Photoshop--

A drawing and animation language for aspiring programmers

Created by
Gilbert Feig (grf2108)
David Figueroa (df2442)
Alana Ramjit (amr2235)

Class
Programming Languages and Translators
Columbia University
Fall 2014

Professor
Stephen A. Edwards

Teaching Assistant
Vaibhav Jagannathan

Table of Contents

1. Introduction
    1.1. Overview
    1.2. Name Origin
    1.3. What Constitutes a Photoshop-- Program
2. Language Tutorial
    2.1. Getting Started
    2.2. Compiling and Running
    2.3. Drawing Shapes
    2.4. Manipulating Shape Location, Size, and Appearance
    2.6. If/Else Statements and Blocks
    2.7. Next Steps
3. Photoshop-- Language Reference Manual
    3.1. Lexical Conventions
        3.1.1 Tokens
        3.1.2 Comments
        3.1.3 Identifiers
        3.1.4 Keywords
        3.1.5 Constants
    3.2. Syntax
        3.2.1 Keyword Glossary
        3.2.2 Basic Types
    3.3. Expressions
        3.3.1 Operators
        3.3.2 Block Calls
    3.4 Declarations
        3.4.1 Declarations of int and bool
        3.4.2 Declarations of shape objects rect and ellipse
        3.4.3 Declarations of functions
    3.5 Statements and Execution
        3.5.1 Statements and Expressions
        3.5.2 Execution
    3.6 Execution
4. Project Plan
    4.1. Process of Planning, Specification, Development, and Testing
    4.2. Programming Style Guide
    4.3. Project Timeline
    4.4. Team Roles and Responsibilities
    4.5. Development Environment, Tools, and Languages
    4.6. Project Log
5. Architectural Design

5.1. Translator Components
5.2. Component Interfaces
5.3. Individual Contributions
6. Test Plan
6.1. Overview
6.2. Test Suites
6.3. Test Cases
6.4. Individual Contributions
7. Lessons Learned
7.1. Group Lessons Learned
7.2. Lessons Learned by Gil Feig
7.3 Lessons Learned by David Figueroa
7.4. Lessons Learned by Alana Ramjit
7.4. Advice for Future Teams
8. Appendix
8.1. scanner.mll
8.2. parser.mly
8.3. ast.ml
8.4. semantic.ml
8.5. codegen.ml
8.6. pmmc.ml

1. Introduction

1.1. Overview

One of the most troublesome aspects of learning to program is the lack of visualization. After "hello world," programs quickly become more complex, adding steps in the production of each output. Photoshop-- poses a solution to the widening gap between number of lines of code and the amount of feedback given for each.

Graphical user interfaces can provide immediate responses to changes in data, but are not within the scope of what new developers are capable. Nonetheless, the visual feedback from developing code that produces an animation is a rewarding introduction to programming. Photoshop-- is a graphics and animation programming language that focuses on rapid learning and ease of use. Developers create shape objects that are automatically displayed on the canvas at runtime.

A shape-manipulation block is automatically run sixty times per second. After each update, the canvas is redrawn. This approach is simple, yet powerful, enabling the most basic of static images to complex physically realistic animations.

1.2. Name Origin

The name Photoshop-- was chosen carefully with the developer target in mind. Firstly, the name makes the language feel more familiar by using the name of a common piece of software. At the same time, Photoshop provides context, indicating that this is a graphical language. Finally, the decrementer "--" is a play on C++, implying simplicity of use and less complex functionality.

1.3. What Constitutes a Photoshop-- Program

A file becomes a compilable Photoshop-- program simply by implementing the `drawloop` block. Implementing shapes, basic types, blocks, and statements within blocks and `drawloop` are all optional.

2. Language Tutorial

This tutorial was created with the beginning programmer in mind. It starts from the basics, introducing the structure of a Photoshop-- program. As you progress, you'll learn how to make custom rectangles and ellipses. Lastly, you'll see how simple it is to implement all the logic necessary to animate your shapes around the canvas.

2.1. Getting Started

As mentioned in Section 1, the simplest program that the Photoshop-- compiler can compile and run is nothing more than the required animation block, indicated by the keyword `drawloop`:

```
drawloop {
    ~ This is a comment… it is ignored by the compiler ~
}
```

This block contains a comment, enclosed within two ~ symbols. The compiler ignores comments, so it's as if it weren't there. After you have written this, choose a name for it and save it as a .pmm file. That's it, you've just written your first Photoshop-- program!

2.2. Compiling and Running

To see your first animation, run:

```
./pmmc <your-file-name.pmm>
```

Where <your-file-name.pmm> is replace by the name of your file. Unsurprisingly, this program doesn't do much more than display a blank canvas. The next sections will introduce you to some basic techniques for drawing shapes and altering their location, size, and appearance.

2.3. Drawing Shapes

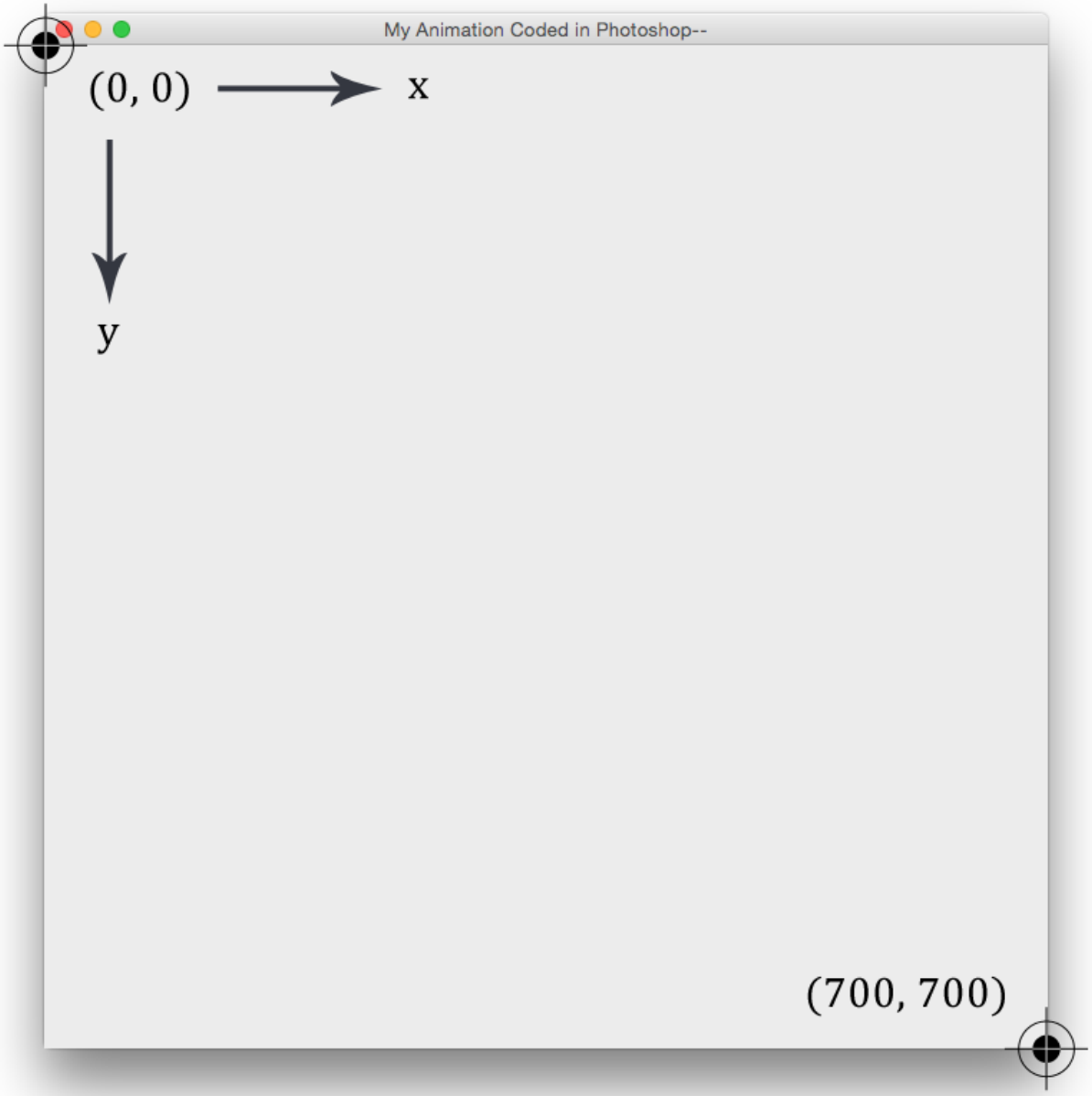
Photoshop-- supports drawing both rectangles and ellipses. To place shapes shapes your canvas, add lines with the following format to the very top of your code file:

```
<shape-type> <name> = <x-origin>, <y-origin>, <width>, <height>,
<color>;

~ Example: Blue rectangle named myRect with the origin (400, 400),
a width of 100, and a height of 200 ~

rectangle myRect = 400, 400, 100, 200, blue;
```

There are two options for `<shape-type>`: `ellipse` and `rectangle`. Name is a string that is entirely up to you. The integer values you specify for `<x-origin>`, `<y-origin>`, `<width>`, and `<height>` will be used to place and size the shape according to a coordinate system where (0, 0) represents the top left corner, and (700, 700) represents the bottom right:



Lastly, for `<color>`, provide the red, green, and blue values for the desired color from 0-255 in the format `(<red>, <green>, <blue>)`. As a means for quick development, instead of providing the comma-separated RGB colors in parentheses, you may use the keywords `red`, `green`, or `blue` alone.

