MANDALA
Geometric Design Language

EDO ROTH enr2116
HARSHA VEMURI hv2169
KANIKA VERMA kv2253
SAMANTHA WIENER srw2168

December 22, 2015
Contents

1 Introduction 1
  1.1 Background ................................................. 1
  1.2 Related Work ............................................ 1
  1.3 Goals ...................................................... 2
    1.3.1 Intuitive ........................................... 2
    1.3.2 Object-Oriented ................................. 2
    1.3.3 Portable ......................................... 2
    1.3.4 Robust .......................................... 2

2 Language Tutorial 3
  2.1 A Simple Example .................................. 3
  2.2 Shapes .................................................. 3
  2.3 Layers ................................................... 4
  2.4 Variables .............................................. 5
  2.5 Functions ............................................. 5
  2.6 A Full Program ...................................... 5

3 Language Reference Manual 7
  3.1 Lexical Conventions ............................. 7
    3.1.1 Tokens ........................................... 7
    3.1.2 Comments ....................................... 7
    3.1.3 Identifiers .................................. 7
    3.1.4 Keywords ..................................... 7
    3.1.5 Punctuation .................................. 9
    3.1.6 Constants .................................... 9
  3.2 Syntax Notation .................................. 9
    3.2.1 Program Structure ......................... 9
    3.2.2 Functions .................................... 9
    3.2.3 Assignment .................................. 10
    3.2.4 Statements ................................ 10
  3.3 Types .................................................. 10
3.3.1 Custom Types .......................................................... 10
3.3.2 Primitive Types ....................................................... 12
3.3.3 Type Conversion ...................................................... 12
3.4 Built-in Functions ....................................................... 13
  3.4.1 addTo ................................................................. 13
  3.4.2 draw ................................................................. 13
3.5 Expressions ............................................................... 13
  3.5.1 Literals ............................................................... 13
  3.5.2 Primary Expressions ............................................. 13
  3.5.3 Arithmetic Operators .......................................... 14
  3.5.4 Assignment Operators ........................................ 14
  3.5.5 Comma Operators .............................................. 14
  3.5.6 Constant Expressions .......................................... 14
  3.5.7 Operator Precedence .......................................... 14
3.6 Declarations ............................................................ 15
  3.6.1 Function Declarations ......................................... 15
  3.6.2 Variable Declarations ....................................... 15
3.7 Scoping ................................................................. 16
  3.7.1 Function scoping .............................................. 16
  3.7.2 Foreach loop scoping ......................................... 16
4 Project Plan ............................................................... 17
  4.1 Planning ............................................................... 17
  4.2 Specification ........................................................ 17
  4.3 Development ........................................................ 17
  4.4 Testing ............................................................... 17
  4.5 Programming Style Guide ....................................... 18
    4.5.1 Introduction .................................................. 18
    4.5.2 General Principles ......................................... 18
    4.5.3 Tabs ............................................................ 18
    4.5.4 Variables ..................................................... 18
    4.5.5 Comments .................................................... 18
  4.6 Roles ................................................................. 18
  4.7 Timeline ............................................................. 19
  4.8 Development Environment ...................................... 19
  4.9 Project Log ........................................................ 20
5 Architectural Design ................................................... 52
  5.1 Compiler Architecture Diagram ................................ 52
  5.2 Components ........................................................ 53
    5.2.1 Preprocessor .................................................. 53
    5.2.2 Scanner ........................................................ 53
Chapter 1

Introduction

The Mandala programming language is designed to allow developers to efficiently prototype, visualize, and discover new design patterns hereafter referred to as Mandalas. Using the Mandala language, it is possible to specify a pattern of sequence of patterns, in order to seamlessly place these abstract models into a visual representation.

Mandala is designed to be simple, intuitive, flexible, and concise. The input of the language closely resembles that of the Python programming language syntactically. The easy-to-learn syntax was created in an effort to reduce implementation errors. Additionally, parts of the syntax of the Mandala language were designed to be similar to the creation of objects in JavaScript in order to be easily understood. The output of the translator is Java code, which is coupled with a Java library to produce a native binary. The semantic features of the Mandala language resemble those of common modern imperative languages. This combination of features makes the Mandala language a strong bridge between modeling and implementation.

1.1 Background

In its most basic representation, a Mandala is a circular, symmetric figure that is composed of a variety of shapes and patterns. The Mandala is considered a spiritual symbol in some religions, and it is thought to represent the universe. Often exhibiting radial balance, Mandalas are used to focus attention during meditation, as a spiritual guidance tool, and to establish a sacred space. The Mandala programming language considers the visual aspects of the Mandala in order to enable developers to easily create such figures.

1.2 Related Work

While there are a number of full-fledged animation and graphics packages and libraries available to use with modern programming languages, many of these are generic enough
such that it would be difficult for a developer to easily relate a Mandala figure to its syntactic and semantic representation. Moreover, many of these libraries are focused on graphical user interfaces and simulations. The Mandala programming language aims to abandon the complexity of graphical packages and bridge the gap between formalism and detailed design. Using common graphics packages as a reference, we reap the benefits of an intuitive and easy-to-use language, without giving up the ability of more intricate frameworks to construct a more exciting image.

1.3 Goals

Mandala is an intuitive, object-oriented, portable, and robust language that can display Mandala figures accurately and efficiently while reducing implementation errors.

1.3.1 Intuitive

The primary goal in the design of this language was to make it easy to learn and use. The developer’s key concern should be in imagining a creative design for the Mandala figure, rather than concentrating on the syntax and semantics of the language. The Mandala language is consistent and intuitive, enabling users to focus on the design patterns themselves.

1.3.2 Object-Oriented

While Mandala is not an entirely object-oriented language like Java, it retains fundamental aspects of that design paradigm. The concept of objects are supported in the sense that the Mandala, each layer that is part of the Mandala, and each shape that is part of a layer are considered to be components that carry their own attributes. Because the language breaks down each figure into these components, users should find Mandala to be intuitive to use.

1.3.3 Portable

By virtue of Mandala code taken as input and converted to Java source code, Mandala is able to attain Java’s portability. The Java source code can be seamlessly integrated with larger Java projects and compiled with any Java compiler. Because Java code is the target platform for Mandala, the Mandala language is as portable as the existence of the JVM.

1.3.4 Robust

The simplicity of the Mandala language significantly decreases the amount of time required to design and generate a particular Mandala design. Mandala’s simple syntax and intuitive semantics ensures that most errors are detected at compilation, therefore making certain that compiled Mandala code behaves precisely as intended by the user.
Chapter 2

Language Tutorial

Mandala uses a clean syntax that is similar to Python in the usage of white space as delimitation and a lack of semi-colons. There are only two requirements for a Mandala program to operate: (1) Each program must create a Mandala, and (2) Each program must draw the Mandala.

2.1 A Simple Example

Here is a very simple program that illustrates the two requirements stated above.

```
Mandala m = Create Mandala
draw : (m)
```

The first line satisfies the first requirement and the second line satisfies the second requirement. The output of executing this program is the creating of a single dot in the center of the window. This center point is the most basic form of a Mandala figure.

2.2 Shapes

Shapes are the building blocks of the Mandala. The Mandala programming language supports three shapes: circles, squares, and triangles. A shape has four required attributes that must be specified upon creation of the shape.

- **Geo** The type of shape (circle, square, or triangle).
- **Size** The radius of a circle or the side length of a square or triangle.
- **Color** The color of the border of the shape.
- **Rotation** The degrees of in-place clockwise rotation of the shape about its center.

The following excerpt from a Mandala program illustrates shape creation.
2.3 Layers

While shapes are the building blocks of the Mandala, we also need a method to properly represent these shapes within the Mandala. To do so, we create layers, which are essentially hidden concentric circles around the center of the Mandala figure. Each layer is composed of any number of shapes, although only one "type" (or Geo) is allowed per layer. However, layers can be stacked (have the same size), which will allow the Mandala to appear as if there are multiple shape types per layer. A layer has five required attributes that must be specified upon creation of the layer.

Radius  The radius of the imaginary concentric circle around the center of the Mandala.

Shape  This is the specific shape variable that was created to be placed in this layer.

Count  This is the number of shapes we want to include in the layer.

Offset  The number of degrees by which we want to rotate the entire layer.

AngularShift  A boolean (0 or 1) that determines whether the shapes are each automatically rotated to point to the center of the figure, or whether they each remain in the same orientation.

Once a layer is created, it must be added to the Mandala in order to be displayed. This is done as follows:

addTo: (m, my_layer)
2.4 Variables

Mandala programs support variable declarations, with variable names using underscore rather than camel case by convention. Variables are defined as follows. Variable names must begin with a lowercase character.

```
Number x = 5.0
```

To reassign a variable to a new value, the type must be restated. For example, to change the value of the above Number variable, write:

```
Number x = 6.0
```

Variable declaration is supported by the following types: Mandala, Layer, Shape, and Number. Note that all Number type variables must be floats. The only exception is in assigning a Count to a layer, which must be an integer value.

2.5 Functions

The Mandala programming language supports functions. The function declaration syntax and semantics are Pythonic. Functions are declared using the Def keyword, and when defining a function, the user must state the return type, function name, and the arguments with their types. The arguments must be enclosed in parentheses, and the function signature must end with a colon. The body of the function should be indented. A function might Return a value as well for use throughout the program. This Return value must match the listed return type in the function signature. If there is no Return statement in the function body, the return type must be specified as Void.

```
Def Number myfunc (Number a, Number b):
    Return a + b
```

The above function takes two values of type Number, adds them together, and Returns the sum, which is also of type Number.

2.6 A Full Program

A complete Mandala program is shown below, combining the various fundamental and required aspects of the language as outlined above.

```
Mandala m = Create Mandala

Shape my_shape = Create Shape:
    Geo circle
    Size 10.0
```
Color blue
Rotation 0.0

Shape your_shape = Create Shape:
  Geo square
  Size 25.0
  Color red
  Rotation 45.0

Layer my_layer = Create Layer:
  Radius 50.0
  Shape my_shape
  Count 8
  Offset 0.0
  AngularShift 0

Layer your_layer = Create Layer:
  Radius 150.0
  Shape your_shape
  Count 8
  Offset 0.0
  AngularShift 0

addTo: (m, my_layer, your_layer)
draw: (m)

This program produces the following Mandala output.
Chapter 3

Language Reference Manual

3.1 Lexical Conventions

3.1.1 Tokens

Mandala breaks down into six classes of tokens: identifiers, keywords, constants, strings, operators and other separators. It uses indentations to group blocks of code. Spaces at the end of the line, other tabs, newlines and more generally "white space" are ignored except to separate tokens and at the beginning of the line to determine indentation.

3.1.2 Comments

Inline comments are indicated by # and extend to the end of the line. Any text following a # will be ignored by the compiler.

3.1.3 Identifiers

An identifier is a combination of letters, numbers and underscores. An identifier must begin with a letter. To distinguish between reserved keywords and variables, Mandala adopts the convention that all created variables must begin with lowercase letters, and that all reserved keywords (as listed below) must begin with capital letters.

3.1.4 Keywords

The following identifiers are keywords and may not be redefined for other purposes.

Foreach is used to define a loop that allows the user to iterate through a range of numbers.

To is a keyword used in Foreach statement to describe the range of the Foreach statement.
**Geo** is a keyword used when defining a Shape, and specifies whether the Shape is a circle, triangle or square. As with all attributes, the Geo keyword must be indented below a Shape construction to properly assign the attribute.

**Size** is an attribute of Shape that describes its scale. To maintain intuitiveness, Size defines the radius for circles, but the side length for triangles and squares.

**Color** is an attribute of Shape that will allow users to write a color and specify blue, red, green, yellow, orange, violet, indigo, teal, aqua and or specify the HEX color.

**Rotation** is an attribute of Shape that specifies the degrees of rotation in clockwise direction from zero degrees at the top of the circle.

**Radius** is an attribute of Layer that defines the distance from the center of the Mandala that all Shapes in the Layer will be placed on.

**Shape** is an attribute of Layer that describes the single Shape that belongs to a Layer. This Shape must be a previously defined Shape object.

**Count** is an attribute of Layer that describes the number of times a Shape is repeated in a given Layer. The shapes will be symmetrically placed around the center of the Mandala depending on this specified Count.

**Offset** is an attribute of Layer that characterizes the offset of a single layer. By default, the first shape is placed at the top of the layer at 12 o’clock. The offset moves the placement of the first shape clockwise the number of degrees specified.

**AngularShift** is an attribute of Layer that indicates the angle at which shapes are placed in the layer depending on where in the shape they are placed. When AngularShift is set to 0, the shapes are all placed at the same original angle no matter where in the layer they are. When AngularShift is set to 0, the shape is rotated along with its position in the layer, and the shapes are angled radially.

**Return** allows users to return entities of any defined type from their functions.

**Def** is used to indicate function declaration.

**Create** is a constructor keyword used when creating new Mandalas, Layers and Shapes. When the Create keyword is used, the attributes of the Mandala, Layer or Shape are assigned.

**Void** is used in function declaration to indicate that the function does not return any value. Functions declared as Void may still have functionality, such as calls to the draw() function.
3.1.5 Punctuation

: Colons are used to indicate the beginning of any kind of declaration or function call. They have three use cases: functions (both for declaration and calling), Create statements, and Foreach loops.

() Parentheses are used to enclose parameters both in function declarations and function calls.

, Commas are used to separate parameters in function declarations and function calls.

{} Braces are used in constructors for Mandala, Layer and Shape when they are being defined.

Arithmetic operators are defined later in the document.

3.1.6 Constants

**Boolean Constants:** 1 represents true and 0 represents false.

**Floating Constants:** floating point constants have an integer part, a decimal point, and a fractional part. They also have an optional ‘-’ sign in front to create negative values.

3.2 Syntax Notation

3.2.1 Program Structure

The user calls various functions to do different actions. To make, fill and draw a mandala, the user must first create a Mandala, then create Layers, which can be filled with Shapes that a user can create. The Layers then must be added to the Mandala using `addTo` and finally the user can draw their Mandala. Some of the main functions such as Create, addTo, and draw can be used to do different things based on the types they are called on. For example, Create can be used to create different things like a Mandala, or a Layer, or Shape based on what is specified and assign a name to the object that was created. See the sample programs in Appendix A of this manual for examples of this syntax.

3.2.2 Functions

**Function Definitions**

```
Def Return_type function_name(Type param1, Type param2, ...):
  function_body
```

**Function Calling**
Type_name var_name = function_name : ( param1, param2, ... )

3.2.3 Assignment

Assignment of typed variables is with the "=" operator. Correct types must be provided for each variable assignment, i.e.

