LSystem Fractal Language

Final Report

PLT (Fall 2011)

Team Members:

Jervis Muindi (jjm2190)

Ethan Hann (eh2413)

Michael Eng (mse2124)

Timothy Sun (ts2578)

12/22/2011

Table of Contents

Section 1: Introduction

1.1 Project Overview

The language we have designed is an L-system based fractal drawing language. L-systems use a variant of context-free grammars to generate fractals and other iterative sequences. Our language is translated from a set of rules to turtle graphics procedures in Java, which will in turn be directly used to display the iterative sequence specified on screen. The language is intended to allow users to quickly model and visualize L-systems on-screen.

Thus, using our language, it will be very easy for a user familiar with L-system grammars to code and draw an L-system in our language, as the logistical details of creating a window and actually drawing the L-system will be automatically handled once the source program is compiled.

Ideally, we would like to minimize the amount of parsing required from a given program. As such, we have Java classes that provide drawing and computational functionality that would execute alongside the outputted code from our compiler. This standard library is added into each program file when it is compiled into a Java file. The net result is a programming language that is accessible to mathematicians having little familiarity with programming.

1.2 Background

L-systems, short for Lindenmayer Systems, are a type of formal grammar. An L-system consists of an alphabet of symbols, an initial sequence of symbols (a string) used to begin a construction, a set of production rules that expand individual symbols into strings, and a set of rules that match terminal symbols to drawing functions in order to translate generated strings into geometric structures. To construct an L-system, the production rules are iteratively applied, starting from the initial string. In each iteration all rules are applied in parallel.

More formally, an L-system is defined as a tuple:

$G = (V, w, P)$

…where, *V* is the alphabet, the set of symbols containing variables, or replaceable elements; *w* is the start or axiom, the string consisting of symbols from *V*, representing the initial state of the system. P is the set of production rules that define how variables can be replaced by combinations of constants and other variables. A given production rule takes the form $p \rightarrow s$, where *p* and *s* are both strings. For any symbol X in V which does not appear as *p* in a production rule in P, the rule $X \rightarrow X$ is assumed, and X is defined as a constant or terminal symbol.

An example L-system representing the Koch Curve follows:

variables : F
constants : + −
start : F
rules : (F → F+F−F−F+F)

F translates to “draw forward”, + translates to “turn left 90 degrees”, and - translates to “turn right 90 degrees”
Example iterative expansions, where n is the number of iterations to carry out:

n = 0:
F
n = 1:
F+F-F-F+F
n = 2:
F+F−F−F+F + F+F−F−F+F − F+F−F−F+F − F+F−F−F+F + F+F−F−F+F
n = 3:
F+F−F−F+F+F+F−F−F+F−F+F−F−F+F−F+F−F−F+F+F+F−F−F+F +
F+F−F−F+F+F+F−F−F+F−F+F−F−F+F−F+F−F−F+F+F+F−F−F+F −
F+F−F−F+F+F+F−F−F+F−F+F−F−F+F−F+F−F−F+F+F+F−F−F+F −
F+F−F−F+F+F+F−F−F+F−F+F−F−F+F−F+F−F−F+F+F+F−F−F+F +
F+F−F−F+F+F+F−F−F+F−F+F−F−F+F−F+F−F−F+F+F+F−F−F+F

1.3 Related Work

There are a number of L-system generator applications available online. They range from applications that create modeling scripts that are used to generate 3-D L-system models in applications like Maya or 3D Studio MAX, to applets that allow for users to input the number of iterations, a single angle degree value, an initial state, and production rules for a single L-system that is then drawn in the browser. A particularly popular L-system generator application is the L-Systems Explorer, available at http://www.generation5.org/content/2002/lse.asp. It allows the user to enter in rules, the number of iterations, an angle value, and a draw distance value. LPFG is a programming language (detailed at *http://algorithmicbotany.org/lstudio/LPFGman.pdf*) that allows users to create L-systems, and is based around a C++ and OpenGL environment.

