octave, n.duration);
        if(sharped.isValid())
            return sharped;
        else
            throw new Exception(\"To convert a scale into a minor chord, you must pass
a scale of at least five notes\");
    }

    public note flat(note n) {
        return new note();
    }

    /**
     * Return a random integer between 0 and i
     * @param i The maximum value for the integer
     * @return A random integer between 0 and i
     */
    public int randint(int i) {
        return (int)(Math.random()*i);
    }

    public chord chordOfNote(note n) {
        return n.toChord();
    }

    public chord rest(int d) {
        ArrayList<note> temp = new ArrayList<note>(1);
        temp.add(new note(-1, 0, d));
        return new chord(temp, d);
    }

    public chord prepend(note n, chord c) {
        chord tmp = c.deepCopy();
        tmp.notelist.add(0, n.deepCopy());
        return tmp;
    }
```

```
public scale prepend(note n, scale s) {
    scale tmp = s.deepCopy();
    tmp.scale_notelist.add(0, n.deepCopy());
    return tmp;
}

public stanza prepend(chord c, stanza s) {
    stanza tmp = s.deepCopy();
    tmp.chordlist.add(0, c.deepCopy());
    return tmp;
}

public score prepend(stanza st, score sc) {
    score tmp = sc.deepCopy();
    tmp.stanzalist.add(0, st.deepCopy());
    return tmp;
}

public chord append(note n, chord c) {
    chord tmp = c.deepCopy();
    tmp.notelist.add(n.deepCopy());
    return tmp;
}

public scale append(note n, scale s) {
    scale tmp = s.deepCopy();
    tmp.scale_notelist.add(n.deepCopy());
    return tmp;
}

public stanza append(chord c, stanza s) {
    stanza tmp = s.deepCopy();
    tmp.chordlist.add(c.deepCopy());
    return tmp;
}

public score append(stanza st, score sc) {
    score tmp = sc.deepCopy();
    tmp.stanzalist.add(st.deepCopy());
    return tmp;
}

public scale concat(scale s1, scale s2) {
    scale tmp = s1.deepCopy();
    for(note n : s2.scale_notelist) {
        tmp.scale_notelist.add(n.deepCopy());
    }
    return tmp;
}

public stanza concat(stanza s1, stanza s2) {
    stanza tmp = s1.deepCopy();
    for(chord c : s2.chordlist) {
        tmp.chordlist.add(c.deepCopy());
    }
    return tmp;
}

public score concat(score s1, score s2) {
    score tmp = s1.deepCopy();
```

```
        for(stanza s : s2.stanzalist) {
            tmp.stanzalist.add(s.deepCopy());
        }
        return tmp;
    }

    public scale repeat(note n, int i) throws Exception {
        scale tmp = new scale();
        if (i < 1) {
            throw new Exception(\"repeat function takes an integer that must be 1 or
greater\");
        }
        for(int j = 0; j < i; j++) {
            tmp.scale_notelist.add(n.deepCopy());
        }
        return tmp;
    }

    public stanza repeat(chord c, int i) throws Exception {
        stanza tmp = new stanza();
        if (i < 1) {
            throw new Exception(\"repeat function takes an integer that must be 1 or
greater\");
        }
        for(int j = 0; j < i; j++) {
            tmp.chordlist.add(c.deepCopy());
        }
        return tmp;
    }

    public score repeat(stanza s, int i) throws Exception{
        score tmp = new score();
        if (i < 1) {
            throw new Exception(\"repeat function takes an integer that must be 1 or
greater\");
        }
        tmp.instrument = 0;
        for(int j = 0; j < i; j++) {
            tmp.stanzalist.add(s.deepCopy());
        }
        return tmp;
    }

    public score repeat(score s, int i) throws Exception{
        score tmp = new score();
        if (i < 1) {
            throw new Exception(\"repeat function takes an integer that must be 1 or
greater\");
        }
        for(int j = 0; j < i; j++) {
            for(stanza st : s.stanzalist) {
                tmp.stanzalist.add(st.deepCopy());
            }
        }
        tmp.instrument = s.instrument;
        return tmp;
    }

"

let main_start =
```

```
"
    public static void main(String[] args) throws Exception { Cb runner = new Cb();
runner.run(); }
"

let run_start = "public void run() throws Exception {"

let block_start = " {
"

let block_end = " }
"

let run_end =
"
    }
"

let class_end =
"
}
"

let rec eval env = function
    Id(name) -> (* (print_string ("Evaluating ID: " ^ name ^ "\n")); *)
        let locals, globals, fdecls = env in
            if NameMap.mem name locals then
                (NameMap.find name locals), env, name
            else if NameMap.mem name globals then
                (NameMap.find name globals), env, name
            else raise (Failure ("undeclared identifier: " ^ name))
    | MemberAccess(vname, memname) -> (* (print_string ("Evaluating Member Access: " ^
vname ^ " Member: " ^ memname ^ "\n")); *)
        let v, env, asJava = eval env (Id vname) in
            let vType = getType v in
            (match vType with
              | "note" ->
                (match memname with
                  "pitch" -> (initIdentifier "int")
                  | "octave" -> (initIdentifier "int")
                  | "duration" -> (initIdentifier "int")
                  | _ -> raise (Failure ("invalid property of note: " ^ memname)))
              | "chord" ->
                (match memname with
                    "chord_duration" -> (initIdentifier "int")
                    | "notelist" -> (initIdentifier "scale")
                  | _ -> raise (Failure ("invalid property of chord: " ^ memname)))
              | "score" ->
                (match memname with
                   "instrument" -> (initIdentifier "int")
                  | _ -> raise (Failure ("invalid property of score: " ^ memname)))
              | "scale" ->
                (match memname with
                  "scale_notelist" -> (initIdentifier "scale")
                  | _ -> raise (Failure ("invalid property of score: " ^ memname)))
              | _ -> raise (Failure ("cannot access " ^ vname ^ "." ^ memname))), env,
(asJava ^ "." ^ memname ^ " ")
    | IntLiteral(i) -> (* (print_string ("Evaluating an intlit: " ^ (string_of_int i) ^
"\n")); *)
        (initIdentifier "int"), env, (string_of_int i)
    | NoteConst(s) -> (* (print_string ("Evaluating a noteConst: " ^ s ^ "\n")); *)
```