2.4. Manipulating Shape Location, Size, and Appearance

A shape's x-origin, y-origin, width, height, and fill color can all be changed after it has been created. There are several convenient ways to do this using action notation:

```
~ Change the origin of myShape to (100, 200)~
put myShape at 100, 200;

~ Move myShape up by 10 pixels ~
move myShape up 10;
```

In the second example, notice the keyword `up`. This may be replaced by any of the other three supported directions: `down`, `left`, and `right` .

You can also perform the same actions using dot syntax, where you refer to the property with the format `<shape-name>.<property-name>`. The compiler recognizes the properties `x`, `y` `width`, `height`, and `color`.

The following example performs the exact same modifications as those shown in action notation above, but through the use of dot syntax:

```
~ Change the origin of myShape to (100, 200)~
myShape.x = 100;
myShape.y = 200;

~ Move myShape up by 10 pixels ~
myShape.y = myShape.y - 10;
```

2.5. Your First Animation

Recall from the language introduction that *every* Photoshop-- program must have a drawloop.

```
drawloop {

}
```

This is what makes the animation magic happen. Everything contained within the braces after drawloop will happen sixty times per second. For your first animation, you will make a circle in the top left move to the bottom right as it grows in size.

First, we need to create the circle. We must carefully choose the point of origin to ensure that it is drawn in the correct starting position on the canvas.

```
circle myFirstCircle = 0, 0, 100, 100, blue;

drawloop {

}
```

When you compile and run, you'll see the blue circle in the top-left corner of the canvas, but it remains static. This is to be expected, as drawloop enables you to perform animations, but yours is empty. Let's fix that by moving the shape down one and right one. At the same time, let's increase its width and height.

```
ellipse myFirstCircle = 0, 0, 100, 100, blue;

drawloop {
      ~ Move the circle right and down ~
      move myFirstCircle down 1;
      move myFirstCircle right 1;

      ~ Increase the size of the circle ~
      myFirstCircle.width = myFirstCircle.width + 1;
      myFirstCircle.height = myFirstCircle.height + 1;
}
```

Compile and run, and you should see your first animation! It looks great, but you may have noticed one big issue -- the shape eventually moves off the canvas. For the next and final portion of the tutorial, you will learn the concepts necessary to make a ball bounce back and forth between the left and right edges.

2.6. If/Else Statements and Blocks

In order to make the ball bounce, we need to somehow detect when it has hit a wall. While there is no magical solution to this problem, there is a way to do it.

A group of code within braces, `{ }`, is called a *block*. This should look familiar, as your first animation has a drawloop, which is a special type of block. When the program starts running through the block, it continues until it reaches the end.

Using an if/else statement, we can execute blocks of code based on a condition. In this case, we want to know if the ball has reached the right side or the left, and based on that, set the direction of the ball:

```
ellipse mySecondCircle = 0, 0, 100, 100, blue;
int velocityX = 1;

drawloop {
      ~ Set the velocity of the ball ~
      if (mySecondCircle.x < 0) {
            ~ The ball is at the left wall; make it move right ~
            velocityX = 1;
      } else if (mySecondCircle.x + mySecondCircle.width > 700) {
            ~ The ball is at the right wall; make it move left ~
            velocityX = -1;
      }

      mySecondCircle.x = mySecondCircle.x + velocityX;
}
```

Run this code and you'll see the ball move from left to right indefinitely, bouncing off the walls. There are two new concepts here. Firstly, a variable is declared at the top with the syntax `int veloxityX = 1;`. This tells the compiler that velocityX is a variable that stores an integer value, and initially holds the value 1. This value can be changed, and that's just how the bouncing is accomplished.

`if (mySecondCircle.x > 0)` determines whether the x-origin of the circle is less than the x-value of the left wall. If so, it sets velocityX to positive 1, forcing it to move to the right. If the first block doesn't run, then the second if statement condition is evaluated. In a similar manner, it checks if the right side of the ball (it's x-origin added to its width) is at an x-value greater than that of the right wall. If this is the case, it changes the velocity to -1, bouncing the ball back to the left. Lastly, velocityX is summed with the x-origin of the circle, moving its origin in the desired direction.

2.7. Next Steps

You're now a Photoshop-- pro! These tutorials have given you the skills necessary to create just about any animation. Though you have learned a good chunk of the language, there is still more you can add to enhance your code, increase organization, and utilize other features. The following Language Reference Manual provides a list of these. Some notable topics to consider reading over are:

- ➔ Creating and running custom blocks
- ➔ The bool (true/false) variable type
- ➔ Styling Photoshop-- code (see section 4.2)
- ➔ Challenge: implement a ball bouncing with gravity

3. Photoshop-- Language Reference Manual

3.1. Lexical Conventions

3.1.1 Tokens

There are three main categories of tokens not mentioned in the other lexical conventions: whitespace, block separators, and semicolons. Whitespace includes the tab, newline, and space characters. Block separators are the '{' and '}' symbols that enclose the component statements of a block. Semicolons indicate the end of an expression, and also indicate that the expression is a statement.

Whitespace is used to separate tokens which can be identifiers, keywords, constants, operators, and comments.

3.1.2 Comments

Comments are strings that are ignored by the compiler. Indicate the start with a single '~' character. Comments may be several lines in length, and are terminated by another single '~' character.

3.1.3 Identifiers

Identifiers are a series of letters and/or digits, always beginning with a letter. The maximum length is 20 characters.

3.1.4 Keywords

Keywords are reserved for special use cases, and may not be used as identifiers or anything else unintended. These consist of:

```
at              green           rect
background      if              red
block           int             right
blue            left            rotate
bool            drawloop        run
down            main            true
ellipse         move            up
else            put             while
false           print
```

An explanation of what each of these individual words mean is found in Section 2.

3.1.5 Constants

The only constants supported are base decimal integer constants. All integers are signed and may be stored in variables of type int only.

3.2. Syntax

3.2.1 Keyword Glossary

at - used in combination with “put” and an identifier to designate an x, y coordinate at which to move a shape.
background - used to change the background color with a tuple
block - declares a function. Must be followed by an ID and a set of braces that group together the statements to be executed when that block is called.
blue - primary color constant representing (0, 0, 255)
bool - declarator for a type that holds either true or false. Used to construct conditional statements.
down - increases the y coordinate of a shape object
drawloop - the special block; any statements in this block will be executed at rate of 60 times per second to enable animation
ellipse - basic round shape type
else - statements to be executed if a preceding “if” clause condition is not true
false - not true, used to construct negative statements in conditionals
green - primary color constant representing (0, 255, 0)
if - indicates a block that are to be executed once if the condition following it is true
int - declarator for a type that holds integer values
left - decreases the x coordinate of a shape object
main - the block of code that is run in the output
move - offsets a shape object in the given direction by the given amount
put - sets the origin of a shape object at the provided x and y values
rect - basic rectangular shape type
red - primary color constant representing (255, 0, 0)
right - increases the x coordinate of a shape object
rotate - offsets a shape object by the given angle
true - not false, used to construct positive statements in conditions
up - increases the y coordinate of a shape object
while - type of loop that continues as long as the given conditional is true

3.2.2 Basic Types

- There are four fundamental types: `bool`, `int`, `rect`, and `ellipse`.
- The `bool` type may only take values `true` and `false`.
- The `int` type may take any signed integer values.
- The `rect` and `ellipse` types have the following properties which may be set or retrieved using dot notation:
    - `x` - the `x` coordinate of the top left corner of the containing frame (defaults to 0)
    - `y` - the `y` coordinate of the top left corner of the containing frame (defaults to 0)
    - `width` - the width of the shape

- `height` - the height of the shape
- `color` - the color of the shape; ex: Accessing properties

```
myRect.x = 100; ~Sets frame x coordinate position to 100~
int i = myrect.x; ~myRect.x returns 100 and sets i to 100~
```

3.3. Expressions

3.3.1 Operators

Multiplicative Operators

The multiplicative operators are * and / and group from left-to-right.

multiplicative-expression:

*multiplicative-expression * int*

multiplicative-expression / int

The operands of * and / must be of type `int`.

The * operator denotes multiplication and returns a product of the operands as an `int`.

The / operator denotes division. If the divisor does not equally divide the dividend, then the integer quotient is returned.

Additive Operators

The additive operators are `+` and `-` and group from left-to-right.

additive-expression:

multiplicative-expression

additive-expression + *multiplicative-expression*

additive-expression - *multiplicative-expression*

The + operator denotes addition and returns the sum of the operands.

The - operator denotes subtraction and returns the difference of the operands.

Relational Operators

The relational operators <, <=, >, >= evaluate to either true or false and group left-to-right such that `x>y>z` is parsed as `(x>y)>z`.

relational-expression:

additive-expression

relational-expression < *additive-expression*

relational-expression <= *additive-expression*

relational-expression > *additive-expression*

relational-expression >= *additive-expression*

The operators `<` (less), <= (less or equal), > (greater), and >= (greater or equal) return true if the relation is true and false otherwise. The return type is of type bool.

Equality Operators

The equality operators == and != evaluate to either true or false

equality-expression:

relational-expression
equality-expression == relational-expression
equality-expression != relational-expression

The operators == (equal) and != (not equal) return true if the equality comparison is true and false otherwise. The return type is of type bool.

Animation Operator

The animation operators `move` (`left`, `right`, `up`, and `down`) and `put at` modify the `x` or `y` position of objects.

animation-expression:
move identifier left additive-expression
move identifier right additive-expression
move identifier up additive-expression
move identifier down additive-expression
put identifier at additive-expression additive-expression

The animation operators change the location of objects. The `move` operator changes the `x` position of an object with the keywords `left` and `right` and changes the `y` position of an object with the keywords `up` and `down`. The `put` operator changes both the `x` and `y` position of an object to the position x,y after `at`.

