corgi

Final Report
COMS W4115
Columbia University
Professor Stephen Edwards

Team Members:
Philippe-Guillaume Losembe (pvl2109)
Alisha Sindhwani (as4312)
Melissa O'Sullivan (mko2110)
Justin Zhao (jxz2101)

December 17, 2014

Table of Contents

Introduction
1. Project Overview
2. Language Goals

Language Tutorial
1. Running the Compiler
2. Hello, world

Language Reference Manual
1. Introduction
2. Types and Type Declaration
3. Lexical Conventions
4. Syntax
5. Scope
6. Standard Library
7. Final Demos

Project Plan
1. Project Plan
2. Team Responsibilities
3. Project Timeline
4. Development Environment

Architectural Design
1. Overview
2. Scanner
3. Parser
4. Abstract Syntax Tree
5. Symbol Table
6. Semantic Check
7. Java Code Generation
8. Libraries
9. Command Line Interface

Testing Plan

Lessons Learned: Advice for Future Groups

Appendix
1. Presentation Slides

2. Complete Code Reference
    a. Root Directory
        i. ast.ml
        ii. check.ml
        iii. javagen.ml
        iv. interpreter.ml
        v. Makefile
        vi. parser.mly
        vii. README.md
        viii. scanner.ml
        ix. table.ml
        x. populatetests.sh
        xi. makejava.sh
        xii. checkjavac.sh
        xiii. runtests.sh
    b. examples
        i. hello_word.corgi
        ii. fib_music.corgi
        iii. search_music.corgi
    c. Tests: tests can be found in github and project files

Acknowledgements

Introduction

Project Overview

corgi is a language centered on music translation, generation, and analysis. It is musical "alg'rhythms" language focused on patterns in music, from a top-down and bottom-up approach. From the top down, we wanted to be able to analyze and find patterns in pre-existing music and from the bottom-up we wanted to have the ability to programmatically generate music from patterns.

corgi reads in a midi file which is a standardized digital file format for interpreting music and translate the files into the appropriate data structures. A user will be able to manipulate and search through these data structures—they allow our language to quantitatively analyze and find patterns in music that would be difficult to do manually. Our data structures make it easy to identify and return the location of specific instances in a given composition. An example program could be finding the longest subsequence of rising notes. This search functionality will allow users to compare multiple pieces of music. Similarly, a user can be able to generate music directly through the implementation of our musical data structures. Our built in functions allow the user to import and export to a midi file as well as play their music.

Language Goals

We wanted our language to make the easiest transition possible from composing music traditionally on paper to creating it programmatically. We wanted the user to easily see how the different parts interconnect instead of hiding it behind abstraction and be flexible in combining data structures. We use simple python-link syntax for objects and arrays as well as easy to read binary operators so that it is very clear to read how the music is being constructed. We looked to see failures and holes in existing music creation programs.

Language Tutorial

Running the Compiler

To run the compiler and execute a corgi file, follow these steps:

```
1. make all
2. ./makejava.sh
3. ./interpreter -javagen < CORGIFILE 2>
   javaclasses/Intermediate.java
4. cd javaclasses
5. javac Intermediate.java
6. java Intermediate
```

Command 1 makes the program. Command 2 compiles the Java libraries. Command 3 executes the interpreter on the input corgi file and outputs the generated java in an Intermediate.java file. Command 4 and 5 are for proper program compilation of the program in the correct directory of java files. There are several steps, and to help with this abstraction, we have provided a single shell script that compiles and executes the corgi program where the output of the program is stout.

```
./corgify.sh CORGIFILE
```

Hello, World

The classic, hello, world!

```
int main() {
    print("Hello, world!");
}
```

And to “corgify”:

```
./corgify.sh examples/hello_world.corgi
```

Language Reference Manual

Types and Type Declaration

Type Declaration
Data is expressed in explicitly declared types similar to Java.

Types

- Integer
  Much like Java, an integer in corgi is a primitive type denoted by the keyword int and representing values ranging from 0 to 2^32-1. An int is declared by:

```
int i;
i = 7;
```

- String
  A String is an array of chars. For example a String can be declared by:

```
string str;
str = "music";
```

- Fractions
  A fraction is reduced division of two integers. It's type declaration is denoted by the keyword frac and each value definition begins with the character '$', followed by the numerator of the fraction, separated by the denominator of the fraction by a '/' and ending with the character '$'. For example a fraction can be declared as:

```
frac f;
f = $¾$;
```

- Duration
  Duration is fraction that meets the constraint that the numerator is less than the denominator. It is used to represent the length of a chord and can be either declared directly or (implicitly) cast from a fraction as shown:

```
duration d;
frac f;
duration fd;
d = $¼$;
f = $¾$;
fd = f;
```

- Pitch

Pitch is defined by an integer. Pitches are in the range 0 to 150. It is declared using the keyword pitch, for example:

```
pitch p;
p = 4;
```

- Rhythm
  Rhythm refers to a sequential list of durations. It is declared using the keyword rhythm as shown:

```
duration d;
rhythm r;
d = <¼>;
r = [d,d,d];
```

- Chord
  A chord is a sequential list of (pitch, duration) tuples. A chord can be declared using the keyword chord as follows:

```
pitch p1;
pitch p2;
duration d1;
duration d2;
chord c;
p1 = 4;
p2 = 5;
d1 = <¼>;
d2 = <⅛>;
c = [(p1,d1), (p1,d2), (p2, d2)];
```

- Track
  A track is a sequential list of chords which can be declared using the keyword track. For example:

```
pitch p1;
pitch p2;
duration d1;
duration d2;
chord c1;
chord c2;
track t1;
p1 = 4;
p2 = 5;
d1 = <¼>;
d2 = <⅛>;
c1 = [(p1,d1), (p1,d2), (p2, d2)];
```

```
c2 = [(p1,d1), (p1,d2), (p2, d2), (p2, d1)];
t = [c1, c2, c2];
```

- Composition
  A composition is a sequential collection of tracks. A composition can be declared using the keyword composition as follows:

```
pitch p1;
pitch p2;
duration d1;
duration d2;
chord c1;
chord c2;
track t1;
track t2;
composition x;
p1 = 4;
p2 = 5;
d1 = <¼>;
d2 = <⅛>;
c1 = [(p1,d1), (p1,d2), (p2, d2)];
c2 = [(p1,d1), (p1,d2), (p2, d2), (p2, d1)];
t1 = [c1, c2, c2];
t2 = [c1, c1];
x = [t1, t2, t1];
```

- Arrays
  In addition to the array-based structures (Rhythm, Chord, Track, Composition), we have arrays just as you would have them in java with the same syntax.

```
int[] x;
x = [1, 2, 3];
```

Lexical Conventions

In corgi, a token is a string of one or more characters consisting of letters, digits, or underscores. corgi has 5 kinds of tokens:

Identifiers
Keywords
Constants
Operators
Newlines

Identifiers

The first character must be a letter and identifiers are case sensitive. The letters are the ASCII characters a-z and A-Z. Digits are the ASCII characters 0-9.

letter → [‘a’-’z’ ‘A’-’Z’]
digit → [‘0’-’9’]
underscore → ‘_’
identifier → *letter* (*letter* | *digit* | *underscore*) *

Keywords

The following identifiers are strictly reserved for use as keywords:

Keywords	**Description**
int	standard 32-bit integer
frac	two integers that represent a fraction
duration	wrapper around fraction
pitch	wrapper around integer, this can also be instantiated as 'C+4'
rhythm	a collection of durations
chord	a collection of pitch duration tuples
track	a sequential list of chords
composition	a collection of tracks
True / False	Boolean constants
if / else	Conditional expressions
random	generate random numbers
print(x,y,z…)	calls System.out.println and concatenates the arguments
main	Declaration of the main program
return	specifies a return statement.

Literals

Defining a string literal is simply done with a sequence of one or more characters enclosed by double quotes.

Type	**Syntax**	**Example**
String	“[str]”	String s; s =“string”
Int	[integer]	int i; t = 5;
Frac	$num/denom$ where num and denom are both integers	Frac f; f = $¾$

Special Escape Character

The only special escape characters are:

Escaped	**Description**
\”	quote
\n	new line
\t	tab

Punctuation

Punctuation	**Use**	**Example**
,	list element separator, function parameters	array = [1, 2, 3]
@	list get and set	c1 @ 1 -> get the value of c1 = @ 1 (p2, d2); -> set the index 1 of c1 to (p2, d2)
()	conditional parameter delimiter, function parameter delimiter	if (array[0] == 3)

{}	statement list delimiter	if (array[0] == 3) { /* work */}
“	string literal delimiter	s = “what\’s up?”
;	end of statement	array = [1, 2, 3];

Comments

Corgi supports java style // comments.

Comment Symbols	Description	Example
//	Single-line comment	// This is a comment

Operators

An operator is a token that specifies an operation on at least one operand and yields some result.

	int	frac	duration	pitch	rhythm	chord	track	composition
"="	assignment	assignment	assignment	assignment	assignment	assignment	assignment	assignment
"+"	addition	addition	addition	adds the pitch values	appends to end	appends to end	appends to end	appends to end
"-"	subtraction	subtraction	subtraction	subtracts the pitch values	removes instances of	removes instances of	removes instances of	removes instances of
"*"	multiplication	multiplication	multiplication	multiplies two pitch values				
"/"	division	division	division					
@					accessor	accessor	accessor	accessor
>	compare value	compare value	compare duration		compare duration			
<	compare value	compare value	compare duration		compare duration			
"=="	check equality	check equality	check equality	check equality	compare duration			
"!="	check equality	check equality	check equality	check equality				
"."						invoke method	invoke method	invoke method

Syntax

Program Structure
A program in corgi is made up of one or more valid statements. A Program begins in a main function which needs to be defined for any statements to be executed.

Expressions
In corgi, an expression is made up of variables, operators, and method calls. An expression must evaluate to a value of one of corgi's data types. An expression is evaluated from left to right as shown:

```
10 - 2 - 3 - 4 //evaluates to 1
```

Variables
A variable refers to a data type. They type and value of a variable is declared and initialized with the type keyword, variable name, and value in a single line as follows:

```
int a = 4;
```

For type specific examples refer to Chapter 2.

Binary Operators
Binary operators can connect variables to create composite expressions. These operators are of the form.

```
x operator x //with x representing an expression
```

Types of Binary Operators include:

- **Arithmetic operators** such as addition (+), subtraction (-), multiplication(*), division (/), and modulus (%). The expressions acting as operands for an arithmetic operator must be both the same type and that type must be int, frac, or duration. The resulting value of the expression composed of two expressions of the same type is a value of that type.

- **Relational operators** such as less than (<), greater than (>), equal (==), or not equal (!=) require operands to be of the same type and of types including int, frac, duration, pitch, or rhythm. The result of a relational operator invoked on two operands of the same type is an integer equal to 0, if the expression evaluates to false or 1 otherwise.

The Role of Parentheses
Parentheses may guide the order of operations on expressions as the expression inside a set of parentheses must be evaluated before that expression can be evaluated with respect to other operators. The surrounding of a set of parentheses around an expression does not change the subexpressions value.

Statements
A statement is an instruction to be executed. An expression on its own is not a valid statement, with the exception of a function call. It is either a single instruction that ends in a ‘;’ or begins a list of statements contained between curly braces ({ }). There are four types of statements in corgi:

- **Assignment**
  An expression’s value can be assignment to a variable with this statement.
  ```
  int a = 4;
  int b = a + 1;
  ```

- **Function Creation**
  Functions can be created much in the style of C functions. The method header includes the return type, function name, and parameters. The return type can be omitted in the case of a function that does not return a value, but the function must return the type declared in the header. This is a function with no parameters which returns a chord:
  ```
  chord function1() {
      chord c = [(1, <1/2>)];
      return c;
  }
  ```
  a function with no return value and two parameters:
  ```
  function2(chord c, int i) {
      …
  }
  ```
- **Return Statement**
  Return statements are specified with the keyword `return`

- **Function Calls**
  A function call consists of the function's name followed by its parameters in parentheses and surrounded by commas. The parameters and the function call itself are expressions whose type are determined from a previous function definition. The function call's value is the function's return value. Functions can be called with no parameters but the parentheses cannot be omitted.
  ```
  chord c = function1();
  function2(c, 2);
  ```
  A function call can be used as a stand alone statement but its return value will be lost if it is not assigned to a variable.