Our L-system language aims to provide the customizability possible only through a full programming language while remaining syntactically intuitive enough to allow users to quickly and easily create and model L-systems.

1.4 Goals of the language

Our L-System language aims to be customizable, intuitive, and portable. It serves as a simple and powerful means of creating and modeling L-systems.

1.4.1 Customizable

Most L-system applications only allow for users to specify fields in an L-system structure- the fields consisting of the system's production rules, initial state, number of iterations, and an angle value. In our language, it is possible for users to map multiple terminals to different angle values (e.g. z = turn(20) and y = turn(40)) and to different draw distances (e.g. z = forward(30) and y = forward(15)). There is a higher level of customizability that comes from creating a programming language instead of an application- users can determine their own parameters instead of merely filling in blanks in a pre-set structure.

1.4.2 Intuitive

Our language has a syntax that is very similar to Java's syntax. The syntax for creating L-system drawing methods is very easy to grasp if the user is familiar with constructing an L-system grammar. Users can easily create and model L-systems without needing to deal with creating a GUI window, instantiating a panel, creating a drawing class, and so on. A perfectly valid program can have an L-system drawing method and a line in the main method that calls the drawing method. As long as a user is familiar with L-systems and their underlying structure, they will be able to use our language to create and model L-systems.

1.4.3 Portable

The L-system language only requires that the user have the Java Development Kit and the Java Runtime Environment on their machine, in order to compile compiled L-system programs from their intermediate Java source code states into class files and to execute those class files. The compiler takes in L-system input files and outputs Java source code. In essence, the language can be used on any machine that has a Java compiler on it.

Section 2: Language Tutorial

2.1 Getting Started with the Compiler

2.1.1 Compiler Requirements

The LSystem compiler can generate intermediate Java source code without the Java compiler installed. In order for the LSystem compiler to create a runnable program a Java compiler of at least version 1.6 needs to be available in the path of the user compiling the LSystem program. The Java compiler is packaged with the Java Development Kit (JDK). To run a compiled LSystem program the Java Run-time Environment (JRE) must be installed.

2.1.2 Installing the Compiler

The LSystem language compiler needs to be copied into a directory that is in your path environment variable, such as **/usr/bin/**. To check if the compiler is accessible from the command line after you have copied it simply attempt to run it, as in Figure 2.1.2.1. Usage instructions will be displayed if the compiler is accessible, with an “InvalidArgument” warning message.

```
$ lsystem
InvalidArgument
 Usage: lsystem [-a|-s|-c] SOURCE_FILE [-t|-v]
```

Figure 2.1.2.1 – Get Compiler Usage Instructions

2.2 A First Example: The Hilbert Curve

2.2.1 The Main Function

Every LSystem program has a main function. This function serves as a main entry point into the application, as shown in figure 2.2.1.1.

```
def compute main()
{
}
```

Figure 2.2.1.1 – Get Compiler Usage Instructions

2.2.2 The Draw Function

As the name implies, **draw functions** draw images. These types of functions define an L-system, which is essentially a list of rules that are used to generate an image. What this means will become clearer as you progress through the tutorial.

We will start with an explanation of the draw function signature. Draw functions always have the same function signature, with the exception of the name of the function which can be different. The **def** keyword indicates that a function is being defined. The **draw** keyword indicates that the function is a draw function, i.e. that it will contain an L-system. The name of the particular draw function given below is **hilbertCurve**. The **level** argument passed into the hilbertCurve function refers to the number of times the L-system rule set is applied.

```
def draw hilbertCurve(int level)
```

Figure 2.2.2.1 – Draw Function Signature

When the name of a rule sequence (e.g. A or B in the example below) is encountered on the right hand side of the rule, the rule sequence is reapplied. The maximum depth of this recursive process is the value passed in for the **level** argument.