<Type> <var_name> = <value>

Assignment of attributes is through adjacency. For example to assign a value to the count attribute in a Layer, a user uses "Count 8". Indentation is used to distinguish a hierarchy. In assignments, after a type like a Layer or a Shape is defined, attributes of those objects such as size or radius are assigned on indented lines within the section of the overall type.

3.2.4 Statements

Expression Statements

Whitespace after a line has no syntactic meaning in Mandala, so an expression statement ends with a newline character. If an expression needs to span more than one line, the continuation operator can be used at the end of the line.

Loop Statements

Foreach i = 1.0 To i = 5.0:
    # Loop contents here

Loops over a given range of numbers (1 to 5 in this example). Use indentation to specify the contents of the loop.

Return Statements

Functions can Return entities of a defined type. The type of the value returned must match the actual value of the Return type specified in function declaration. If the Return type is specified as Void, then no Return statement is needed.

3.3 Types

3.3.1 Custom Types

Mandala represents the entire design that will be created by the user. A new Mandala object must be instantiated with the call:

Mandala <name> = Create Mandala
where name can be any string. The value of name will then be used for all functionality pertaining to the Mandala object. There are two additional functions that may be used with a Mandala object – the built-in addTo function allows any created layers to be added to the design in the following way:

addTo: (Mandala, Layer1, Layer2, \ldots, LayerN)

Note that any layers that are never added to a Mandala will never be drawn - they stand alone in an abstract manner but not pictorially. Finally, any Mandala object can be drawn with the call:

This will bring up the display window, and show the complete creation represented by the Mandala object.

Layer represents an abstract circle upon which shapes can be placed. Like Mandala, a Layer is instantiated with the create constructor, but unlike Mandala, it has additional parameters that may be provided to specify additional properties. Syntax is of the form:

Layer <name> = Create Layer:
  Shape <Shape>
  Radius <Number>
  Count <Number>
  AngularShift <Boolean digit: 0 or 1>

Indentation indicates description of the given layer. These attributes must be defined in the given order, and they must include the attribute name correctly. All attribute definitions should be indented exactly once beneath the initial creation of the Layer. The only additional existing functionality of the Layer type is to be added to Mandala objects. As described above, the syntax is as follows:

addTo: (Mandala, Layer)

Shape is a type which represents various shapes (circles, triangles, and squares) that can be added to Layers and then drawn on Mandalas. Like the syntax for Layer, a Shape is created with an initial create constructor statement, and then provided parameters that must be indented below the initial statement. The attributes are Geo, Size, Color, and Rotation. Again, all attributes are required, and must be specified in the correct order with correct attribute name. Syntax looks as follows:

Shape <name> = Create Shape:
  Geo <Geo>
  Size <Number>
  Color <Color>
  Rotation <Number>
Create is the constructor for all of these custom types. The constructor creates a new instance of the type and takes parameters to fill in the various attributes of the type.

```plaintext
Type variable_name = Create Type:
  attributeType attributeValue
  attributeType attributeValue
  ... 

For example:
Shape my_shape = Create Shape:
  Geo circle
  Size 5
  ... 
```

### 3.3.2 Primitive Types

**Number** represents a floating point value, identical to the float type in C. The number range is from 1.2E-38 to 3.4E+38, and has 6 digits of precision. Numbers can be used when assigning other properties, but may also be declared on their own and assigned to variables. Examples:

```plaintext
Number x = 100.0 
Layer l = Create Layer:
  Radius 4.5
  Shape <Shape>
  Count 2
  Offset -1.25
  AngularShift 0
```

**Geo** can be one of either circle, square or triangle, and is used to define a Shape.

### 3.3.3 Type Conversion

There is no type conversion in Mandala. Where some languages differentiate between integers and floats for instance, Mandala just has one Number type, which is used for all numerical values. Any created variable must be created with a corresponding type, and it will remain that type for its entire existence during compilation and runtime. The exception to this rule is Count, which must be assigned as an integer. This logically follows from the fact that Count fundamentally represents an integer value, the number of times a shape appears in a layer.
3.4 Built-in Functions

3.4.1 addTo

addTo: (<Mandala>, <Layer>, <Layer>, ..., <Layer>)

Once a Layer is defined, in order to actually include Layer in the drawable Mandala, the addTo function must be used. The addTo function must have at least two arguments – the Mandala, and at least one layer to be added. These added layers now become a part of the Mandala. Once a layer has been added to a Mandala, it remains in that Mandala and will be drawn accordingly, regardless of whether the layer variable itself has gone out of scope.

3.4.2 draw

draw: (<Mandala>, <Mandala>, ...)

Draw is used to execute the program and actually draw the Mandala figure. Without this function call, the Mandala will exist as an abstract structure, but will never materialize on a user’s screen. Draw takes all layers and their corresponding shapes that have been added to the Mandala, and displays them to the user’s screen. Draw takes one or more Mandala arguments.

3.5 Expressions

3.5.1 Literals

Literals are floats and integers.

3.5.2 Primary Expressions

Identifiers
Identifiers are primary expressions.

Literals
Literals are primary expressions. They are described above.

Constant
A float constant is a primary expression.

(expression)
Parenthesized expressions are primary expressions. The type and value of a parenthesized expressions is the same as that of the expression without the delimiters. Parentheses allow expressions to be evaluated in a desired precedence. Parenthesized expressions are evaluated relative to each other starting with the expression that is most deeply nested.
3.5.3 Arithmetic Operators

expression * expression
The result is the product of the two expressions. The types of the expressions and the result must be Number.

expression / expression
The result is the quotient of the expressions, where the first expression is the dividend and the second is the divisor. The types of the expressions and the result must be Number.

expression + expression
The result is the sum of the expressions. The types of the expressions must be Number.

expression - expression
The result is the difference of the first and second expressions. The types of the expressions must be Number.

3.5.4 Assignment Operators

Assignment operators have left associativity.

lvalue = expression
The result is the assignment of the expression to the lvalue. The type of the expression is the same as that of the lvalue.

3.5.5 Comma Operators

expression, expression
A pair of expressions separated by a comma is evaluated left to right and the value of the left expression is discarded. The type and value of the result are the type and value of the right expression. This expression should be avoided in those situations wherein the comma operator has a different meaning, such as in function calls.

3.5.6 Constant Expressions

Syntactically, constant expressions are expressions restricted to a subset of operators. These are expressions that evaluate to a constant. Constant expressions may not contain assignments, function calls, or comma operators.

3.5.7 Operator Precedence

Primary expressions have left associativity. Unary operators have right associativity. Assignment operators have left associativity.

The precedence of operators is determined by the order of the sections in which they
are shown above (with the highest precedence operators at the top). Operators within a section have the same precedence.

3.6 Declarations

3.6.1 Function Declarations

Mandala supports user-defined functions that are defined using the keyword Def preceding each function definition. Arguments are given as a list, along with their corresponding types. The function signature ends with a colon and the body of the function is denoted via indentation. If a function declaration specifies a non-Void return type, it must contain a return statement that returns a value of corresponding type. If a function declaration specifies Void as its return type, any Return statement will be ignored.

```plaintext
Def return_type func_name (arg_type func_arg1, arg_type func_arg2):
    # function body
    Return <func_Return_value>
```

3.6.2 Variable Declarations

For the custom types in Mandala (Mandala, Layer, and Shape), the create keyword is used to instantiate variables. For Numbers, this is unnecessary. However, for all types, the type being created must be specified upon variable instantiation.

```plaintext
Number varName = <float>

Mandala m = Create Mandala

Shape <name> = Create Shape:
    Geo <Geo>
    Size <Number>
    Color <Color>
    Rotation <Number>

Layer <name> = Create Layer:
    Radius <Number>
    Shape <Shape>
    Count <Integer>
    Offset <Number>
```
3.7 Scoping

Mandala uses block scoping, which means that any variable defined within a given level of indentation is accessible only within that level and any deeper level of indentation. Note that any shapes and layers that are added to a mandala within a limited scope will still be drawn, but the variable names are no longer accessible once outside of the Layer’s indentation block.

3.7.1 Function scoping

Functions only have access to the parameters passed into the corresponding function call. The only value that will remain in scope after a function call is the return value, if applicable.

3.7.2 Foreach loop scoping

Unlike functions, foreach loops do have access to the variables declared before their call. However, any variables declared within that call will no longer be in scope once the loop has terminated.
Chapter 4

Project Plan

4.1 Planning
We began with an initial meeting to discuss team roles, programming guidelines, and to set times to meet each week. We scheduled additional meetings with our TA Prachi, who would help us gauge our progress and discuss any complications we were encountering. In each team meeting we would assign action items to complete before the next meeting. Luckily, we were able to develop in a fairly modular fashion and with constant communication, which allowed us to prevent bottlenecks and dependencies.

4.2 Specification

4.3 Development
The development process largely followed the workflow of compiler architecture. For example, we began with the preprocessor, which handled whitespace, comments, and added syntactic features such as semi-colons. Then we worked through the scanner and the parser, and then moved and to the semantic checking, the intermediate representations, and code generation. We ran into trouble at one point when trying to proceed directly from the SAST to code generation, but we fixed the problem by introducing an JAST (Java AST). Of course, throughout this process we returned to the earlier components to make small modifications and add new features.

4.4 Testing
Although we began fullstack testing towards the end of the timeline, we were able to test the intermediate elements individually via unit tests throughout the project development. Unit tests for the preprocessor, the AST, and the parser were written upon completion of these components. Once we started testing end-to-end, we generated larger programs and
analyzed both the visual output as well as the Java source code to catch any errors. A comparison script was written in Python to compare the output of the program to our pre-determined expected output, which allowed us to check whether the tests passed or failed. See the test plan section for more detail.

4.5 Programming Style Guide

4.5.1 Introduction

The purpose of this style guide is to provide basic guidelines for seamless collaborative code development. The standards contained in this style guide reflect the fundamental coding best practices agreed upon by the team members prior to development. In an effort to make the project codebase readable and maintainable, these guidelines should be followed as closely as possible during project development.

4.5.2 General Principles

Code should be easy to read. Whitespace should be used where appropriate and comments should be utilized heavily. Indentations should be consistent and variables names should clearly indicate their purpose. Java code should follow accepted Java coding conventions.

4.5.3 Tabs

Code should not contain tabs. Instead, use four spaces to indent. This is due to the fact that the team members use a variety of hardware and software to collaborate on the project. If a developer wishes to use the tab key, the key should be re-mapped to four spaces.

4.5.4 Variables

Variable names should use underscore rather than camelCase. Global variables should be avoided, but explained thoroughly if employed.

4.5.5 Comments

Comments should be used liberally. Each function should contain a comment that explains the purpose of the function, including inputs, types, and return values. Each file should contain a header that explains the overall purpose of the file.

4.6 Roles

Although we initially set team roles as shown below, we quickly realized that responsibilities were extremely fluid, with each person taking on the responsibilities of two or more
of the roles.

<table>
<thead>
<tr>
<th>Team Member</th>
<th>Role</th>
<th>Responsibilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kanika Verma</td>
<td>Project Manager</td>
<td>Back-End, Semantic Checking, Code Generation</td>
</tr>
<tr>
<td>Samantha Wiener</td>
<td>Language Guru</td>
<td>Front-End, Semantic Checking, Code Generation</td>
</tr>
<tr>
<td>Edo Roth</td>
<td>System Architect</td>
<td>Back-End, Code Generation, Testing</td>
</tr>
<tr>
<td>Harsha Vemuri</td>
<td>Tester</td>
<td>Front-End, Semantic Checking, Testing</td>
</tr>
</tbody>
</table>

### 4.7 Timeline

<table>
<thead>
<tr>
<th>Date</th>
<th>Milestone</th>
</tr>
</thead>
<tbody>
<tr>
<td>September 20</td>
<td>Broadly defined language</td>
</tr>
<tr>
<td>September 21</td>
<td>Project repository created, first commit</td>
</tr>
<tr>
<td>September 30</td>
<td>Language proposal completed</td>
</tr>
<tr>
<td>October 22</td>
<td>Preprocessor completed</td>
</tr>
<tr>
<td>October 25</td>
<td>Scanner completed</td>
</tr>
<tr>
<td>October 26</td>
<td>Language reference manual completed</td>
</tr>
<tr>
<td>November 11</td>
<td>Parser and AST completed, determined graphics package</td>
</tr>
<tr>
<td>November 15</td>
<td>SAST and Semantic checker completed</td>
</tr>
<tr>
<td>November 17</td>
<td>Began unit testing for components</td>
</tr>
<tr>
<td>November 18</td>
<td>Hello World</td>
</tr>
<tr>
<td>December 1</td>
<td>JAST and Code Generation completed</td>
</tr>
<tr>
<td>December 18</td>
<td>Finished updates to components, regression testing</td>
</tr>
<tr>
<td>December 20</td>
<td>Code completed, testing completed</td>
</tr>
<tr>
<td>December 21</td>
<td>Presentation and final submission</td>
</tr>
</tbody>
</table>

### 4.8 Development Environment

The following technologies were used.

- OCaml 4.02.1 with OCamlyacc and OCamllex extensions used for scanner and parser.
- Python 2.7.8 was used for the preprocessor and the comparison script.
- Java 7 was used for the target source code. The Java Turtle library was used for graphics.

The following environments were used.
• Sublime Text 2
• Vim
• Vagrant

We also used a Git repository hosted on Github for version control.