- **Control Statements**
  - **for** loop

A for statement takes two assignment statements and a Boolean expression and executes its statement list until its condition evaluates to False, the first assignment is executed when the for statement is encountered and the second one after each iteration of the loop:

```
for ( assignment1; condition; assignment2 ) {
      …
}

ex:

int i;
for(i = 0; i<10; i = i + 1){
      print(i);
}
```

- **while** loop
  A while statement takes a Boolean expression and executes its statement list until the expression evaluates to False:

```
while ( condition ) {
      …
}

ex:

      int i;
      i = 0;
      while(i<10){
            print(i);
            i = i+1;
      }
```

- **if else**
  An if else statement takes a Boolean expression and executes one statement list if its value is True and the other statement list otherwise:

```
      if (condition) { // condition is not 0
            …
      } else { // condition is 0
```

- **Combining data structures**
  We felt that it was very important to be able to easily combine data structures because when you're writing music by hand, you can very easily combine notes to create a chord and the chords to create a track and we felt that current music programs really

lacked in their inflexible data structures that were difficult to use. corgi's python-like syntax lets you very easily construct higher level data structures from lower level ones, lets you easily add arrays together, and intermix literals with variables. We also ensure that equivalent data types can be used interchangeably, so wherever you use a Duration, you can also use a Fraction or an int because all of those can represent a duration.

Ex: In this example we show how you can construct a chord from predefined variables as well as through the use of literals

```
pitch p1;
pitch p2;
duration d1;
duration d2;
chord c1;
chord c2;

p1 = 4;
p2 = 5;
d1 = $1/4$;
d2 = $1/8$;
//using literals and variables to construct tuples for a chord
c1 = [(p1,d1), (6,d2), (p2, $1/8$)];
print(c1);
c2 = [(p1,d1), (p1,d2), (p2, d2), (p2, d1)];
```

Scope

Global Variables

Everything has access to global variables. In the case that a local variable is defined in a block with the same name as a global variable, the local variable will be used. We do semantic checking to ensure that you can't declare variables with the same name inside a scope.

Block scoping

A block is a list of statements enclosed between two braces. Blocks can be nested and have their own local variables. A variable is only accessible in the block in which it was defined and blocks inside this one.

```
int x = 5;
{
      int y = x + 1;
      x = y + 1;
}
```

```
if (x > 5) { // This is true
      y = 0; // This is not allowed, y has no type or value
}
```

Function scoping
Functions only have access to variables in their parameter list and local variables declared inside the function.

Standard Library

import()

Usage:
```
composition c = import("filepath/test.mid");
```

By using the import function, one can read in a midi file from the file system into a composition variable.

export()

Usage:
```
export(c, "filepath/masterpiece.mi");
```

By using the export function, one can export a composition "c" of theirs to a music xml file for further processing and alteration.

print()

Usage:
```
      Frac f;
      f = $5/3$;
print("My fraction is: ", f);
```

The print function will take a variable number of arguments that will be concatenated and printed to stdout. For datatypes like Frac, print will call the toString method of the datatype.

length()

When given a function argument of an array or data structure that employs an array (Rhythm, Chord, Composition, Track), length will return the length of the array or the base array of the data structure.

Demo Program #1

This demo finds the number of interesting chords in a composition found in examples/search_music.corgi

```
      int main() {

          // Declaring variables
          composition compositionAnalysis;
          int index;
          int index2;
          chord interestingChord;
          chord tempChord;
          track interestingTrack;
          pitch c5;
          pitch g5;
          track trackHelper;
          duration quarterNote;
          int count;

          // Set constants
          c5 = 60;
          g5 = 67;
          quarterNote = $1/4$;
          interestingChord = [(c5, quarterNote)];
          tempChord = interestingChord;
          // interestingTrack = [interestingChord];

          // Import composition analysis
          compositionAnalysis = import("result.mid");

          count = 0;
          // Iterate through the composition and check
          for (index = 0; index < length(compositionAnalysis);
index=index+1) {
              trackHelper = compositionAnalysis @ index;

              for (index2 = 0; index2 < length(trackHelper);
index2=index2+1) {
                  tempChord = trackHelper @ index2;
                  if (interestingChord == tempChord) {
                      count = count + 1;
                  }
```

```
            }
        }

        print("There are ", count, " interesting chords in this
composition!");
    }
```

Demo Program #2

This demo creates a composition from a fibonacci sequence of notes and plays it found in examples/fib_music.corgi

```
    /*
     *  Function that returns the n'th fibonacci number
     */
    int fib(int n) {
        int sum;
        int i;
        if (n == 1) {
            return 1;
        }
        if (n == 2) {
            return 1;
        }
        sum = 1;

        for (i=2; i<n; i=i+1) {
            sum = sum + i;
        }
        return sum;
    }

    /*
     *  Function that uses the fibonacci number sequence to
generate melodies
     */
    int main() {

        // Variable declarations
        int i;
```

```
    chord tempChord;

    int fibNum;
    pitch p;
    duration d;

    track cumulativeTrack;
    track helperTrack;

    composition finalComposition;

  // Use a constant quarter note as the duration
    d = $1/4$;
    // Use a starting pitch of 60
    p = 60;

    tempChord = [(p,d)];
    cumulativeTrack = [tempChord];

    for (i=1; i<30; i=i+1) {
        fibNum = fib(i);

        // Keep it between 60 and 70
        fibNum = fibNum % 10 + 60;
        p = fibNum;
        tempChord = [(p, d)];

        helperTrack = [tempChord];
        print(helperTrack);

        // Add the helper track to the cumulative
        cumulativeTrack + helperTrack;
    }

    // initialize the final composition
    finalComposition = [cumulativeTrack];

    play(finalComposition);
    export(finalComposition, "fib_sequence.mid");

}
```

Project Plan

Meetings

To maintain continual progress over the semester, we met consistently each Friday to where we discussed the design of our language, reviewed our progress, and assigned individual work to do during the week. We planned our meetings with our TA at the start of our Friday meetings when necessary. Towards the end of the semester, as the project grew larger, we met for full days to program together.

Team Responsibilities

We found the best way to make progress was to give everyone individual, substantial features to work on and then pair program when it was time to unify the parts together and to debug. Once we identified individual strengths and passions, we started coding much faster and more cohesively. We all functioned as the Language Guru in some sense as we all decided together what we felt our language should look like and do.

Philippe-Guillaume Losembe:

Role: System Architect
Contributions: Scanner, Parser, AST, Semantic checking

Alisha Sindhwani:

Role: Project Manager
Contributions: Scanner, Parser, AST, Java generation, Java object classes

Melissa O'Sullivan:

Role: System Architect
Contributions: Symbol Table, Semantic checking, testing scripts

Justin Zhao:

Role: Testing lead
Contributions: Tests, testing scripts, Java object classes, Java built in functions

Project Timeline

https://github.com/melissaosullivan/corgi/commits/master

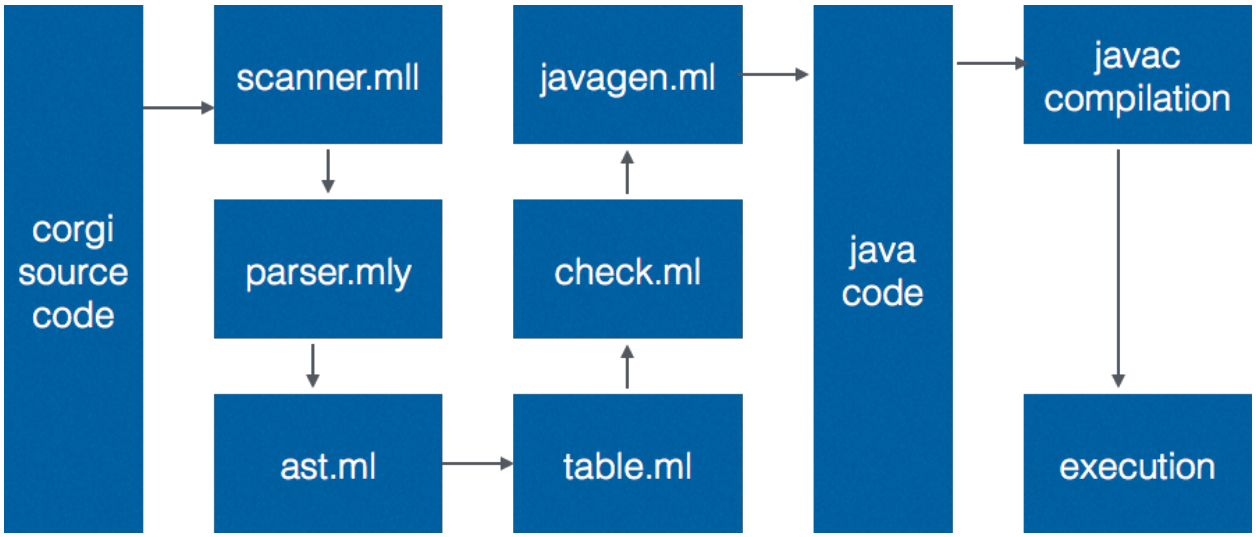
Development Environment

The corgi team developed on Mac OS X and Ubuntu machines. We used OCaml version x OCaml version 4.01.0 and Sublime Text 2 as an IDE. We had bash scripts to (corgify, populatetests, runtests, checkjavac) and Makefiles to make building and running our code cleaner and more streamlined. We used git hosted on github for version control.

Architectural Design

Overview

The corgi compiler takes as input a single .corgi file and outputs Java source code. Java is then compiled using javac and executed. The compiler breaks down into the following stages: scanning, parsing, populating the symbol table, semantic analysis by building a semantic abstract syntax tree, and finally Java code generation. Code generation always includes a library of jFugue and abstracted Java classes to support music library functions like importing from a midi, exporting to a midi, and playing music.



Scanner

The scanner scans through the .corgi source file and converts the file into a stream of tokens using ocamllex.

Parser

The parser analyzes the stream of tokens read in by the scanner and decides whether or not they are in the language that is specified by our abstract syntax tree (CFG). With ideas borrowed from Lorax, a scope number is bundled with blocks, types are deduced and constructed and the abstract syntax tree for the program is built.

Abstract Syntax Tree

The AST defines the CFG rules and structure for corgi. This includes all of corgi's primitive types and code flow like variables, blocks, functions, and main.

Symbol Table

Generating the symbol table is the very next step after generating the abstract syntax tree. Using the block ids set by the parser, we translate these block ids into scope number. The symbol table is ultimately a string map that keeps track of declared variables and functions. The symbol table is also used to enforce unique function and variables names within each scope and to verify that each variable and function is visible within the current scope.

Semantic Checking

The semantic checker accomplishes two primary tasks. One, it constructs a semantic abstract symbol tree which are essentially types from the regular abstract syntax tree with additional information of the type attached. Two, through the construction of the semantic abstract syntax tree, the additional typing information allows us to check for type compatibility. For our primitives, this required extensive checking because flexible data operations was a strong point for corgi. Chords, for example, can be constructed from pitches or ints and durations or fractions. Additionally, we check to make sure that the calls to functions and return types of the functions match the declarations of the functions that we parsed.

Java Code Generation

Java code generation takes the semantically checked code and converts it into Java, that utilizes our our base Java object classes—Pitch, Frac, Duration, Pitch_Duration_Tuple, Rhythm, Chord, Track, and Composition as well as our built in functions that we wrote in Java. It deals with flexible data structures in two ways: it either uses method overloading in Java so that our classes can handle multiple data types for all of their methods or the semantic checker tags binary operations and assignments that use different data types with the datatype of the higher precedence (so an operation between a duration and either an int or a fraction will tag the int and the fraction as a duration) and then we call a constructor on the lower precedence data type to convert it to the higher precedence data type.