In the example L-system rule set, the **l** constant means "turn left" by 90 degrees, and **r** constant means "turn right" by negative 90 degrees. In terms of the underlying drawing system, the turn radius refers to the degree by which the drawing cursor turns when an **l** or **r** constant is encountered. The **f** constant means “draw forward" by some integer increment.

```
alphabet:  (A,B);                   # The rules in the l-system.
rules:{
  lambda -> A;                      # The first rule to call.
  A  ->  l B f r A f A r f B l;     # the A rule sequence
  B  ->  r A f l B f B l f A r;     # the B rule sequence
  f = forward(1);
  r = turn(-90);
  l = turn(90);
}
```

Figure 2.2.2.2 – Example L-system Alphabet and Rule Set

Combining these concepts together we get a program that draws a Hilbert curve, i.e. the “Hello, World” program of the LSystem language. The image that this program produces is shown in the next section, after the full program is compiled.

```
def draw hilbertCurve (int level){
  alphabet: (A,B);
  rules: {
    lambda -> A;
    A  ->  l B f r A f A r f B l;
    B  ->  r A f l B f B l f A r;
    f = forward(1);
    r = turn(-90);
    l = turn(90);
  }
}

def compute main()
{
  hilbertCurve(5);
}
```

Figure 2.2.2.3 – Hilbert Curve Code

2.2.3 Compiling the Hilbert Program

The output of the program will be a file called "hilbert.class" that contains the Java bytecode representation of the LSystem program. Follow the following steps to compile the Hilbert example program.

1. Create a text file called "hilbert.ls" (sans quotes).
2. Open the text file with your favorite text editor.
3. Copy the example program into the file, then save the file.
4. Enter the following command into the command line.

```
$ lsystem -c hilbert.ls
```

Figure 2.2.3.1 – Hilbert Compilation Command

2.2.4 Running the Hilbert Program

The LSystem language compiles to a Java program. To run the example program, you will need to have the Java JRE installed. After compiling the Hilbert example program, to run the Java program enter the following command.

```
$ java Hilbert
```

Figure 2.2.4.1 – Hilbert Run Command

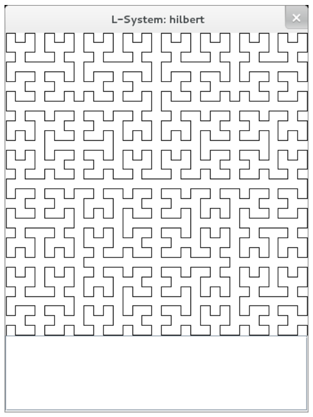

Figure 2.2.4.2 – The Running Hilbert Program

2.3 Additional Examples

This section includes additional drawing examples, with notes that explain why they are interesting. An image of the running program is included, without the text printing area. The code for each program is also included.

2.3.1 Dragon Curve

The code for the dragon curve example, shown in figure 2.3.1.1, is very similar in format to the Hilbert curve code, shown in figure 2.2.2.3. The difference of interest is that the dragon curve example has a different L-system. The other change, the draw function being called "dragonCurve" instead of "hilbertCurve," is irrelevant to the output of the program.

There are two interesting concepts to note about this example: rules are not reflections of each other as in the Hilbert curve example, in fact the Z rule is only used once, and there are no drawing constants defined. Though self-similar, the dragon curve is not symmetrical like the Hilbert curve. This is due to the rules not being symmetrical. The **l**, **r**, and **f** drawing constants, when not explicitly defined, have default values of **turn(90)**, **turn(-90)**, and **forward(1)**, respectively.

```
def draw dragonCurve(int level){
    alphabet:  (X,Y,Z);
    rules:{
        lambda -> Z;
        Z -> f X;
        X -> X l Y f;
        Y -> f X r Y;
    }
}

def compute main(){
      dragonCurve(12);
}
```