4.9 Project Log

This project log shows a history of 375 commits starting from September 21 and ending December 22.

commit 1410a65310e3eb41db7bee0627007e8b9b96454e423
Author: Harsha Vemuri <hv2169@columbia.edu>
Date: Tue Dec 22 23:07:02 2015 -0500
commit 26d6402cef591638fbb1fd4d2b85a8f82912f1a5
Author: samw7823 <samw7823@users.noreply.github.com>
Date: Tue Dec 22 21:53:50 2015 -0500
commit dbbb0a709c127e554050c1bb409ae11f7f05ecea
Author: samw7823 <srwiener@gmail.com>
Date: Tue Dec 22 21:53:10 2015 -0500
commit 42c8e656e403160ab9c3ebffdd0fbbf13f6747778
Author: samw7823 <srwiener@gmail.com>
Date: Tue Dec 22 20:56:32 2015 -0500
commit 9c1d2a335b63a0dbb0a7
Author: samw7823 <samw7823@users.noreply.github.com>
Date: Tue Dec 22 21:53:50 2015 -0500
commit 50ecae73c67a9d26deef8c696c267d19d9ca558f
Author: Harsha Vemuri <hv2169@columbia.edu>
Date: Tue Dec 22 19:41:08 2015 -0500
commit 638a8ce34fb766e6a679751297ec4565685c25a3
Merge: 06c93f3 50ecae7
Author: hvemuri <hv2169@columbia.edu>
commit 9e51af292abcc9cbc4fc65a1a197c209db9c636d
Merge: 17e8837 cf493a5
Author: edoroth <edoroth@gmail.com>
Date: Sat Nov 21 23:08:45 2015 +0000
commit cf493a537519d2b63512e2476e942331e17d0aaf
Author: edoroth <edoroth@gmail.com>
Date: Sat Nov 21 23:01:51 2015 +0000
commit 7aaf9db9a73518e85151195adecdde24daee7c6b
Merge: ba23ac5 1612874
Author: edoroth <edoroth@gmail.com>
Date: Sat Nov 21 22:57:37 2015 +0000
commit ba23ac5723b3d58165993e20798d3d2a8e283da5
Author: edoroth <edoroth@gmail.com>
Date: Sat Nov 21 22:57:15 2015 +0000
commit 161287493ffac0ffe1913357396a91d3b9baa4c5
Author: Harsha Vemuri <hv2169@columbia.edu>
Date: Sat Nov 21 17:46:12 2015 -0500
commit 68cd0e8081fb86d38a55a240eb84cf601bc92f6e
Merge: 9522c5a bd8ea1a8
Author: samw7823 <srwiener@gmail.com>
Date: Sat Nov 21 22:57:37 2015 +0000
commit bd8ea1a82f3f1147176b6b2239d84e534b4bf4a17
Author: Harsha Vemuri <hv2169@columbia.edu>
Date: Sat Nov 21 17:13:26 2015 -0500
commit 8e733562f12d8c3a2414f2d6447bc7919c7a6db7
Merge: 6a823a6 ea12f91
Author: Edo Roth <enr2116@columbia.edu>
Date: Sat Nov 21 17:21:38 2015 -0500
commit ea12f91ae98b82d2a9d8a89daee2bf4874e6c75
Author: samw7823 <srwiener@gmail.com>
Date: Sat Nov 21 21:19:33 2015 +0000
commit fab66a3 6a823a6
Author: edoroth <edoroth@gmail.com>
Date: Sat Nov 21 21:19:33 2015 +0000
commit fab66a3eb97dec78a10a78c21fa834fb34e218ca9
Merge: fab66a3 6a823a6
Author: edoroth <edoroth@gmail.com>
Date: Sat Nov 21 21:19:33 2015 +0000
commit 52c083085b09897a651291783d3648e23bff9f87
Author: samw7823 <srwiener@gmail.com>
Date: Tue Nov 17 12:16:59 2015 -0500
commit 6a823a6bab71c5513e5a078ece32f2bbbd2a993
Merge: 465785a 3b315f7
Author: samw7823 <samw7823@users.noreply.github.com>
commit 457b582961ab8f43c009ebe87964cb1555d1c61fe
Author: Harsha Vemuri <hv2169@columbia.edu>
Date: Sun Nov 15 02:15:02 2015 -0500

commit 8d16c9988b86a61d2afdf253e140c508dba6258ee8
Merge: d5aedf7 fe45cbe
Author: hvemuri <hv2169@columbia.edu>
Date: Sun Nov 15 01:29:06 2015 -0500

commit fe45cbe7cf4def90549dfc002cccad55e64fc839
Author: Harsha Vemuri <hv2169@columbia.edu>
Date: Sun Nov 15 01:25:13 2015 -0500

commit 2747c2a0caea8dfbac91f58bbafc581ffbcc988f
Author: edoroth <edoroth@gmail.com>
Date: Sun Nov 15 05:43:53 2015 +0000

commit 93c165c3cfff875010810b2eb352112a7d58498e5
Author: Kanika Verma <vermakaniaka@hotmail.com>
Date: Sun Nov 15 00:00:22 2015 -0500

commit 8e0cb935ef0dccc1b3f8d1411f44d737e81edca7d
Author: edoroth <edoroth@gmail.com>
Date: Sun Nov 15 04:09:38 2015 +0000

commit d5aedf7794a379c9faa4be5ded0e56e573e62ef
Author: edoroth <edoroth@gmail.com>
Date: Sun Nov 15 00:23:26 2015 +0000

commit 555d50a0e23b063947deb0022605870112a95de
Author: edoroth <edoroth@gmail.com>
Date: Sun Nov 15 00:21:59 2015 +0000

commit 8128199cc8d674ec26eab133656687fdd45cbe2c
Author: Harsha Vemuri <hv2169@columbia.edu>
Date: Sat Nov 14 18:46:08 2015 -0500

commit 7a210345da765e87cc832e397e9d115f6f69e7
Merge: eaeda0b 8bf66ac
Author: samw7823 <srwiener@gmail.com>
Date: Sat Nov 14 11:38:27 2015 -0500

commit eaeda0bdf9989ab99d90d362f0b2a7862dfb3
Author: samw7823 <srwiener@gmail.com>
Date: Sat Nov 14 11:37:43 2015 -0500

commit 8bf66acc80b83cb4df1338d199560f8425e8875b
Merge: 0f7254a de2ec1f
Author: Edo Roth <enr2116@columbia.edu>
Date: Sat Nov 14 00:21:51 2015 -0500

commit de2ec1f49841021512cd80d58250d3f05f321f44
Author: edoroth <edoroth@gmail.com>
commit 7e4bb2456cea548616c1dfc22b37b08353f2bd48
Merge: 4e02b81 e2d9c8a
Author: Edo Roth <enr2116@columbia.edu>
Date: Tue Nov 10 12:47:19 2015 -0500
commit e2d9c8a684b3d52730772b6b0b55a2a8e1901d61
Author: edoroth <edoroth@gmail.com>
Date: Tue Nov 10 17:32:36 2015 +0000
commit 4b0b71f9f9d9c2f92cfc41578a59a65f3716c64a13
Merge: 2688676 4e02b81
Author: edoroth <edoroth@gmail.com>
Date: Tue Nov 10 16:05:54 2015 +0000
commit 4e02b819d7053eabd27c522802623a7c8760db65
Merge: 15da623 c32f6f3
Author: hvemuri <hv2169@columbia.edu>
Date: Mon Nov 9 15:46:00 2015 -0500
commit c32f6f3177bc5d1efef4f1b3beb1d63e0374b2541
Author: Harsha Vemuri <hv2169@columbia.edu>
Date: Mon Nov 9 15:45:35 2015 -0500
commit 15da623650e7fc7252e2f2b61f10f3e9b7bbde96f
Author: Harsha Vemuri <hv2169@columbia.edu>
Date: Sun Nov 8 21:20:14 2015 -0500
commit be519f0271c6f8df5955cd98e7c7b07743a8b341
Author: Harsha Vemuri <hv2169@columbia.edu>
Date: Sun Nov 8 21:01:10 2015 -0500
commit 84954b911ebdc738718253ace21d6ef3c09d208
Merge: ed3d0da 5a552b4
Author: hvemuri <hv2169@columbia.edu>
Date: Sun Nov 8 20:29:52 2015 -0500
commit ed3d0daa4f10079oc42ef68b4a63bcdf848d9da
Merge: edff14 41cbaa7
Author: hvemuri <hv2169@columbia.edu>
Date: Sun Nov 8 20:29:13 2015 -0500
commit 41cbaa711f7a1d711b7f74905033f393988f182
Author: Harsha Vemuri <hv2169@columbia.edu>
Date: Sun Nov 8 19:29:40 2015 -0500
commit 3503360c1583bb74d61c1ee6fb97bf71410fa01
Author: Harsha Vemuri <hv2169@columbia.edu>
Date: Sun Nov 8 17:17:38 2015 -0500
commit 5a552b4c900766b6177c49d6b1f099e40757205e
Merge: db3767 edff14
Author: samw7823 <srwiener@gmail.com>
commit b3a50cf947761e12d641fea83e7a5ccd8e2053de
Merge: 4426954 ec0ff4
Author: samw7823 <srwiener@gmail.com>
Date: Fri Oct 23 14:56:49 2015 -0400
commit 44269542cdf4bb04408adea0c41ee06fc810ab4e
Author: samw7823 <srwiener@gmail.com>
Date: Fri Oct 23 14:56:35 2015 -0400
commit ec0ff4f0382f2478eb5d94c839fd3bae9e94059
Author: Harsha Vemuri <hv2169@columbia.edu>
Date: Fri Oct 23 14:23:13 2015 -0400
commit 2911e6a8f7d98d39f749b0b46301a908d10d6d2a
Author: Harsha Vemuri <hv2169@columbia.edu>
Date: Thu Oct 22 16:24:18 2015 -0400
commit 07839659931cee049972cefbf88bf60c2e900f2f
Author: Harsha Vemuri <hv2169@columbia.edu>
Date: Thu Oct 22 16:06:00 2015 -0400
commit c8568b024e6fed5308dd1ee36ac659ce02721a89
Author: Harsha Vemuri <hv2169@columbia.edu>
Date: Wed Oct 21 21:02:13 2015 -0400
commit efff97feaffa88c2a474bdcb49275823506dcee3
Author: Harsha Vemuri <hv2169@columbia.edu>
Date: Wed Oct 21 20:49:16 2015 -0400
commit 5632659e75b67658325408a85f7a7ddd450f3808
Author: Harsha Vemuri <hv2169@columbia.edu>
Date: Wed Oct 21 16:20:57 2015 -0400
commit 16b527927dc50681573d015ef0ed685f2e699d0
Author: Harsha Vemuri <hv2169@columbia.edu>
commit 94dc21cc16117b2bf3756c815480239bb85f6ed6
Author: Harsha Vemuri <hv2169@columbia.edu>
Date: Wed Oct 21 01:15:34 2015 -0400
commit 47e7f00a4e9801f75d58cb3a5c64cf9cc8e79a88
Merge: dea149e 2bf2cd7
Author: Harsha Vemuri <hv2169@columbia.edu>
Date: Tue Oct 20 21:20:15 2015 -0400
Chapter 5

Architectural Design

5.1 Compiler Architecture Diagram
5.2 Components

5.2.1 Preprocessor

Implemented by: Harsha

preprocessor.py

A preprocessor was written in Python to take the *.mandala source code and convert it into a form that could be lexically analyzed and parsed. The preprocessor reads the source code line by line and replaces whitespace delimiters with braces, removes comments and extra whitespace, and inserts semi-colons at the end of statements. The preprocessor also checks for characters that are invalid in the Mandala programming language and throws an error since these would break the scanner. The preprocessor produces an intermediate output file designated with the *.mandala.proc extension.

5.2.2 Scanner

Implemented by: Edo, Harsha

scanner.mll

The scanner was written in OCamllex. It takes the intermediate preprocessed file from the preprocessor and tokenizes it into keywords, identifiers, operators, and values. It removes any extraneous whitespace not already removed by the preprocessor. The scanner throws an error if it encounters a character that cannot be lexed. The tokens produced by the scanner are used by the parser to create an abstract syntax tree.

5.2.3 Parser and AST

Implemented by: Edo, Sam

parser.mly, ast.mli

The parser was written in OCamllyacc. It takes the tokens generated by the scanner and uses the grammar and the data types to generated an abstract syntax tree. The grammar is defined using productions and rules. Code that successfully passes through the parser is syntactically correct.
5.2.4 Semantic Checker and Intermediate Generator

Implemented by: Edo, Harsha, Kanika, Sam

`semantic.ml`, `sast.mli`, `sast_to_jast.ml`, `jast.mli`

The semantic checker traverses the AST and converts it into an extended abstract syntax tree that includes a semantic check called the SAST. Semantic.ml checks that all types are matching and everything is semantically correct. The SAST enables the compiler to keep track of objects rather than identifiers and variables. The SAST-to-JAST converts the SAST to several intermediate representations that allows the JAST to create an abstract syntax tree for Java code.

5.2.5 Code Generation

Implemented by: Edo, Kanika, Sam

`jast.mli`, `gen_java.ml`

The code generator traverses the Java abstract syntax tree (JAST) to generate Java code by analyzing the objects and variables defined by the JAST. The Java code that is generated is ready to compile using a Java compiler such as `javac`.

5.2.6 Java Library

Implemented by: Edo, Harsha

`Turtle.java`

Because it would be impractical to generate pure Java code that could implement all the graphics features we wanted, we used an external Java library called Turtle. This library made it possible to generate Java code that supported the various features that we wished to include in our programs.
Chapter 6

Test Plan

6.1 Source to Target Sample

Source Program:

```plaintext
Mandala m = Create Mandala

Shape s = Create Shape:
  Geo circle
  Size 50.0
  Color red
  Rotation 0.0

Shape s2 = Create Shape:
  Geo circle
  Size 75.0
  Color blue
  Rotation 0.0

Shape s3 = Create Shape:
  Geo circle
  Size 100.0
  Color green
  Rotation 0.0

Layer l = Create Layer:
  Radius 0.0
  Shape s
  Count 1
  Offset 0.0
```
AngularShift 0

Layer l2 = Create Layer:
    Radius 0.0
    Shape s2
    Count 1
    Offset 0.0
    AngularShift 0

Layer l3 = Create Layer:
    Radius 0.0
    Shape s3
    Count 1
    Offset 0.0
    AngularShift 0

addTo: (m, l, l2, l3)
draw: (m)

Target Result:

```java
public class Program{

    public static void drawCircle(Turtle t, double radius, double x, double y, String color) {
        t.penColor(color);
        t.up(); t.setPosition(x , y + radius); t.down();
        for (int i = 0; i < 360; i++) {
            t.forward(radius * 2 * Math.PI / 360);
            t.right(1);
        }
    }

    public static void drawSquare(Turtle t, double size, double x, double y, double rotation, String color) {
        t.penColor(color);
        t.up();
        t.setPosition(x - size/2, y + size/2);
        rotation = rotation % 90;
        double radius = Math.sqrt(2) * size / 2;
        if (rotation > 0 ) t.left(45);
        for (int i = 0; i < rotation; i++) {
```
```java
public static void drawTriangle(Turtle t, double size, double x, double y, double rotation, String color) {
    t.penColor(color);
    t.up(); t.setPosition(x - size / 2, y + Math.sqrt(3) * size / 6);
    rotation = rotation % 120;
    double radius = size / Math.sqrt(3);
    if (rotation > 0) t.left(60);
    for (int i = 0; i < rotation; i++) {
        t.forward(radius * 2 * Math.PI / 360); t.right(1);
    }
    t.down(); if (rotation > 0) t.right(60); int turn = 120;
    t.forward(size); t.right(turn);
    t.forward(size); t.right(turn);
    t.forward(size); t.right(turn);
    t.left(rotation);
}
```

```java
public static void main(String[] args) {
    Turtle t = new Turtle();
    t.hide();
    t.speed(0);
    drawCircle(t, 100.0, 0.0, "green");
    drawCircle(t, 75.0, 0.0, "blue");
    drawCircle(t, 50.0, 0.0, "red");
```

```java
}
```
6.2 Test Suite

During compiler development, unit tests were written for each component upon completion in order to verify that the features we implemented were working as intended. Once the compiler was ready to execute programs end-to-end, we wrote a test suite of 50 test programs.