3.3.2 Block Calls

A `block` is called by calling `run` followed by the name of the `block`.
ex.

```
run myBlock; ~function named myFunction is being called~
```

3.4 Declarations

Declarations of an identifier can be associated with one of the four basic types or a function. All declarations must be accompanied by an initial value or declaration. All variables and functions must be declared before they are referenced.

3.4.1 Declarations of `int` and `bool`

Identifiers of type `int` are declared as `int <identifier> =` with an assignment to a constant integer value. Identifiers or type `bool` are declared in the same manner `bool <identifier> =` but are assigned to either `false`, `true`, or the value of some boolean expression.
ex:

```
int i = 0;
bool b = true;
```

3.4.2 Declarations of shape objects `rect` and `ellipse`

Shape objects `rect` and `ellipse` are declared in a similar format of `<type> <identifier> =` . They must be assigned initial properties in order: an x position, a y position, a height, a width, and a color (using keywords red, green, blue, or an rgb triple). These properties are separated by commas.
ex.

```
rect r = 10, 10, 10, 10, red;
ellipse e = 10,10,10,10,(255,255,0);
```

3.4.3 Declarations of functions

Declarations of functions are specified by the keyword “block” followed by an identifier and curly braces containing the group of statements associated with that block. Statements are discussed in section 5.1 of this reference manual.
Note that the block `drawloop` is a special `block`, and therefore does not take the `block` declarator.

3.5 Statements and Execution

3.5.1 Statements and Expressions

An expression is a syntactically valid variable declaration, boolean or relational evaluation, arithmetic expression, function call, or animator operation on a shape as discussed in section 3.

A statement is any expression that is terminated with a semi-colon. Expressions such as variable declarations, function calls, and animator operations must always end with a semi-colon and are always statements. Relational evaluations, or arithmetic expressions may evaluated as part of a declaration or as a condition within an **if** block. **if, block, while,** and **drawloop,** are all followed by braces that must group together a set of statements.
ex.

```
block myBlock { <stmt-list> }
drawloop { <stmt-list> }
```

3.5.2 Execution

Execution begins at the top of the file. The file must include the special block `drawloop` in order to compile. All statements within the `drawloop` block will be executed continuously at a rate of 60 frames per second, enabling animation simulation.

All variables and blocks must be declared before they are referenced in a non-declarative statement following it in the execution path.

3.6 Execution

Any variable declared within a block, understood to mean a group of statements between braces, is only visible within those braces. If a block is nested within another block, and a variable is declared with the same name in the inner block as a variable in the outer block, then the inner block copy takes precedence and the outer block copy is rendered invisible.

Global variables are visible within any block but must be declared at the beginning of the file before any block declarations.

4. Project Plan

4.1. Process of Planning, Specification, Development, and Testing

The initial plan for our language was to generate a semantically-checked intermediate C or C++ source file. The set of recognized key words was relatively small, as our language pared down programming concepts to the essentials needed to write descriptive algorithms.

The outlines for the scanner, ast, and parser were completed in late October, and early November fine-tuning the syntax for our language. Naturally, some of our initial plans for the syntax were modified to help ease shift/reduce and reduce/reduce conflicts. Examples of this include introducing comma separators between initializers for shapes, the inclusion of the tuple format for RGB values, and the drawloop syntax.

Researching various C and C++ graphics library pinpointed GTK+ for hardware rendering paired with Cairo for drawing and SDL as the two most viable C-compatible libraries. However, the code generation developer had issues installing and using these libraries, and given that Java has generally universal hardware rendering capabilities and is a familiar language, the decision was made to use Java on our backend.

The first major concern for testing was to make sure our ast was correctly parsing our language and translating every keyword and possible valid statement construction into the correct corresponding Java code. During this and the next phase of testing, we assume a beneficent and always correct user. The next phase was ensuring that we could correctly express simple algorithms and mathematical expressions, such as in the collision test file. Finally, the last phase is making sure that malicious or incorrect input is handled safely, handled by the error test cases.

4.2. Programming Style Guide

The conventions for our programming followed the basics rules of ocaml. Specific conventions are as follows.

Names of Variables

- ➔ Names of variables are concise but with some easy-to-recognize pattern for what they represent. For instance, “whitespace” is too long but “ws” is an appropriate name for a tokenizing regex as it is easy to recall what it might stand for.

- ➔ The members of structs usually start with the letter of the struct type to differentiate it from the general type, i.e. “color” represents an RGB tuple int * int * int but the corresponding member of the shape type is scolor.

➔ Duplicate variable names were allowed for things that represented similar objectives, such as f_decl and func_decl as long their usage was clear in context.

Names of Functions

➔ Function names are generally more descriptive and longer because they are used less frequently.

➔ Code generating functions that were at some point to be injected in the Java source all include "string" somewhere in their name. Validity checks are indicated usually by "is" or "check'.

Spacing and Indentation

➔ Used liberally. Ocaml tends to have layers of nested functions but in otherwise, code should be uniformly indented and well-spaced. We as a group believe that readability should be valued.

4.3. Project Timeline

Rough goals and milestones:
Mid November - solidify language syntax and write scanner
End of November - finish front-end, pick a graphics library, and set up files for code generation
Early December - finish code generation, produce working programs and an executor, and begin writing tests
Mid December - implement semantic checking, resolve test issues, and write demos

4.4. Team Roles and Responsibilities

Lana - Originally the systems and language guru, but also team manager in assigning tasks and general nagging about due dates. Had initial idea for a visual educational language, designed front-end of compiler, wrote large part of ast, parser, scanner, makefile, shell script, and pmmc.ml, and general background research and architecture.

David - Originally test writer and stayed pretty true to testing. Generally pointed out inconsistencies in syntax and animation ideas, responsible for test suite, helped resolve errors in parser/ast, wrote ast-traversal code-generative functions in the latter half of the ast, demo programs, contributed to makefile and basic semantic checker.

Gil - Originally manager; wrote codegen, a basic semantic checker, and the Java backend using Swing with threaded animations. Responsible for the entirety of the static code and code-injection functions. Also wrote the demo programs like "hello," and contributed extra functionality to parser, ast, and scanner, not included in original language scope, such as rotation and background.

4.5. Development Environment, Tools, and Languages

Each of used personal laptops, so for Gil and David these were Mac OS and for Lana, Linux's Ubuntu 14.04. We all have our personal favorite text editor (Sublime vs vim). We used git version control software to push updates and merge code. The original repository is hosted on Lana's github at https://github.com/alanamramjit/Photoshop--. Informally, Facebook messenger group chats played a large role in group communication and task distribution. The front-end is of course coded in ocaml, the backend in Java using the Swing graphics library, and the swift compiler is a bash script.

4.6. Project Log

Early November

The skeleton for our code was created in mid-November and design ideas discussed.

Mid November

Small and mostly boilerplate portions of code completed. Researching typical syntax for front-end, hashing out what the syntax and goals of our language should look like in finer granularity.

Late November

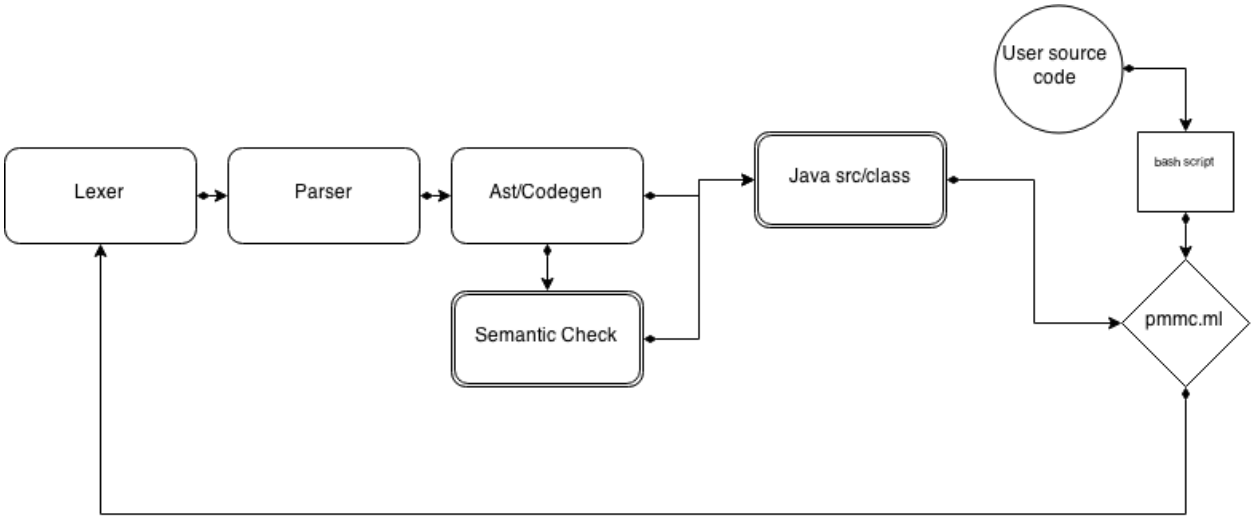
A basic, complete first attempt at a parser, ast, and scanner. Largely incorrect but needed testing. Research into C/C++ graphics libraries.

Early December

- Completed the front end, and also began debugging the ast
- Completed the static Java code
- Implemented a broader range of functionality
- Created an executor to generate combined files
- Code injection and shell script compile Java code with options
- Added basic semantic checking and tests

5. Architectural Design

5.1. Translator Components



5.2. Component Interfaces

The flow of a PMM program happens as follows: running ./pmmc <name of file> causes the .ppm source file to be scanned in tokens, parsed into the ast and semantically checked at a very basic level. Then the codegenerator traverses the semantic tree, translating each node into a corresponding Java statement which is injected into a larger, static Java string with a predefined graphics framework. This is written to a .java source file. The pmmc file, which is really just a bash script, then compiles and executes the generated class file.