```
        (initIdentifier "int"), env, (string_of_int (NameMap.find s noteMap))
    | BoolLiteral(b) -> (* (print_string ("Evaluating a bool literal: " ^
(string_of_bool b) ^ "\n")); *)
        (initIdentifier "bool"), env, (string_of_bool b)
    | ChordExpr(el, e) -> (* (print_string ("Evaluating a chordexpr\n")); *)
        let note_list = List.map (fun (note_elem) ->
            (let chord_elem, env, asJava = eval env note_elem in
                let vType = (getType chord_elem) in
                    if ( vType = "note") then (getNote (chord_elem))
                    else raise (Failure ("Chord must be composed of notes "))
            )) el in
                let javaStrList = List.map (fun (note_elem) ->
                    (let chord_elem, env, asJava = eval env note_elem in
                        let vType = (getType chord_elem) in
                            if ( vType = "note") then ("add(" ^ asJava ^ ");")
                            else raise (Failure ("Chord must be composed of notes "))
                    )) el in
                let chordAsJava = String.concat "\n" javaStrList in
                    let dur, env, durAsJava = eval env e in
                        let durType = getType dur in
                            if durType = "int" then
                                (initIdentifier "chord"),env,("new chord(new
ArrayList<note>() {{\n" ^ chordAsJava ^ "}}, " ^ durAsJava ^ ")")
                            else raise (Failure ("Duration does not evaluate to an
integer"))
    | DurConst(s) -> (* (print_string ("Evaluating a durConst: " ^ s ^ "\n")); *)
        if s = "whole" then (initIdentifier "int"), env, "64"
            else if s = "half" then (initIdentifier "int"), env, "32"
            else if s = "quarter" then (initIdentifier "int"), env, "16"
            else raise (Failure ("Duration constant unknown"))
    | NoteExpr(s,e,e1) -> (* (print_string ("Evaluating a NoteExpr\n")); *)
        let oct, env, octAsJava = eval env e in
            let octType = getType oct in
                if octType = "int" then (let dur, env, durAsJava = eval env e1 in
                                    let durType = getType dur in
                                        if durType = "int" then
                                            (initIdentifier "note"),env,(" new note(" ^
(string_of_int (NameMap.find s noteMap)) ^ "," ^ octAsJava ^ "," ^ durAsJava ^ ")")
                                        else raise (Failure ("Duration does not
evaluate to an integer")))
                else  raise (Failure ("Octave does not evaluate to an integer"))
    | BinOp(e1,o,e2) -> (* (print_string ("Evaluating a binop\n")); *)
        let v1, env, v1AsJava = eval env e1 in
        let v2, env, v2AsJava = eval env e2 in
        let v1Type = getType v1 in
        let v2Type = getType v2 in
        (* Two variables have to be of the same type for binop *)
        if v1Type = v2Type then
            (match o with (* Only accept ints for now *)
                Add ->
                    if v1Type = "int" then
                         (initIdentifier "int")
                    else raise (Failure ("incorrect type: " ^ v1Type ^ " + " ^ v2Type))
                | Sub ->
                    if v1Type = "int" then
                        (initIdentifier "int")
                    else raise (Failure ("incorrect type: " ^ v1Type ^ " - " ^ v2Type))
                | Mult ->
                    if v1Type = "int" then
                        (initIdentifier "int")
                    else raise (Failure ("incorrect type: " ^ v1Type ^ " * " ^ v2Type))
```

```
                | Div ->
                    if v1Type = "int" then
                        (initIdentifier "int")
                    else raise (Failure ("incorrect type: " ^ v1Type ^ " / " ^ v2Type))
                | Mod -> (* (print_string ("Doing a mod binop\n")); *)
                    if v1Type = "int" then
                        (initIdentifier "int")
                    else raise (Failure ("incorrect type: " ^ v1Type ^ " % " ^ v2Type))
                | And ->
                    if v1Type = "bool" then
                        (initIdentifier "bool")
                    else raise (Failure ("incorrect type: " ^ v1Type ^ " and " ^
v2Type))
                | Or ->
                    if v1Type = "bool" then
                        (initIdentifier "bool")
                    else raise (Failure ("incorrect type: " ^ v1Type ^ " or " ^
v2Type))
                | Eq ->
                    if v1Type = "int" then
                        (initIdentifier "bool")
                    else raise (Failure ("incorrect type: " ^ v1Type ^ " is " ^
v2Type))
                | NEq ->
                    if v1Type = "int" then
                        (initIdentifier "bool")
                    else
                        raise (Failure ("incorrect type: " ^ v1Type ^ " isnt " ^
v2Type))
                | Less ->
                    if v1Type = "int" then
                        (initIdentifier "bool")
                    else raise (Failure ("cannot compare: " ^ v1Type ^ " < " ^ v2Type))
                | LEq ->
                    if v1Type = "int" then
                        (initIdentifier "bool")
                    else raise (Failure ("cannot compare: " ^ v1Type ^ " <= " ^
v2Type))
                | Greater ->
                    if v1Type = "int" then
                        (initIdentifier "bool")
                    else raise (Failure ("cannot compare: " ^ v1Type ^ " > " ^ v2Type))
                | GEq ->
                    if v1Type = "int" then
                        (initIdentifier "bool")
                    else raise (Failure ("cannot compare: " ^ v1Type ^ " >= " ^
v2Type))
                | _ -> raise (Failure ("Unknown binary operation"))
            ), env, (v1AsJava ^ (string_of_op o) ^ v2AsJava)
        else raise (Failure ("type mismatch: " ^ v1Type ^ " and " ^ v2Type))
    | MethodCall("print", [e]) ->
        let arg, env, eAsJava = eval env e in
            (Bool true), env, ("System.out.println(" ^ eAsJava ^ ")")
    | MethodCall("major", [e; dur]) ->
        let arg, env, eAsJava = eval env e in
        let arg2, env, durAsJava = eval env dur in
            if getType arg = "scale" && getType arg2 = "int" then
                (initIdentifier "chord"), env, ("major(" ^ eAsJava ^ "," ^ durAsJava ^
")")
            else raise (Failure ("argument of major must be a scale"))
    | MethodCall("minor", [e; dur]) ->
```