The first 15 tests are designed to fail, and the component of the compiler that finds the error is included in the filename of the test program. The remaining 35 tests are designed to pass and to test various features of the Mandala programming language, such as variables, arithmetic, loops, functions, and drawing multiple Mandalas.

| Output Correct: [y] for f_01_semantic.mandala |
| Output Correct: [y] for f_02_semantic.mandala |
| Output Correct: [y] for f_03_semantic.mandala |
| Output Correct: [y] for f_04_semantic.mandala |
| Output Correct: [y] for f_05_semantic.mandala |
| Output Correct: [y] for f_06_preprocessor.mandala |
| Output Correct: [y] for f_07_preprocessor.mandala |
| Output Correct: [y] for f_08_preprocessor.mandala |
| Output Correct: [y] for f_09_preprocessor.mandala |
| Output Correct: [y] for f_10_preprocessor.mandala |
| Output Correct: [y] for f_11_parser.mandala |
| Output Correct: [y] for f_12_parser.mandala |
| Output Correct: [y] for f_13_parser.mandala |
| Output Correct: [y] for f_14_parser.mandala |
| Output Correct: [y] for f_15_scanner.mandala |
| Output Correct: [y] for p_01_dot.mandala |
| Output Correct: [y] for p_02_square.mandala |
| Output Correct: [y] for p_03_circle.mandala |
| Output Correct: [y] for p_04_triangle.mandala |
| Output Correct: [y] for p_05_rotation.mandala |
| Output Correct: [y] for p_06_color.mandala |
| Output Correct: [y] for p_07_concentric.mandala |
| Output Correct: [y] for p_08_circles.mandala |
| Output Correct: [y] for p_09_offset.mandala |
| Output Correct: [y] for p_10_overlap.mandala |
| Output Correct: [y] for p_11_many_layers.mandala |
| Output Correct: [y] for p_12_many_shapes_layers.mandala |
| Output Correct: [y] for p_13_angular_shift.mandala |
| Output Correct: [y] for p_14_rotation_angular_shift.mandala |
| Output Correct: [y] for p_15_arithmetic.mandala |
6.3 Automated Testing

With a test suite containing 50 test programs, it was necessary to automate regression testing. A bash script called `regression_tester.sh` was written that runs each program, checks the filename of the test program to determine whether the test should pass or fail, verifies the output against the predetermined expected output, and displays the results. This regression testing script is included in Appendix A. The test suite was created by Harsha and Kanika.
Chapter 7

Lessons Learned

7.1 Edo

I learned that bash scripts exhibit completely nondeterministic behavior, List.rev solves all of your problems, and that variable naming in OCaml is one of the great unsolved mysteries in today’s day and age. On a more serious note, I learned how important it was to set goals and plan ahead as a team, and to be generous in allocating time to finish a task, because you never know how long you can be stuck on a bug. It’s also really important to communicate with team members so everybody is on the page in how things are being implemented. Finally, I think it’s really important to completely understand the functionality of a feature before beginning to implement it, so you don’t end up half-implementing a function and confusing both yourself and your teammates when you look back at it later.

Advice: Communication is everything! Have a good way to keep track of what you’ve done so far and what still needs to happen.

7.2 Harsha

The most important thing is to pick a project that you won’t get bored working on. Having strong communication and frequent meetings throughout the term is important. Make sure team members are very well-versed in the target language, whether it’s Python, C, Java, or something else. Testing each component with unit tests was useful. Bash scripting is the worst thing ever. Don’t commit broken code.

Advice: Pick chill teammates. Choosing to create a graphics-based language is going down a dangerous road but worth it in the end.
7.3 Kanika

One of the most important aspects to keep in mind is that this is a team project. So choose your team wisely!! Some of the most challenging parts of completing a successful project are working well in a team and maintaining good communication. Make sure to start meeting early in the semester and keep up the momentum throughout. Pro-tip: The hello world milestone is actually really important for making sure your team is on track, so don’t cut corners and actually fully implement hello world by the milestone date. From a coding perspective, make sure to allocate sufficient time because OCaml can be a tricky thing and you don’t know how long it might take to fix a bug. Also make sure to communicate well between teammates so that the different parts of the code fit well together.

**Advice:** Start early, communicate well, test often, have fun!

7.4 Sam

Working with a group you trust and can communicate with is key. We had a great group and it was a lot of fun to work on a visual language and be able to see a figure generated with the language you created. Having weekly meetings to touch base on progress and reevaluate a timeline for milestones is really helpful in staying on task. It’s also important to recognize that if you are new to OCaml it might take some time to get familiar with the language, but making mistakes along the way makes it a lot easier to recognize and fix bugs later on. It’s also really helpful to explain your code to your teammates, especially when you are making a significant design decision for the language so that they can account for that decision in their code.

**Advice:** Pick a good team, update one another frequently, and learn to love OCaml!
Appendix A

Code Listing

A.1 preprocessor.py

```python
#!/usr/bin/python
#
# Author: Harsha Vemuri
#
# import os
import re
import sys

# Find the best implementation available based on the platform
try:
    from cStringIO import cStringIO
except:
    from StringIO import StringIO

invalid_characters = (';', '?', '~') # characters not in the language
comment_symbol = '#' # character for commenting
blockcomment = ['//', '#/']
extensions = ('.mndl', '.mandala') # file extensions for the language

def process(input_file):
    stack = [0]
    output = StringIO()
    newindent = False
    commented = False
    linejoin = False

    for i, line in enumerate(input_file):
        clean_line = sanitize(line) # remove comments

        if clean_line:
            # throw error on invalid characters
            for char in invalid_characters:
```
if char in clean_line:
    sys.exit("Invalid character: [0]. Found on line: [1]".format(char, i))

stripped_line = clean_line.lstrip()

if len(stripped_line) > 1 and blockcomment[0] == stripped_line[:2]:
    commented = True

if commented:
    if len(clean_line) > 1 and blockcomment[1] == clean_line[-2:]:
        commented = False

else:
    if not linejoin:
        wcount = len(clean_line) - len(clean_line.lstrip(' '))

        if newindent:
            if wcount > stack[-1]:
                stack.append(wcount)
                newindent = False
        else:
            sys.exit("Indentation error on line {}".format(i))

if wcount > stack[-1]:
    print(clean_line)
    sys.exit("Indentation error on line {}".format(i))

else:
    while wcount < stack[-1]:
        clean_line = "} ;
" + clean_line
    stack.pop()
    if wcount != stack[-1]:
        sys.exit("Indentation error on line {}".format(i))

if clean_line[-1] == ':':
    newindent = True
    clean_line = clean_line + "\n"

elif clean_line[-1] == "\":
    linejoin = True
    clean_line = clean_line[:-1]

else:
    linejoin = False
    clean_line = clean_line + " ;\n"

output.write(clean_line)

while 0 < stack[-1]:
    output.write("\")
stack.pop()

output = StringIO(remove_semis(output))

output = StringIO(handle_funcs_and_loops(output))

return output

# remove semicolons from custom type creation
def remove_semis(text_io):
text = text_io.getvalue()
in_braces = False
output_text = ""

for line in text.splitlines():
    if '{' in line:
        in_braces = True
    if in_braces:
        if '}' in line:
            in_braces = False
    if in_braces:
        if ':' in line:
            line += '{
        output_text += line[:-1]
        output_text += "\n"
    else:
        output_text += line
        output_text += "\n"

return output_text

# fixes semicolons in functions and loops
def handle_funcs_and_loops(text_io):
text = text_io.getvalue()
output_text = ""

for line in text.splitlines():
    if line[-1] == '{' or ';' in line:
        output_text += line
        output_text += "\n"
    elif 'Geo' in line or 'Size' in line or 'Color' in line or 'Rotation' in line:
        output_text += line
        output_text += "\n"
    elif 'Radius' in line or 'Shape' in line or 'Count' in line or 'Offset' in line or 'AngularShift' in line:
        output_text += line
        output_text += "\n"
    else:
        output_text += line + ';'
        output_text += "\n"
return output_text

# removes comments from the line
def sanitize(line):
    if blockcomment[0] not in line and blockcomment[1] not in line and
        comment_symbol in line:
        regex_pattern = "^\(\*?\)\#\.*\.*$
        match = re.match(regex_pattern, line)
        sans_comments = match.group(1)
    else:
        sans_comments = line
    return sans_comments.rstrip()

# main
if __name__ == "__main__":
    # sanitize usage
    if len(sys.argv) != 2:
        sys.exit("usage: python preprocessor.py <input.mandala>")

    # open the file
    try:
        infile = open(sys.argv[1], 'r')
    except IOError:
        sys.exit("Cannot read input file.")

    # get the path
    filename = os.path.basename(infile.name)
directory = os.path.dirname(infile.name) + '/'

    # get the filename without extension
    if filename.lower().endswith(extensions):
        new_filename = os.path.splitext(filename)[0]
    else:
        sys.exit("Input file must have Mandala file extension.")

    # process the input file
    output = process(infile)

    # create output file
    outfile = open(directory + new_filename + " .mandala.proc", 'w')
    outfile.write(output.getvalue())

A.2 scanner.mll

(* Authors: Edo Roth, Harsha Vemuri *)
[ open Parser;; ]

(* numbers and literals *)
let digit = ['0'-'9']
let alpha = ['a'-'z' 'A'-'Z' '_']
let number = '-'? digit+ '.' digit* | '-'? digit* '.' digit+

rule token = parse

(* white space *)
| [ ' ' '	' '' '
' ] { token lexbuf }

(* literals and variables *)
| '-'? digit+ as lit { LITERAL (int_of_string lit) }
| number as lit { FLOAT_LITERAL (float_of_string lit) }
| ['a'-'z']+ (alpha | digit)* as lxm { ID (lxm) }

(* comments *)
| "/#" { comment lexbuf }

(* arithmetic operators *)
| '+' { PLUS } | '*' { TIMES }
| '-' { MINUS } | '/' { DIVIDE }

(* assignment *)
| '=' { ASSIGN } | ':' { COLON }

(* loop words *)
| "To" { TO } | "Foreach" { FOREACH }

(* punctuation and delimiters *)
| '(' { LPAREN } | ')' { RPAREN }
| '[' { LBRACKET } | ']' { RBRACKET }
| '{' { LBRACE } | '}' { RBRACE }
| ',' { COMMA }
| ';' { SEMI }

(* built-in functions and constructors *)
| "Def" { DEF } | "Return" { RETURN }
| "Create" { CREATE }

(* language specific keywords *)
| "Radius" { RADIUS } | "Count" { COUNT }
| "Size" { SIZE } | "Color" { COLOR }
| "Rotation" { ROTATION } | "Offset" { OFFSET }
| "AngularShift" { ANGULARSHIFT }

(* types *)
| "Number" { NUMBER } | "Void" { VOID }
| "Shape" { SHAPE } | "Geo" { GEO }
| "Layer" { LAYER } | "Mandala" { MANDALA }
A.3 parser.mly

```ml
/* punctuation and delimiters */
/%token LPAREN RPAREN LBRACKET RBRACKET LBRACE RBRACE COMMA SEMI
/* arithmetic operators */
/%token PLUS MINUS TIMES DIVIDE
/* loop operators */
/%token FOREACH TO
/* assignment */
/%token ASSIGN COLON
/* built-in functions and constructors */
/%token DEF RETURN CREATE
/* language specific keywords */
/%token RADIUS COUNT SIZE COLOR ROTATION OFFSET ANGULARSHIFT
/* types */
/%token NUMBER BOOLEAN VOID SHAPE GEO LAYER MANDALA
/* geo types */
/%token CIRCLE TRIANGLE SQUARE
/* literals and variables */
/%token <float> FLOAT_LITERAL
/%token <int> LITERAL
/%token <string> ID
/* end of file */
/%token EOF
/* right ASSIGN EOF */
/* left PLUS MINUS TIMES DIVIDE */
/* start program */
/* type <Ast.program> program */
```
program:
  decls EOF [ $1 ]
/* Parse function declarations and statements */
decls:
  */ nothing */
  | decls fdecl [ fst $1, ( $2 :: snd $1 ) ]
  | decls stmt [ ( $2 :: fst $1 ),
                snd $1 ]

fdecl:
  DEF any_id ID LPAREN formals_opt RPAREN COLON LBRACE stmt_list RBRACE SEMI [{
    fname = $3;
    returntype = $2;
    formals = $5;
    body = List.rev $9
  }]
/* Formal parameters used in function declaration */
formals_opt:
  */ nothing */
  | formal_list [ List.rev $1 ]

formal_list:
  formal [ $1 ]
  | formal_list COMMA formal [ $3 :: $1 ]
/* Formal parameters */
formal:
  any_id ID
  | [ kind = $1;
      vname = $2;
  ]

any_id:
  custom_types [ $1 ]
  | basic_types [ $1 ]
/* Custom types to create Mandalas */
custom_types:
  MANDALA [ Mandalat ]
  | LAYER [ Layert ]
  | SHAPE [ Shapet ]
/* Variable types */
basic_types:
stmt_list:
  /* nothing */
  stmt_list stmt

stmt:
  expr SEMI
  | RETURN expr SEMI
  | FOREACH ID ASSIGN FLOAT_LITERAL TO FLOAT_LITERAL COLON
     LBRACE stmt_list RBRACE SEMI
  /* Constructor statements for Mandala, Shape and Layer */
  | assign_expr ASSIGN CREATE MANDALA SEMI
  | assign_expr ASSIGN CREATE SHAPE COLON LBRACE GEO expr
     SIZE expr
     ROTATION expr RBRACE SEMI
  | assign_expr ASSIGN CREATE LAYER COLON LBRACE RADIUS expr
     SHAPE expr
     COUNT expr
     OFFSET expr
     ANGULARSHIFT expr RBRACE SEMI
  | assign_expr ASSIGN expr SEMI

expr:
  LITERAL
  | FLOAT_LITERAL
  | ID
  | expr PLUS expr
  | expr MINUS expr
  | expr TIMES expr
  | expr DIVIDE expr
  | LPAREN expr RPAREN
  | ID COLON LPAREN actuals_opt RPAREN

assign_expr:
  any_id ID
  ||
 kind = $1;
 vname = $2;
]]