Figure 2.3.1.1 – Dragon Curve Code

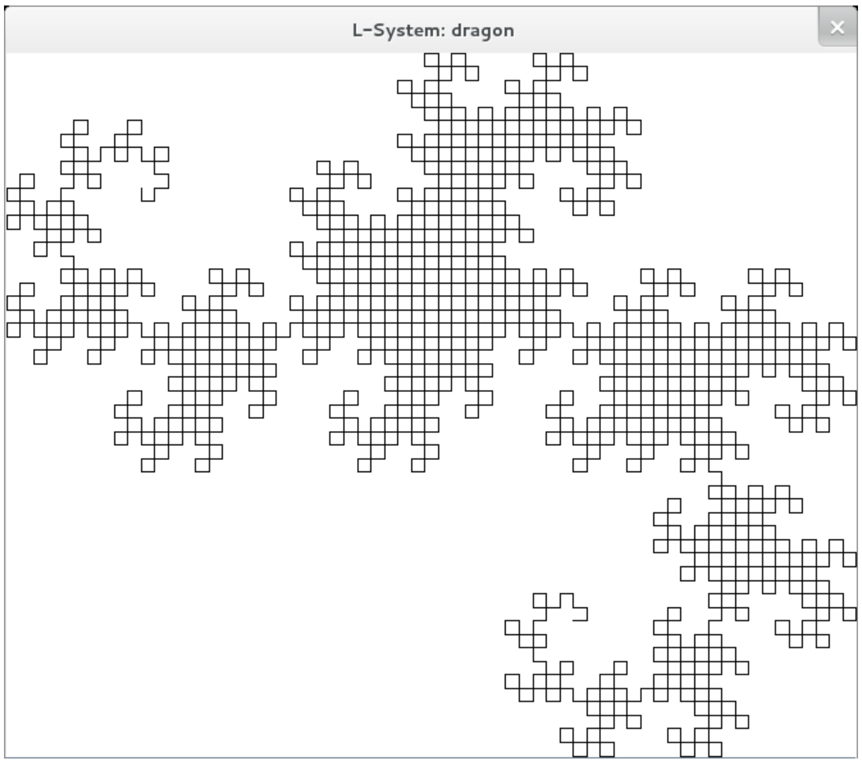

Figure 2.3.1.2 – Dragon Curve Program

2.3.2 Hilbert Curve Derivation

The Hilbert curve derivation is a fractal created by Ethan Hann. The L-system rules that define it, shown in figure 2.3.2.1, are similar to the Hilbert curve, hence the name “Hilbert Derivation.” It demonstrates that with those few rule changes, and the angle of the turn constants changed to 80 degrees, a very different fractal is created, as shown in figure 2.3.2.2.

```
def draw hilbertDerivation(int level){
    alphabet:  (A,B);
    rules:{
        lambda -> A;
        A  ->  l B f r A f A r f B l;
        B  ->  r C f l B f B l f C r;
        C  ->  l D f r C f C r f D l;
        D  ->  r A f l D f D l f A r;
        l = turn(-80);
        r = turn(80);
    }
}

def compute main()
{
    hilbertDerivation(7);
}
```