```
        let arg, env, eAsJava = eval env e in
        let arg2, env, durAsJava = eval env dur in
            if (getType arg = "scale") && (getType arg2 = "int") then
                (initIdentifier "chord"), env, ("minor(" ^ eAsJava ^ "," ^ durAsJava ^
")")
            else raise (Failure ("argument of minor must be a scale"))
    | MethodCall("sharp", [e]) ->
        let arg, env, eAsJava = eval env e in
            if getType arg = "note" then
                (initIdentifier "note"), env, ("sharp(" ^ eAsJava ^ ")")
            else raise (Failure ("argument of flat must be a note"))
    | MethodCall("flat", [e]) ->
        let arg, env, eAsJava = eval env e in
            if getType arg = "note" then
                (initIdentifier "note"), env, ("flat(" ^ eAsJava ^ ")")
            else raise (Failure ("argument of flat must be a note"))
    | MethodCall("randint", [e]) ->
        let v, env, eAsJava = eval env e in
            if getType v = "int" then
                (initIdentifier "int"), env, ("randint(" ^ eAsJava ^ ")")
            else raise (Failure ("argument of randint must be an integer"))
    | MethodCall("chordOfNote",[e]) ->
        let v, env, eAsJava = eval env e in
            if getType v = "note" then
                (initIdentifier "chord"), env, ("chordOfNote(" ^ eAsJava ^ ")")
            else raise (Failure ("argument of chordOfNote must be a note"))
    | MethodCall("rest", [e]) ->
        let v, env, eAsJava = eval env e in
            if getType v = "int" then
                (initIdentifier "chord"), env, ("rest(" ^ eAsJava ^ ")")
            else raise (Failure ("argument of rest must be an integer"))
    | MethodCall("prepend", [item; alist]) ->
        let arg1, env, itemAsJava = eval env item in
            let arg2, env, listAsJava = eval env alist in
                if getType arg1 = "note" then
                    (if getType arg2 = "scale" then
                        (initIdentifier "scale"), env, ("prepend(" ^ itemAsJava ^ "," ^
listAsJava ^ ")")
                    else if getType arg2 = "chord" then (* Returns a new chord with the
note appended *)
                        (initIdentifier "chord"), env, ("prepend(" ^ itemAsJava ^ "," ^
listAsJava ^ ")")
                    else raise (Failure ("A note can only be prepended to a chord or
scale")))
                else if getType arg1 = "chord" then
                    (if getType arg2 = "stanza" then
                        (initIdentifier "stanza"), env, ("prepend(" ^ itemAsJava ^ ","
^ listAsJava ^ ")")
                    else raise (Failure ("A chord can only be prepended to a stanza")))
                else if getType arg1 = "stanza" then
                    (if getType arg2 = "score" then
                        (initIdentifier "score"), env, ("prepend(" ^ itemAsJava ^ "," ^
listAsJava ^ ")")
                    else raise (Failure ("a stanza can only be prepended to a score")))
                else raise (Failure ("First argument for prepend must be of type note,
chord, or stanza"))
    | MethodCall("append", [item; alist]) ->
        let arg1, env, itemAsJava = eval env item in
            let arg2, env, listAsJava = eval env alist in
                if getType arg1 = "note" then
                    (if getType arg2 = "scale" then
```

```
                        (initIdentifier "scale"), env, ("append(" ^ itemAsJava ^ "," ^
listAsJava ^ ")")
                    else if getType arg2 = "chord" then (* Returns a new chord with the
note appended *)
                        (initIdentifier "chord"), env, ("append(" ^ itemAsJava ^ "," ^
listAsJava ^ ")")
                    else raise (Failure ("A note can only be appended to a chord or
scale")))
                else if getType arg1 = "chord" then
                    (if getType arg2 = "stanza" then
                        (initIdentifier "stanza"), env, ("append(" ^ itemAsJava ^ "," ^
listAsJava ^ ")")
                    else raise (Failure ("A chord can only be appended to a stanza")))
                else if getType arg1 = "stanza" then
                    (if getType arg2 = "score" then
                        (initIdentifier "score"), env, ("append(" ^ itemAsJava ^ "," ^
listAsJava ^ ")")
                    else raise (Failure ("a stanza can only be appended to a score")))
                else raise (Failure ("First argument for append must be of type note,
chord, or stanza"))
    | MethodCall("concat", [list1; list2]) ->
        let arg1, env, arg1AsJava = eval env list1 in
            let arg2, env, arg2AsJava = eval env list2 in
                if getType arg1 = getType arg2 then
                    (if getType arg1 = "stanza" then
                        (initIdentifier "stanza"), env, ("concat(" ^ arg1AsJava ^ "," ^
arg2AsJava ^ ")")
                    else if getType arg1 = "scale" then
                        (initIdentifier "scale"), env, ("concat(" ^ arg1AsJava ^ "," ^
arg2AsJava ^ ")")
                    else if getType arg1 = "score" then
                        (initIdentifier "score"), env, ("concat(" ^ arg1AsJava ^ "," ^
arg2AsJava ^ ")")
                    else raise (Failure ("concat works only on stanzas, scales, and
scores")))
                else raise (Failure ("Both arguments to concat must be of the same
type"))
    | MethodCall("repeat", [e; n]) -> (*Takes the argument and returns its container
type with arg repeated n times*)
        let arg1, env, eAsJava = eval env e in
            let arg2, env, nAsJava = eval env n in
                if getType arg2 = "int" then
                    (if getType arg1 = "note" then
                        (initIdentifier "scale"), env, ("repeat(" ^ eAsJava ^ "," ^
nAsJava ^ ")")
                    else if getType arg1 = "chord" then
                        (initIdentifier "stanza"), env, ("repeat(" ^ eAsJava ^ "," ^
nAsJava ^ ")")
                    else if getType arg1 = "stanza" then
                        (initIdentifier "score"), env, ("repeat(" ^ eAsJava ^ "," ^
nAsJava ^ ")")
                    else if getType arg1 = "score" then
                        (initIdentifier "score"), env, ("repeat(" ^ eAsJava ^ "," ^
nAsJava ^ ")")
                    else raise (Failure ("The first argument must be a note, chord,
stanza, or score")))
                else raise (Failure ("The second argument to repeat must be an integer
number of times to repeat"))
    | MethodCall("compose", e) ->  (* Writes the specified part to a java file to be
written into midi *)
        ignore(
```