Ex: d1 - $1/5$ will generate to (d1).subtract(new Duration(new Frac(1,5))) where d1 is declared as a Duration.

Libraries

The primary music library that was used to interface with midi file parsing, playing, importing, and exporting was jFugue. As jFugue had little support for reading in MIDI files into our particular way of organizing musical structures, we wrote a supplementary library to do this. To support our particular data types for music structures, we created Java classes to represent these types. While this may seem like simply a direct translation to object oriented design, extra consideration was taken to support flexible data type construction. This involved, for example, writing several overloaded versions of method for many data types to be able to perform operations with other data types.

Command Line Interface

The command line interface allows for the user to inspect output at each stage of the compilations process by specifying a variety of flags. This proved to be useful for testing and debugging. Lastly, this design was also motivated by trying to create an easy flow for a user to compile and run their program.

Testing Plan

Frequent and thorough testing was an important aspect of debugging the compiler. Using a set of successful and failure base test cases, we generated standardized output at every stage of compilation to make sure testing was thorough. This relied on two primary shell scripts: populatetests.sh and runtests.sh. populatetests.sh runs through each test corgi file and generates “golden” outputs for each stage of compilation. runtests.sh is a script that would run each test corgi file through each stage of compilation like populatetests.sh, but would compare the results with each test corgi file’s respective golden output for each stage of compilation. Golden outputs for the abstract syntax tree, symbol table, semantic checking, and intermediate java output code are in the subdirectories of tests: astout, symout, checkout, and intermedout, respectively.

Lessons Learned: Advice for Future Groups

Philippe-Guillaume Losembe

Under time constraints and frustration, it's easy to lose sight of the bigger picture. While working on semantically checking, for instance, we obsessed over how everything needed to be semantically checked. We would even get bogged down sometimes for hours on the tiniest of test cases and try to make semantic checking as thorough as possible.

My advice to future groups is that there will always be more to semantically check, but no matter what, don't get stuck in the details too much if it means losing sight of the bigger picture for the project.

Alisha Sindhwani

I think the biggest challenges of creating corgi was coordinating a semester-long group project, managing time well, and getting through the immense amount of details you need to think about when creating a language. We had a slow start to our project because of conflicting schedules and interviews, we all weren't able to meet at a consistent time on a regular basis which really hampered the initial start of our project. Additionally, our work was slow because we arbitrarily assigned tasks rather than considering the individual strengths and passions we weren't being efficient and when coupled with not being able to meet together, brought our project down to a halt. We also struggled with accountability and group members would be significantly late to meetings or not show up at all. It's very important to have a discussion at the beginning of the semester about accountability and being fair to all the group members.

One piece of advice I would give to future students is to always consider the big picture in mind of what the purpose of your language is and what kind of programs you want to create with it. We made countless tiny tests to check every single possible minutia that would possibly need to be semantically checked (you don't realize really what goes into a language until you start writing it!) and we started obsessing over it. But when you have a time crunch, you really need think about what you need to get done to get an interesting program you're passionate about it work rather than thinking of every possible incorrect program someone can input. Obviously, if you have ample time (which will not happen no matter how hard you try), it would be really fun to get your language as perfect as possible.

I would also warn students to be cautious when referencing past student's materials. Old projects are definitely enormously helpful but it is important to remember that they too were students and had other classes and time constraints to deal with and that their projects

weren't perfect. Other students' implementation may not be the most ideal way to do things, so critically think about how you think things should work when referencing other material.

Finally, always stay positive! You'll be creating something really cool regardless of what the final outcome is.

Melissa O'Sullivan

With project this size, it's hard to tell where to begin. You can spend a lot of time trying to figure out the best way to approach a problem. More important than starting with a flawless plan is just starting somewhere, quickly. Don't be afraid of trying something and throwing it out if it doesn't work. You gain a lot in this process and are well prepared to make the next iteration.

Testing is extremely important. It's very frustrating when working code stops working and you're not sure why. Setting up a strong testing infrastructure initially and ensuring that tests pass at every step can save an incredible amount of time.

Understand your limits when it comes to sleep deprivation and make sure you start early!

Justin Zhao

Looking back, I would say that spending the extra time to really think about how you are going to design your language really pulls through in the end. Towards the end of our project, for example, we realized that there were inefficiencies or better ways to do what we wanted to accomplish. We spent valuable time making design decisions that should have been made long before.

Testing cannot be emphasized enough. In my opinion, the real progress we made on corgi didn't come until we had solid test cases to run our compiler against. Adopt a testing framework early on and make sure everyone in the group follows that testing protocol.

For our group, the time constraint was arguably our largest problem. On several occasions in the nights before the project was due, we hypothetically wondered: "What if we had reached this point a week ago, or even a month ago?" We procrastinated working on the project out of a fear of getting stuck, but in the end, OCaml is actually kind of great and we should have started earlier.

What is also interesting is that Professor Edwards actually emphasizes a lot of the points here on the first few days of class, yet we, being the naive students we were, didn't take his advice to heart, and here we are spewing out the same advice. Listen to Professor Edwards -- he knows what's up!

Lastly, no matter what, stay positive, try your best, and if at all possible, start early. Ultimately, it’s an intense growing and learning experience with your teammates, and that in itself is gratifying enough.

Acknowledgements

For this project, we acknowledge the incredible resources and direction provided by our mentor Vaibhav (thanks for answering our emails late at night!), professor Stephen Edwards, and the collection of past projects to reference to help us through when we got stuck. In particular, pubCrawl and Lorax and, of course, microc.

Corgi Appendix: Root Directory

ast.ml

```
type op = Add | Sub | Mult | Div | Mod | Equal | Neq | Less | Leq | Greater |
Geq | And | Or

type uop = Neg | Not

type prim_type =
    Bool_Type
  | Int_Type
  | Pitch_Type
  | String_Type
  | Frac_Type
  | Rhythm_Type
  | Duration_Type
  | PD_Type
  | Chord_Type
  | Track_Type
  | Composition_Type
  | Null_Type

type types =
  Corgi_Prim of prim_type

type var = string * bool * prim_type

type expr =
    Bool_Lit of bool
  | Int_Lit of int
  | String_Lit of string
  | Frac_Lit of expr * expr (* int * int or Id's of type int *)
  | Id of string
  | Array_Lit of expr list
  | Binop of expr * op * expr
  | Unop of expr * uop
  (* | Create of types * string * expr  *)
  | Call of string * expr list
  | Access of string * expr
  | Tuple of expr * expr
  | Noexpr

type stmt =
```

```
    Block of block
  | Expr of expr
  | Assign of string * expr
  | Array_Assign of string * expr * expr
  | Return of expr
  | If of expr * block * block
  | For of stmt * stmt * stmt * block
  | While of expr * block

and block = {
    locals : var list;
    statements: stmt list;
    block_id: int;
}

(*type variable = {
  vname : string;
  vtype : types;
 vexpr : expr;
}*)

type parameter = {
  pname : string;
  ptype : prim_type;
}

type func = {
    ret_type : prim_type;
    fname : string;
    formals : var list;
    fblock : block;
  }

type program = var list * func list

(* Added from Lorax *)

type scope_var_decl = string * bool * prim_type * int

type scope_func_decl = string * prim_type * prim_type list * int

type decl =
    Func_Decl of scope_func_decl
  | Var_Decl of scope_var_decl


let string_of_prim_type = function
```

```
    Bool_Type -> "bool"
  | Int_Type -> "int"
  | Pitch_Type -> "pitch"
  | String_Type -> "string"
  | Frac_Type -> "frac"
  | Rhythm_Type -> "rhythm"
  | Duration_Type -> "duration"
  | Chord_Type -> "chord"
  | Track_Type -> "track"
  | Composition_Type -> "composition"
  | PD_Type -> "(pitch, duration)"
  | Null_Type -> "null"

let string_of_types = function
  Corgi_Prim(t) -> string_of_prim_type t

let string_of_unop = function
    Neg -> "-"
  | Not -> "!"

let string_of_binop = function
    Add -> "+"
  | Sub -> "-"
  | Mult -> "*"
  | Div -> "/"
  | Mod -> "%"
  | Equal -> "=="
  | Neq -> "!="
  | Less -> "<"
  | Leq -> "<="
  | Greater -> ">"
  | Geq -> ">="
  | And -> "&&"
  | Or -> "||"


let rec string_of_expr = function
  Bool_Lit(b) -> string_of_bool b
  | Int_Lit(i) -> string_of_int i
  | String_Lit(s) -> s
  | Frac_Lit(n, d) -> "$" ^ string_of_expr n ^ "/" ^ string_of_expr d ^ "$"
  | Array_Lit(e) -> String.concat ", " (List.map string_of_expr e)
  | Id(s) -> s
  | Access(ar, i) -> ar ^ "@" ^ string_of_expr i
  | Binop(e1, o, e2) ->
      string_of_expr e1 ^ " " ^
      string_of_binop o ^ " " ^
```

```
      string_of_expr e2
  | Unop(e, o) ->
      (match o with
          Neg -> "-" ^ string_of_expr e
        | Not -> "!" ^ string_of_expr e)
  (* |  Create(t, id, rhs) -> string_of_types t ^ " " ^ id ^ " = " ^
string_of_expr rhs  *)
  | Tuple(e1, e2) -> "(" ^ string_of_expr e1 ^ ", " ^ string_of_expr e2 ^ ")"
  | Call(f, e) ->
    f ^ "(" ^ String.concat ", " (List.map string_of_expr e) ^ ")"
  | Noexpr -> ""

(* let string_of_elif (expr, stmt) =
  "elif (" ^ string_of_expr expr ^ ") { \n" ^
  string_of_stmt stmt ^ "\n}\n"

let string_of_elseifs elseifs =
  String.concat "" (List.map (function(expr, stmt) -> string_of_expr expr ^
string_of_stmt stmt) elseifs) ^ "\n"  *)

(*
let string_of_vdecl vdecl = string_of_types vdecl.vtype ^ " " ^ vdecl.vname ^
              " = " ^ string_of_expr vdecl.vexpr ^ ";\n"
*)
let string_of_array_bool a =
  if a then "[] " else ""

let string_of_vdecl v =
  let (n, a, t) = v in
    string_of_prim_type t ^ " " ^ string_of_array_bool a ^ n

let rec string_of_stmt = function
    Block(b) -> string_of_block b
  | Expr(expr) -> string_of_expr expr ^ ";\n";
  | Assign(id, rhs) -> id ^ " = " ^ string_of_expr rhs ^ "; \n"
  | Return(expr) -> "return " ^ string_of_expr expr ^ ";\n";
  | If(e, b1, b2) ->
    (match b2.statements with
        [] -> "if (" ^ string_of_expr e ^ ")\n" ^ string_of_block b1
      | _  -> "if (" ^ string_of_expr e ^ ")\n" ^
              string_of_block b1 ^ "else\n" ^ string_of_block b2)
  | For(a1, c, a2, b) ->
      "for (" ^ string_of_stmt a1 ^ string_of_stmt c ^
      string_of_stmt a2  ^ ") " ^ string_of_block b
  | While(e, b) -> "while (" ^ string_of_expr e ^ ") " ^ string_of_block b

and string_of_block (b:block) =
```

```
  "{\n" ^
  String.concat ";\n" (List.map string_of_vdecl b.locals) ^ (if (List.length
b.locals) > 0 then ";\n" else "") ^
  String.concat "" (List.map string_of_stmt b.statements) ^
  "}\n"

let string_of_fdecl fdecl =
  (string_of_prim_type fdecl.ret_type) ^ " " ^
  fdecl.fname ^ "(" ^ String.concat ", " (List.map string_of_vdecl
fdecl.formals) ^ ")\n" ^
  string_of_block fdecl.fblock

(* need to rewrite *)
let string_of_decl = function
      Var_Decl(n, a, t, id)     -> string_of_vdecl (n, a, t)
    | Func_Decl(n, t, f, id) ->
      (string_of_prim_type t) ^ " " ^
      n ^ "(" ^
      String.concat ", " (List.map string_of_prim_type f) ^ ")"

(* ____________________________________ *)

let string_of_program (vars, funcs) =
  String.concat "" (List.map string_of_vdecl (List.rev vars) ) ^ "\n" ^
  String.concat "\n" (List.map string_of_fdecl (List.rev funcs) ) ^ "\n"
```