/* actual parameters passed into functions */
 actuals_opt:
  /* nothing */
  | actuals_list

 actuals_list:
  expr
  | actuals_list , expr

A.4 ast.mli

```ocaml
type op = Add | Sub | Mult | Div

(* Mandala variable types. *)
type mndlt =
  | Number
  | Boolean
  | Shape
  | Geo
  | Layer
  | Mandalat
  | Array
  | Color
  | Void

type expr =
  | Literal of int
  | Float_Literal of float
  | Id of string
  | Binop of expr * op * expr
  | Call of string * expr list

type var_decl = {
  kind : mndlt;
  vname : string;
}

type stmt =
  | Expr of expr
  | Assign of var_decl * expr
  | Return of expr
  | Foreach of string * float * float * stmt list
  | Shape of var_decl * expr * expr * expr * expr
```

A.5  semantic.ml

open Ast
open Sast

exception Error of string

(* Storing all variables, including parent for coping *)
type symbol_table = {
  parent : symbol_table option;
  variables : (string * smndlt) list
}

(* Storing all functions *)
type function_table = {
  functions : (string * smndlt * svar_decl list * sstmt list) list
}

(* Complete environment *)
type translation_environment = {
  var_scope : symbol_table;
  fun_scope : function_table;
}

(* List of java built-in colors, for use for color in shape *)
let list_of_colors = [
  "black"; "red"; "blue"; "cyan"; "darkGray"; "gray"; "green"
  ; "lightGray"; "orange"; "pink"; "white"; "yellow"
]

(* Returns the name, type and value *)
let find_variable (scope: symbol_table) name=
  try
    List.find (fun (s,_) -> s=name) scope.variables
  with Not_found -> raise (Error ("Unable to find variable in lookup table "^name"))
let rec find_function (scope: function_table) name =  
  try  
    List.find (fun (s, _, _, _) -> s=name) scope.functions  
  with Not_found ->  
    raise (Error("Function not found in function table! " ^ name))

let add_to_var_table (env, name, typ) =  
  try  
    let (n, t) = List.find (fun (s,_) -> s=name) env.var_scope.variables in  
    env  
  with Not_found ->  
    let new_vars = (name, typ)::env.var_scope.variables in  
    let new_sym_table = {parent = env.var_scope.parent;  
        variables = new_vars;} in  
    let new_env = {env with var_scope = new_sym_table} in  
    new_env

let add_to_func_table env sfunc_decl =  
  let func_table = env.fun_scope in  
  let old_functions = func_table.functions in  
  let func_name = sfunc_decl.sfname in  
  let func_type = sfunc_decl.sreturntype in  
  let func_formals = sfunc_decl.sformals in  
  let func_body = sfunc_decl.sbody in  
  let new_functions = (func_name, func_type, func_formals, func_body)::old_functions  
  in  
    let new_fun_scope = {functions = new_functions} in  
    let final_env = {env with fun_scope = new_fun_scope} in  
    final_env

let rec find_function (scope: function_table) name =  
  List.find (fun (s, _, _, _) -> s=name) scope.functions

let rec extract_type (scope: function_table) name = function  
  (smndlt, string) -> (smndlt)

let get_formal_arg_types env = function  
  (smndlt, string) -> (smndlt)

(* Process a single expression, checking for type matching and compatibility *)  
let rec semantic_expr (env:translation_environment):(Ast.expr -> Sast.sexpr *  
    translation_environment) = function

Ast.Id(vname) ->  
  (* Check for built-in Ids for shapes like circle, triangle, and square *)  
  if (vname="circle" || vname="triangle" || vname="square")  
    then
let geo_typ = Sast.Geot in
let name = vname in
(Sast.Id(name), geo_typ, env)

else (/* Checks for build in Id of color */)
let return_thing = try let color = List.find (fun s -> s=vname) list_of_colors in
let color_typ = Sast.Colort in
let name = vname in
(Sast.Id(name), color_typ, env)

with Not_found ->
(* Otherwise name is treated as a variable *)
let vdecl = try find_variable env.var_scope vname
with Not_found ->
raise (Error("undeclared identifier: "^vname))
(* Want to add the symbol to our symbol table *) in
let (name, typ) = vdecl in
(Sast.Id(name), typ, env)
in return_thing

(* AST Call of string * expr list *)
| Ast.Float_Literal(num) ->
(Sast.Float_Literal(num), Sast.Numbert, (*Sast.SNumber(num),*) env)
| Ast.Literal(num) ->
(Sast.Literal(num), Sast.Integert, env)
| Ast.Binop(term1, operator, term2) ->
(* convert to Sast.Binop *)

let (eval_term1, typ1, new_env) = semantic_expr env term1 in
let (eval_term2, typ2, new_env) = semantic_expr env term2 in
(* now translate Ast.operator to Sast.operator *)
if not (typ1 = typ2)
then raise (Error("Mismatched types, invalid operation"))
else
(* Checking the types for binary operators and will do evaluation of binop *)
in sast_to_jast (*)
(Sast.Binop(eval_term1, operator, eval_term2), typ1, env)

| Ast.Call(fid, args) ->

if not ( ((List.length args) > 0) ) then (/* Make sure that func_decl has no formal arguments * )
let (_, ret_typ, decl_list, _) = find_function env.fun_scope fid in
let decl_size = List.length decl_list in
if (decl_size > 0) then
    raise (Error("This function expects parameters but none were provided"))
else
    (Sast.Call(fid, [], ret_typ, env)
)
else

let actual_types = List.map (fun expr -> semantic_expr env expr) args in
(* let actual_type_names = List.iter extract_type actual_types*)
let actual_len = List.length args in
let actual_types_list = List.fold_left (fun a (_, typ, ret_env) -> typ :: a) [] actual_types in
(*get list of just types from list of (type, string) tuples, [] is an accumulator*)
let actual_expr_list = List.fold_left (fun a (expr, _, ret_env) -> expr :: a) [] actual_types in
let len = List.length actual_expr_list in
if (fid = "draw")
    then
        if (len == 1)
            then (Sast.Call(fid, actual_expr_list), Sast.Voidt, env)
            else raise (Error("Draw function has incorrect parameters"^ string_of_int actual_len))
        else raise (Error("addTo function has incorrect parameters"^ string_of_int actual_len))
    else
        try (let (fname, fret, fargs, fbody) =
            find_function env.fun_scope fid in

        let formal_types = List.map (fun farg -> let arg_type =
            get_formal_arg_types env (farg.skind, farg.svname) in arg_type) fargs in
        if not (actual_types_list= formal_types)
        then
            raise (Error("Mismatching types in function call"))
        else
            let actual_expr_list = List.fold_left (fun a (expr, _, ret_env) -> expr :: a) [] actual_types in
            (Sast.Call(fname, actual_expr_list), fret, env)
            (* Call of string * sexpr list*)
with Not_found ->
  let numFuncs = List.length env.fun_scope.functions in
  raise (Error(fid^"undeclared function"^string_of_int numFuncs))
|
_ -> raise (Error("invalid expression, was not able to match expression"))

let proc_type = function
  Ast.Boolean -> Sast.Boolean
  | Ast.Shapet -> Sast.Shapet
  | Ast.Layert -> Sast.Layert
  | Ast.Mandalat -> Sast.Mandalat
  | Ast.Arrayt -> Sast.Arrayt
  | Ast.Numbert -> Sast.Numbert
  | Ast.Voidt -> Sast.Voidt

let proc_var_decl = function
  (var_decl, env) ->
  let k = var_decl.kind in
  let v = var_decl.vname in
  let sskind =
    if (k = Ast.Numbert) then
      Sast.Numbert
    else if (k = Ast.Geot) then
      Sast.Geot
    else if (k = Ast.Colort) then
      Sast.Colort
    else
      proc_type k in
    let new_svar_decl = {
      skind = sskind;
      svname = v;
    } in
    let new_env = add_to_var_table (env, new_svar_decl.skbind, new_svar_decl.
      skbind) in
    (new_svar_decl, new_env)

let rec proc_formals (var_decl_list, env, update_var_decl_list: Ast.var_decl list
  * translation_environment * Sast.svar_decl list) = match var_decl_list
with [] -> (update_var_decl_list, env)
| [var_decl] -> let (new_var_decl, new_env) = proc_var_decl(var_decl, env) in
  (update_var_decl_list@new_var_decl, env)
| var_decl :: other_var_decls ->
  let (new_var_decl, new_env) = proc_var_decl(var_decl, env) in
  proc_formals (other_var_decls, new_env, update_var_decl_list@new_var_decl))

let var_empty_table_init = {parent=None; variables=[]}
let fun_empty_table_init = { functions = []; }
let empty_environment =
let rec semantic_stmt (env: translation_environment) : (Ast.stmt -> Sast.sstmt * smndlt * translation_environment) = function

Ast.Mandala(mandala_arg) ->

let {vname=name} = mandala_arg in
let typ = Sast.Mandalat in
(* add to current env *)
let new_env = add_to_var_table (env, name, typ) in

(Sast.Mandala(|skind=typ; svname=name|), typ, new_env)

| Ast.Layer(v_name, v_radius, v_shape, v_count, v_offset, v_angular_shift) ->

let {vname=name} = v_name in
let typ = Sast.Layert in
let (s_radius, s_r_typ, env) = semantic_expr env v_radius in
let (s_shape, s_s_typ, env) = semantic_expr env v_shape in
let (s_count, s_c_typ, env) = semantic_expr env v_count in
let (s_offset, s_o_typ, env) = semantic_expr env v_offset in
let (s_angular_shift, s_a_typ, env) = semantic_expr env v_angular_shift in
let new_env = add_to_var_table (env, name, typ) in

(Sast.Layer(|skind=typ; svname=name|, s_radius, s_shape, s_count, s_offset, s_angular_shift), typ, new_env)

| Ast.Shape(v_name, v_geo, v_size, v_color, v_rotation) ->

let {vname=name} = v_name in
let typ = Sast.Shapet in
let s_geo = match v_geo with

Ast.Id(v_geo) -> let new_geo = v_geo in new_geo
| _ -> raise (Error("WRONG FORMAT FOR GEO IN SHAPE!")) in
let updated_s_geo = Sast.SGeo(s_geo) in
let (size_stmt, typ, env) = semantic_expr env v_size in
(* Checking that the shape’s size is a float and returning a sexpr *)

let size_value = match typ with
Sast.Numbert -> size_stmt
| _ -> raise (Error("Size wasn’t a number!")) in

let s_color = match v_color with
let new_color = v_color in new_color

let updated_s_color = Sast.SColor(s_color) in

let (rotation_stmt, typ, env) = semantic_expr env v_rotation in

let rotation_value = match typ with
  Sast.Numbert -> rotation_stmt
| _ -> raise (Error("Rotation wasn't a number!"))

let new_env = add_to_var_table (env, name, typ) in

(Sast.Shape([skind = typ; svname=name;], updated_s_geo, size_value, updated_s_color, rotation_value), typ, new_env)

let newExpr = try
  semantic_expr env expression
with Not_found ->
  raise (Error("undefined expression"))

let (x, typ, ret_env) = newExpr in

let {kind=typ2; vname=name2} = lefthand

let result = match typ with
  (*Assign is of form var_decl*expr*)
  typ2 -> let new_env = add_to_var_table (env, name2, typ2)
          in (Sast.Assign([|skind = typ2; svname = name2|], assign_val), typ,
             new_env) (* check structural equality *)
| _ -> raise (Error("Assignment could not be typechecked"))

let (_, returntype) = List.find (fun (s,_) -> s="return") env.var_scope.variables in
let newExpr = semantic_expr env x in
let (x, typ, ret_env) = newExpr in
let result = match typ with
returntype -> (Sast.Return(x), typ, env)
  | _ -> raise (Error("User defined function is returning something of the
  | wrong type"))

in result

Ast.ForEach(varName, countStart, countEnd, body) ->
(*create custom env for the scope of the for loop*)
let body = List.rev body in
let func_env=
  |
  var_scope = {parent = env.var_scope.parent; variables=(varName,Sast.
  Numbert)::env.var_scope.variables};
  fun_scope = env.fun_scope;
  |
  let empty_list=[] in
let (statements, func_env) = separate_statements (body, func_env, empty_list)
  |
  (Sast.ForEach(Sast.Id(varName), Sast.Float_Literal(countStart), Sast.
  Float_Literal(countEnd), statements), Sast.Loopt, env)
  |
  _ -> raise (Error("Unable to match statement"))

and separate_statements (stmts, env, update_list: Ast.stmt list *
translatation_environment * Sast.stmt list) = match stmts
with [] -> (update_list, env)
  |
  [stmt] -> let (new_stmt, typ, new_env) = semantic_stmt env stmt in (update_list@[new_stmt], new_env)
  |
  stmt :: other_stmts ->
    let (new_stmt, typ, new_env) = semantic_stmt env stmt in
    separate_statements (other_stmts, new_env, update_list@[new_stmt])

let rec semantic_func (env: translation_environment): (Ast.func_decl -> Sast.
sfuncdecl * translation_environment) = function
my_func ->
  let fname = my_func.fname in
  let returntype = my_func.returntype in
  let formals = my_func.formals in
  let body = my_func.body in

  let empty_list = [] in
  let new_returntype = proc_type returntype in
  let func_env=
    |
    var_scope = {parent = env.var_scope.parent; variables=[("return",
    new_returntype)]};
    fun_scope = fun_empty_table_init;
let (new_formals, func_env) = proc_formals(formals, func_env, empty_list) in
(*walks through body of function, checking types etc.*)
let (new_stmts, func_env) = separate_statements(body, func_env, empty_list) in
(*check that function returned the right thing—get the return stmt from stmt list, check its typ against returntyp*)
(let rettyp = findReturnStmt new_stmts in *)
(*CHECK IF rettyp is same as new_returntype*)

let sfuncdecl = {
    sfname = fname;
    sreturntype = new_returntype;
    sformals = new_formals;
    sbody = new_stmts;
} in

let env = add_to_func_table env sfuncdecl in

let rec separate_functions (functions, env, update_list: Ast.func_decl list * translation_environment * Sast.sfuncdecl list) = match functions with [ ] -> (update_list, env)
| [func] ->
    let (new_func, new_env) = semantic_func env func in (update_list@[new_func ], new_env)
| func :: other_funcs ->
    let (new_func, new_env) = semantic_func env func in
    separate_functions (other funcs, new_env, update_list@[new_func])

let rec semantic_check (check_program: Ast.program): (Sast.sprogram) =
    let (prog_stmts, prog_funcs) = check_program in
    let env = empty_environment in
    let empty_list = [ ] in
    let reverse_prog_stmts = List.rev prog_stmts in
    let (resulting_functions, env) = separate_functions (prog_funcs, env, empty_list) in
    let (statements, env) = separate_statements (reverse_prog_stmts, env, empty_list) in
    Sast.SProg(statements, resulting_functions)
A.6 sast.mli

open Ast

(* Mandala specific data types *)
type smndlt =
  | Numbert
  | Booleant
  | Shapet
  | Geot
  | Layert
  | Mandalat
  | Arrayt
  | Color
  | Integert
  | Voidt
  | Loopt