```
            if ( (List.length e) > 16) then
                raise (Failure ("only up to 16 scores can be composed at once"));
        );
        let score_names = (List.map ( fun e1 ->  match e1 with
                                        Id(i) -> i;
                                        | _ ->  raise (Failure ("compose takes an
identifier as input"))) (List.rev e);
                            );
        in
        let actuals, env = (* make sure all ids are known *)
            List.fold_left (fun (al, env) actual ->
                                let v, env, _ = ((eval env) actual) in (v :: al), env
                            ) ([], env) e;
        in
        ignore (
            List.map (fun act ->  (* ids need to be scores *)
                        match (getType act) with
                            "score" -> act; (* print_string ("score"^ "\n\n"); *)
                            | _ -> raise (Failure ("compose takes a score only"));
                        ) (List.rev actuals);
        );
        let scoreListAsJava = String.concat "\n" (List.map (fun scor ->
                                            "\n\tadd("^ scor ^");"
                                        ) (List.rev score_names);)
        in Bool true, env, ("\n  compose(new ArrayList<score>() {{" ^ scoreListAsJava ^
"\n}})")
    | MethodCall(name, el) -> (* Check that method exists and passing correct args, do
not actually call *)
        (* (print_string ("User defined method call: " ^ name ^ "\n")); *)
        let locals, globals, fdecls = env in
            let fdecl = try (NameMap.find name fdecls)
                        with Not_found -> raise (Failure ("Undefined function: " ^
name))
            in
                let actuals, env = List.fold_left
                    (fun (al, env) actual ->
                        let v, env, _ = ((eval env) actual) in (v :: al), env
                    ) ([], env) el
                in
                let actualsAsJava = String.concat "," (List.map(fun arg -> let _, _,
asJava = eval env arg in asJava)el) in
                    let l1 =
                        try List.fold_left2 (fun locals formal actual ->
                                                if (getType actual) = (string_of_cbtype
formal.paramtype) then
                                                    (NameMap.add formal.paramname
(initIdentifier (getType actual)) locals)
                                                else
                                                    raise (Failure ("Wrong parameter
type in method call to " ^ fdecl.fname))
                                            ) NameMap.empty fdecl.formals (List.rev
actuals)
                        with Invalid_argument(_) -> raise (Failure ("wrong number of
arguments to: " ^ fdecl.fname))
                    in (initIdentifier (string_of_cbtype fdecl.rettype)), (l1, globals,
fdecls), (name ^ "(" ^ actualsAsJava ^ ")")
    | UnaryOp(uo,e) -> (* (print_string ("Evaluating a unary op\n")); *)
        let v, env, eAsJava = eval env e in
        let vType = getType v in
        if ( vType = "note" or vType = "chord" ) then
            (match uo with (* Only accept notes for now *)
```

```
                Raise ->
                    if vType = "note" then
                        (initIdentifier "note"), env, (eAsJava ^ ".inc_oct()")
                    else if vType = "chord" then
                            (initIdentifier "chord"), env, (eAsJava ^ ".inc_oct()")
                    else
                        raise (Failure ("cannot raise: " ^ vType))
                | Lower ->
                    if vType = "note" then
                        (initIdentifier "note"), env, (eAsJava ^ ".dec_oct()")
                    else if vType = "chord" then
                        (initIdentifier "chord"), env, (eAsJava ^ ".dec_oct()")
                    else
                        raise (Failure ("cannot raise: " ^ vType)))
        else raise (Failure ("type mismatch: " ^ vType ^ " is not suitable, must be a
note or chord"))
    | ListExpr(el) -> (* el is elment list *)
        (* (print_string ("Evaluating a listExpr\n")); *)
        let master, _, _ = (eval env (List.hd el)) in (* pull of the first element in
el and evalute *)
            let master_type = (getType master) in (* the type of the first element,
everything gets compared to this *)
            begin
                match master_type with (* what is the master type? *)
                    "note" -> (* if it is a note create a scale *)
                        let javaStrList = List.map (fun (note_elem) ->
                            (let chord_elem, env, asJava = eval env note_elem in
                                let vType = (getType chord_elem) in
                                    if ( vType = "note") then ("add(" ^ asJava ^ ");")
                                    else raise (Failure ("List expressions must contain
all of same type"))
                            )) el in
                        let notesAsJava = String.concat "\n" (List.rev javaStrList) in
                                (initIdentifier "scale"), env, ("new scale(new
ArrayList<note>() {{\n" ^ notesAsJava ^ "}})")
                    | "chord" -> (* if it is a chord create a stanza *)
                        let javaStrList = List.map (fun (note_elem) ->
                            (let chord_elem, env, asJava = eval env note_elem in
                                let vType = (getType chord_elem) in
                                    if ( vType = "chord") then ("add(" ^ asJava ^ ");")
                                    else raise (Failure ("List expressions must contain
all of same type"))
                            )) el in
                        let chordsAsJava = String.concat "\n" (List.rev javaStrList) in
                                (initIdentifier "stanza"), env, ("new stanza(new
ArrayList<chord>() {{\n" ^ chordsAsJava ^ "}})")
                    | "stanza" -> (* if it is a stanza create a score *)
                        let javaStrList = List.map (fun (note_elem) ->
                            (let chord_elem, env, asJava = eval env note_elem in
                                let vType = (getType chord_elem) in
                                    if ( vType = "stanza") then ("add(" ^ asJava ^
");")
                                    else raise (Failure ("List expressions must contain
all of same type"))
                            )) el in
                        let stanzasAsJava = String.concat "\n" (List.rev javaStrList)
in
                                (initIdentifier "score"), env, ("new score(new
ArrayList<stanza>() {{\n" ^ stanzasAsJava ^ "}})")
                    | _ -> raise (Failure ("List expression must only contain notes or
chords or stanzas"))
```