5.3. Individual Contributions

Lana

Implemented grammar construction (parsing/lexing, ast), and execution architecture.

David

Created test suites and contributed to scanner, parser, and ast.

Gil

Implemented codegen, semantic checking, and Java graphics. Also contributed to scanner, parser, and ast with additional functionality.

6. Test Plan

6.1. Overview

Our initial tests started off simple and short to ensure that we had the ability to compile a simple program with the minimum amount of code to compile a test. Our goal was then to build on those tests to ensure that each of our keywords and key functionalities worked individually as well as when put together for more complex programs.

6.2. Test Suites

We have divided out tests into two suites: `test_programs` and `sample_programs`.
The `test_programs` directory includes tests of basic functionality in which we isolate the functionality we want to focus on. We also created tests that we purposefully fail to ensure we cannot compile and output generated code with errors. These tests are numbered with the prefix `error`.
The sample_programs directory holds some of our early tests in which we wanted to manipulate created objects from the moment our compiler started to work. These include some more interesting programs such as a `collision`, `explosion`, and our most complex program `hello` featuring a bouncing multicolored hello created solely from the ellipses and rectangles that can be created in Photoshop--.

6.3. Test Cases

Below are three test cases: `simple_rect`, `blocks`, and `hello`.

The `simple_rect` test displays a simple red rectangle in the middle of the screen and shows how simple it can be to put something on the screen with Photoshop--.

`simple_rect`

```
~Simple Rectangle~

rect r = 300,300,100,100,red;

drawloop {}
```

`simple_rect Java output:`

```
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.Rectangle;
import java.awt.geom.Ellipse2D;
import java.awt.geom.Rectangle2D;
import java.awt.geom.AffineTransform;
```

```
import java.util.ArrayList;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.SwingUtilities;
public class PSMMAnimator {
	public static void main(String[] args) {
		SwingUtilities.invokeLater(new Runnable() {
			public void run() {
				createAndDisplayGUI();
			}
		});
	}
	public static void createAndDisplayGUI() {
		JFrame frame = new JFrame("My Animation Coded in
Photoshop--");
		frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
		PSMMAnimatedPanel panel = new PSMMAnimatedPanel();
		frame.add(panel);
		frame.pack();
		frame.setVisible(true);
		Thread t = new Thread(panel);
		t.start();
	}
}
class PSMMAnimatedPanel extends JPanel implements Runnable {
	private static final long serialVersionUID = 1L;
	public ArrayList<Shape> shapes;
Shape r = new Shape(new Rectangle(300, 300, 100, 100), new
Color(255, 0, 0), Shape.Type.RECTANGLE);
	public PSMMAnimatedPanel() {
		shapes = new ArrayList<Shape>();
		// Create and add shapes
shapes.add(r);
	}
public void drawloop() {

}
	@Override
	public void run() {
		while (true) {
			recalculateShapes();
			repaint();
			try {
				Thread.sleep(1000 / 60);
			} catch (InterruptedException e) {
			}
		}
```

```
	}
	private void recalculateShapes() {
		// Do stuff to shapes
		drawloop();
	}
	public Dimension getPreferredSize() {
		return new Dimension(700, 700);
	}
	@Override
	public void paintComponent(Graphics g) {
		super.paintComponent(g);
		Graphics2D g2 = (Graphics2D) g;
		for (Shape shape : shapes) {
			AffineTransform old = g2.getTransform();
shape.angle %= 360; g2.rotate(Math.toRadians(shape.angle));
g2.setTransform(old);
			g2.setPaint(shape.color);
			Rectangle frame = shape.frame;
			if (shape.type == Shape.Type.ELLIPSE) {
				g2.fill(new Ellipse2D.Double(frame.x, frame.y,
frame.width, frame.height));
			} else if (shape.type == Shape.Type.RECTANGLE) {
				g2.fill(new Rectangle2D.Double(frame.x,
frame.y, frame.width, frame.height));
			}
		}
	}
}
class Shape {
	public enum Type {
		RECTANGLE, ELLIPSE
	}
	public Rectangle frame;
	public Color color;
	public Type type;
	public int angle;
	public Shape(Rectangle frame, Color color, Type type) {
		this.frame = frame;
		this.color = color;
		this.type = type;
		this.angle = 0;
	}
}
```

The `blocks` test displays is a slightly more complex program that tests the ability to create and run blocks. We create two blocks, one called `checkAndIncrement` and the other `putInQuadrant2or4`. By running them consecutively after 30 runs of the `drawloop`, the check will change the boolean value which changes the location of the rectangle in the put block.

`blocks:`

```
~Test blocks~

rect r = 10,10,330,330,red;
bool b = true;
int i = 0;

block checkAndIncrement {
      if(i == 30) {
            if(b) {
                  b = false;
            } else {
                  b = true;
            }
            i = 0;
      }
      i = i+1;
}

block putInQuadrant2or4 {
      if(b) {
            ~Quadrant 4~
            put r at 360,360;
      } else {
            ~Quadrant 2~
            put r at 10,10;
      }
}

drawloop {
      run putInQuadrant2or4;
      run checkAndIncrement;
}
```

`blocks Java output:`

```
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Graphics;
```

```
import java.awt.Graphics2D;
import java.awt.Rectangle;
import java.awt.geom.Ellipse2D;
import java.awt.geom.Rectangle2D;
import java.awt.geom.AffineTransform;
import java.util.ArrayList;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.SwingUtilities;
public class PSMMAnimator {
     public static void main(String[] args) {
          SwingUtilities.invokeLater(new Runnable() {
               public void run() {
                    createAndDisplayGUI();
               }
          });
     }
     public static void createAndDisplayGUI() {
          JFrame frame = new JFrame("My Animation Coded in
Photoshop--");
          frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
          PSMMAnimatedPanel panel = new PSMMAnimatedPanel();
          frame.add(panel);
          frame.pack();
          frame.setVisible(true);
          Thread t = new Thread(panel);
          t.start();
     }
}
class PSMMAnimatedPanel extends JPanel implements Runnable {
     private static final long serialVersionUID = 1L;
     public ArrayList<Shape> shapes;
Shape r = new Shape(new Rectangle(10, 10, 330, 330), new Color(255,
0, 0), Shape.Type.RECTANGLE);
boolean b = true;
int i = 0;
     public PSMMAnimatedPanel() {
          shapes = new ArrayList<Shape>();
          // Create and add shapes
shapes.add(r);
     }
public void checkAndIncrement() {
     if (i == 30)
{
if (b)
{
b = false;}
```

```
else
{
b = true;}

i = 0;}

     i = i + 1;
}
public void putInQuadrant2or4() {
     if (b)
{
r.frame.x = 360; r.frame.y = 360;}
else
{
r.frame.x = 10; r.frame.y = 10;}

}
public void drawloop() {
     putInQuadrant2or4();
     checkAndIncrement();
}
     @Override
     public void run() {
          while (true) {
               recalculateShapes();
               repaint();
               try {
                    Thread.sleep(1000 / 60);
               } catch (InterruptedException e) {
               }
          }
     }
     private void recalculateShapes() {
          // Do stuff to shapes
          drawloop();
     }
     public Dimension getPreferredSize() {
          return new Dimension(700, 700);
     }
     @Override
     public void paintComponent(Graphics g) {
          super.paintComponent(g);
          Graphics2D g2 = (Graphics2D) g;
          for (Shape shape : shapes) {
               AffineTransform old = g2.getTransform();
shape.angle %= 360; g2.rotate(Math.toRadians(shape.angle));
g2.setTransform(old);
```

```
                g2.setPaint(shape.color);
                Rectangle frame = shape.frame;
                if (shape.type == Shape.Type.ELLIPSE) {
                     g2.fill(new Ellipse2D.Double(frame.x, frame.y,
frame.width, frame.height));
                } else if (shape.type == Shape.Type.RECTANGLE) {
                     g2.fill(new Rectangle2D.Double(frame.x,
frame.y, frame.width, frame.height));
                }
           }
     }
}
class Shape {
     public enum Type {
          RECTANGLE, ELLIPSE
     }
     public Rectangle frame;
     public Color color;
     public Type type;
     public int angle;
     public Shape(Rectangle frame, Color color, Type type) {
          this.frame = frame;
          this.color = color;
          this.type = type;
          this.angle = 0;
     }
}
```

Below are three test cases: `simple_rect`, `blocks`, and `hello`.
Below is a more complex program, `hello`, that tests more complex capabilities of Photoshop--.