check.ml

```
open Ast

let fst_of_three (t, _, _) = t
let snd_of_three (_, t, _) = t
let thrd_of_three (_, _, t) = t

type d_expr =
      D_Bool_Lit of bool * prim_type
    | D_Int_Lit of int * prim_type
    | D_String_Lit of string * prim_type
    | D_Frac_Lit of d_expr * d_expr  * prim_type (* Expressions of type int *)
    | D_Id of string * prim_type
    | D_Array_Lit of d_expr list * prim_type
    | D_Binop of d_expr * op * d_expr * prim_type
    | D_Unop of d_expr * uop * prim_type
    | D_Call of string * d_expr list * prim_type
    | D_Tuple of d_expr * d_expr * prim_type  (* Come back and fix tuples *)
```

```
    | D_Access of string * d_expr * prim_type
    | D_Noexpr

type d_stmt =
      D_CodeBlock of d_block
    | D_Expr of d_expr
    | D_Assign of string * d_expr * prim_type
    | D_Array_Assign of string * d_expr * d_expr * prim_type
    | D_Return of d_expr
    | D_If of d_expr * d_stmt * d_stmt (* stmts of type D_CodeBlock *)
    | D_For of d_stmt * d_stmt * d_stmt * d_block (* stmts of type D_Assign |
D_Noexpr * D_Expr of type bool * D_Assign | D_Noexpr *)
    | D_While of d_expr * d_block

and d_block = {
    d_locals : scope_var_decl list;
    d_statements: d_stmt list;
    d_block_id: int;
}


type d_func = {
    d_fname : string;
    d_ret_type : prim_type; (* Changed from types for comparison error in
verify_stmt*)
    d_formals : scope_var_decl list;
    d_fblock : d_block;
}

type d_program = {
    d_gvars: scope_var_decl list;
    d_pfuncs: d_func list;
}

let type_of_expr = function
    D_Int_Lit(_,t) -> t
  | D_Bool_Lit(_,t) -> t
  | D_String_Lit(_,t) -> t
  | D_Frac_Lit(_,_,t) -> t
  | D_Id(_,t) -> t
  | D_Binop(_,_,_,t) -> t
  | D_Array_Lit (_, t) -> t
  | D_Unop (_, _, t) -> t
  | D_Call (_, _, t) -> t
  | D_Tuple (_, _, t) -> t (* Come back and fix tuples *)
  | D_Access (_, _, t) -> t
  | D_Noexpr -> Null_Type
```

```
let rec map_to_list_env func lst env =
    match lst with
          [] -> []
        | head :: tail ->
            let r = func head env in
                r :: map_to_list_env func tail env

let verify_gvar gvar env =
    let decl = Table.get_decl (fst_of_three gvar) env in
    match decl with
        Var_Decl(v) -> let (vname, varray, vtype, id) = v in
            (vname, varray, vtype, id)
        | _ -> raise(Failure("global" ^ (fst_of_three gvar) ^ " not a
variable"))

let verify_var var env =
    let decl = Table.get_decl (fst_of_three var) env in
    match decl with
        Func_Decl(f) -> raise(Failure("symbol is not a variable"))
      | Var_Decl(v) -> let (vname, varray, vtype, id) = v in
            (vname, varray, vtype, id)

let verify_is_func_decl name env =
    let decl = Table.get_decl name env in
    match decl with
        Func_Decl(f) -> name
        | _ -> raise(Failure("id " ^ name ^ " not a function"))

let verify_unop_and_get_type e unop =
    let e_type = type_of_expr e in
    match e_type with
          Bool_Type ->
            if unop = Neg then raise (Failure "incorrect negation operator
applied to Bool")
            else Bool_Type
        | Int_Type -> if unop = Not then raise (Failure "incorrect negation
operator applied to Int")
            else Int_Type
        | Frac_Type -> if unop = Not then raise (Failure "incorrect negation
operator applied to Frac")
            else Frac_Type
        | _ -> raise (Failure "negation operator applied to type that doesn't
support negation")

let verify_id_get_type id env =
    let decl = Table.get_decl id env in
```

```
    match decl with
        Var_Decl(v) -> let (_, _, t, _) = v in t
        | _ -> raise(Failure("id " ^ id ^ " not a variable."))

let verify_id_is_array id env =
    let decl = Table.get_decl id env in
    match decl with
        Var_Decl(v) -> let(_, is_array, _, _ ) = v in is_array
        | _ -> raise(Failure("id " ^ id ^ " not an array."))

let verify_binop l r op =
    let tl = type_of_expr l in
    let tr = type_of_expr r in
    match op with
        Add | Sub | Mult | Div  -> (match (tl, tr) with
            Int_Type, Int_Type -> Int_Type
            | Int_Type, Pitch_Type -> Pitch_Type
            | Int_Type, Frac_Type -> Frac_Type
            | Int_Type, Duration_Type -> Duration_Type
            | Pitch_Type, Int_Type -> Pitch_Type
            | Pitch_Type, Pitch_Type -> Pitch_Type
            | Frac_Type, Int_Type -> Frac_Type
            | Frac_Type, Frac_Type -> Frac_Type
            | Frac_Type, Duration_Type -> Duration_Type
            | Duration_Type, Int_Type -> Duration_Type
            | Duration_Type, Frac_Type -> Duration_Type
            | Duration_Type, Duration_Type -> Duration_Type
            | Track_Type, Track_Type -> Track_Type
            | _, _ -> raise(Failure("Cannot apply + - * / op to types " ^
string_of_prim_type tl ^ " + " ^ string_of_prim_type tr)))
        | Mod -> (match (tl, tr) with
            Int_Type, Int_Type -> Int_Type
            | _, _ -> raise(Failure("Can only apply % to int operands.")))
        | Equal | Neq -> if tl = tr then Bool_Type else (match(tl, tr) with
            | Int_Type, Pitch_Type -> Bool_Type
            | Int_Type, Frac_Type -> Bool_Type
            | Int_Type, Duration_Type -> Bool_Type
            | Pitch_Type, Int_Type -> Bool_Type
            | Frac_Type, Int_Type -> Bool_Type
            | Frac_Type, Duration_Type -> Bool_Type
            | Duration_Type, Int_Type -> Bool_Type
            | Duration_Type, Frac_Type -> Bool_Type
            | _, _ -> raise(Failure("Cannot apply == !=  op to types " ^
string_of_prim_type tl ^ " + " ^ string_of_prim_type tr)))
        | Less | Greater | Leq | Geq-> (match (tl, tr) with
            Int_Type, Int_Type -> Bool_Type
            | Int_Type, Pitch_Type -> Bool_Type
```

```
            | Int_Type, Frac_Type -> Bool_Type
            | Int_Type, Duration_Type -> Bool_Type
            | Pitch_Type, Int_Type -> Bool_Type
            | Pitch_Type, Pitch_Type -> Bool_Type
            | Frac_Type, Int_Type -> Bool_Type
            | Frac_Type, Frac_Type -> Bool_Type
            | Frac_Type, Duration_Type -> Bool_Type
            | Duration_Type, Int_Type -> Bool_Type
            | Duration_Type, Frac_Type -> Bool_Type
            | Duration_Type, Duration_Type -> Bool_Type
            | String_Type, String_Type -> Bool_Type
            | _, _ -> raise(Failure("Cannot apply < > <= >=  op to types " ^
string_of_prim_type tl ^ " + " ^ string_of_prim_type tr)))
        | And | Or -> (match (tl, tr) with
            Bool_Type, Bool_Type -> Bool_Type
            | _, _ -> raise(Failure("Cannot apply && ||  op to types " ^
string_of_prim_type tl ^ " + " ^ string_of_prim_type tr)))

let verify_tuple_types p d =
    match type_of_expr p with
        Int_Type | Pitch_Type -> (match type_of_expr d with
            Int_Type | Frac_Type | Duration_Type -> true
            | _ -> raise(Failure("Second term in tuple must be of type duration
(,*)"))
        )
        | _ -> raise(Failure("First term in tuple must be of type pitch (*,)"))

let verify_expr_as_pitch p env = match p with
    Int_Lit(i) -> D_Int_Lit(i, Pitch_Type)
    | Id(s) -> (match (verify_id_get_type s env) with
        Int_Type | Pitch_Type -> D_Id(s, Pitch_Type)
        | _ -> raise(Failure("expected expression of type pitch (*,)")))
    | _ -> raise(Failure("expected expression of type pitch (*,)"))

let set_dexpr_type e t = match e with
        D_Int_Lit(i,_) -> D_Int_Lit(i,t)
      | D_Bool_Lit(b,_) -> D_Bool_Lit(b,t)
      | D_String_Lit(s,_) -> D_String_Lit(s,t)
      | D_Frac_Lit(e1,e2,_) -> D_Frac_Lit(e1,e2,t)
      | D_Id(s,_) -> D_Id(s,t)
      | D_Binop(e1,o,e2,_) -> D_Binop(e1,o,e2,t)
      | D_Array_Lit (l, _) -> D_Array_Lit (l, t)
      | D_Unop (e, u, _) -> D_Unop (e, u, t)
      | D_Call (s, a, _) -> D_Call (s, a, t)
      | D_Tuple (p, d, _) -> D_Tuple (p, d, t)
      | D_Access (a, i, _) -> D_Access (a, i, t)
      | D_Noexpr -> D_Noexpr
```

```
let rec verify_expr expr env =
    match expr with                                          (* expr evaluates to
*)
          Bool_Lit(b)     -> D_Bool_Lit(b,Bool_Type)         (* D_Bool_Lit *)
        | Int_Lit(i)      -> D_Int_Lit(i, Int_Type)          (* D_Int_Lit *)
        | String_Lit(s)   -> D_String_Lit(s, String_Type)    (* D_String_Lit*)
        | Frac_Lit(n,d)   ->                                 (* D_Frac_Lit *)
            let vn = verify_expr n env in
            let vd = verify_expr d env in
            if type_of_expr vn <> Int_Type || type_of_expr vd <> Int_Type then
                raise(Failure("Fraction literal must have integer numerator and
denominator."))
            else D_Frac_Lit(vn, vd, Frac_Type)
        | Id(s)           ->                                 (* D_Id_Lit *)
            let vid_type = verify_id_get_type s env in
            D_Id(s, vid_type)
        | Binop(l, op, r) ->
            let vl = verify_expr l env in
            let vr = verify_expr r env in
            let vtype = verify_binop vl vr op in
            (* if vtype = Bool_Type && (op <> And || op <> Or) then *)
            let vtl = type_of_expr vl in
            let vtr = type_of_expr vr in
            if vtl = vtr then D_Binop(vl, op, vr, vtype)
            else (match (vtl, vtr) with
                Int_Type, Frac_Type | Frac_Type, Int_Type ->
D_Binop(set_dexpr_type vl Frac_Type, op, set_dexpr_type vr Frac_Type, vtype)
                | Int_Type, Pitch_Type | Pitch_Type, Int_Type ->
D_Binop(set_dexpr_type vl Pitch_Type, op, set_dexpr_type vr Pitch_Type, vtype)
                | Int_Type, Duration_Type | Duration_Type, Int_Type ->
D_Binop(set_dexpr_type vl Duration_Type, op, set_dexpr_type vr Duration_Type,
vtype)
                | Frac_Type, Duration_Type | Duration_Type, Frac_Type->
D_Binop(set_dexpr_type vl Duration_Type, op, set_dexpr_type vr Duration_Type,
vtype)
                | _, _ -> raise(Failure("Congratulations on raising the
impossible failure.")))
            (* else D_Binop(vl, op, vr, vtype) *)                   (* D_Binop
*)
        | Unop(e, uop) ->
            let ve = verify_expr e env in
            let ve_type = verify_unop_and_get_type ve uop in
            D_Unop(ve, uop, ve_type)                         (* D_Unop *)
        | Array_Lit (ar) ->
```

```
            let (va, va_type) = verify_array ar env in
            D_Array_Lit(va, va_type)                        (* D_Array_Lit *)
        | Call (name, args) ->
            let va = verify_expr_list args env in
            let vt = verify_call_and_get_type name va env in
            D_Call(name, va, vt)                             (* D_Call *)
        | Tuple(e1, e2) ->                                   (* D_Tuple *)
            let ve1 = verify_expr_as_pitch e1 env in
            let ve2 = verify_expr_as_duration e2 env in
            if verify_tuple_types ve1 ve2 then D_Tuple(ve1, ve2, PD_Type)
            else raise(Failure("Invalid tuple."))
        | Access(ar, i) ->
            let is_array = verify_id_is_array ar env in
            let ar_type = verify_id_get_type ar env in
            let vi = verify_expr i env in
            let vit = type_of_expr vi in
            if vit = Int_Type && is_array then
                let accessed_type = (match ar_type with
                    Composition_Type -> Track_Type
                    | Track_Type -> Chord_Type
                    | Chord_Type -> PD_Type
                    | Rhythm_Type -> Duration_Type
                    | _ -> ar_type) in D_Access(ar, vi, accessed_type)
            else raise(Failure("symbol " ^ ar ^ " must be an array, index must
be of type int"))
        | Noexpr -> D_Noexpr

and verify_expr_as_duration d env = match d with
    Int_Lit(i) -> D_Int_Lit(i, Duration_Type)
    | Frac_Lit(n, d) ->
        let vn = verify_expr n env in
        let vd = verify_expr d env in
        if type_of_expr vn <> Int_Type || type_of_expr vd <> Int_Type then
            raise(Failure("Fraction literal must have integer numerator and
denominator."))
        else D_Frac_Lit(vn, vd, Duration_Type)
    | Id(s) -> (match (verify_id_get_type s env) with
        Int_Type | Frac_Type | Duration_Type -> D_Id(s, Duration_Type)
        | _ -> raise(Failure("expected expression of type duration (,*)")))
    | _ -> raise(Failure("expected expression of type duration (,*)"))

and verify_array arr env =
    match arr with
    [] -> ([], Null_Type) (* Empty *)
    | head :: tail ->
        let verified_head = verify_expr head env in
        let head_type = type_of_expr verified_head in
```

```
            let rec verify_list_and_type l t e = match l with
                [] -> ([], t)
                | hd :: tl ->
                    let ve = verify_expr hd e in
                    let te = type_of_expr ve in
                    if t = te then (ve :: (fst (verify_list_and_type tl te e)),
t)
                    else raise (Failure "Elements of inconsistent types in
Array_Lit")
            in
        (verified_head :: (fst (verify_list_and_type tail head_type env)),
head_type)

and verify_expr_list lst env =
    match lst with
    [] -> []
    | head :: tail -> verify_expr head env :: verify_expr_list tail env

and verify_call_and_get_type name vargs env =
    let decl = Table.get_decl name env in (* function name in symbol table *)
    let fdecl = match decl with
        Func_Decl(f) -> f                    (* check if it is a function *)
        | _ -> raise(Failure (name ^ " is not a function")) in
    if name = "print" then Int_Type          (* Add more builtins when we have
more builtins *)
    (* else if name = "import" then Composition_Type
    else if name = "export" then Int_Type *)
    else if name = "length" then Int_Type
    else
        let (_,rtype,params,_) = fdecl in
        if (List.length params) = (List.length vargs) then
            let arg_types = List.map type_of_expr vargs in
            if params = arg_types then rtype
            else raise(Failure("Argument types in " ^ name ^ " call do not match
formal parameters."))
        else raise(Failure("Function " ^ name ^ " takes " ^ string_of_int
(List.length params) ^
                           " arguments, called with " ^ string_of_int
(List.length vargs)))

let verify_id_match_type (id:string) ve env =
    let decl = Table.get_decl id env in
    let vdecl = match decl with (* check that id refers to a variable *)
    Var_Decl(v) -> v
    | _ -> raise(Failure (id ^ " is not a variable")) in
    let (_,is_array, id_type, _) = vdecl in
    let vt = type_of_expr ve in
```

```
    if is_array then
        (match ve with
        D_Array_Lit(_, _) -> if id_type = vt then id_type(* Check that it goes
into id's type *)
            else (match(id_type, vt) with
                Rhythm_Type, Duration_Type
                | Rhythm_Type, Frac_Type
                | Composition_Type, Track_Type
                | Chord_Type, PD_Type
                | Track_Type, Chord_Type -> id_type
                | _, _ -> raise(Failure("Cannot assign " ^ string_of_prim_type
vt ^ " to " ^ id ^ " of type " ^ string_of_prim_type id_type)))
                | D_Id(s, _) -> if verify_id_is_array s env then (
                        if id_type = vt then id_type
                        else (match(id_type, vt) with (* Compatible simple types
*)
                            Frac_Type, Int_Type
                            | Duration_Type, Int_Type
                            | Duration_Type, Frac_Type
                            | Pitch_Type, Int_Type -> id_type
                            | _, _ -> raise(Failure("Cannot assign  " ^
string_of_prim_type vt ^ " to " ^ id ^ " of type " ^ string_of_prim_type id_type
))
                        )
                    ) else raise(Failure("Cannot assign single element to
array."))
        | D_Tuple(_, _, _) -> (match (id_type, vt) with
            Chord_Type, PD_Type -> id_type
            | _, _ -> raise(Failure("Can only assign (pitch, duration) to
rhythms")))
        | D_Binop(_,_,_,t) -> t
        | D_Access(_,_,t) -> t
        | D_Call(_,_,t) -> t
        | _ -> raise(Failure("Cannot assign ...." ^ string_of_prim_type vt ^ "
to " ^ id ^ " of type " ^ string_of_prim_type id_type )))
    else (* id is not an array *)
        if id_type = vt then id_type else (match (id_type, vt) with
            Frac_Type, Int_Type
            | Duration_Type, Int_Type
            | Duration_Type, Frac_Type
            | Pitch_Type, Int_Type  -> id_type
            | _, _ -> raise(Failure("Cannot assign " ^ string_of_prim_type vt ^
" to " ^ id ^ " of type " ^ string_of_prim_type id_type )))

let rec verify_stmt stmt ret_type env =
    match stmt with
    Return(e) ->
```

```
        let verified_expr = verify_expr e env in
        if ret_type = type_of_expr verified_expr then D_Return(verified_expr)
        else raise(Failure "return type does not match")
    | Expr(e) ->
        let verified_expr = verify_expr e env in
        D_Expr(verified_expr)
    | Assign(id, e) -> (* Verify that id is compatible type to e *)
        let ve = verify_expr e env in
        let vid_type = verify_id_match_type id ve env in
        let ve_type = type_of_expr ve in
        if (match vid_type with Rhythm_Type | Chord_Type | Track_Type |
Composition_Type -> true | _ -> false)
            then D_Assign(id, ve, vid_type)
        else D_Assign(id, set_dexpr_type ve vid_type, vid_type)

    | Array_Assign(id, e, i) ->
        let ve = verify_expr e env in
        let vid_type = verify_id_match_type id ve env in
        let vi = verify_expr i env in
        if type_of_expr vi = Int_Type then D_Array_Assign(id, ve, vi, vid_type)
        else raise(Failure("Array index must be of type int."))
    | Block(b) ->
        let verified_block = verify_block b ret_type (fst env, b.block_id) in
        D_CodeBlock(verified_block)
    | If(e, b1, b2) ->
        let verified_expr = verify_expr e env in
        if (type_of_expr verified_expr) = Bool_Type then
            let vb1 = verify_block b1 ret_type (fst env, b1.block_id) in
            let vb2 = verify_block b2 ret_type (fst env, b2.block_id) in
            D_If(verified_expr, D_CodeBlock(vb1), D_CodeBlock(vb2))
        else raise(Failure("Condition in if statement must be a boolean
expression."))
    | For(assignment1, condition, assignment2, block) ->
        let va1 = (match assignment1 with
            Assign(_, _) | Expr(_) ->  verify_stmt assignment1 ret_type env
            | _ -> raise(Failure("First term in For statement must be assignment
or no expression. (*;;)"))) in
        let vc = (match condition with
            Expr(e) ->
                let ve = verify_expr e env in
                let vt = type_of_expr ve in
                if vt = Bool_Type || vt = Null_Type then verify_stmt condition
ret_type env
                else let () = print_endline ("vt = " ^ string_of_prim_type vt)
in
                    raise(Failure("Condition in For statement must be boolean or
no expression. (;*;)"))
```

```
            | _ -> raise(Failure("Condition in For statement must be boolean or
no expression. (;*;)"))) in
        let va2 = (match assignment1 with
            Assign(_, _) | Expr(_) ->  verify_stmt assignment2 ret_type env
            | _ -> raise(Failure("Last term in For statement must be assignment
or no expression. (;;*)"))) in
        let vb = verify_block block ret_type (fst env, block.block_id) in
        D_For(va1, vc, va2, vb)
    | While(condition, block) ->
        let vc = verify_expr condition env in
        let vt = type_of_expr vc in
        if vt = Bool_Type then
            let vb = verify_block block ret_type (fst env, block.block_id) in
            D_While(vc, vb)
        else raise(Failure("Condition in While statement must be boolean."))



and verify_stmt_list stmt_list ret_type env =
    match stmt_list with
          [] -> []
        | head :: tail -> (verify_stmt head ret_type env) :: (verify_stmt_list
tail ret_type env)

and verify_block block ret_type env =
    let verified_vars = map_to_list_env verify_var block.locals (fst env,
block.block_id) in
    let verified_stmts = verify_stmt_list block.statements ret_type env in
    { d_locals = verified_vars; d_statements = verified_stmts; d_block_id =
block.block_id }

(*verify formals, get return type, verify function name, verify fblock *)
let verify_func func env =
    (* let () = Printf.printf "verifying function \n" in *)
    let verified_block = verify_block func.fblock func.ret_type (fst env,
func.fblock.block_id) in
    (* let () = Printf.printf "func.fname" in *)
    let verified_formals = map_to_list_env verify_var func.formals (fst env,
func.fblock.block_id) in
    let verified_func_decl = verify_is_func_decl func.fname env in
    { d_fname = verified_func_decl; d_ret_type = func.ret_type; d_formals =
verified_formals; d_fblock = verified_block }

let verify_semantics program env =
    let (gvar_list, func_list) = program in
    let verified_gvar_list = map_to_list_env verify_var gvar_list env in
    (* let () = Printf.printf "after verifying gvars \n" in *)
```

```
    let verified_func_list = map_to_list_env verify_func func_list env in
    (* let () = Printf.printf "after verifying functions \n" in *)
    let () = prerr_endline "// Passed semantic checking \n" in
        { d_pfuncs = verified_func_list; d_gvars = verified_gvar_list}
```