```
            end
    | Assign(toE, fromE) ->
        let lft_expr, env, lft_expr_jString = eval env toE in
            let rht_expr, (locals, globals, fdecls), rht_expr_jString = eval env fromE
in
                let lftInfo =
                    match toE with
                        Id(i) -> ("id", (i, ""))
                        | MemberAccess(i, j) -> ("member", (i, j))
                        | _ -> raise (Failure ("left side of assignment must be an
identifier or member access")) in
                let lftIdType = fst lftInfo in
                    let lftName = snd lftInfo in
                        let lftType = (* ("note", "locals") *)
                            (if NameMap.mem (fst lftName) locals then
                                (getType (NameMap.find (fst lftName) locals), "locals")
                            else if NameMap.mem (fst lftName) globals then
                                (getType (NameMap.find (fst lftName) globals),
"globals")
                            else raise (Failure ("undeclared identifier: " ^ fst
lftName))) in
                        let lftRetType = getType lft_expr in
                            let rhtType = getType rht_expr in
                            if lftRetType = rhtType then
                                match lftRetType with
                                    "int" ->
                                        if lftIdType = "id" then
                                            (if snd lftType = "locals" then
                                                (initIdentifier (getType rht_expr)),
(locals, globals, fdecls), ("\n\t" ^ lft_expr_jString ^ " = " ^ rht_expr_jString)
                                            else if snd lftType = "globals" then
                                                (initIdentifier (getType rht_expr)),
(locals, globals, fdecls), ("\n\t" ^ lft_expr_jString ^ " = " ^ rht_expr_jString)
                                            else raise (Failure ("fatal error")))
                                        (* MEMBER METHODS *)
                                        else if lftIdType = "member" then
                                            (* NOTE MEMBER METHODS *)
                                            if fst lftType = "note" then
                                                if snd lftName = "pitch" then
                                                        if snd lftType = "locals" then
                                                            (initIdentifier (getType
rht_expr)), (locals, globals, fdecls), ("\n\t" ^ lft_expr_jString ^ " = " ^
rht_expr_jString)
                                                        else if snd lftType = "globals"
then
                                                            (initIdentifier (getType
rht_expr)), (locals, globals, fdecls), ("\n\t" ^ lft_expr_jString ^ " = " ^
rht_expr_jString)
                                                        else raise (Failure ("fatal
error"))
                                                else if snd lftName = "duration" then
(* min max checking *)
                                                    if snd lftType = "locals" then
                                                        (initIdentifier (getType
rht_expr)), (locals, globals, fdecls), ("\n\t" ^ lft_expr_jString ^ " = " ^
rht_expr_jString)
                                                    else if snd lftType = "globals"
then
                                                        (initIdentifier (getType
rht_expr)), (locals, globals, fdecls), ("\n\t" ^ lft_expr_jString ^ " = " ^
rht_expr_jString)
```

```
                                                    else raise (Failure ("undeclared
identifier: " ^ fst lftName))
                                                else if snd lftName = "octave" then (*
min max checking *)
                                                        if snd lftType = "locals" then
                                                            (initIdentifier (getType
rht_expr)), (locals, globals, fdecls), ("\n\t" ^ lft_expr_jString ^ " = " ^
rht_expr_jString)
                                                        else if snd lftType = "globals"
then
                                                            (initIdentifier (getType
rht_expr)), (locals, globals, fdecls), ("\n\t" ^ lft_expr_jString ^ " = " ^
rht_expr_jString)
                                                        else raise (Failure
("undeclared identifier: " ^ fst lftName))
                                                else raise (Failure ("fatal error"))
                                            else if fst lftType = "chord" then
                                                if snd lftName = "chord_duration" then
                                                        if snd lftType = "locals" then
                                                            (initIdentifier (getType
rht_expr)), (locals, globals, fdecls), ("\n\t" ^ lft_expr_jString ^ " = " ^
rht_expr_jString)
                                                        else if snd lftType = "globals"
then
                                                            (initIdentifier (getType
rht_expr)), (locals, globals, fdecls), ("\n\t" ^ lft_expr_jString ^ " = " ^
rht_expr_jString)
                                                        else raise (Failure
("undeclared identifier: " ^ fst lftName))
                                                else raise (Failure ("fatal error"))
                                            else if fst lftType = "score" then
                                                if snd lftName = "instrument" then
                                                    if snd lftType = "locals" then
                                                        (initIdentifier (getType
rht_expr)), (locals, globals, fdecls), ("\n\t" ^ lft_expr_jString ^ " = " ^
rht_expr_jString)
                                                    else if snd lftType = "globals"
then
                                                        (initIdentifier (getType
rht_expr)), (locals, globals, fdecls), ("\n\t" ^ lft_expr_jString ^ " = " ^
rht_expr_jString)
                                                    else raise (Failure ("fatal
error"))
                                                else raise (Failure ("fatal error"))
                                            else raise (Failure ("cannot assign to: " ^
fst lftType))
                                        else raise (Failure ("cannot assign to: " ^
(fst lftType)))
                                    | "scale" ->
                                        if lftIdType = "id" then
                                            (if snd lftType = "locals" then
                                                (initIdentifier (getType rht_expr)),
(locals, globals, fdecls), ("\n\t" ^ lft_expr_jString ^ " = " ^ rht_expr_jString)
                                            else if snd lftType = "globals" then
                                                (initIdentifier (getType rht_expr)),
(locals, globals, fdecls), ("\n\t" ^ lft_expr_jString ^ " = " ^ rht_expr_jString)
                                            else raise (Failure ("fatal error")))
                                        (* MEMBER METHODS *)
                                        else if lftIdType = "member" then
                                            (* NOTE MEMBER METHODS *)
                                            if fst lftType = "scale" then
```