Figure 2.3.2.1 – Hilbert Curve Derivation Code

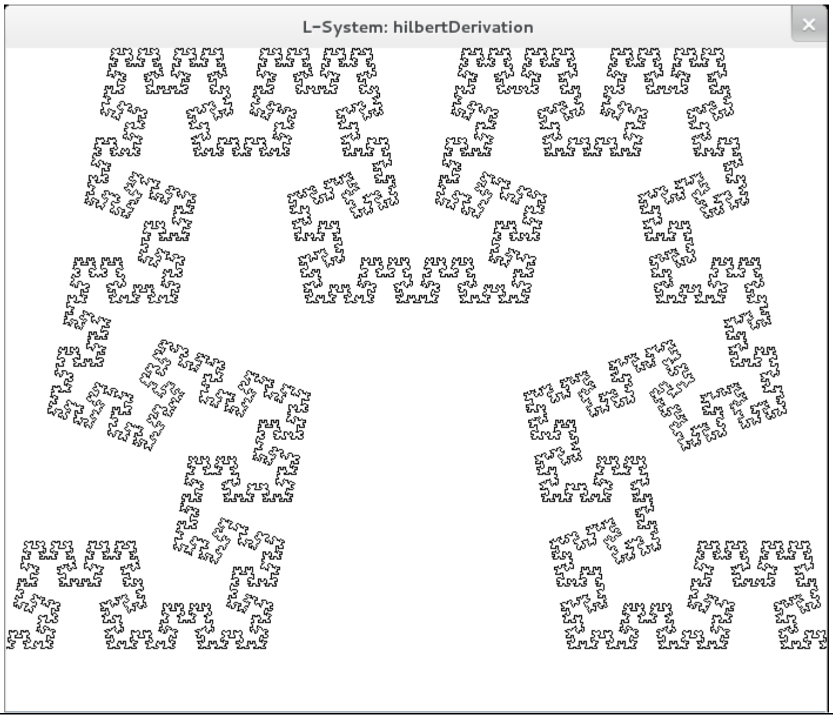

Figure 2.3.2.2 – Hilbert Curve Derivation Code

2.3.3 Koch Curve

The interesting thing about the Koch curve example, shown in figure 2.3.3.1, is that its L-system contains only one rule. The LSystem language is extremely powerful in that a small amount of code can produce a very complex image. This is due to the language providing an almost identical representation of the mathematical notation for L-systems in its syntax.

```
def draw kochCurve(int level){
    alphabet:  (X,f,r,l);
    rules:{
        lambda -> X;
        X -> f X l f X r f X r f X l f X;
    }
}

def compute main(){
      kochCurve(5);
}
```