javagen.ml

```
open Ast
open Check

(* To Do:
    length and access
    D_call???
*)

let remove_semi s =
    if String.contains s ';' then
    let i = String.index s ';' in
    String.sub s 0 i
    else s

let write_type = function
      Bool_Type -> "Boolean"
    | Int_Type -> "int"
    | String_Type -> "String"
    | Pitch_Type -> "Pitch"
    | Frac_Type -> "Frac"
    | Rhythm_Type -> "Rhythm"
    | Duration_Type -> "Duration"
    | Chord_Type -> "Chord"
    | Track_Type -> "Track"
    | Composition_Type -> "Composition"
    | PD_Type -> "Pitch_Duration_Tuple"
    | _ -> raise(Failure "Type string of PD_Tuple or Null_Type being generated")

let write_types ts =
    match ts with Corgi_Prim(t) -> write_type t

let write_op_primitive = function
    Add -> " + "
    | Sub -> " - "
    | Mult -> " * "
    | Div -> " / "
    | Equal -> " == "
    | Neq -> " != "
```

```
    | Less -> " < "
    | Leq -> " <= "
    | Greater -> " > "
    | Geq -> " >= "
    | Mod -> " % "
    | _ -> raise (Failure "and/or begin applied to a java primitive")

let write_op_compares e1 op e2 =
    match op with
    Equal -> "(" ^ e1 ^ ").equals(" ^ e2 ^ ")"
    | Less ->  "(" ^ e1 ^ ").compareTo(" ^ e2 ^ ")" ^ " < 0"
    | Leq -> "(" ^ e1 ^ ").compareTo(" ^ e2 ^ ")" ^ " <= 0"
    | Greater -> "(" ^ e1 ^ ").compareTo(" ^ e2 ^ ")" ^ " > 0"
    | Geq -> "(" ^ e1 ^ ").compareTo(" ^ e2 ^ ")" ^ " >= 0"
    | Neq -> "(" ^ e1 ^ ").compareTo(" ^ e2 ^ ")" ^ " != 0"
    | _ -> raise (Failure "not a comparator operation")

let rec get_typeof_dexpr = function
      D_Bool_Lit(boolLit, t) -> t
    | D_Int_Lit(intLit, t) -> t
    | D_String_Lit(strLit, t) -> t
    | D_Frac_Lit(num_expr, denom_expr, t) -> t
    | D_Id (str, t) -> t
    | D_Array_Lit(dexpr_list, t) -> t
    | D_Unop(d_expr, uop, t) -> t
    | D_Binop (dexpr1, op, dexpr2, t) -> t
    | D_Tuple(dexpr1, dexpr2, t) -> t
    (* | D_Null_Lit -> "null" *)
    | D_Noexpr -> Null_Type
    | D_Call(str,dexpr_list,t) -> t
    | D_Access(str,dexpr,t) -> t


let rec write_expr = function
      D_Bool_Lit(boolLit, t) -> string_of_bool boolLit
    | D_Int_Lit(intLit, t) -> (match t with
                                    Int_Type -> string_of_int intLit
                                  | Pitch_Type -> "new Pitch(" ^ string_of_int
intLit  ^ ")"
                                  | Duration_Type -> "new Duration(" ^
string_of_int intLit ^ ")" ^ string_of_int intLit
                                  | _ -> raise(Failure(write_type t ^ " is not a
integer")))
    | D_String_Lit(strLit, t) -> "\"" ^ strLit ^ "\""
    | D_Frac_Lit(num_expr, denom_expr, t) -> (match t with
                                       Frac_Type -> "new Frac(" ^ write_expr
num_expr ^ "," ^ write_expr denom_expr ^ ")"
```

```
                                    | Duration_Type -> "new Duration(new Frac("
^ write_expr num_expr ^ "," ^ write_expr denom_expr ^ "))"
                                    | _ -> raise(Failure(write_type t ^ " is not
a fraction")))
    | D_Id (str, yt) -> str
    | D_Array_Lit(dexpr_list, t) -> write_array_expr dexpr_list t
    | D_Unop(d_expr, uop, t) -> write_unop_expr d_expr uop t
    | D_Binop (dexpr1, op, dexpr2, t) -> write_binop_expr dexpr1 op dexpr2 t
    | D_Tuple(dexpr1, dexpr2, t) ->  "new Pitch_Duration_Tuple(" ^ write_expr
dexpr1 ^ "," ^ write_expr dexpr2 ^")"
    (* | D_Null_Lit -> "null" *)
    | D_Noexpr -> ""
    | D_Call(str,dexpr_list,t) -> (match str with
                                    "print" -> "System.out.println("  ^
String.concat "+" (List.map tostring_str dexpr_list) ^ ")"
                                    | "play" -> "Utils." ^ str ^ "(" ^
String.concat "," (List.map write_expr dexpr_list) ^ ")"
                                    | "export" -> "Utils.exportMidi(" ^
String.concat "," (List.map write_expr dexpr_list) ^ ")"
                                    | "import" -> "Utils.importMidi(" ^
String.concat "," (List.map write_expr dexpr_list) ^ ")"
                                    | "length" ->  String.concat "," (List.map
write_expr dexpr_list) ^ ".length()" (* semantic checking ensures length has 1
arg *)
                                    | _ -> str ^ "(" ^ String.concat "," (List.map
write_expr dexpr_list) ^ ")")
    | D_Access(str,dexpr,t) -> (match t with
                                  (Bool_Type | Int_Type | Frac_Type |
Duration_Type | String_Type | Pitch_Type) ->  str ^ "["  ^ write_expr dexpr ^
"]"
                                 | _ -> str ^ ".get(" ^ write_expr dexpr ^ ")")




and write_binop_expr expr1 op expr2 t =
    let e1 = write_expr expr1 and e2 = write_expr expr2 in
        let write_binop_expr_help e1 op e2 =
            match t with
                Int_Type -> (match op with
                    (Add | Sub | Mult | Div | Equal | Neq | Less | Leq | Mod |
Greater | Geq | And | Or) ->
                    e1 ^ write_op_primitive op ^ e2)
              | String_Type -> (match op with
                     Add -> " + "
                    | (Equal | Less | Leq | Greater | Geq) -> write_op_compares
e1 op e2
```

```
                    | _ -> raise(Failure(write_op_primitive op ^ " is not a
supported operation for String_Type")))
               | Bool_Type -> (match op with
                       And -> e1 ^ " && " ^ e2
                     | Or -> e1 ^ " || " ^ e2
                     | _ -> write_binop_expr expr1 op expr2 (get_typeof_dexpr
expr1))
                     (* this function assumes that the return type of the binop
is the return type of dexpr1 and dexpr2,
                      but in the case of comparaters (like i < 10 where i is an
int. the return type is boolean even
                      though dexprs are ints! so fool this method by calling it
again with the return type of int!*)
               | (Pitch_Type | Frac_Type | Rhythm_Type | Duration_Type |
Chord_Type | Track_Type | Composition_Type) -> (match op with
                     (Equal | Less | Leq | Greater | Geq | Neq) ->
write_op_compares e1 op e2
                     | Add -> "(" ^ e1 ^ ").add(" ^ e2 ^ ")"
                     | Sub -> "(" ^ e1 ^ ").subtract(" ^ e2 ^ ")"
                     | Mult -> "(" ^ e1 ^ ").multiply(" ^ e2 ^ ")"
                     | Div -> "(" ^ e1 ^ ").divide(" ^ e2 ^ ")"
                     | _ -> raise(Failure(write_op_primitive op ^ " is not a
supported operation for" ^ write_type t)))
               | _ -> raise(Failure(write_op_primitive op ^ " is not a supported
operation for" ^ write_type t))
        in write_binop_expr_help e1 op e2

and write_unop_expr dexpr uop t =
    (match uop with
      Neg -> "-(" ^ write_expr dexpr ^ ")"
    | Not -> "!" ^ write_expr dexpr)

and write_array_expr dexpr_list t =
      match t with
      PD_Type -> "new Pitch_Duration_Tuple[]"  ^ " {" ^ String.concat ","
(List.map write_expr dexpr_list) ^ "}"
    | _ -> "new " ^ write_type t ^ " []"  ^ " {" ^ String.concat "," (List.map
write_expr dexpr_list) ^ "}"

and write_tostr_class dexpr =
    let t = get_typeof_dexpr dexpr in
    match t with
          Bool_Type -> "Boolean"
        | Int_Type -> "Integer"
        | _ -> raise (Failure "toString method should already be in class")

and tostring_str dexpr =
```

```
    let t = get_typeof_dexpr dexpr in
    match t with
          (Bool_Type | Int_Type) -> write_tostr_class dexpr ^ ".toString(" ^
write_expr dexpr ^ ")"
        | String_Type -> write_expr dexpr
        | _ -> "(" ^ write_expr dexpr ^ ").toString()"


let write_scope_var_decl_func svd =
    let (n, b, t, _) = svd in
        match b with
          true -> (match t with
                      (Bool_Type | Int_Type | Frac_Type | Duration_Type |
String_Type | Pitch_Type) -> write_type t ^ "[]" ^ n (* true if it is an array
*)
                    | _ -> write_type t ^ " " ^ n)
        | false -> write_type t ^ " " ^ n

let write_scope_var_decl svd =
    write_scope_var_decl_func svd ^ ";\n"

let write_global_scope_var_decl gsvd =
    "static " ^ write_scope_var_decl_func gsvd ^ ";\n"


let write_assign name dexpr t =
    (match t with
      Bool_Type | Int_Type | String_Type | Frac_Type -> name ^ " = " ^
write_expr dexpr
    | Pitch_Type | Duration_Type | Rhythm_Type | Chord_Type | Track_Type |
Composition_Type -> name ^ " = new " ^ write_type t ^ "(" ^ write_expr dexpr ^
")"
    | _ -> raise(Failure(write_type t ^ " is not a valid assign_type")))

let rec write_stmt = function
      D_CodeBlock(dblock) -> write_block dblock
    | D_Expr(dexpr) -> write_expr dexpr ^ ";"
    | D_Assign (name, dexpr, t) -> write_assign name dexpr t ^ ";\n"
    | D_Return(dexpr) -> "return " ^ write_expr dexpr ^ ";\n"
    | D_If(dexpr, dstmt1, dstmt2) -> "if(" ^ write_expr dexpr ^  ")" ^
write_stmt dstmt1 ^ "else"  ^ write_stmt dstmt2
    | D_For(dstmt1, dstmt2, dstmt3, dblock) -> "for(" ^ write_stmt dstmt1  ^
write_stmt dstmt2 ^ remove_semi (write_stmt dstmt3) ^ ")" ^ write_block dblock
    | D_While(dexpr, dblock) -> "while(" ^ write_expr dexpr ^ ")"  ^ write_block
dblock
    | D_Array_Assign(str,dexpr_value, dexpr_index, t) -> str ^ ".set(" ^
write_expr dexpr_index ^ "," ^ write_expr dexpr_value ^ ");"
```

```
and write_block dblock =
    "{\n" ^  String.concat "\n" (List.map write_scope_var_decl dblock.d_locals)
^ String.concat "\n" (List.map write_stmt dblock.d_statements) ^ "\n}"

let write_func dfunc =
    match dfunc.d_fname with
    "main" -> "public static void main(String[] args)" ^ write_block
dfunc.d_fblock
    | _ -> "static " ^ write_type dfunc.d_ret_type ^ " " ^ dfunc.d_fname ^ "("
^ String.concat "," (List.map write_scope_var_decl_func dfunc.d_formals) ^ ")" ^
write_block dfunc.d_fblock

let write_pgm pgm =
     "public class Intermediate {\n" ^ String.concat "\n" (List.map
write_global_scope_var_decl pgm.d_gvars) ^ String.concat "\n" (List.map
write_func pgm.d_pfuncs) ^ "}"
```