(* Stores the values and types *)
type sdata_val =
  SInt
  | SLiteral
  | SFloat
  | SVoid
  | SNumber of float
  | SBoolean of int
  | SShape
  | SGeo of string
  | SLayer
  | SMandala
  | SArray
  | SColor of string

type sexpr =
  Literal of int
  | Float_Literal of float
  | Id of string
  | Binop of sexpr * op * sexpr
  | Call of string * sexpr list

and svar_decl = {
  skind : smndlt;
  svname : string;
}

and sfucndecl = {
  sfname : string;
  sreturntype : smndlt;
  sformals : svar_decl list;
  sbody : stmt list;
A.7  sast_to_jast.ml

open Ast
open Sast
open Jast
open Semantic

(*Define constant for mathematical calculations*)
let pi = 3.14159

(*Environment used to store all variables, functions, and drawing structure*)
type environment = {
  drawing: Jast.drawing;
  functions: Sast.sfuncdecl list;
}

(*Creates an SAST by going through the scanner, parser, and semantic_check*)
let sast =
  let lexbuf = Lexing.from_channel stdin in
  let ast = Parser.program Scanner.token lexbuf in
  Semantic.semantic_check ast

(*Looks up function from function table*)
let find_function (scope: environment) fid =
  try
    List.find (fun s -> s.sfname = fid) scope.functions
  with Not_found -> raise (Error ("Function not properly declared: "^fid))

(*Looks up variable from variable table*)
let find_variable (scope: environment) name=
  try
    List.find (fun (s,_) -> s=name) (List.rev scope.drawing.variables)
  with Not_found -> raise (Error("Variable not properly declared: "/name))

(*Looks up return value and ensures return type matches the specification in function declaration*)
let find_variable_check_return_type (scope, return_typ: environment * smndlt)
  name=
  try
    List.find (fun (s,_) -> s=name) scope.drawing.variables
  with Not_found ->
    if (not (return_typ = Sast.Voidt)) then
      raise (Error("No return statement found for non-void function. Must return a value of corresponding type."))
    else
      (", Jast.JVoid)

(*Looks up mandala from mandala table*)
let find_mandala (scope: environment) mandala_name =
  try List.find (fun (str, mandala) -> str = mandala_name) scope.drawing.mandala_list
  with Not_found -> raise (Error("Mandala not properly created: "/^mandala_name))

(*Processes a binary operation recursively*)
let rec proc_bin_expr (scope: environment):(Sast.sexpr -> Sast.sexpr) = function
  | Sast.Float_Literal(term1) -> Sast.Float_Literal(term1)
  | Sast.Id(var) ->
    let (n,v) = find_variable scope var in
    Jast.JNumber(my_float) = v in
    Sast.Float_Literal(my_float)
  | Sast.Binop(t1, op, t2) ->
    let eval_term1 = proc_bin_expr scope t1 in
    let eval_term2 = proc_bin_expr scope t2 in
    Sast.Float_Literal(float_term_one) = eval_term1 in
    Sast.Float_Literal(float_term_two) = eval_term2 in
    let result = match op
      with Add -> float_term_one +. float_term_two
      | Sub -> float_term_one -. float_term_two
      | Mult -> float_term_one *. float_term_two
      | Div -> float_term_one /. float_term_two
    in Sast.Float_Literal(result)

(*Looks up given layer names and returns the actual structure of these layers to add to a Mandala structure*)
let rec get_layer_info(env, actual_args, layer_list: environment * Sast.sexpr list * Jast.layer list): (Jast.layer list * environment) = match actual_args
with [] → raise (Error("Invalid call of addTo: must be adding at least one layer.
"));

| [layer_arg] → let (new_env, ret_typ) = proc_expr env layer_arg in
| (* Check to see if the layer has been defined *)
| let layer_name = match layer_arg
| with Sast.Id(l) → l
| | _ → raise (Error("Parameter provided to addTo is not a layer.
"));
| in
| let (my_layer_name, my_layer_typ) = find_variable new_env layer_name in
| let my_layer_info = match my_layer_typ
| with Jast.JLayer(m) → m
| | _ → raise (Error("Failure in retrieving layer information");)
| in
| (layer_list@[my_layer_info], new_env)
| | layer_arg :: other_layers → let (new_env, ret_typ) = proc_expr env layer_arg in
| (*Check to see if the layer has been defined*)
| let layer_name = match layer_arg
| with S.Id(l) → l
| | _ → raise (Error("Parameter provided to addTo is not a layer.
"));
| in
| let (my_layer_name, my_layer_typ) = find_variable new_env layer_name in
| let my_layer_info = match my_layer_typ
| with Jast.JLayer(m) → m
| | _ → raise (Error("Failure in retrieving layer information");)
| in
| get_layer_info (new_env, other_layers, layer_list@[my_layer_info])

(*Match the declared arguments of a function with its given parameters in a function call*)
and match_formals (formals, params, env: Sast.svar_decl list * Sast.sexpr list * environment) = match formals
with [] → env
| [formal] → let namer = formal.svname in
| let result =
| match params
| with [] → env
| [[param] →
| | let (_, my_val) = proc_expr env param in
| | let new_variables = env.drawing.variables@[(namer, my_val)] in
| | let drawing = env.drawing in
| | let new_drawing = |drawing with variables = new_variables| in
| | let new_env = |env with drawing = new_drawing| in
| | new_env in
| result
| | formal :: other_formals → let namee = formal.svname in
| | match params
| | with [] → env
| | ((param :: other_params) →
| | | let (_, my_val) = proc_expr env param in
| | | let new_variables = env.drawing.variables@[namee, my_val] in
let drawing = env.drawing in
let new_drawing = {drawing with variables = new_variables} in
let new_env = {env with drawing = new_drawing} in
match_formals (other_formals, other_params, new_env)

(* Pull out the values of the arguments passed into a function *)
and process_arguments (params, l: Sast.sexpr list * string list) = match params with [] -> l
| [param] -> let result = match param with Sast.Float_Literal(term1) -> l
| Sast.Id(var) -> l @ [var] in result
| param :: other_params -> let result = match param with Sast.Float_Literal(term1) -> l
| Sast.Id(var) -> l @ [var] in
process_arguments (other_params, result)

(* Process an SAST expression and return the new environment along with resulting JAST type *)
and proc_expr (env:environment): (Sast.sexpr -> environment * Jast.jdata_type) = function
Sast.Id(vname) ->
(* Want to go from Sast.Id to Jast.jexpr or Jast.JId, and Jast.drawing *)
let var_info = try
  find_variable env vname
with Not_found ->
  raise (Error("undeclared identifier: "^vname))
in let (name, value) = var_info in
(env, value)
| Sast.Literal(literal_var) ->
  (env, Jast.JInt(literal_var))
| Sast.Float_Literal(number_var) ->
  (env, Jast.JNumbert(number_var))
| Sast.Binop(term1, operator, term2) ->
  (* Recursively calls a binary operator *)
  let eval_term1 = proc_bin_expr env term1 in
  let eval_term2 = proc_bin_expr env term2 in
  (* Can be a variable or a float literal *)
  let float_term_one = match eval_term1 with Sast.Float_Literal(term1) -> term1
  | Sast.Id(var) ->
    let (n, v) = find_variable env var in
    let Jast.JNumbert(my_float) = v in
    my_float
  | _ -> raise (Error("Operand one is not a float literal, invalid operand "))
in
  let float_term_two = match eval_term2
with Sast.Float_Literal(term2) -> term2
  | Sast.Id(var) ->
  let (n,v) = find_variable env var in
  let Jast.JNumbert(my_float) = v in
  my_float
  _ _ _ _ _ _ raise(Exception("Operand two is not a float literal, invalid operand "))

(*Calls supported binary operator*)
let result = match operator
  | Add -> float_term_one +. float_term_two
  | Sub -> float_term_one -. float_term_two
  | Mult -> float_term_one *. float_term_two
  | Div -> float_term_one /. float_term_two

in (env, Jast.JNumbert(result))

(*Process function calls*)
| Sast.Call(fid, args) ->

  let old_variables = env.drawing.variables in

  if not ( List.length args > 0 ) then (
    (*Make sure that func_decl has no formal arguments*)
    let my_func_decl = find_function env fid in
    let my_body = my_func_decl.body in
    let env_with_return = separate_statements_s(my_body, env) in
    let return_name = "return" in

    let var = find_variable_check_return_type (env_with_return, my_func_decl.
      sreturntype) return_name in

    let (n, v) = var in
    let new_env = |
      drawing = [mandala_list = env_with_return.drawing.mandala_list; variables
        = old_variables; java_shapes_list = env_with_return.drawing.java_shapes_list
      ];
      functions = env_with_return.functions;
    | in
    (new_env, v)
  )

  else

    (*Add all variables only to this function's scope — everything is the same
      except for variables*)
    (*At end, empty out variables, store them, put in the arg variables, later
      add back at end (but remove arg variables)*)
    let all_param_names = process_arguments (args, []) in
    let only_param_variables = List.filter ( fun (n, v) -> if ( List.mem n
      all_param_names ) then true else false ) env.drawing.variables in
let env_with_param_vars = {
  drawing = { mandala_list = env.drawing.mandala_list; variables = only_param_variables; java_shapes_list = env.drawing.java_shapes_list; }
  functions = env.functions;
} in

(*Grab the function from its table*)
if ( not(fid = "draw") && not (fid = "addTo")) then ( 
  let my_func_decl = find_function env_with_param_vars fid in 
  let my_formals = my_func_decl.sformals in 
  let new_env = match_formals(my_formals, args, env_with_param_vars) in 
  let func_stmts = my_func_decl.sbody in 
  (*Process statements with limited scope*)
  let env_with_return = separate_statements_s(func_stmts, new_env) in 
  let return_name = "return" in

  (*Get return value (will check if return type is void if applicable)*) 
  let var = find_variable_check_return_type (env_with_return, my_func_decl.returntype) return_name in

  let (n, v) = var in 
  let new_env = {
    drawing = { mandala_list = env_with_return.drawing.mandala_list; variables = old_variables; java_shapes_list = env_with_return.drawing.java_shapes_list; }
    functions = env_with_return.functions;
  } in
  (new_env, v) )
else
  let len = List.length args in 
  if (fid ="draw") then 
  if (len == 1) then (*Drawing one mandala*) 
    let check_arg = List.hd args in 
    let curr_name = match check_arg with Sast.Id (check_arg) -> let new_check_arg = check_arg in
    new_check_arg | _ -> raise (Error("This mandala has not been defined"))
  in

  (*Find mandala from mandala_list*)
  let (mandala_name, actual_mandala) = find_mandala env curr_name in 

  let updated_current_mandala = {
    name = curr_name;
    list_of_layers = actual_mandala.list_of_layers;
    max_layer_radius = actual_mandala.max_layer_radius;
    is_draw = true;
  } in
let filtered_vars = List.filter (fun (var_name, var_typ) -> if (var_name=curr_name) then false else true) env.drawing.variables in

let filtered_mandalas = List.filter (fun (var_name, var_typ) -> if (var_name=curr_name) then false else true) env.drawing.mandala_list in

let mandalas_to_be_drawn = filtered_mandalas@[(curr_name, updated_current_mandala)] in
let updated_vars = filtered_vars@[(curr_name, Jas.cont.Mandalat(updated_current_mandala))] in
let new_draw_env = { mandala_list = mandalas_to_be_drawn; variables = updated_vars; java_shapes_list = env.drawing.java_shapes_list; } in
let new_env = { drawing = new_draw_env; functions = env.functions; } in

(new_env, Jas.cont.JVoid)

else raise (Error("Draw function has incorrect parameters"^ string_of_int len))

else if (fid="addTo")
then
(* Check that length is greater than 1 — args must contain a mandala and at least one layer*)
if (len > 1)
then
(* Pull out the first argument, which should be the mandala that a layer(s) is being added to *)
let rev_args = List.rev args in
let update_mandala = List.hd rev_args in
let update_mandala_name = match update_mandala
with Sas.ext.Id(update_mandala) -> update_mandala
| _ -> raise (Error("This name is not a string! "))
in

let (mandala_name, untyped_mandala) = List.find (fun (s,_) -> s=update_mandala_name) env.drawing.variables in

let actual_mandala = match untyped_mandala
with Jas.cont.Mandalat(untyped_mandala) -> untyped_mandala
| _ -> raise (Error("The variable returned is invalid because it is not of type mandala. "))
in
let old_layer_list = actual_mandala.list_of_layers in
(*Get layers by looking up all arguments and checking whether they’ve been defined*)

```ocaml
let new_layers_list = match rev_args
                      with hd :: tail -> get_layer_info (env, tail, old_layer_list)
                            | _ -> raise (Error("This doesn’t have a mandala and layers! " ^
                                            update_mandala_name))
                           in
                      let (actual_layer_list, layer_updated_env) = new_layers_list in
                      let updated_layer_list = actual_layer_list in
                      let rec find_max l = match l with
                         | [] -> 0.0
                         | h :: t -> max h (find_max t) in
                      let get_max_layer_radius = function
                                        updated_layer_list ->
                                        let layer_radius_list = List.fold_left (fun a layer -> layer.radius
                                                                 :: a) [] updated_layer_list in
                                        find_max layer_radius_list in
                      let updated_current_mandala = {
                          name = update_mandala_name;
                          list_of_layers = updated_layer_list;
                          max_layer_radius = get_max_layer_radius updated_layer_list;
                          is_draw = false;
                      } in
                      let env = layer_updated_env in
                      (* Leave in all mandalas except the current mandala (pull this one out) *)
                      let unchanged_variables = List.filter (fun (m_name, m_typ) ->
                                                      if (m_name=update_mandala_name) then false else true) env.drawing.variables in
                      (* Then add back in the updated mandala to the list of all variables *)
                      let updated_variables = unchanged_variables@[(update_mandala_name,
                                                                 Jast.JMandalat(updated_current_mandala))] in
                      (*Take out this mandala and add it back in with updated stuff*)
                      let unchanged_mandalas = List.filter (fun (m_name, m_typ) ->
                                                          if (m_name=update_mandala_name) then false else true) env.drawing.mandala_list in
                      let updated_mandala_list = unchanged_mandalas@[updated_current_mandala] in
                      let new_draw_env = {mandala_list = updated_mandala_list; variables =
                                          updated_variables; java_shapes_list = env.drawing.java_shapes_list;} in
                      let new_env = {drawing = new_draw_env; functions = env.functions} in
                      (new_env, Jast.JMandalat(updated_current_mandala))
                      else
```