Figure 2.3.3.1 – Koch Curve Code

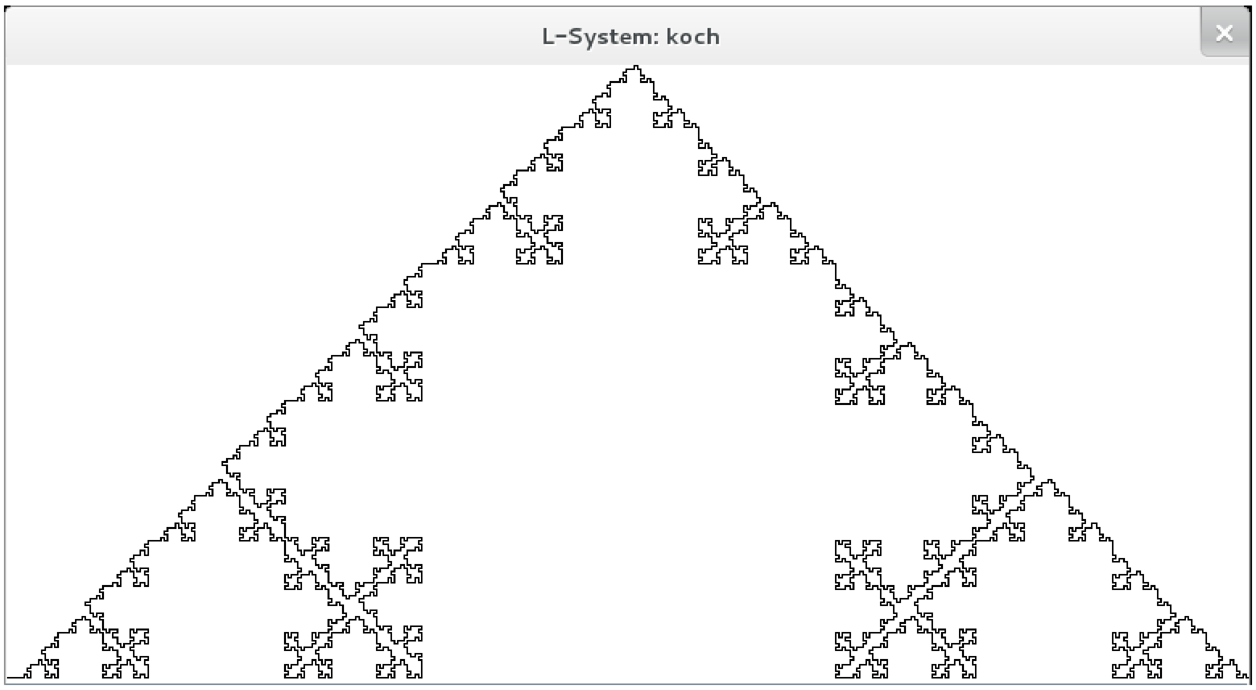

Figure 2.3.1.2 – Koch Curve Program

2.3.4 Lévy C Curve

At high levels of recursion, i.e. setting the level argument to greater than 20, the Lévy C curve program shown in figure 2.3.3.1, compiles correctly, but it crashes the Java virtual machine. The Lévy C curve is very ornate and requires a lot of heap memory. This highlights a very important limitation of the LSystem compiler: LSystem programs can easily reach hardware limits.

```
def draw levycCurve(int level){
    alphabet:  (X);
    rules:{
        lambda -> X;
        X -> r f X l l f X r;
        l = turn(-45);
        r = turn(45);
    }
}

def compute main(){
      levycCurve(20);
}
```