`hello:`

```
~ PHYSICS VARIABLES~
int height = 200;
int minY = 50;
int maxY = minY + height;
int minX = 100;
int maxX = 100;
int velocityY = 1;
int velocityX = 1;

~ BACKGROUND ~
int bgCounter = 474;

~ H ~
```

```
rect hLeft = minX, minY, 30, height, red;
rect hMiddle = minX, minY + 85, 90, 30, red;
rect hRight = minX + 60, minY, 30, height, red;

~ E ~
rect eLeft = minX + 100, minY, 30, height, (255, 127, 0);
rect eTop = minX + 100, minY, 90, 30, (255, 127, 0);
rect eMiddle = minX + 100, minY + 85, 90, 30, (255, 127, 0);
rect eBottom = minX + 100, maxY - 30, 90, 30, (255, 127, 0);

~ L ~
rect firstLLeft = minX + 200, minY, 30, height, (255, 255, 0);
rect firstLBottom = minX + 200, maxY - 30, 90, 30, (255, 255, 0);

~ L ~
rect secondLLeft = minX + 300, minY, 30, height, green;
rect secondLBottom = minX + 300, maxY - 30, 90, 30, green;

~ O ~
ellipse oOuter = minX + 400, minY, 90, height, blue;
ellipse oInner = minX + 430, minY + 30, 30, height - 60, (236, 236,
236);

drawloop {
     velocityY = velocityY + 1;

     ~ Bounce HELLO if it is at the bottom of the screen and make
it stick a bit ~
     if (hLeft.y + hLeft.height >= 700) {
          if (velocityY > 30) {
               velocityY = 30;
          }
          velocityY = velocityY - 2 * velocityY;
     }

     ~ Horizontal motion ~
     if (oOuter.x + oOuter.width >= 700) {
          velocityX = -1;
     } else if (hLeft.x <= 0) {
          velocityX = 1;
     }

     run updatePositions;
     run updateBackgroundColor;
}

~ Updates shape positions ~
```

```
block updatePositions {
     put hLeft at hLeft.x + velocityX, hLeft.y + velocityY;
     put hMiddle at hMiddle.x + velocityX, hMiddle.y + velocityY;
     put hRight at hRight.x + velocityX, hRight.y + velocityY;

     put eLeft at eLeft.x + velocityX, eLeft.y + velocityY;
     put eTop at eTop.x + velocityX, eTop.y + velocityY;
     put eMiddle at eMiddle.x + velocityX, eMiddle.y + velocityY;
     put eBottom at eBottom.x + velocityX, eBottom.y + velocityY;

     put firstLLeft at firstLLeft.x + velocityX, firstLLeft.y +
velocityY;
     put firstLBottom at firstLBottom.x + velocityX, firstLBottom.y
+ velocityY;

     put secondLLeft at secondLLeft.x + velocityX, secondLLeft.y +
velocityY;
     put secondLBottom at secondLBottom.x + velocityX,
secondLBottom.y + velocityY;

     put oOuter at oOuter.x + velocityX, oOuter.y + velocityY;
     put oInner at oInner.x + velocityX, oInner.y + velocityY;
}

~ Updates the background color ~
block updateBackgroundColor {
     ~ Set the background color ~
     if (bgCounter > 575) {
          background red;
          oInner.color = red;
     } else if (bgCounter > 550) {
          background (255, 127, 0);
          oInner.color = (255, 127, 0);
     } else if (bgCounter > 520) {
          background (255, 255, 0);
          oInner.color = (255, 255, 0);
     } else if (bgCounter > 500) {
          background green;
          oInner.color = green;
     } else if (bgCounter > 475) {
          background blue;
          oInner.color = blue;
     } else {
          background (236, 236, 236);
          oInner.color = (236, 236, 236);
     }
```

```
      ~ Decrement bgCounter 600 -> 0 and then back to 600 ~
      if (bgCounter < 0) {
            bgCounter = 600;
      } else {
            bgCounter = bgCounter - 1;
      }
}
```

hello Java output:

```
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.Rectangle;
import java.awt.geom.Ellipse2D;
import java.awt.geom.Rectangle2D;
import java.awt.geom.AffineTransform;
import java.util.ArrayList;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.SwingUtilities;
public class PSMMAnimator {
      public static void main(String[] args) {
            SwingUtilities.invokeLater(new Runnable() {
                  public void run() {
                        createAndDisplayGUI();
                  }
            });
      }
      public static void createAndDisplayGUI() {
            JFrame frame = new JFrame("My Animation Coded in
Photoshop--");
            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            PSMMAnimatedPanel panel = new PSMMAnimatedPanel();
            frame.add(panel);
            frame.pack();
            frame.setVisible(true);
            Thread t = new Thread(panel);
            t.start();
      }
}
class PSMMAnimatedPanel extends JPanel implements Runnable {
      private static final long serialVersionUID = 1L;
      public ArrayList<Shape> shapes;
```

```
int height = 200;
int minY = 50;
int maxY = minY + height;
int minX = 100;
int maxX = 100;
int velocityY = 1;
int velocityX = 1;
int bgCounter = 474;
Shape hLeft = new Shape(new Rectangle(minX, minY, 30, height), new
Color(255, 0, 0), Shape.Type.RECTANGLE);
Shape hMiddle = new Shape(new Rectangle(minX, minY + 85, 90, 30),
new Color(255, 0, 0), Shape.Type.RECTANGLE);
Shape hRight = new Shape(new Rectangle(minX + 60, minY, 30,
height), new Color(255, 0, 0), Shape.Type.RECTANGLE);
Shape eLeft = new Shape(new Rectangle(minX + 100, minY, 30,
height), new Color(255, 127, 0), Shape.Type.RECTANGLE);
Shape eTop = new Shape(new Rectangle(minX + 100, minY, 90, 30), new
Color(255, 127, 0), Shape.Type.RECTANGLE);
Shape eMiddle = new Shape(new Rectangle(minX + 100, minY + 85, 90,
30), new Color(255, 127, 0), Shape.Type.RECTANGLE);
Shape eBottom = new Shape(new Rectangle(minX + 100, maxY - 30, 90,
30), new Color(255, 127, 0), Shape.Type.RECTANGLE);
Shape firstLLeft = new Shape(new Rectangle(minX + 200, minY, 30,
height), new Color(255, 255, 0), Shape.Type.RECTANGLE);
Shape firstLBottom = new Shape(new Rectangle(minX + 200, maxY - 30,
90, 30), new Color(255, 255, 0), Shape.Type.RECTANGLE);
Shape secondLLeft = new Shape(new Rectangle(minX + 300, minY, 30,
height), new Color(0, 255, 0), Shape.Type.RECTANGLE);
Shape secondLBottom = new Shape(new Rectangle(minX + 300, maxY -
30, 90, 30), new Color(0, 255, 0), Shape.Type.RECTANGLE);
Shape oOuter = new Shape(new Rectangle(minX + 400, minY, 90,
height), new Color(0, 0, 255), Shape.Type.ELLIPSE);
Shape oInner = new Shape(new Rectangle(minX + 430, minY + 30, 30,
height - 60), new Color(236, 236, 236), Shape.Type.ELLIPSE);
     public PSMMAnimatedPanel() {
          shapes = new ArrayList<Shape>();
          // Create and add shapes
shapes.add(hLeft);
shapes.add(hMiddle);
shapes.add(hRight);
shapes.add(eLeft);
shapes.add(eTop);
shapes.add(eMiddle);
shapes.add(eBottom);
shapes.add(firstLLeft);
shapes.add(firstLBottom);
shapes.add(secondLLeft);
```

```
shapes.add(secondLBottom);
shapes.add(oOuter);
shapes.add(oInner);
     }
public void drawloop() {
     velocityY = velocityY + 1;
     if (hLeft.frame.y + hLeft.frame.height >= 700)
{
if (velocityY > 30)
{
velocityY = 30;}

velocityY = velocityY - 2 * velocityY;}

     if (oOuter.frame.x + oOuter.frame.width >= 700)
{
velocityX = -2;}
else
if (hLeft.frame.x <= 0)
{
velocityX = 1;}

     updatePositions();
     updateBackgroundColor();
}
public void updatePositions() {
     hLeft.frame.x = hLeft.frame.x + velocityX; hLeft.frame.y =
hLeft.frame.y + velocityY;
     hMiddle.frame.x = hMiddle.frame.x + velocityX; hMiddle.frame.y
= hMiddle.frame.y + velocityY;
     hRight.frame.x = hRight.frame.x + velocityX; hRight.frame.y =
hRight.frame.y + velocityY;
     eLeft.frame.x = eLeft.frame.x + velocityX; eLeft.frame.y =
eLeft.frame.y + velocityY;
     eTop.frame.x = eTop.frame.x + velocityX; eTop.frame.y =
eTop.frame.y + velocityY;
     eMiddle.frame.x = eMiddle.frame.x + velocityX; eMiddle.frame.y
= eMiddle.frame.y + velocityY;
     eBottom.frame.x = eBottom.frame.x + velocityX; eBottom.frame.y
= eBottom.frame.y + velocityY;
     firstLLeft.frame.x = firstLLeft.frame.x + velocityX;
firstLLeft.frame.y = firstLLeft.frame.y + velocityY;
     firstLBottom.frame.x = firstLBottom.frame.x + velocityX;
firstLBottom.frame.y = firstLBottom.frame.y + velocityY;
     secondLLeft.frame.x = secondLLeft.frame.x + velocityX;
secondLLeft.frame.y = secondLLeft.frame.y + velocityY;
     secondLBottom.frame.x = secondLBottom.frame.x + velocityX;
```

```
secondLBottom.frame.y = secondLBottom.frame.y + velocityY;
     oOuter.frame.x = oOuter.frame.x + velocityX; oOuter.frame.y =
oOuter.frame.y + velocityY;
     oInner.frame.x = oInner.frame.x + velocityX; oInner.frame.y =
oInner.frame.y + velocityY;
}
public void updateBackgroundColor() {
     if (bgCounter > 575)
{
setBackground(new Color(255, 0, 0));
oInner.color = new Color(255, 0, 0);}
else
if (bgCounter > 550)
{
setBackground(new Color(255, 127, 0));
oInner.color = new Color(255, 127, 0);}
else
if (bgCounter > 520)
{
setBackground(new Color(255, 255, 0));
oInner.color = new Color(255, 255, 0);}
else
if (bgCounter > 500)
{
setBackground(new Color(0, 255, 0));
oInner.color = new Color(0, 255, 0);}
else
if (bgCounter > 475)
{
setBackground(new Color(0, 0, 255));
oInner.color = new Color(0, 0, 255);}
else
{
setBackground(new Color(236, 236, 236));
oInner.color = new Color(236, 236, 236);}

     if (bgCounter < 0)
{
bgCounter = 600;}
else
{
bgCounter = bgCounter - 1;}

}
     @Override
     public void run() {
          while (true) {
```

```
                recalculateShapes();
                repaint();
                try {
                     Thread.sleep(1000 / 60);
                } catch (InterruptedException e) {
                }
           }
     }
     private void recalculateShapes() {
           // Do stuff to shapes
           drawloop();
     }
     public Dimension getPreferredSize() {
           return new Dimension(700, 700);
     }
     @Override
     public void paintComponent(Graphics g) {
           super.paintComponent(g);
           Graphics2D g2 = (Graphics2D) g;
           for (Shape shape : shapes) {
                AffineTransform old = g2.getTransform();
shape.angle %= 360; g2.rotate(Math.toRadians(shape.angle));
g2.setTransform(old);
                g2.setPaint(shape.color);
                Rectangle frame = shape.frame;
                if (shape.type == Shape.Type.ELLIPSE) {
                     g2.fill(new Ellipse2D.Double(frame.x, frame.y,
frame.width, frame.height));
                } else if (shape.type == Shape.Type.RECTANGLE) {
                     g2.fill(new Rectangle2D.Double(frame.x,
frame.y, frame.width, frame.height));
                }
           }
     }
}
class Shape {
     public enum Type {
           RECTANGLE, ELLIPSE
     }
     public Rectangle frame;
     public Color color;
     public Type type;
     public int angle;
     public Shape(Rectangle frame, Color color, Type type) {
           this.frame = frame;
           this.color = color;
           this.type = type;
```

```
            this.angle = 0;
      }
}
```