interpreter.ml

```
type action = Ast | Symtab | Sem | Javagen | Help

let usage (name:string) =
  "usage:\n" ^ name ^ "\n" ^
    "        -ast < source.corg              (Print AST of source)\n" ^
    "        -sym < source.corg              (Print Symbol Table of source)\n" ^
    "            -sem < source.corg                  (Print Semantic Analysis
of source)\n" ^
    "        -javagen < source.corg          (Print java intermediate code of
source)\n"

let _ =
  let action =
  if Array.length Sys.argv > 0 then
    (match Sys.argv.(1) with
        "-ast" -> Ast
      | "-sym" -> Symtab
      | "-sem" -> Sem
      | "-javagen" -> Javagen
      | _ -> Help)
  else Help in

  match action with
      Help -> print_endline (usage Sys.argv.(0))
    | _ ->
      let lexbuf = Lexing.from_channel stdin in
      let program = Parser.program Scanner.token lexbuf in
      (match action with
          Ast -> let listing = Ast.string_of_program program
                 in prerr_string listing
        | Symtab -> let env = Table.build_table program in
                    prerr_string (Table.string_of_table env)
        | Sem -> let env = Table.build_table program in
                    let checked = Check.verify_semantics program env in
                        ignore checked;
        | Javagen -> let env = Table.build_table program in
                let checked = Check.verify_semantics program env in
                let outstring = Javagen.write_pgm checked in
                prerr_string outstring
        | Help -> print_endline (usage Sys.argv.(0)))
```