```
                                                let str_len = (length rht_expr_jString
- 10) in
                                                if snd lftName = "scale_notelist" then
                                                        if snd lftType = "locals" then
                                                            (initIdentifier (getType
rht_expr)), (((getChord (NameMap.find (fst lftName) locals)).notelist <- (getScale
rht_expr).scale_notelist); (locals, globals, fdecls)), ("\n\t" ^ lft_expr_jString ^ " =
" ^ (String.sub rht_expr_jString 9 str_len) )
                                                        else if snd lftType = "globals"
then
                                                            (initIdentifier (getType
rht_expr)), (((getChord (NameMap.find (fst lftName) globals)).notelist <- (getScale
rht_expr).scale_notelist); (locals, globals, fdecls)), ("\n\t" ^ lft_expr_jString ^ " =
" ^ (String.sub rht_expr_jString 9 str_len) )
                                                        else raise (Failure
("undeclared identifier: " ^ fst lftName))
                                                else raise (Failure ("fatal error"))
                                            else raise (Failure ("cannot assign to: " ^
fst lftType))
                                        else raise (Failure ("cannot assign to: " ^
(fst lftType)))
                                    | _ -> (* bool, note, chord, staff, part *)
                                        if lftIdType = "id" then
                                            (if snd lftType = "locals" then
                                                (initIdentifier (getType rht_expr)),
(NameMap.add (fst lftName) (initIdentifier (getType rht_expr)) locals, globals,
fdecls), ("\n\t" ^ lft_expr_jString ^ " = " ^ rht_expr_jString)
                                            else if snd lftType = "globals" then
                                                (initIdentifier (getType rht_expr)),
(locals, NameMap.add (fst lftName) (initIdentifier (getType rht_expr)) globals,
fdecls), ("\n\t" ^ lft_expr_jString ^ " = " ^ rht_expr_jString)
                                            else raise (Failure ("fatal error")))
                                        else raise (Failure ("cannot assign to: " ^
(fst lftType)))
                            else if lftIdType = "id" then
                                raise (Failure ("cannot assign: " ^ fst lftType ^ " = "
^ rhtType))
                            else if lftIdType = "member" then
                                raise (Failure ("cannot assign: " ^ lftRetType ^ " = "
^ rhtType))
                            else raise (Failure ("fatal error"))
    | NoExpr -> Bool true, env, ""
and exec env fname = function
        Expr(e) -> let _, env, asJava = (eval env e) in
            (* (print_string ("Exec an expr\n")); *)
            env, (asJava ^ ";\n")
        | Return(e) -> (* (print_string ("Return stmt in exec\n")); *)
            let v, (locals, globals, fdecls), asJava = (eval env e) in
                let fdecl = NameMap.find fname fdecls in
                    if (getType v) = (string_of_cbtype fdecl.rettype) then
                        (locals, globals, fdecls), ("return " ^ asJava ^ ";\n")
                    else raise (Failure ("function " ^ fdecl.fname ^ " returns: " ^
(getType v) ^ " instead of " ^ (string_of_cbtype fdecl.rettype)))
        | Block(s1) -> (* (print_string ("Block stmt in exec\n")); *)
            let (locals, globals, fdecls) = env in
                let (l, g), jStr = call s1 locals globals fdecls fname "" (* Check to
make sure we don't need a rev *)
                in (l, g, fdecls), jStr
        | If(e, ibl, s) -> (* (print_string ("If stmt with else in exec\n")); *)
            let (locals, globals, fdecls) = env in
                let v, env, evalJavaString = eval env e in
```

```
                    if (getType v) = "bool" then (env, ("if(" ^ evalJavaString ^ ")
{\n" ^ (snd (call (List.rev ibl) locals globals fdecls fname "")) ^ "}\n else {\n" ^
(snd ((exec env fname) s)) ^ "}\n"))
                    else raise (Failure ("If statement must be given boolean
expression"))
        | If(e, s, Block([])) -> (* (print_string ("If stmt without else in exec\n"));
*)
            let (locals, globals, fdecls) = env in
                let v, env, evalJavaString = eval env e in
                    if (getType v) = "bool" then (env, ("if(" ^ evalJavaString ^ ")
{\n" ^ (snd (call (List.rev s) locals globals fdecls fname "")) ^ "}\n"))
                    else raise (Failure ("If statement must be given boolean
expression"))
       | Foreach(par_decl, list_name, sl) ->
            let locals, globals, fdecls = env in (* env *)
                let list1 = (*check for var existence in locals *)
                    if NameMap.mem list_name locals then (*check if list_name is in
locals *)
                        NameMap.find list_name locals (*let list equal to the variable
found in locals map*)
                    else
                        if NameMap.mem list_name globals then (*let list equal to the
variable found in globals map*)
                            NameMap.find list_name globals
                        else
                            raise (Failure ("list variable undeclared"))
                in let vType = getType list1 in
                begin
                    match vType with
                        "chord" ->
                            (*notes*)
                            if (string_of_cbtype par_decl.paramtype) = "note" then
                                let (l, g), jStr = (call (List.rev sl) (NameMap.add
par_decl.paramname (initIdentifier "note") locals) globals fdecls fname "") in
                                    (env, "for (note " ^ par_decl.paramname ^ " : " ^
list_name ^ ".notelist " ^ ") { " ^ jStr ^ " } ")
                            else
                                raise (Failure ("failure of type matching with chord
list"))
                        | "scale" ->
                            (*notes*)
                            if (string_of_cbtype par_decl.paramtype) = "note" then
                                let (l, g), jStr = (call (List.rev sl) (NameMap.add
par_decl.paramname (initIdentifier "note") locals) globals fdecls fname "") in
                                    (env, "for (note " ^ par_decl.paramname ^ " : " ^
list_name ^ ".scale_notelist " ^ ") { " ^ jStr ^ " } ")
                            else
                                raise (Failure ("failure of type matching with scale
list"))