88
raise (Error("addTo function has incorrect parameters"))

else
(env, Jast.JVoid)

|_ -> raise(Error("Other call found"))

(*Process an entire statement list by recursively processing each statement in
the list*)

and separate_statements_s (stmts, env:Sast.sstmt list * environment) = match
stmts
with [] -> env
| [stmt] -> proc_stmt env stmt (*let new_env = proc_stmt env stmt in new_env*)
| stmt :: other_stmts ->
  let new_env = proc_stmt env stmt in
  separate_statements_s (other_stmts, new_env)

(*Process an individual statement and return the resulting environment*)

and proc_stmt (env:environment):(Sast.sstmt -> environment) = function
Sast.Mandala(var_decl) ->
(*Create new mandala object of name vname*)
  let {skind = typ1; svname= name1;}= var_decl in
  (* Create a new mandala *)
  let new_mandala =
  [ name= name1;
    list_of_layers= [];
    max_layer_radius= 0.0;
    is_draw= false;
  ] in
  let new_mandalas = env.drawing.mandala_list @ [(name1, new_mandala)] in
  let new_vars = env.drawing.variables @ [(name1, Jast.JMandalat(new_mandala))] in
  let new_drawing = [mandala_list=new_mandalas; variables = new_vars;
    java_shapes_list = env.drawing.java_shapes_list;] in
  let new_env = {mandala_list=new_mandalas;
   v offsets = env.drawing.ja v a_s h a p e s _ l i s t ;} in
  new_env
| Sast.Layer(var_decl, v_radius, v_shape, v_count, v_offset, v_angular_shift)
  ->
  (* Return the var_decl for Jast*)
  let {skind = typ; svname = name;} = var_decl in
  let (env, j_radius) = proc_expr env v_radius in
  (* Match with JData_types to get type of float *)
  let actual_radius = match j_radius
    with Jast.JNumbert(j_radius) -> let new_num = j_radius in new_num
    | _ -> raise (Error("Incorrect type for radius in layer"))
  in
  let (env, j_shape_typ) = proc_expr env v_shape in
  let actual_jshape = match j_shape_typ
    with Jast.JShapet(j_shape_typ) -> j_shape_typ

89
let (env, j_count) = proc_expr env v_count in
(* Match with Jdata_typ to get the float count *)
let actual_count = match j_count
  with Jast.JInt(j_count) -> let new_count = j_count in new_count
| _ -> raise (Error("Incorrect type for shape when adding to layer"))

let (env, j_offset) = proc_expr env v_offset in
let actual_offset = match j_offset
  with Jast.JNumber(j_offset) -> let new_offset = j_offset in new_offset
| _ -> raise (Error("Incorrect type for offset"))

let (env, j_angular_shift) = proc_expr env v_angular_shift in
let actual_angular_shift = match j_angular_shift
  with Jast.JInt(j_angular_shift) -> let new_angular_shift = j_angular_shift in new_angular_shift
| _ -> raise (Error("Incorrect type for angular shift"))

let new_layer =
  {
    name = name;
    radius = actual_radius;
    shape = actual_j_shape;
    count = actual_count;
    offset = actual_offset;
    angular_shift = actual_angular_shift;
  } in

(* Add to variable list and mandala list and update environment*)
let new_variables = env.drawing.variables @ [(name, Jast.JLayer(new_layer))] in
let new_drawing = { mandala_list = env.drawing.mandala_list; variables = new_variables; java_shapes_list = env.drawing.java_shapes_list; } in
let new_env = { drawing = new_drawing; functions = env.functions; } in
new_env

let Sast.Shape(v_name, v_geo, v_size, v_color, v_rotation) ->
let { skind = typ; svname = name; } = v_name in
let Sast.SGeo(s_geo) = v_geo in
let actual_size = match v_size with
  Sast.Float_Literal(s_size) -> s_size
| Sast.Id(var_name) -> let (name, value) = find_variable env var_name in
  let Jast.JNumber(real_val) = value in real_val in
let Sast.SColor(s_color) = v_color in

let actual_rotation = match v_rotation with
  Sast.Float_Literal(s_rotation) -> s_rotation
| Sast.Id(var_name) -> let (name, value) = find_variable env var_name in
  let Jast.JNumber(real_val) = value in real_val in
let new_shape = {
  name = name;
  geo = s_geo;
  size = actual_size;
  color = s_color;
  rotation = actual_rotation;
}
in let new_variables = env.drawing.variables @ [(name, Jast.JShape(new_shape))]
in let new_drawing = {mandala_list = env.drawing.mandala_list; variables = new_variables; java_shapes_list = env.drawing.java_shapes_list;}
in let new_env = {drawing = new_drawing; functions = env.functions;}
in new_env

(*Process an expression*)
| Sast.Expr(expression) ->
  (* Add this expression to the mandala list *)
| let updated_expr = proc_expr env expression in
| let (new_env, j_typ) = updated_expr in
  (* Now return new environment and java statement *)
| new_env

(*Process foreach loop*)
| Sast.Foreach(i_var, i_start_var, i_end_var, for_statements) ->
  (*Get Jdata type values for start and end points*)
| let Sast.Id(i) = i_var in
| let i_start = match i_start_var with
  Sast.Float_Literal(x) -> Jast.JNumber(x)
  _ -> raise(Error("Start value of this for loop is not a float")) in
| let i_end = match i_end_var with
  Sast.Float_Literal(x) -> Jast.JNumber(x)
  _ -> raise(Error("End value of this for loop is not a float")) in
  (*Remove i from list if it was found*)
| let new_variables = List.filter (fun (n, v) -> if (n = i) then false else true) env.drawing.variables in
  (*Add i with its updated value*)
| let updated_vars = new_variables @[[(i, i_start)]] in
  (*Storing for later*)
| let store_old_vars = updated_vars in
  (*Create environment to pass to statement processing*)
| let updated_drawing = {env.drawing with variables = updated_vars} in
| let updated_env = {env with drawing = updated_drawing} in
  (*Pull actual values from for loop start end end*)
let Sast.Float_Literal(k_start) = i_start_var in
let Sast.Float_Literal(k_end) = i_end_var in

(*Increasing loops*)
let rec pos_loop = function
  (env, var_name, k_cur, k_end) ->

  (*i_cur is the data type to insert into variable table*)
  let i_cur = Jast.JNumbert(k_cur) in

  (*Need to update actual value of i in the table and then update environment*)
  let new_variables = List.filter (fun (n, v) -> if (n = var_name) then false else true) env.drawing.variables in
  let updated_vars = new_variables @[(var_name, i_cur)] in
  let updated_drawing = {env.drawing with variables = updated_vars} in
  let updated_env = {env with drawing = updated_drawing} in

  (*Go through all statements*)
  let fresh_env = separate_statements_s(for_statements, updated_env) in
  let returning_env = if not (k_cur >= k_end) then pos_loop(fresh_env, var_name, k_cur + 1.0, k_end) else fresh_env in
  returning_env in

(*Decreasing loops*)
let rec neg_loop = function
  (env, var_name, k_cur, k_end) ->

  (*i_cur is the data type to insert into variable table*)
  let i_cur = Jast.JNumbert(k_cur) in

  (*Need to update actual value of i in the table and then update environment*)
  let new_variables = List.filter (fun (n, v) -> if (n = var_name) then false else true) env.drawing.variables in
  let updated_vars = new_variables @[(var_name, i_cur)] in
  let updated_drawing = {env.drawing with variables = updated_vars} in
  let updated_env = {env with drawing = updated_drawing} in

  (*Go through all statements*)
  let fresh_env = separate_statements_s(for_statements, updated_env) in
  let returning_env = if not (k_cur <= k_end) then neg_loop(fresh_env, var_name, k_cur - 1.0, k_end) else fresh_env in
  returning_env in

(*Process statements in the for loop*)
let new_env =
if (k_start <= k_end) then
    pos_loop (updated_env, i, k_start, k_end)
else
    neg_loop (updated_env, i, k_start, k_end)
in
(*Put last value of i into the stored variables*)
let old_variables_minus_i = List.filter (fun (n, v) -> if (n = i) then false else true) store_old_vars in
let old_vars_with_update_i = old_variables_minus_i @[(i, i_end)] in

let updated_drawing = {new_env.drawing with variables = old_vars_with_update_i} in
let updated_env = {new_env with drawing = updated_drawing} in

(*Process return statement*)
l Sast.Return(expr) ->
    let (new_env, eval_expr) = proc_expr env expr in
    let return_val = eval_expr in
    (*Signal for a function call to grab the return statement*)
    let return_name = "return" in
    let updated_vars = new_env.drawing.variables @ [(return_name, return_val)] in
    let updated_drawing = {mandala_list= new_env.drawing.mandala_list; variables = updated_vars; java_shapes_list= new_env.drawing.java_shapes_list;} in
    let updated_env = {drawing = updated_drawing; functions = new_env.functions} in
    updated_env

(*Process assignment*)
l Sast.Assign(vardecl, assign_expr) ->
    (* TODO: Finish this*)
    let (new_env, eval_expr) = proc_expr env assign_expr in
    let {skind = typ; svname = name;} = vardecl in
    (*Now get the variable*)
    let get_val_and_type = match eval_expr
    | Jast.JNumbert(eval_expr) -> Jast.JNumbert(eval_expr)
    | Jast.JBooleant(eval_expr) -> Jast.JBooleant(eval_expr)
    | Jast.JShapet(eval_expr) -> Jast.JShapet(eval_expr)
    | Jast.JGegot(eval_expr) -> Jast.JGegot(eval_expr)
    | Jast.JLayert(eval_expr) -> Jast.JLayert(eval_expr)
    | Jast.JMandalat(eval_expr) -> Jast.JMandalat(eval_expr)
    | Jast.JColort(eval_expr) -> Jast.JColort(eval_expr)
    | Jast.JVoid -> Jast.JVoid
    | Jast.JArrayType -> Jast.JArrayType
    | _ -> raise (Error("This expression does not have a supported type here!"))) in
let (n, v) = try List.
find (fun (s, _) -> s=name) env.
drawing.
variables with Not_found -> (name, get_val_and_type) in

let new_variables = List.
filter (fun (n, v) -> if (n = name) then false else
true) new_env.
drawing.
variables in

let updated_vars = new_variables @[(n, get_val_and_type)] in

let updated_drawing = { mandala_list= new_env.
drawing.
mandala_list; variables = updated_vars; java_shapes_list= new_env.
drawing.
java_shapes_list;} in

let updated_env = { drawing = updated_drawing; functions = new_env.
functions } in updated_env

| _ -> raise (Error("unsupported statement found"))

(*Add function declaration to our environment *)

let proc_func (env: environment):(Sast.
funcdecl -> environment) = function

my_func ->

let new_env = {
  drawing = env.
drawing;
  functions = env.
functions @ [my_func];
} in

new_env

(*Processes list of functions and keeps track of environment by recursively
processing individual functions*)

let rec separate_functions_s (funcs, env: Sast.
funcdecl list * environment) = match funcs
with [] -> env
| [func] -> proc_func env func
| func :: other_funcs ->
  let new_env = proc_func env func in
  separate_functions_s (other_funcs, new_env)

(*Given the entire SAST program, creates the resulting environment by processing
the entire program*)

let gen_java (env:environment):(Sast.
sprogram -> environment)= function

Sast.
SProg(s,f) ->
  (* Check if the program has at least one statement *)
  let x = List.
length s in
  if (x>0) then (
    (* Already reversed the statements in semantic when going from ast to jast,
so don’t need to reverse again *)
    let updated_env = separate_functions_s (f, env) in
    let updated_env = separate_statements_s (s, updated_env) in
    (* List.
map(fun stmt_part -> separate_statements_s prog_stmts env) in *)
    updated_env
  )

94
else
    raise (Error("A valid Mandala program must consist of at least one statement.")

(*Process a layer and load them all into the shapes structure in environment*)
let extract_shapes_from_layer (new_list:Jast.jShape list):(Jast.layer * float ->
Jast.jShape list) = function
    (my_layer, big_radius) ->

    let listed_shape = my_layer.shape in

    let count = my_layer.count in

    (* Goes through the layer and calculates position and size for all squares *)
    if (count >= 1 && listed_shape.geo = "square")
    then
        let rec loop = function
            (new_list, k) ->
            let rad_offset = my_layer.offset *. pi /. 180.0 in
            let my_angle = -1.0 *. rad_offset +. pi /. 2.0 -. (float_of_int k) *. 2.0 *. pi /. (float_of_int) my_layer.count in
            let x_pos = cos (my_angle) *. my_layer.radius in
            let y_pos = sin (my_angle) *. my_layer.radius in
            let extra_rotation =
                if (my_layer.angularshift = 1)
                then
                    (pi /. 2.0 -. my_angle) *. 180.0 /. pi
                else
                    0.0
            in
            let rotat = listed_shape.rotation +. extra_rotation in
            let color = listed_shape.color in
            let new_shape = Jast.Square listed_shape.size, x_pos, y_pos, rotat, color
            in
                if (k > 0) then
                    let updated_k = k - 1 in
                    loop (new_list@new_shape, updated_k)
                else
                    new_list@new_shape
            in
            loop(new_list, count - 1)

    (* Goes through the layer and calculates position and size for all circles *)
else if (count >= 1 && listed_shape.geo = "circle")
then
    let rec loop = function
        (new_list, k) ->
        let rad_offset = my_layer.offset *. pi /. 180.0 in
        let my_angle = -1.0 *. rad_offset +. pi /. 2.0 -. (float_of_int k) *. 2.0 *. pi /. (float_of_int) my_layer.count in

let x_pos = cos (my_angle) *. my_layer.radius in
let y_pos = sin (my_angle) *. my_layer.radius in
let color = listed_shape.color in
let new_shape = Jast.Circle(listed_shape.size, x_pos, y_pos, color) in
if (k > 0) then
  let updated_k = k - 1 in
  loop (new_list@new_shape, updated_k)
else
  new_list@new_shape
in
loop (new_list, count - 1)