Makefile

```
#OBJS = ast.cmo symtab.cmo parser.cmo scanner.cmo interpreter.cmo
```

```
OBJS = ast.cmo table.cmo check.cmo parser.cmo scanner.cmo javagen.cmo
interpreter.cmo

interpreter: $(OBJS)
    ocamlc -o interpreter -g $(OBJS)

scanner.ml: scanner.mll
    ocamllex scanner.mll

parser.ml parser.mli : parser.mly
    ocamlyacc -v parser.mly

%.cmo : %.ml
    ocamlc -g -c $<

%.cmi : %.mli
    ocamlc -g -c $<

.PHONY : clean
clean :
    rm -rf interpreter parser.ml parser.mli scanner.ml *.cmo *.cmi *.output

all : clean interpreter


ast.cmo:
ast.cmx:

symtab.cmo: ast.cmo
symtab.cmx: ast.cmx

check.cmo: table.cmo
check.cmx: table.cmx

javagen.cmo: check.cmo
javagen.cmx: check.cmx

interpreter.cmo: scanner.cmo parser.cmi ast.cmo symtab.cmo check.cmo javagen.cmo
interpreter.cmx: scanner.cmx parser.cmx ast.cmx symtab.cmx check.cmx javagen.cmx

parser.cmo: ast.cmo parser.cmi
parser.cmx: ast.cmx parser.cmi
scanner.cmo: parser.cmi
scanner.cmx: parser.cmx
parser.cmi: ast.cmo
```

parser.mly

```
%{ open Ast

let scope_id = ref 1

let inc_block_id (u:unit) =
    let x = scope_id.contents in
    scope_id := x + 1; x
%}

%token SEMI LPAREN RPAREN LBRACE LBRACKET RBRACE RBRACKET COMMA
%token PLUS MINUS TIMES DIVIDE MOD ASSIGN ARRAY_ASSIGN AT
%token EQ NEQ LT LEQ GT GEQ DOLLAR
%token RETURN IF ELIF ELSE FOR WHILE
/* %token BREAK CONTINUE */
%token TRUE FALSE NULL
%token AND OR NOT

%token BOOL INT
%token STRING RHYTHM CHORD TRACK COMPOSITION
%token FRAC PITCH DURATION

%token <string> ID
%token <int> INT_LIT
%token <bool> BOOL_LIT
%token <string> FRAC_LIT
%token <string> STRING_LIT
%token <string> ARRAY_LIT
%token EOF

%nonassoc NOELSE
%nonassoc ELSE ELIF
%right ASSIGN
%right DOLLAR
%left OR
%left AND
%left EQ NEQ
%left LT GT LEQ GEQ
%left PLUS MINUS
%left TIMES DIVIDE MOD
%left NEG NOT

%start program
%type <Ast.program> program
```

```
%%

program:
   /* nothing */ { [], [] }
 | program vdecl { ($2 :: fst $1), snd $1 }
 | program fdecl { fst $1, ($2 :: snd $1) }

prim_type:
    BOOL {Bool_Type}
  | INT {Int_Type}
  | STRING {String_Type}
  | FRAC {Frac_Type}
  | PITCH {Pitch_Type}
  | DURATION {Duration_Type}
  | RHYTHM {Rhythm_Type}
  | CHORD {Chord_Type}
  | TRACK {Track_Type}
  | COMPOSITION {Composition_Type}

fdecl:
   prim_type ID LPAREN formals_opt RPAREN LBRACE vdecl_list stmt_list RBRACE
     { { ret_type = $1;
         fname = $2;
           formals = $4;
           fblock = {locals = List.rev $7; statements = List.rev $8; block_id =
inc_block_id ()} } }

formals_opt:
    /* nothing */ { [] }
  | formal_list   { List.rev $1 }

formal_list:
    fvdecl                { [$1] }
  | formal_list COMMA fvdecl { $3 :: $1 }

/* Suggested to make vdecls a statement */
vdecl_list:
    /* nothing */    { [] }
  | vdecl_list vdecl { $2 :: $1 }

/* Was here before
vdecl:
   prim_type ID { ({vname = $2; vtype = $1; vexpr = Noexpr}) }
   | prim_type ID ASSIGN expr { {vname = $2; vtype = $1; vexpr = $4}}
*/
vdecl:
  prim_type ID SEMI{ ($2, false, $1) }
```

```
  | prim_type LBRACKET RBRACKET ID SEMI { ($4, true, $1)}

fvdecl:
  prim_type ID { ($2, false, $1) }
  | prim_type LBRACKET RBRACKET ID { ($4, true, $1) }

stmt_list:
    /* nothing */  { [] }
  | stmt_list stmt { $2 :: $1 }

stmt:
    block { Block($1) }
  | expr SEMI { Expr($1) }
  | RETURN expr SEMI { Return($2) }
  | IF LPAREN expr RPAREN block %prec NOELSE { If($3, $5, {locals = [];
statements = []; block_id = inc_block_id ()}) }
  | IF LPAREN expr RPAREN block ELSE block { If($3, $5, $7) }
  | FOR LPAREN expr_opt SEMI expr_opt SEMI expr_opt RPAREN block { For($3, $5,
$7, $9) }
  | WHILE LPAREN expr RPAREN block { While($3, $5) }
  | ID ASSIGN expr SEMI{ Assign($1, $3) }
  | ID ASSIGN AT int_expr expr SEMI { Array_Assign($1, $5, $4)}
  /* array_variable = @ 2 4; */

block:
  LBRACE stmt_list RBRACE { {locals = []; statements = List.rev $2; block_id =
inc_block_id ()} }

/*
elifs:
  { [] }
  | elifs ELIF LPAREN expr RPAREN stmt { ($4, $6) :: $1 }
*/

expr_opt:
    /* nothing */ { Expr(Noexpr) }
  | expr          { Expr($1) }
  | ID ASSIGN expr{ Assign($1, $3) }

expr:
  literal {$1}
  | DOLLAR int_expr DIVIDE int_expr DOLLAR {Frac_Lit($2, $4)}
  | LBRACKET expr_list RBRACKET { Array_Lit($2) }
  | LPAREN expr COMMA expr RPAREN { Tuple($2, $4)}
  | ID               { Id($1) }
  | expr PLUS   expr { Binop($1, Add,   $3) }
  | expr MINUS  expr { Binop($1, Sub,   $3) }
```

```
  | expr TIMES  expr { Binop($1, Mult,  $3) }
  | expr DIVIDE expr { Binop($1, Div,   $3) }
  | expr MOD    expr { Binop($1, Mod, $3) }
  | expr EQ     expr { Binop($1, Equal, $3) }
  | expr NEQ    expr { Binop($1, Neq,   $3) }
  | expr LT     expr { Binop($1, Less,  $3) }
  | expr LEQ    expr { Binop($1, Leq,   $3) }
  | expr GT     expr { Binop($1, Greater,  $3) }
  | expr GEQ    expr { Binop($1, Geq,   $3) }
  | expr AND    expr             { Binop($1, And, $3) }
  | expr OR     expr             { Binop($1, Or, $3) }
  | MINUS expr %prec NEG         { Unop($2, Neg) }
  | NOT expr                     { Unop($2, Not) }
  | ID LPAREN actuals_opt RPAREN { Call($1, $3) }
  | ID AT int_expr{ Access($1, $3) }
  | LPAREN expr RPAREN { $2 }

int_expr:
  ID               { Id($1) }
  | INT_LIT        { Int_Lit($1) }

expr_list:
  /* nothing */ { [] }
  | expr { [$1] }
  | expr COMMA expr_list {$1 :: $3}

literal:
    BOOL_LIT   { Bool_Lit($1) }
  | INT_LIT    { Int_Lit($1) }
  | STRING_LIT { String_Lit($1) }

actuals_opt:
    /* nothing */ { [] }
  | actuals_list  { List.rev $1 }

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

README.md

```
# corgi

## Instructions for Compiling and Running a corgi

    ./corgify.sh examples/hello_world.corg

It's that easy!
```

scanner.mll

```
    { open Parser
      exception ParsingError of string }


    (* regular definitions *)

    let char_lit = ['a'-'z' 'A'-'Z']?
    let int_lit = ['0'-'9']+
    let string_lit = '\"' ([^'\"']* as lxm) '\"'
    (*let frac_lit = '$'(int_lit '/' int_lit | int_lit)'$'*)
    let id = ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']*
    (*let rhythm = '['((int_lit ',' )* int_lit)? ']'
    let pd_tuple = '(' int_lit ',' frac_lit ')'
    let chord = '[' ((pd_tuple ',' )*pd_tuple)? ']'
    let track = '[' ((chord ',' )*chord)? ']'
    let composition = '[' ((track ',' )*track)? ']'*)
    (*let array_content = (char_lit | int_lit | string_lit | frac_lit | id
    let array_lit = '['((array_content ',' )* array_content)? ']'*)

    rule token = parse
      [' ' '\t' '\r' '\n'] { token lexbuf } (* Whitespace *)
    | "//"     { comment lexbuf }           (* Single-line comments *)
    | "/*"     { comments lexbuf }          (* Multi-line comments *)
    | '('      { LPAREN }                   (* Punctuation *)
    | ')'      { RPAREN }
    | '{'      { LBRACE }
    | '}'      { RBRACE }
    | '['      { LBRACKET }
    | ']'      { RBRACKET }
    | ';'      { SEMI }
    | ','      { COMMA }
    | '+'      { PLUS }
    | '-'      { MINUS }
    | '*'      { TIMES }
```

```
| '/'      { DIVIDE }
| '='      { ASSIGN }
| "=>"     { ARRAY_ASSIGN }
| "!"      { NOT }
| "&&"     { AND }
| "||"     { OR }
| "=="     { EQ }
| "!="     { NEQ }
| '<'      { LT }
| "<="     { LEQ }
| '>'      { GT }
| ">="     { GEQ }
| '$'      { DOLLAR }
| '@'      { AT }
| "%"      { MOD }

| "if"     { IF }                        (* Keywords *)
| "elif"   { ELIF }
| "else"   { ELSE }
| "for"    { FOR }
| "while"  { WHILE }
| "return" { RETURN }
| "null"   { NULL }


| "int"    { INT }                       (* Types *)
(*| "char"   { CHAR }*)
| "bool"   { BOOL }
| "string" { STRING }
| "frac"   { FRAC }
| "pitch"  { PITCH }
| "rhythm" { RHYTHM }
| "duration" { DURATION }
| "chord"    { CHORD }
| "track"    { TRACK }
| "composition" { COMPOSITION }


| ("true"|"false") as lit { BOOL_LIT(bool_of_string lit) }
| int_lit as lit { INT_LIT(int_of_string lit) }
| '\"' ([^'\"']* as lit) '\"' { STRING_LIT(lit) }
(*| frac_lit as lit { FRAC_LIT(lit) }*)
| id as lit { ID(lit) }
(*| '[' ((array_content ',' )* array_content)? as lit ']' {ARRAY_LIT(lit)}*)
| eof { EOF }
| _ as char { raise (Failure("illegal character " ^ Char.escaped char)) }
```

```
    and comment = parse
      "\n" { token lexbuf }
    | eof  { EOF }
    | _ { comment lexbuf }

    and comments = parse
      "*/" { token lexbuf }
    | eof { raise (ParsingError("unterminated comment"))}
    | _    { comments lexbuf }
```

table.ml

```
open Ast

module StrMap = Map.Make(String)

let env_table (table,_) = table
let env_scope (_,scope) = scope
let type_of_funct_args (_,_,p_type) = p_type

let parent_scope = Array.make 1000 0

let rec map_to_list_env func lst env =
    match lst with
          [] -> env
        | head :: tail ->
            let new_env = func head env in
                map_to_list_env func tail new_env