Figure 2.3.3.1 – Lévy C Curve Code

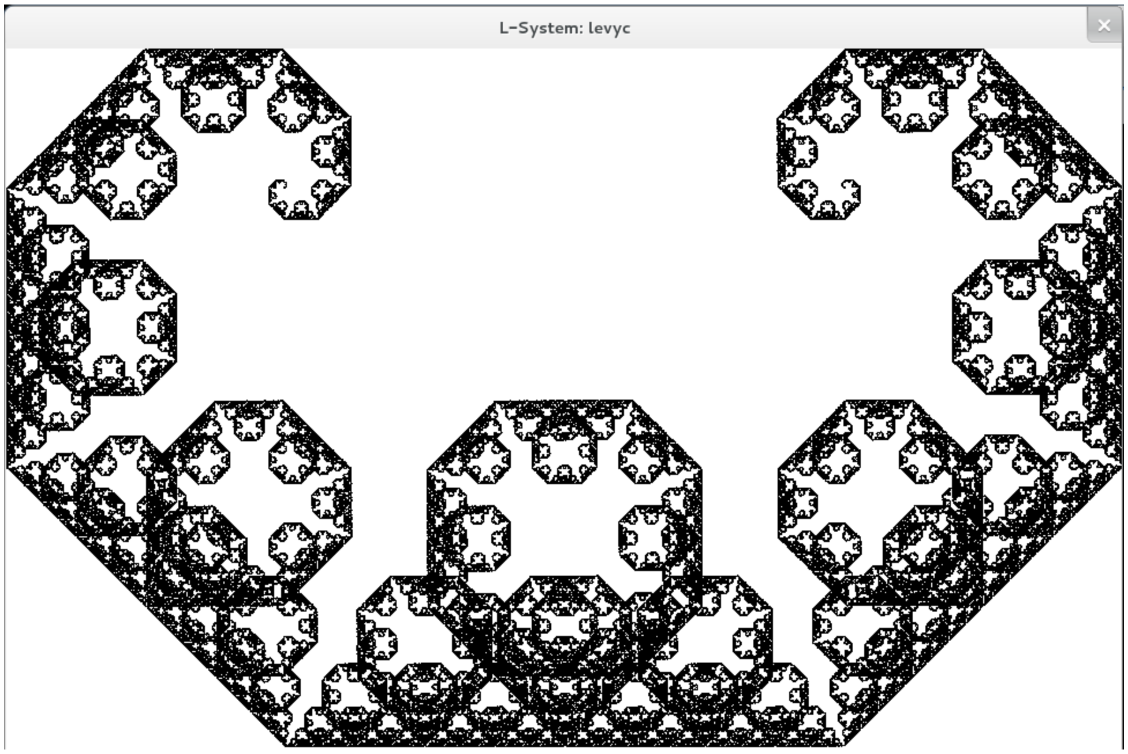

Figure 2.3.1.2 – Lévy C Curve Program

2.3.5 Sierpinski Triangle and Derivation

Changing the value of the forward constant to something other than 1 can have interesting effects on the programs output. The L-system shown in the Sierpinski triangle example, figure 2.3.5.1, is identical to those of the Sierpinski derivation L-system (not shown), except that the A forward rule has a value of -1 and the B forward rule has a value of -10. The image generated by the Sierpinski derivation, figure 2.3.5.2, is a horizontal reflection of the original, figure 2.3.5.1, due to the sign being flipped. Also, some of the inner triangles are smaller in the derivation due to the A and B rules not having the same value.

Notice that the **f** constant is not used. It is not needed as the A and B rules serve a dual role: rule and constant. This feature was added to the language to make rule sets even more flexible. This feature allows for rules to not merely inform the shape of the generated fractal, as with the other examples, but they are directly responsible for creating it.

```
def draw sierpinskiDerivation(int level){
    alphabet:  (A,B);
    rules:{
        lambda -> A;
        A -> B l A l B;
        B -> A r B r A;
        A = forward(-1);
        B = forward(-10);
        l = turn(-60);
        r = turn(60);
    }
}

def compute main(){
      sierpinskiDerivation(8);
}
```