(* Goes through the layer and calculates position and size for all triangles *)
else if (count >= 1 && listed_shape.geo = "triangle")
then
  let rec loop = function
  (new_list, k) ->
  let rad_offset = my_layer.offset *. pi /. 180.0 in
  let my_angle = -1.0 *. rad_offset +. pi /.2.0 -. (float_of_int k) *. 2.0 *. pi /. (float_of_int my_layer.count) in
  let x_pos = cos (my_angle) *. my_layer.radius in
  let y_pos = sin (my_angle) *. my_layer.radius in
  let extra_rotation =
    if (my_layer.angularshift = 1)
    then
      (pi /.2.0 -. my_angle) *. 180.0 /. pi
    else
      0.0
    in
  let rotat = listed_shape.rotation +. extra_rotation in
  let color = listed_shape.color in
  let new_shape = Jast.Triangle(listed_shape.size, x_pos, y_pos, rotat, color) in
  if (k > 0) then
    let updated_k = k - 1 in
    loop (new_list@new_shape, updated_k)
  else
    new_list@new_shape
  in
  loop (new_list, count - 1)
else
  raise (Error ("Only circles, squares, and triangles supported. Must have count at least 1. "))

(* Pulls out all layers and deals with max radius given a mandala *)
let get_layers = function
mandala ->
  let radius = mandala.max_layer_radius in
  let list_of_layers = mandala.list_of_layers in
let result = List.fold_left (fun a layer -> (layer, radius) :: a) [] list_of_layers in result

(* Checks mandala and outputs list of shapes generated. Only draws those with is_draw boolean *)
let process_mandala = function
  mandala ->
  if (mandala.is_draw = true) then
    let layers_with_radii = get_layers mandala in
    List.fold_left extract_shapes_from_layer [] layers_with_radii
  else
    []

(* Create empty initial environment *)
let empty_drawing_env=
  {
    mandala_list = []; variables = []; java_shapes_list = [];
  }

let empty_environment = {
  drawing = empty_drawing_env;
  functions = [];
}

(*Go through all mandalas and eventually convert into shape structures *)
let rec process_mandalas (mandalas, shapes, total:Jast.mandala list * Jast.jShape list * float) = match mandalas with [] -> shapes
  | [mandala] ->
    let new_mandala = {
      name = mandala.name;
      list_of_layers = mandala.list_of_layers;
      max_layer_radius = total;
      is_draw = mandala.is_draw
    } in
    (shapes @ process_mandala new_mandala)
  | mandala :: other_mandalas ->
    let new_mandala = {
      name = mandala.name;
      list_of_layers = mandala.list_of_layers;
      max_layer_radius = total;
      is_draw = mandala.is_draw
    } in
    (let new_shapes = process_mandala new_mandala in
      process_mandalas (other_mandalas, (shapes @ new_shapes), total))

(*Final conversion from Sast program to Jast program which runs all statements and moves into final structure *)
let actual_final_convert (check_program: Sast.sprogram): (Jast.javaprogram) =
  let env = empty_environment in
  (*Parse all statements and update environment*)
  let new_draw_env = gen_java env sast in
  let mandala_lists = new_draw_env.drawing.mandala_list in
  let all_mandalas = List.rev (List.fold_left (fun a (_, mandala) -> mandala :: a ) [] mandala_lists) in
  let total_radius = 0.0 in
  (*Get shapes from mandalas*)
  let all_shapes = process_mandalas (all_mandalas, [], total_radius) in
  (*All classes will have same name to allow java compilation*)
  let prog_name = Jast.CreateClass("Program") in
  Jast.JavaProgram(prog_name, all_shapes)

A.8 jast.mli

open Sast

(* Operators for jast *)
type op = Add | Sub | Mult | Div

(* Mandala specific types for java ast *)
type jmndlt =
  | Numbert
  | Booleant
  | Shapet
  | Geot
  | Layert
  | Mandalat
  | Arrayt
  | Colort

type jPrimative =
  | JBooleant of bool
  | JInt of int

type jValue =
  JValue of jPrimative

(* Create shape to store attributes of shape *)
type shape =
  { name: string;
    geo : string;
    size : float;
    color: string;
    rotation: float
  }
32 (* Create layer to define shape drawn in layer *)
    and layer = |
        name : string;
        radius : float;
        shape : shape;
        count : int;
        offset : float;
        angularshift : int
    |
32 (* Create mandala to store list of layers *)
    and mandala=[
        name : string;
        list_of_layers : layer list;
        max_layer_radius : float;
        is_draw: bool
    ]

and jdata_type =
    JInt of int
    | JVoid
    | JNumber of float
    | JBoolean of int
    | JShape of shape
    | JGeo of string
    | JLayer of layer
    | JMandala of mandala
    | JArray
    | JColor of string

32 (* Defines orientation of the shapes *)
    type jShape =
        Circle of float * float * float * string
    | Square of float * float * float * float * string
    | Triangle of float * float * float * float * string

32 (* drawing stores information about figures we will draw *)
    type drawing=[
        mandala_list : (string * mandala) list; (* figures to be drawn *)
        variables: (string * jdata_type) list; (* store variables and type *)
        java_shapes_list: jShape list; (* store shapes coordinates *)
    ]

    type java_shapes = |
        shape_list : shape list
    ]

32 (* Our environment stores a drawing *)
    type symbol_table = |
        draw_stmts : drawing
    ]
type javaClass = CreateClass of string

type javaprogram = JavaProgram of javaClass * jShape list

A.9 gen_java.ml

open Ast
open Sast
open Sast_to_jast
open Jast
open Semantic
open Lexing

exception Error of string

(*Generates jast by running through scanner, parser, semantic check, and sast_to_jast*)
let jast =
  let lexbuf = Lexing.from_channel stdin in
  let ast = Parser.program Scanner.token lexbuf in
  let sast = Semantic.semantic_check ast in
  Sast_to_jast.actual_final_convert sast

(*Generates primitive functions for drawing shapes*)
let draw_circle = function
  (radius, x, y, color) ->
  print_string "drawCircle(t,";
  print_float radius;
  print_string ",";
  print_float x;
  print_string ",";
  print_float y;
  print_string ",";
  print_string "\";
  print_string color;
  print_string \";
  print_string ");\n"

let draw_square = function
  (side, x, y, rotation, color) ->
  print_string "drawSquare(t,";
  print_float side;
  print_string ",";
  print_float x;
  print_string ",";
```plaintext
print_float y;
print_string ",";
print_float rotation;
print_string ",";
print_string "\"";
print_string color;
print_string "\"";
print_string ");\n"

let draw_triangle = function
  (side, x, y, rotation, color) ->
  print_string "  drawTriangle(t,";
  print_float side;
  print_string ",";
  print_float x;
  print_string ",";
  print_float y;
  print_string ",";
  print_float rotation;
  print_string ",";
  print_string color;
  print_string ");\n"

(*Match on shapes*)
let proc_shape = function
  Just.Circle(radius, x, y, color) ->
  draw_circle(radius, x, y, color)
  | Just.Square(side, x, y, rotation, color) ->
     draw_square(side, x, y, rotation, color)
  | Just.Triangle(side, x, y, rotation, color) ->
     draw_triangle(side, x, y, rotation, color)

(*Build primitive methods in java*)
let define_methods = function
  x -> if (x > 0) then (
      (* CIRCLES *)
      print_string "public static void drawCircle(Turtle t, double radius, double
x, double y, String color) {\n";
      print_string "  t.penColor(color);\n";
      print_string "  t.up(); t.setPosition(x, y + radius); t.down();\n";
      print_string "  for (int i = 0; i < 360; i++) {\n";
      print_string "    t.forward(radius * 2 * Math.PI / 360);\n";
      print_string "  }\n";
    print_string "}\n";
      (* SQUARES *)
      print_string "public static void drawSquare(Turtle t, double size, double x
, double y, double rotation, String color) {\n";
      print_string "  t.penColor(color);\n";
```

101
print_string = t.up()\n; 
print_string = t.setPosition(x – size/2, y + size/2);\n; 
print_string = rotation = rotation % 90;\n; 
print_string = double radius = Math.sqrt(2) * size / 2;\n; 
print_string = if (rotation > 0) t.left(45);\n; 
print_string = for (int i = 0; i < rotation; i++) \n{\n}; 
print_string = t.forward(radius * 2 * Math.PI / 360);\n; 
print_string = t.right(1);\n; 
print_string = \n; 
print_string = t.down();\n; 
print_string = if (rotation > 0) t.right(45);\n; 
print_string = int turn = 90;\n; 
print_string = t.forward(size); t.right(turn);\n; 
print_string = t.forward(size); t.right(turn);\n; 
print_string = t.forward(size); t.right(turn);\n; 
print_string = t.forward(size); t.right(turn);\n; 
print_string = t.left(rotation);\n; 
print_string = }\n; 

(* TRiANGLES *)
print_string = \n; 
print_string = public static void drawTriangle(Turtle t, double size, double x, double y, double rotation, String color) {\n; 
print_string = t.penColor(color);\n; 
print_string = t.up(); t.setPosition(x – size/2, y + Math.sqrt(3)*size/6);\n; 
print_string = rotation = rotation % 120;\n; 
print_string = double radius = size / Math.sqrt(3);\n; 
print_string = if (rotation > 0) t.left(60);\n; 
print_string = for (int i = 0; i < rotation; i++) \n{\n}; 
print_string = t.forward(radius*2*Math.PI / 360); t.right(1);\n; 
print_string = \n; 
print_string = t.down(); if (rotation > 0) t.right(60); int turn = 120;\n; 
print_string = t.forward(size); t.right(turn);\n; 
print_string = t.forward(size); t.right(turn);\n; 
print_string = t.forward(size); t.right(turn);\n; 
print_string = t.left(rotation);\n; 
print_string = }\n; 

else print_string = ""
}

(* Default classname is set to "Program")
let get_string_of_classname = function
  Jast.createClass(string_of_classname) -> string_of_classname
)

(* Final function that parses Jast program and generates code *)
let gen_java_final = function
  Jast.JavaProgram(classname, shapes) ->
    let l = List.length shapes in
    let string_of_classname = get_string_of_classname classname in
print_string "public class ";
print_string string_of_classname; (*Print the string of class name for class header*)
print_string "\n\n";

(*Only defines method if we need to use them to create shapes*)
define_methods l;
print_string "public static void main(String[] args) {\n\n";
(*Only defines method if we need to use them to create shapes*)
define_methods l;
print_string "public static void main(String[] args) {

";
print_string "Turtle t = new Turtle();\n"
print_string "t.hide();\n"
print_string "t.speed(0);\n"

(*Go through and print all the shapes*)
if (l > 0) then
(List.map proc_shape shapes)
else if (l == 0) then
(*Just draw a dot if we have no shapes*)
(print_string "t.setPosition(0,0);\n t.dot();\n"
 List.map proc_shape shapes)
else
(List.map proc_shape shapes);

print_string "}\n\n}\n"

let _ =
gen_java_final jast

---

### A.10 Makefile

default: run semantic sast_to_jast
run: scanner parser semantic sast_to_jast gen_java
   ocamlc -o run scanner.cmo parser.cmo semantic.cmo sast_to_jast.cmo gen_java.cmo

gen_java: sast
   ocamlc -c gen_java.ml

sast_to_jast_o: scanner parser semantic sast_to_jast
   ocamlc -o semantic sast_to_jast parser.cmo scanner.cmo semantic.cmo
   sast_to_jast.cmo

sast_to_jast: jast sast
   ocamlc -c sast_to_jast.ml

semantic_o: scanner parser semantic
   ocamlc -o semantic parser.cmo scanner.cmo semantic.cmo

103
A.11 regression_tester.sh

```bash
# Automated regression testing
#!/bin/bash

# Author: Harsha Vemuri

# COMPONENTS
preprocessor="";run="";j_file=Program.java"
warnings="";compare=""

# BUILDING
  cd ../../../compiler
  echo "" > $warnings
```
make 2> $warnings

18 cd ../tests/fullstack

# GET ALL MANDALA FILES
mandala_files=$(find suite -name \*.mandala)

22 for m_file in $mandala_files
24 do
26 # PASSING TESTS
28 if [[ $m_file == *"p_"* ]]
30 then
32 # PREPROCESSING
34 python $preprocessor $m_file
36 p_file=$(find suite -name \*.proc)
38 # JAVA GENERATION
./$run < $p_file > "suite/Program.java"
40 # JAVA COMPILATION
42 cd suite
44 javac $j_file
46 #COMPARING
48 t_filename=${m_file%.*}
50 compareTo="solutions/"$t_filename
52 diff=$(python $compare Program.java $compareTo)
54 if [[ $diff -eq 0 ]]; then
56 echo "Output Correct: [y]" for $m_file"
58 else
60 echo "Output Correct: [n]" for $m_file"
62 fi
64 # TESTS THAT FAIL
66 else
68 t_filename=${m_file%.*}
70 t_filename="suite/solutions/"$t_filename".txt"
72 err=$(<${t_filename})
74 if [[ $err == "ERROR" ]]
76 then
78 echo "Output Correct: [y]" for $m_file"
80 else
82 echo "Output Correct: [n]" for $m_file"
84 fi
86 cd suite
# CLEANING
rm -f *.proc
mv Turtle.java Turtle.java.keep
rm -f *.java
mv Turtle.java.keep Turtle.java
rm -f *.class
cd ..
done

A.12 mandala.sh

# compile and execute a mandala program
#!/bin/bash
filename="src/$1"
preprocessor="compiler/preprocessor.py"
run="compiler/run"
j_file="Program.java"
exe="Program"
warnings="tests/fullstack/warnings.txt"

# BUILDING
echo "" > $warnings
make 2> $warnings

# PREPROCESSING
python $preprocessor $filename
p_file=$filename".proc"

# JAVA GENERATION
./run < $p_file > "$src/$j_file &

# JAVA COMPILATION
cd src
javac $j_file

# EXECUTION
java $exe

# CLEANING
rm -f *.proc
mv Turtle.java Turtle.java.keep
A.13 compare.py

```python
#!/usr/bin/python
# Author: Harsha Vemuri

import sys

hashmap = {}
hashmap2 = {}

try:
f = file(sys.argv[1], 'r') # generated program
f2 = file(sys.argv[2], 'r') # expected output
except IOError:
    print -1
    sys.exit()

def main():
    build_hash_1()
    build_hash_2()
    if hashmap == hashmap2:
        print 0 # equal
    else:
        print -1 # unequal

# hashmap of lines from first file
def build_hash_1():
    for line in f:
        line = line.strip()
        if line in hashmap:
            hashmap[line] += 1
        else:
            hashmap[line] = 1

# hashmap of lines from second file
def build_hash_2():
    for line in f2:
        line = line.strip()
        if line in hashmap2:
            hashmap2[line] += 1
        else:
```

```
```python
    hashmap2[line] = 1

if __name__ == "__main__":
    main()
```
Appendix B

Mandalas

The following pages illustrate interesting Mandalas we generated during development.

The final page shows an image of a 3D printed Mandala. To demonstrate the future possibilities of what can be created with Mandala, we used one of the jpg images we generated and converted it to a .stl file to 3D print.