(*   need to rewrite *)
let string_of_table env =
    let symlist = StrMap.fold
        (fun s t prefix -> (string_of_decl t) :: prefix) (fst env) [] in
    let sorted = List.sort Pervasives.compare symlist in
    String.concat "\n" sorted

(* ____________________________________________ *)

let name_scope_str (name:string) env =
    name ^ "_" ^ (string_of_int (env_scope env))

let rec get_scope name env =
    if StrMap.mem (name_scope_str name env) (fst env) then (snd env)
    else if (snd env) = 0 then raise(Failure("symbol " ^ name ^ " not
declared."))
```

```
    else get_scope name (fst env, parent_scope.(snd env))

let rec get_decl name env =
    let key = name_scope_str name env in
    if StrMap.mem key (fst env) then StrMap.find key (fst env)
    else
        if (snd env) = 0 then raise (Failure("symbol " ^ name ^ " not declared
in current scope"))
        else get_decl name ((fst env), parent_scope.(snd env))

let add_symbol (name:string) (decl:decl) env =
    let key = name_scope_str name env in
    if StrMap.mem key (env_table env)
    then raise(Failure("symbol " ^ name ^ " declared twice in same scope"))
    else ((StrMap.add key decl (env_table env)), (env_scope env))


let add_var var env =
    let (name, isAr, p_type) = var in
    let is_implicit_array =
        (match p_type with
          (Chord_Type | Track_Type | Composition_Type | Rhythm_Type) -> true
          | _ -> false) in

    add_symbol name (Var_Decl(name, (isAr || is_implicit_array), p_type,
(env_scope env))) env

let rec add_stmt stmt env =
    match stmt with
      Block(block) -> add_block block env
      | If(expr, block1, block2) ->
            let env = add_block block1 env in add_block block2 env
      | For(expr1, expr2, expr3, block) -> add_block block env
      | While(expr, block) -> add_block block env
      | _ -> env

and add_block block env =
    let (table, scope) = env in
    let id = block.block_id in
    let env = map_to_list_env add_var block.locals (table, id) in
    let env = map_to_list_env add_stmt block.statements env in
    parent_scope.(id) <- scope;
    ((env_table env), scope)

and add_func func env =
    let (table, scope) = env in
    let arg_names = List.map type_of_funct_args func.formals in
```

```
    let env = add_symbol func.fname (Func_Decl(func.fname, func.ret_type,
arg_names, scope)) env in
    let env = map_to_list_env add_var func.formals ((env_table env),
func.fblock.block_id) in
    add_block func.fblock ((env_table env), scope)

let base_env =
    let table = StrMap.add "print_0" (Func_Decl("print", Int_Type, [], 0))
StrMap.empty in
    let table = StrMap.add "import_0" (Func_Decl("import", Composition_Type,
[String_Type], 0)) table in
    let table = StrMap.add "export_0" (Func_Decl("export", Int_Type,
[Composition_Type; String_Type], 0)) table in
    let table = StrMap.add "play_0" (Func_Decl("play", Int_Type,
[Composition_Type], 0)) table in
    let table = StrMap.add "length_0" (Func_Decl("length", Int_Type, [], 0))
table in
    (table, 0)

let build_table p =
    let (vars, funcs) = p in
    let env = base_env in
    let env =  map_to_list_env add_var vars env  in
    let env = map_to_list_env add_func funcs env in
    env
```

populatetests.sh

```
#!/bin/bash
if ! [ -e interpreter ]
    then make all
fi

tests=$(find tests -name *\.corgi)
had_failures="0"
ast_suffix=".astout"
sym_suffix=".symout"
sem_suffix=".semout"
intermed_suffix=".java"

ast_outdir="astout"
sym_outdir="symout"
sem_outdir="semout"
intermed_outdir="intermedout"
final_outdir="finalout"
```

```
# Remove all previous test results
rm -rf tests/$ast_outdir/*
rm -rf tests/$sym_outdir/*
rm -rf tests/$intermed_outdir/*
rm -rf tests/$final_outdir/*


get_test_name () {
    local fullpath=$1
    testpath="${fullpath%.*}"
    test_name="${testpath##*/}"
}

# Testing AST
for file in $tests
do
    get_test_name "$file"
    ./interpreter -ast < "$file" 2> "tests/$ast_outdir/$test_name$ast_suffix"
    ./interpreter -sym < "$file" 2> "tests/$sym_outdir/$test_name$sym_suffix"
    ./interpreter -sem < "$file" 2> "tests/$sem_outdir/$test_name$sem_suffix"
    ./interpreter -javagen < "$file" 2>
"tests/$intermed_outdir/$test_name$intermed_suffix"
done

# Populate the java output tests
# ./populatejavatests.sh


echo "Tests are populated"

make clean
exit $had_failures
```

makejava.sh

```
cd javaclasses
javac Frac.java
javac Duration.java
javac Pitch.java
javac Rational.java
javac Rhythm.java
javac Track.java
javac Pitch_Duration_Tuple.java
javac Composition.java
javac Chord.java
javac -cp ./jfugue-4.0.3-with-musicxml.jar:./ Utils.java
cd ..
```

checkjavac.sh

```
tests=$(find tests/intermedout -name *\.java)

get_test_name () {
    local fullpath=$1
    testpath="${fullpath%.*}"
    test_name="${testpath##*/}"
}

for file in $tests
do
    get_test_name "$file"

    cp $file javaclasses/Intermediate.java

    cd javaclasses
    javac Intermediate.java 2> ../tests/javacresults/$test_name.txt

    rm Intermediate.java

    cd ..

done
```

runtests.sh

```
#!/bin/bash
if ! [ -e interpreter ]
```

```
    then make all
fi

tests=$(find tests -name *\.corgi)
had_failures="0"
ast_suffix=".astout"
sym_suffix=".symout"
sem_suffix=".semout"
intermed_suffix=".java"

ast_outdir="astout"
sym_outdir="symout"
sem_outdir="semout"
intermed_outdir="intermedout"
final_outdir="finalout"

get_test_name () {
    local fullpath=$1
    testpath="${fullpath%.*}"
    test_name="${testpath##*/}"
}

# Testing AST
echo ""
echo "----------------Testing Abstract Syntax Tree Output----------------"
echo ""

for file in $tests
do
    get_test_name "$file"
    ./interpreter -ast < "$file" 2> ".test_out"
    if [[ ! $(diff ".test_out" "tests/$ast_outdir/$test_name$ast_suffix") ]]
    then
        echo "success: $test_name"
    else
        echo "FAIL:    $test_name"
        had_failures="1"

        printf "Expected: {\n"
        cat "tests/$ast_outdir/$test_name$ast_suffix"
        printf "}\n"
        echo

        printf "Recieved: {\n"
        cat ".test_out"
        printf "}\n"
        echo
```

```
    fi
done

# Testing Symbol Tables
echo ""
echo "----------------Testing Symbol Table Output----------------"
echo ""
for file in $tests
do
    get_test_name "$file"
    ./interpreter -sym < "$file" 2> ".test_out"
    if [[ ! $(diff ".test_out" "tests/$sym_outdir/$test_name$sym_suffix") ]]
    then
        echo "success: $test_name"
    else
        echo "FAIL:    $test_name"
        had_failures="1"

        printf "Expected: {\n"
        cat "tests/$sym_outdir/$test_name$sym_suffix"
        printf "}\n"
        echo

        printf "Recieved: {\n"
        cat ".test_out"
        printf "}\n"
        echo
    fi
done


# Testing Symbol Tables
echo ""
echo "----------------Testing Semantic Checking Output----------------"
echo ""
for file in $tests
do
    get_test_name "$file"
    ./interpreter -sem < "$file" 2> ".test_out"
    if [[ ! $(diff ".test_out" "tests/$sem_outdir/$test_name$sem_suffix") ]]
    then
        echo "success: $test_name"
    else
        echo "FAIL:    $test_name"
        had_failures="1"

        printf "Expected: {\n"
```

```
        cat "tests/$sem_outdir/$test_name$sem_suffix"
        printf "}\n"
        echo

        printf "Recieved: {\n"
        cat ".test_out"
        printf "}\n"
        echo
    fi
done


# Testing Final Output
: '
echo ""
echo "----------------Testing Final Output----------------"
echo ""
for file in $tests
do
    get_test_name "$file"
    ./interpreter -javagen < "$file" 2> ".test_out"

    cp .test_out javaclasses/Intermediate.java

    cd javaclasses
    javac Intermediate.java
    java Intermediate > ../.test_out

    rm Intermediate.java
    rm Intermediate.class

    cd ..

    if [[ ! $(diff ".test_out" "tests/$final_outdir/$test_name.txt") ]]
    then
        echo "success: $test_name"
    else
        echo "FAIL:    $test_name"
        had_failures="1"

        printf "Expected: {\n"
        cat "tests/$final_outdir/$test_name.txt"
        printf "}\n"
        echo

        printf "Recieved: {\n"
        cat ".test_out"
```

```
        printf "}\n"
        echo
    fi
done
'


echo ""
echo "----------------Finished Testing, Running Make Clean----------------"
echo ""


rm -f ".test_out"

make clean

exit $had_failures
```

corgify.sh

```
#!/bin/bash

# Make if not made yet
if ! [ -e interpreter ]
    then make all
fi

./interpreter -javagen < $1 2> javaclasses/Intermediate.java
# shift
cd javaclasses

echo ""
echo ""

javac Intermediate.java

java Intermediate

rm Intermediate.class
rm Intermediate.java

cd ..
```

Examples

hello_word.corgi

```
int main() {
    print("Hello, world!");
}
```

fib_music.corgi

```
/*
 *  Function that returns the n'th fibonacci number
 */
int fib(int n) {
    int sum;
    int i;
    if (n == 1) {
        return 1;
    }
    if (n == 2) {
        return 1;
    }
    sum = 1;

    for (i=2; i<n; i=i+1) {
        sum = sum + i;
    }
    return sum;
}

/*
 *  Function that uses the fibonacci number sequence to generate melodies
 */
int main() {

    // Variable declarations
    int i;
    chord tempChord;

    int fibNum;
    pitch p;
    duration d;
```

```
    track cumulativeTrack;
    track helperTrack;

    composition finalComposition;

    // Use a constant quarter note as the duration
    d = $1/4$;
    // Use a starting pitch of 60
    p = 60;

    tempChord = [(p,d)];
    cumulativeTrack = [tempChord];

    for (i=1; i<30; i=i+1) {
        fibNum = fib(i);

        // Keep it between 60 and 70
        fibNum = fibNum % 10 + 60;
        p = fibNum;
        tempChord = [(p, d)];

        helperTrack = [tempChord];
        print(helperTrack);

        // Add the helper track to the cumulative
        cumulativeTrack + helperTrack;
    }

    // intialize the final composition
    finalComposition = [cumulativeTrack];

    play(finalComposition);
    export(finalComposition, "fib_sequence.mid");

}
```

search_music.corgi

```
int main() {

    // Declaring variables
    composition compositionAnalysis;
    int index;
    int index2;
    chord interestingChord;
    chord tempChord;
    track interestingTrack;
    pitch c5;
    pitch g5;
    track trackHelper;
    duration quarterNote;
    int count;

    // Set constants
    c5 = 60;
    g5 = 67;
    quarterNote = $1/4$;
    interestingChord = [(c5, quarterNote)];
    tempChord = interestingChord;
    // interestingTrack = [interestingChord];

    // Import composition analysis
    compositionAnalysis = import("result.mid");

    count = 0;
    // Iterate through the composition and check
    for (index = 0; index < length(compositionAnalysis); index=index+1) {
        trackHelper = compositionAnalysis @ index;

        for (index2 = 0; index2 < length(trackHelper); index2=index2+1) {
            tempChord = trackHelper @ index2;
            if (interestingChord == tempChord) {
                count = count + 1;
            }
        }
    }

    print("There are ", count, " interesting chords in this composition!");
}
```