                       | "stanza" ->
                            (*chords*)
                            if (string_of_cbtype par_decl.paramtype) = "chord" then
                                let (l, g), jStr = (call (List.rev sl) (NameMap.add
par_decl.paramname (initIdentifier "chord") locals) globals fdecls fname "") in
                                    (env, "for (chord " ^ par_decl.paramname ^ " : " ^
list_name ^ ".chordlist " ^ ") { " ^ jStr ^ " } ")
                            else
                                raise (Failure ("failure of type matching with stanza
list"))
                        | "score" ->
```

```
                            if (string_of_cbtype par_decl.paramtype) = "stanza" then
                                let (l, g), jStr = (call (List.rev sl) (NameMap.add
par_decl.paramname (initIdentifier "stanza") locals) globals fdecls fname "") in
                                    (env, "for (stanza " ^ par_decl.paramname ^ " : " ^
list_name ^ ".stanzalist " ^ ") { " ^ jStr ^ " } ")
                            else
                                raise (Failure ("failure of type matching with score
list"))
                        | _ ->
                            raise (Failure ("undesired list type for for_each loop"))
                end
        | While(e, sl) ->
            let boolarg, env, javaString = eval env e in
                if (getType boolarg) = "bool" then (
                    let locals, globals, fdecls = env in
                        let (locals, globals), jStr = call (List.rev sl) locals globals
fdecls fname "" in
                            (locals, globals, fdecls), ("while(" ^ javaString ^ ") {\n"
^ jStr ^ "}\n")
                )
                else raise (Failure ("while loop argument must decompose to a boolean
value"))
        | _ -> raise (Failure ("Unable to match the statment"))
(* Execute the body of a method and return an updated global map *)
and call fdecl_body locals globals fdecls fdecl_name jStr =
       (*  (print_string ("Call jStr = " ^ jStr ^ "\n")); *)
        match fdecl_body with
            [] -> (* (print_string ("Done with a full call\n")); *)
                (locals, globals), jStr (*When we are done return the updated globals*)
            | head::tail ->
                match head with
                    VDecl2(head) -> (* (print_string ("Working on a vdecl in call\n"));
*)
                        if(fdecl_name = "") then
                            ((if NameMap.mem head.varname globals then raise (Failure
("Variable " ^ head.varname ^ " declared twice")));
                                let rtype = if (string_of_cbtype head.vartype) = "bool"
then "boolean" else (string_of_cbtype head.vartype) in
                                    (call tail locals (NameMap.add head.varname
(initIdentifier (string_of_cbtype head.vartype)) globals) fdecls fdecl_name (jStr ^
("\n" ^ rtype ^ " " ^ head.varname ^ (NameMap.find (string_of_cbtype head.vartype)
defaultInitMap)))))
                        else
                            ((if NameMap.mem head.varname locals then raise (Failure
("Variable " ^ head.varname ^ " declared twice")));
                                let rtype = if (string_of_cbtype head.vartype) = "bool"
then "boolean" else (string_of_cbtype head.vartype) in
                                    (call tail (NameMap.add head.varname
(initIdentifier (string_of_cbtype head.vartype)) locals) globals fdecls fdecl_name
                                    (jStr ^ ("\n" ^ rtype ^ " "  ^ head.varname ^
(NameMap.find (string_of_cbtype head.vartype) defaultInitMap)))))
                    | FullDecl2(head) -> (* (print_string ("Working on a full decl in
call\n")); *)
                        (if(fdecl_name = "") then (if NameMap.mem head.fvname globals
then raise (Failure ("Variable " ^ head.fvname ^ " declared twice"))));
                        (if(fdecl_name <> "") then (if NameMap.mem head.fvname locals
then raise (Failure ("Variable " ^ head.fvname ^ " declared twice"))));
                        let v, env, rhsJavaString = eval (locals, globals, fdecls)
head.fvexpr in
                            let vType = getType v in
                                if vType = (string_of_cbtype head.fvtype)
```