6.4. Individual Contributions

Gil

Created several demo programs such as “hello,” contributed to finding and fixing parsing and logic issues.

David

Implemented all of the tests above, finding many issues and determining the appropriate course of action.

7. Lessons Learned

7.1. Group Lessons Learned

One of the main lessons we learned is just how powerful OCaml can really be. While its highly-functional syntax appeared unfamiliar, we grew to really love just how ideal it is for pattern matching and building a compiler. We also learned the importance of starting early when given such a large assignment.

7.2. Lessons Learned by Gil Feig

I gained a true understanding for the way compilers work, by at least contributing to every step along the compilation path. Furthermore, I realized just how important communication is, as this project is not easily separable. Though there are components that can be written and grouped in separate files, every piece of the system depends on every other. With this, I learned how critical it is to use more of Git's rich functionalities. Working with a group on such a large project requires absolute cooperation, and I realized the importance of committing and pushing often, merging properly, and using branches.

7.3 Lessons Learned by David Figueroa

The major lesson I learned while working on this project was how important it is to start early and seek help from the TAs as they are a great resource. I believe I am walking away from this course with a true understanding of the process of writing and the structure of a compiler. I have gained very valuable experience working on this large project in a group and have learned the many advantages of using Git and working with others who are familiar with the tool.

7.4. Lessons Learned by Alana Ramjit

-You really can fix most problems by adding another level of indirection
-An understanding of the capabilities of ocamlyacc for implementing CFGs
-When faced with a huge task and not knowing where to start, start somewhere
-Also, ask for help early on. Being more proactive about setting up weekly meetings with Vaibhav and checking in with the professor and TAs about progress and what steps to take next in retrospect would have made a world of difference and would have been much more useful early-on.
-the interdependency of all the parts--one "finished" and working section of code could easily become unfinished if someone else updated or added another file. Ex: building a working, type-matching top-level to read compiler flags could be reverted to a much earlier stage by the addition of a semantic checking file, and many sections of the AST/parser were continually under revision as other group members contributed to their sections.

7.4. Advice for Future Teams

Start as early as possible. Even if that means reading through past years' projects just to get an understanding of the scope of the assignment, do not wait until the last couple weeks.

Communicate at least weekly throughout the semester, and determine who will work on what, and how those pieces will be connected.

8. Appendix

8.1. scanner.mll

```
{ open Parser }

let letter = ['a'-'z' 'A' - 'Z']
let digit = ['0'-'9']
let identifier = (letter)(letter | digit)*
let stringy = (letter | digit)*
let ws = [' ''\t' '\r' '\n']

rule token = parse
     [' ' '\t' '\r' '\n' ]                    { token lexbuf }
     | '~'                                    { comment lexbuf }
     | ';'                                    { SEMICOLON }
     | '{'                                    { LBRACE }
     | '}'                                    { RBRACE }
     | '('                                    { LPAREN }
     | ','                                    { COMMA }
     | ')'                                    { RPAREN }
     | '='                                    { ASSIGN }
     | '<'                                    { LTHAN }
     | '>'                                    { GTHAN }
     | '!'                                    { NOT }
     | '*'                                    { TIMES}
     | '+'                                    { PLUS }
     | '-'                                    { MINUS }
     | "=="                                   { EQ }
     | '+'                                    { PLUS }
     | '-'                                    { MINUS }
     | "=="                                   { EQ }
     | "!="                                   { NEQ }
     | "<="                                   { LEQ }
     | ">="                                   { GEQ }
     | ".angle"                               { GETANGLE}
     | ".x"                                   { GETX }
     | ".y"                                   { GETY }
     | ".width"                               { WIDTH }
     | ".height"                              { HEIGHT }
     | ".color"                               { GETCOLOR }
     | "at"                                   { AT }
     | "background"                           { BACKGROUND }
```

```
      | "block"                                        { BLOCK }
      | "blue"                                       { BLUE }
      | "bool"                                       { BOOL }
      | "down"                                       { DOWN }
      | "drawloop"                                   { DRAWLOOP }
      | "ellipse"                                    { ELLIPSE }
      | "else"                                       { ELSE }
      | "false"                                      { FALSE }
      | "green"                                      { GREEN }
      | "if"                                         { IF }
      | "int"                                        { INT }
      | "left"                                       { LEFT }
      | "main"                                       { MAIN }
      | "move"                                       { MOVE }
      | "print"                                      { PRINT }
    | "put"                                          { PUT }
      | "rect"                                       { RECT }
      | "red"                                        { RED }
      | "right"                                      { RIGHT }
      | "rotate"                                     { ROTATE }
    | "run"                                          { RUN }
      | '"'(letter | digit)+(letter | digit | ws)*'"' as str
{ STRING(str) }
    | "true"                                         { TRUE }
      | "right"                                      { RIGHT }
      | "run"                                        { RUN }
      | "true"                                       { TRUE }
      | "up"                                         { UP }
      | "while"                                      { WHILE }
      | identifier as lxm                            { ID(lxm) }
      | eof                                          { EOF }
      | digit+ as lxm                 { LITERAL(int_of_string lxm) }
      | _ as char                                      { raise
(Failure("Illegal Character: " ^ Char.escaped char)) }

and comment = parse
      '~'                                    { token lexbuf }
      | _                                    { comment lexbuf }
      | eof                                  { raise (Failure
("Unclosed Comment: All comments must have both opening and closing
squiggles" )) }
```

8.2. parser.mly

```
%{ open Ast %}

%token SEMICOLON LBRACE RBRACE LPAREN RPAREN EQ LTHAN GTHAN
%token NOT TIMES NEQ LEQ GEQ AT BLOCK BLUE DOWN INT NOELSE EOF
%token ELSE FALSE GREEN IF LEFT LOOP MAIN MOVE PUT ELLIPSE COMMA
%token RED RIGHT RUN TRUE UP WHILE ASSIGN BOOL RECT DRAWLOOP DOT
%token GETX GETY WIDTH HEIGHT GETCOLOR GETANGLE PLUS MINUS PRINT
QUOTE
%token BACKGROUND ROTATE
%token <string> ID
%token <int> LITERAL
%token <string> STRING

%nonassoc NOELSE
%nonassoc ELSE
%nonassoc COMMA
%right ASSIGN
%left EQ NEQ
%left LTHAN GTHAN LEQ GEQ
%left PLUS MINUS
%left TIMES DIVIDE
%left MOVE


%start program
%type <Ast.program> program

%%

program:
                      {[], []}
  | program vdecl     { $2::fst $1, snd $1  }
  | program fdecl     { fst $1, $2 :: snd $1}



fdecl:
    BLOCK ID LBRACE stmt_list RBRACE
    {
      {
        fname = $2;
        body = List.rev $4;
      }
```

```
    }

  | DRAWLOOP LBRACE stmt_list RBRACE
    {
      {
        fname = "drawloop";
        body = List.rev $3;
      }
    }

color:
    RED           { (255, 0, 0) }
  | GREEN         { (0, 255, 0) }
  | BLUE          { (0, 0, 255) }
  | LPAREN LITERAL COMMA LITERAL COMMA LITERAL RPAREN { ($2, $4,
$6) }

shape:
    RECT { Rect }
  | ELLIPSE { Ellipse }

vdecl:
  shape ID ASSIGN expr COMMA expr COMMA expr COMMA expr COMMA color
SEMICOLON
    { Shape(
      {
        stype = $1;
        sname = $2;
        x = $4;
        y = $6;
        w = $8;
        h = $10;
        scolor = $12;
      }
    )}
  | INT ID                          {Def(Int, $2, Literal(0) )}
  | BOOL ID                         {Def(Bool, $2, Literal(0) )}
  | INT ID ASSIGN expr SEMICOLON    {Def(Int, $2, $4)}
  | BOOL ID ASSIGN expr SEMICOLON   {Def(Bool, $2, $4)}

stmt_list:
                      { [] }
  | stmt_list stmt    { $2 :: $1 }

stmt:
    expr SEMICOLON                              { Expr($1) }
  | LBRACE stmt_list RBRACE                     { Block(List.rev $2)
```

```
}
  | IF LPAREN expr RPAREN stmt %prec NOELSE   { If($3, $5,
Block([])) }
  | IF LPAREN expr RPAREN stmt ELSE stmt      { If($3, $5, $7) }
  | WHILE LPAREN expr RPAREN stmt             { While($3, $5) }
  | RUN ID SEMICOLON                          { Run($2) }
  | PUT ID AT expr COMMA expr SEMICOLON       { Put($2, $4, $6) }
  | MOVE ID LEFT expr SEMICOLON         { Animator($2, Left, $4) }
  | MOVE ID RIGHT expr SEMICOLON        { Animator($2, Right, $4) }
  | MOVE ID UP expr SEMICOLON           { Animator($2, Up, $4) }
  | MOVE ID DOWN expr SEMICOLON         { Animator($2, Down, $4) }
  | ROTATE ID expr SEMICOLON        { Animator($2, Degoffset, $3) }
  | vdecl                                     { Vdecl($1) }
  | PRINT STRING SEMICOLON                    { Print($2) }
  | BACKGROUND color SEMICOLON                { Background($2) }

expr:
    LITERAL                   { Literal($1) }
  | MINUS LITERAL             { Literal(-1 * 2) }
  | ID                        { Id($1) }
  | TRUE                      { Boolean("true")}
  | FALSE                     { Boolean("false")}
  | expr PLUS expr            { Binop($1, Add,   $3) }
  | expr MINUS expr           { Binop($1, Sub,   $3) }
  | expr TIMES expr           { Binop($1, Mult,  $3) }
  | expr DIVIDE expr          { Binop($1, Div,   $3) }
  | expr EQ expr              { Binop($1, Equals, $3) }
  | expr NEQ expr             { Binop($1, Neq,   $3) }
  | expr LTHAN expr           { Binop($1, Less,  $3) }
  | expr LEQ expr             { Binop($1, Leq,   $3) }
  | expr GTHAN expr           { Binop($1, Greater,  $3) }
  | expr GEQ expr             { Binop($1, Geq,   $3) }
  | ID ASSIGN expr            { Vassign($1, $3)}
  | LPAREN expr RPAREN        { $2 }
  | color                     { Rgb($1) }
  | ID GETX                   { Get($1, X) }
  | ID GETY                   { Get($1, Y) }
  | ID WIDTH                  { Get($1, Width) }
  | ID HEIGHT                 { Get($1, Height) }
  | ID GETCOLOR               { Get($1, Color) }
  | ID GETANGLE               { Get($1, Angle) }
  | ID GETX ASSIGN expr       { Set($1, X, $4) }
  | ID GETY ASSIGN expr       { Set($1, Y, $4) }
  | ID WIDTH ASSIGN expr      { Set($1, Width, $4) }
  | ID HEIGHT ASSIGN expr     { Set($1, Height, $4) }
  | ID GETCOLOR ASSIGN color  { Set($1, Color, Rgb($4)) }
  | ID GETANGLE ASSIGN expr   { Set($1, Angle, $4) }
```

8.3. ast.ml

```
type op = Add | Sub | Mult | Div | Equals | Neq | Less | Leq | Geq
| Greater

type animop = Left | Right | Up | Down | Degoffset

type s_type = Rect | Ellipse

type sdesc = Width | Height | X | Y | Color | Angle

type color = int * int * int

type expr =
    Literal of int
  | Id of string
  | Vassign of string * expr
  | Binop of expr * op * expr
  | Get of string * sdesc
  | Set of string * sdesc * expr
  | Rgb of (color)
  | Boolean of string

type p_type = Int | Bool

type shape = {
  stype: s_type;
  sname: string;
  x: expr;
  y: expr;
  w: expr;
  h: expr;
  scolor: color;
}

type v_decl =
    Shape of shape
  | Def of p_type * string * expr

type stmt =
    Block of stmt list
  | Expr of expr
  | If of expr * stmt * stmt
  | While of expr * stmt
  | Run of string
```

```
  | Animator of string * animop * expr
  | Put of string * expr *  expr
  | Vdecl of v_decl
  | Print of string
  | Background of color

type f_decl = {
  fname: string;
  body: stmt list;
}

type program = v_decl list * f_decl list

type prog_funcs =
     Var of v_decl
   | Fun of f_decl

(* Returns a string representation of the given binary operation *)
let string_of_op = function
    Add -> "+"
  | Sub -> "-"
  | Mult -> "*"
  | Div -> "/"
  | Equals -> "=="
  | Neq -> "!="
  | Less -> "<"
  | Leq -> "<="
  | Greater -> ">"
  | Geq -> ">="

(* Returns a string ID suffix for the given property *)
let string_of_prop = function
    X -> ".frame.x"
  | Y -> ".frame.y"
  | Width -> ".frame.width"
  | Height -> ".frame.height"
  | Color -> ".color"
  | Angle -> ".angle"

(* Returns a string for the given basic type *)
let string_of_type = function
    Int -> "int"
  | Bool -> "boolean"

(* Returns a string ID suffix for the given move direction *)
let string_of_direction = function
    Left -> ".frame.x -="
```

```
  | Right -> ".frame.x +="
  | Up -> ".frame.y -="
  | Down -> ".frame.y +="
  | Degoffset -> ".angle +="

(* Returns a string for the given color *)
let string_of_color col =
  let (r, g, b) = col in
    "new Color(" ^ string_of_int r ^ ", " ^ string_of_int g ^ ", "
^ string_of_int b ^ ")"

(* Returns a string identifier for the given shape type *)
let string_of_stype = function
    Rect -> "Shape.Type.RECTANGLE"
  | Ellipse -> "Shape.Type.ELLIPSE"

(* Returns a string for the given expression *)
let rec string_of_expr = function
    Literal(l) -> string_of_int l
  | Id(id) -> id
  | Boolean(b) -> b
  | Binop(e1, op, e2) ->
      string_of_expr e1 ^ " " ^ string_of_op op ^ " " ^
string_of_expr e2
  | Vassign(e1, e2) -> e1 ^ " = " ^ string_of_expr e2
  | Rgb(col) -> string_of_color col
  | Get(id, prop) -> id ^ string_of_prop prop
  | Set(id, prop, ex) -> id ^ string_of_prop prop ^ " = " ^
string_of_expr ex

(* Returns a string for the given variable declaration *)
let string_of_vdecl = function
    Def(ty, id, ex) -> string_of_type ty ^" " ^ id ^ " = " ^
string_of_expr ex ^ ";\n"
  | Shape(shape_defn) -> "Shape " ^ shape_defn.sname ^ " = new
Shape(new Rectangle(" ^ string_of_expr shape_defn.x ^ ", " ^
string_of_expr shape_defn.y ^ ", " ^ string_of_expr shape_defn.w ^
", " ^ string_of_expr shape_defn.h ^ "), " ^ string_of_color
shape_defn.scolor ^ ", " ^ string_of_stype shape_defn.stype ^
");\n"

(* Returns a string for the given statement *)
let rec string_of_stmt = function
    Expr(ex) -> string_of_expr ex ^";"
  | Block(s) -> "{\n"^String.concat "\n" (List.map string_of_stmt
s) ^ "}\n"
  | If(ex, s, Block([])) -> "if (" ^ string_of_expr ex ^")\n"
```

```
^string_of_stmt s
  | If(ex, s1, s2) -> "if (" ^ string_of_expr ex
^")\n"^string_of_stmt s1 ^ "else\n" ^ string_of_stmt s2
  | While(ex, s) -> "while (" ^ string_of_expr ex ^")\n"^
string_of_stmt s
  | Run(id) -> id^"();"
  | Put(id, ex1, ex2) -> id ^ ".frame.x = " ^ string_of_expr ex1 ^
"; " ^ id ^ ".frame.y = " ^ string_of_expr ex2 ^ ";"
  | Animator(id, dir, ex) -> id ^ string_of_direction dir ^
string_of_expr ex ^ ";"
  | Vdecl(var) -> string_of_vdecl var ^ ";"
  | Print(str) -> "System.out.println("^str^");"
  | Background(color) -> "setBackground(" ^ string_of_color color ^
");"

(* Returns a string add statement for the given v_decl()->Shape *)
 let string_of_add = function
  Shape(shape_defn) -> "shapes.add(" ^ shape_defn.sname ^ ");\n"
  | Def(_, _, _) -> ""

(* Returns a string for the given function declaration *)
let string_of_func f_decl = "public void " ^ f_decl.fname ^ "()
{\n\t" ^ String.concat "\n\t" (List.map string_of_stmt f_decl.body)
^ "\n}"

(* Returns a single string with the program's contents *)
let program_string (gl, funs) =
  String.concat "" (List.map string_of_vdecl (List.rev gl)) ^"\n" ^
String.concat "\n" (List.map string_of_func (List.rev funs)) ^ "\n"

(* Returns a tuple of strings in the form (v_decls, add_stmts,
f_decls) *)
let program_string_split (gl, funs) =
  (String.concat "" (List.map string_of_vdecl (List.rev gl)),
String.concat "" (List.map string_of_add (List.rev gl)),
String.concat "\n" (List.map string_of_func (List.rev funs)) ^
"\n")
```