```
                                    then
                                        match vType with
                                            "int" -> (* print_string ("\n\n<" ^
head.fvname ^ ">"); *) call tail (NameMap.add head.fvname (initIdentifier "int")
locals) globals fdecls fdecl_name (jStr ^ ("int " ^ head.fvname ^ " = " ^ rhsJavaString
^ ";\n"))
                                          | "note" -> call tail (NameMap.add
head.fvname (initIdentifier "note") locals) globals fdecls fdecl_name (jStr ^ ("note "
^ head.fvname ^ " = " ^ rhsJavaString ^ ";\n"))
                                          | "chord" -> call tail (NameMap.add
head.fvname (initIdentifier "chord") locals) globals fdecls fdecl_name (jStr ^ ("chord
" ^ head.fvname ^ " = " ^ rhsJavaString ^ ";\n"))
                                          | "bool" -> call tail (NameMap.add
head.fvname (initIdentifier "bool") locals) globals fdecls fdecl_name (jStr ^ ("boolean
" ^ head.fvname ^ " = " ^ rhsJavaString ^ ";\n"))
                                          | "scale" -> call tail (NameMap.add
head.fvname (initIdentifier "scale") locals) globals fdecls fdecl_name (jStr ^ ("scale
" ^ head.fvname ^ " = " ^ rhsJavaString ^ ";\n"))
                                          | "stanza" -> call tail (NameMap.add
head.fvname (initIdentifier "stanza") locals) globals fdecls fdecl_name (jStr ^
("stanza " ^ head.fvname ^ " = " ^ rhsJavaString ^ ";\n"))
                                          | "score" -> call tail (NameMap.add
head.fvname (initIdentifier "score") locals) globals fdecls fdecl_name (jStr ^ ("score
" ^ head.fvname ^ " = " ^ rhsJavaString ^ ";\n"))
                                          | _ -> raise (Failure ("Unknown type: " ^
vType))
                                else
                                    raise (Failure ("LHS = " ^ (string_of_cbtype
head.fvtype) ^ " <> RHS = " ^ vType))
                    | Stmt2(head) -> (* (print_string ("Working on a stmt in call\n"));
*)
                        let (locals, globals, fdecls), execJavaString = (exec (locals,
globals, fdecls) fdecl_name head) in
                            call tail locals globals fdecls fdecl_name (jStr ^
execJavaString)
and translate prog env =
    let locals, globals, fdecls = env in
        match prog with
            [] -> (* everything went well, write the java file and quit *)
                (* (print_string ("Finished the program\n")); *)
                let javaOut = open_out ("Cb.java") in
                Printf.fprintf javaOut "%s" (import_decl ^ class_start ^
globalJava.contents ^ methJava.contents ^ main_start ^ run_start ^ mainJava.contents ^
run_end ^ class_end);
                (close_out javaOut);
                Bool true, (locals, globals, fdecls)
            | head::tail ->
                match head with
                    VDecl(head) -> (* (print_string ("Translate sees a vdecl\n")); *)
                        (if NameMap.mem head.varname globals then raise (Failure
("Variable " ^ head.varname ^ " defined more than once"))
                        else
                            ((if (string_of_cbtype head.vartype) = "bool"
                                then ((globalJava := globalJava.contents ^ "\nboolean "
^ head.varname ^ ";\n");
                                    (mainJava := mainJava.contents ^ "\n" ^
head.varname ^ (NameMap.find (string_of_cbtype head.vartype) defaultInitMap)))
                            else ((globalJava := globalJava.contents ^ "\n" ^
(string_of_cbtype head.vartype) ^ " " ^ head.varname ^ ";\n");
                                (mainJava := mainJava.contents ^ "\n" ^ head.varname ^
(NameMap.find (string_of_cbtype head.vartype) defaultInitMap))));
```

```
                            translate tail (locals, (NameMap.add head.varname
(initIdentifier (string_of_cbtype head.vartype)) globals), fdecls)))
                    | FullDecl(head) -> (* (print_string ("Translate sees a
fulldecl\n")); *)
                        (if NameMap.mem head.fvname globals then raise (Failure
("Variable " ^ head.fvname ^ " defined more than once")));
                        let v, env, asJava = eval (locals, globals, fdecls) head.fvexpr
in
                            let vType = getType v in
                                if vType = (string_of_cbtype head.fvtype)
                                    then
                                        ((if (string_of_cbtype head.fvtype) = "bool"
                                            then ((globalJava := globalJava.contents ^
"\nboolean " ^ head.fvname ^ ";\n");
                                                    (mainJava := mainJava.contents ^
head.fvname ^ " = " ^ asJava ^ ";\n"))
                                            else ((globalJava := globalJava.contents ^ "\n"
^ (string_of_cbtype head.fvtype) ^ " " ^ head.fvname ^ ";\n");
                                                    (mainJava := mainJava.contents ^ "\n" ^
head.fvname ^ " = " ^ asJava ^ ";\n")));
                                            match vType with
                                                "int" -> translate tail (locals,
(NameMap.add head.fvname (initIdentifier "int") globals), fdecls)
                                                | "note" -> translate tail (locals,
(NameMap.add head.fvname (initIdentifier "note") globals), fdecls)
                                                | "chord" -> translate tail (locals,
(NameMap.add head.fvname (initIdentifier "chord") globals), fdecls)
                                                | "bool" -> translate tail (locals,
(NameMap.add head.fvname (initIdentifier "bool") globals), fdecls)
                                                | "scale" -> translate tail (locals,
(NameMap.add head.fvname (initIdentifier "scale") globals), fdecls)
                                                | "stanza" -> translate tail (locals,
(NameMap.add head.fvname (initIdentifier "stanza") globals), fdecls)
                                                | "score" -> translate tail (locals,
(NameMap.add head.fvname (initIdentifier "score") globals), fdecls)
                                                | _ -> raise (Failure ("Unknown type: " ^
vType)))
                                    else
                                        (raise (Failure ("LHS = " ^ (string_of_cbtype
head.fvtype) ^ "<> RHS = " ^ vType)))
                    | MDecl(head) -> (* (print_string ("Translate sees a methdecl\n"));
*)
                        (if (NameMap.mem head.fname fdecls) then raise (Failure
("Method with same name already defined"))
                        else
                            let newlocals = List.fold_left(fun acc arg -> (NameMap.add
arg.paramname (initIdentifier (string_of_cbtype arg.paramtype)) acc)) locals
head.formals in
                            let (_, _), javaBody = call head.body newlocals globals
(NameMap.add head.fname head fdecls) head.fname "" in
                                let rtype = if (string_of_cbtype head.rettype) = "bool"
then "boolean" else (string_of_cbtype head.rettype) in
                                    (methJava := methJava.contents ^ "\npublic " ^
rtype ^ " " ^ head.fname ^ "(" ^
                                    (String.concat "," (List.map(fun arg -> (if
(string_of_cbtype arg.paramtype) = "bool" then "boolean" else (string_of_cbtype
arg.paramtype)) ^ " " ^ arg.paramname)head.formals)) ^
                                    ") {" ^ javaBody ^ "\n}\n");
                            translate tail (locals, globals, (NameMap.add head.fname
head fdecls))
                        )
```

```
                    | Stmt(head) -> (* (print_string ("Translate sees a stmt\n")); *)
                        let env, jString = exec (locals, globals, fdecls) "" head in
                            (mainJava := mainJava.contents ^ jString);
                            translate tail env

let helper prog = translate prog (NameMap.empty, NameMap.empty, NameMap.empty)
```