8.4. semantic.ml

```
open Ast

module TypeMap = Map.Make
( struct
     type t = string
     let compare x y = Pervasives.compare x y
```

```
end )

let type_map = ref TypeMap.empty

let print_map_entry id typ =
     print_string(id ^ " = " ^ typ)

let print_map =
     TypeMap.iter print_map_entry !type_map

(* Returns a string representation of the given binary operation *)
let string_of_op = function
    Add -> "+"
  | Sub -> "-"
  | Mult -> "*"
  | Div -> "/"
  | Equals -> "=="
  | Neq -> "!="
  | Less -> "<"
  | Leq -> "<="
  | Greater -> ">"
  | Geq -> ">="

(* Returns a string ID suffix for the given property *)
let string_of_prop = function
    X -> ".frame.x"
  | Y -> ".frame.y"
  | Width -> ".frame.width"
  | Height -> ".frame.height"
  | Color -> ".color"
  | Angle -> ".angle"

(* Returns a string for the given basic type *)
let string_of_type = function
    Int -> "int"
  | Bool -> "boolean"

(* Returns a string ID suffix for the given move direction *)
let string_of_direction = function
    Left -> ".frame.x -="
  | Right -> ".frame.x +="
  | Up -> ".frame.y -="
  | Down -> ".frame.y +="
  | Degoffset -> ".angle +="

let is_valid_rgb rgb =
  if rgb > 255 || rgb < 0
```

```
    then false
  else
    true

(* Returns a string for the given color *)
let string_of_color col =
  let (r, g, b) = col in
    if (is_valid_rgb r) && (is_valid_rgb g) && (is_valid_rgb b)
      then "new Color(" ^ string_of_int r ^ ", " ^ string_of_int g
^ ", " ^ string_of_int b ^ ")"
    else
      raise(Failure("Invalid color! Colors must be one of: red,
green, blue, or (0-255, 0-255, 0-255)"))

(* Returns a string identifier for the given shape type *)
let string_of_stype = function
    Rect -> "Shape.Type.RECTANGLE"
  | Ellipse -> "Shape.Type.ELLIPSE"

(* Returns a string for the given expression *)
let rec string_of_expr = function
    Literal(l) -> string_of_int l
  | Id(id) -> id
  | Boolean(b) -> b
  | Binop(e1, op, e2) ->
      string_of_expr e1 ^ " " ^ string_of_op op ^ " " ^
string_of_expr e2
  | Vassign(e1, e2) -> e1 ^ " = " ^ string_of_expr e2
  | Rgb(col) -> string_of_color col
  | Get(id, prop) -> id ^ string_of_prop prop
  | Set(id, prop, ex) -> id ^ string_of_prop prop ^ " = " ^
string_of_expr ex

(* Returns a string for the given variable declaration *)
let string_of_vdecl = function
    Def(ty, id, ex) -> if TypeMap.mem id !type_map
                              then raise(Failure("Redeclaration
of variable named " ^ id))
                          else
                              type_map := TypeMap.add id
(string_of_type ty) !type_map;
                              (* print_string("Printing Map:");
                              print_map; *)
                              string_of_type ty ^" " ^ id ^ " = "
^ string_of_expr ex ^ ";\n"
  | Shape(shape_defn) -> "Shape " ^ shape_defn.sname ^ " = new
Shape(new Rectangle(" ^ string_of_expr shape_defn.x ^ ", " ^
```

```
string_of_expr shape_defn.y ^ ", " ^ string_of_expr shape_defn.w ^
", " ^ string_of_expr shape_defn.h ^ "), " ^ string_of_color
shape_defn.scolor ^ ", " ^ string_of_stype shape_defn.stype ^
");\n"

(* Returns a string for the given statement *)
let rec string_of_stmt = function
    Expr(ex) -> string_of_expr ex ^";"
  | Block(s) -> "{\n"^String.concat "\n" (List.map string_of_stmt
s) ^ "}\n"
  | If(ex, s, Block([])) -> "if (" ^ string_of_expr ex ^")\n"
^string_of_stmt s
  | If(ex, s1, s2) -> "if (" ^ string_of_expr ex
^")\n"^string_of_stmt s1 ^ "else\n" ^ string_of_stmt s2
  | While(ex, s) -> "while (" ^ string_of_expr ex ^")\n"^
string_of_stmt s
  | Run(id) -> id^"();"
  | Put(id, ex1, ex2) -> id ^ ".frame.x = " ^ string_of_expr ex1 ^
"; " ^ id ^ ".frame.y = " ^ string_of_expr ex2 ^ ";"
  | Animator(id, dir, ex) -> id ^ string_of_direction dir ^
string_of_expr ex ^ ";"
  | Vdecl(var) -> string_of_vdecl var ^ ";"
  | Print(str) -> "System.out.println("^str^");"
  | Background(color) -> "setBackground(" ^ string_of_color color ^
");"

(* Returns a string add statement for the given v_decl()->Shape *)
 let string_of_add = function
  Shape(shape_defn) -> "shapes.add(" ^ shape_defn.sname ^ ");\n"
  | Def(_, _, _) -> ""

(* Returns a string for the given function declaration *)
let string_of_func f_decl = "public void " ^ f_decl.fname ^ "()
{\n\t" ^ String.concat "\n\t" (List.map string_of_stmt f_decl.body)
^ "\n}"

(* Returns a single string with the program's contents *)
let check_program (gl, funs) =
  ignore(List.map string_of_vdecl (List.rev gl));
  ignore(List.map string_of_func (List.rev funs));
```

8.5. codegen.ml

```
open Printf

let file_name = "PSMMAnimator"
```

```
let window_size = 700

(* Returns the complete Java string given tuple of strings
(shape_decls, add_stmts, func_decls)  *)
let java_code (s_decls, add, funs) =
     "import java.awt.Color;\n" ^
     "import java.awt.Dimension;\n" ^
     "import java.awt.Graphics;\n" ^
     "import java.awt.Graphics2D;\n" ^
     "import java.awt.Rectangle;\n" ^
     "import java.awt.geom.Ellipse2D;\n" ^
     "import java.awt.geom.Rectangle2D;\n" ^
     "import java.awt.geom.AffineTransform;\n" ^
     "import java.util.ArrayList;\n" ^

     "import javax.swing.JFrame;\n" ^
     "import javax.swing.JPanel;\n" ^
     "import javax.swing.SwingUtilities;\n" ^


     "public class " ^ file_name ^ " {\n" ^

     "    public static void main(String[] args) {\n" ^
     "         SwingUtilities.invokeLater(new Runnable() {\n" ^
     "              public void run() {\n" ^
     "                   createAndDisplayGUI();\n" ^
     "              }\n" ^
     "         });\n" ^
     "    }\n" ^

     "    public static void createAndDisplayGUI() {\n" ^
     "         JFrame frame = new JFrame(\"My Animation Coded in
Photoshop--\");\n" ^
     "
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);\n" ^

     "         PSMMAnimatedPanel panel = new
PSMMAnimatedPanel();\n" ^
     "         frame.add(panel);\n" ^
     "         frame.pack();\n" ^
     "         frame.setVisible(true);\n" ^

     "         Thread t = new Thread(panel);\n" ^
     "         t.start();\n" ^
     "    }\n" ^
     "}\n" ^
```

```
     "class PSMMAnimatedPanel extends JPanel implements Runnable
{\n" ^
     "    private static final long serialVersionUID = 1L;\n" ^
     "    public ArrayList<Shape> shapes;\n" ^

     s_decls ^

     "    public PSMMAnimatedPanel() {\n" ^
     "         shapes = new ArrayList<Shape>();\n" ^

     "         // Create and add shapes\n" ^
     add ^
     (* Create and add shapes *)


     "    }\n" ^

     funs ^

     "    @Override\n" ^
     "    public void run() {\n" ^
     "         while (true) {\n" ^
     "              recalculateShapes();\n" ^
     "              repaint();\n" ^

     "              try {\n" ^
     "                   Thread.sleep(1000 / 60);\n" ^
     "              } catch (InterruptedException e) {\n" ^

     "              }\n" ^
     "         }\n" ^
     "    }\n" ^

     "    private void recalculateShapes() {\n" ^
     "         // Do stuff to shapes\n" ^
     "         drawloop();\n" ^
     "    }\n" ^

     "    public Dimension getPreferredSize() {\n" ^
     "         return new Dimension(" ^ string_of_int window_size
^ ", " ^ string_of_int window_size ^ ");\n" ^
     "    }\n" ^

     "    @Override\n" ^
     "    public void paintComponent(Graphics g) {\n" ^
     "         super.paintComponent(g);\n" ^
     "         Graphics2D g2 = (Graphics2D) g;\n" ^
```

```
     "          for (Shape shape : shapes) {\n" ^
     "               AffineTransform old = g2.getTransform();
shape.angle %= 360; g2.rotate(Math.toRadians(shape.angle));
g2.setTransform(old); \n" ^
     "               g2.setPaint(shape.color);\n" ^

     "               Rectangle frame = shape.frame;\n" ^

     "               if (shape.type == Shape.Type.ELLIPSE) {\n" ^
     "                    g2.fill(new Ellipse2D.Double(frame.x,
frame.y, frame.width, frame.height));\n" ^
     "               } else if (shape.type == Shape.Type.RECTANGLE)
{\n" ^
     "                    g2.fill(new Rectangle2D.Double(frame.x,
frame.y, frame.width, frame.height));\n" ^
     "               }\n" ^
     "          }\n" ^
     "     }\n" ^
     "}\n" ^

     "class Shape {\n" ^
     "     public enum Type {\n" ^
     "          RECTANGLE, ELLIPSE\n" ^
     "     }\n" ^

     "     public Rectangle frame;\n" ^
     "     public Color color;\n" ^
     "     public Type type;\n" ^
     "     public int angle;\n" ^

     "     public Shape(Rectangle frame, Color color, Type type)
{\n" ^
     "          this.frame = frame;\n" ^
     "          this.color = color;\n" ^
     "          this.type = type;\n" ^
     "          this.angle = 0;\n" ^
     "     }\n" ^
     "}\n"

(* Generates the Java code and prints it to a file *)
let generate_code fnv =
        let oc = open_out (file_name ^ ".java") in
            fprintf oc "%s" (java_code fnv);
            close_out oc;
```

8.6. pmmc.ml

```
let filename = Sys.argv.(1) in
     let lexbuf = Lexing.from_channel (open_in filename) in
          let src = Parser.program Scanner.token lexbuf in
          let string_split = Ast.program_string_split src in
               Semantic.check_program src;
               Codegen.generate_code string_split
```