Figure 2.3.5.1 – Sierpinski Derivation Code

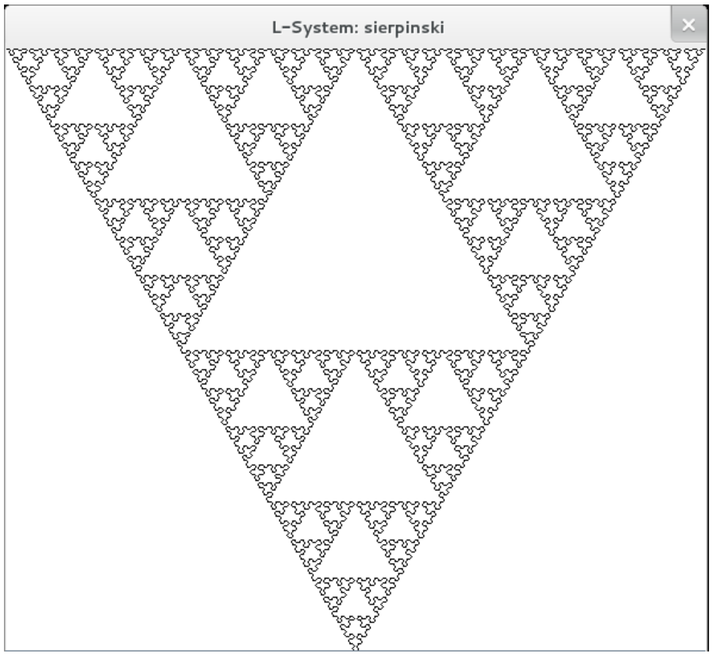

Figure 2.3.5.2 – Sierpinski Triangle Program

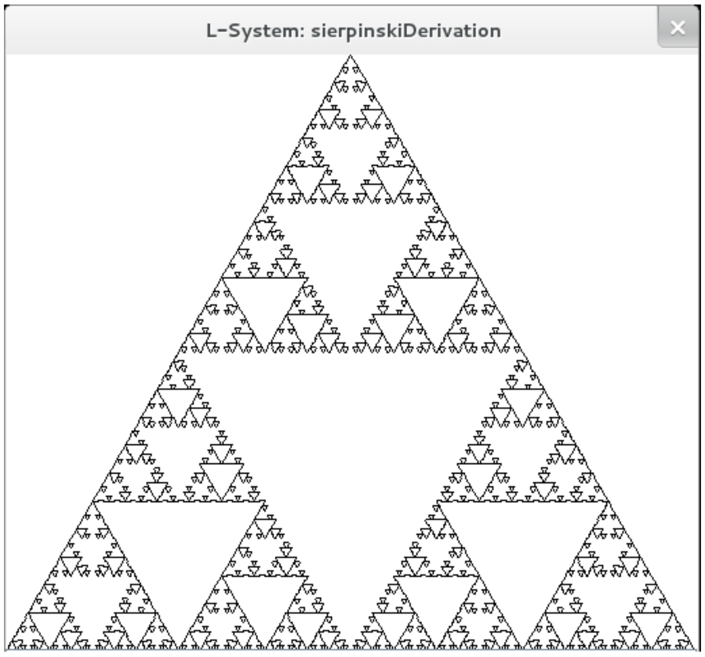

Figure 2.3.5.3 –Sierpinski Derivation Program

Section 3: Language Manual

3.1 Program Definition

A program in our language is made up of statements which consist of function declarations, function implementations and expression statements. A program in our language is to be written in a single source file, and combining different source files is not currently supported. Thus, the structure of the program in the source file is as follows:

<<function declarations and implementations in source file>>

That is both function declaration and implementation happen at the same time. Also note that one of the function declarations must be a main compute function as this is the function that serves as the entry point of the program.

Currently, it is not possible to pass in user supplied command-line level arguments and so the main function should have the following signature:

```
def compute main()
{

}
```

Also, any user defined functions that are used must have been previously declared at the time of use. It is illegal to refer to functions that have not yet been declared and implemented even if their implementation comes on later. An example of a simple program is given below:

```
def compute sqrt(double x){ # computes square root of x
      x = x^(0.5);
      return x;
}

def compute main() { # main function
  double x = 25;
  double root = 0;
  root = sqrt(25);
  print(root);
}
```

3.2 Lexical Conventions

3.2.1 Comments

Comments are single-line and are prefaced by the # symbol. Comments are in effect until the end of the line.

```
#This is a comment.

def draw hilbert(double s) #This is also a comment
```

3.2.2 Identifiers

An identifier is a sequence of letters, numbers and underscores (_) in which the first character is not a number. An identifier can consist of both upper and lower case letters. The language is case-sensitive and as such will differentiate identifiers with identical letters but have different cases.

3.2.3 Keywords

The following terms are a list of reserved keywords and built-in functions in the language and cannot be used for any other purpose:

Reserved Words		**Functions**
alphabet	int	print
boolean	lambda	forward
compute	print	turn
def	return	up
double	rules	down
else	string	setX
false	true	setY
if	while	

3.2.4 Constants and literals

Our language provides functionality for literals (also known as constants) of type int, double, string, and boolean. If any of these literals are assigned to a variable, that variable's declared type must match up with the literal's type - no automatic conversion will occur except in the case of up-converting from an int to a double.

Integer constants

An integer constant consists of a sequence of numbers without a decimal point. In the example below 45 is the integer constant. We do not provide support for octal or hexadecimal representation of integers.

```
int x = 45;
```

String constants

A string constant is enclosed in double quotation marks, such as “x”. We provide support for the following escape sequences within string constants:

Character name	Escape sequence
Newline	\n
Horizontal tab	\t
Double quotation marks	\”
Backslash	\\

As such, examples of the use of string constants include:

```
string s = "string\n";
print("|column 1 \t column 2 \t column 3 \t |\n");
```

Note that string concatenation is currently not natively supported in our language.

Double-precision floating point constants

A double constant consists of a sequence of numbers and a decimal point. At least one number must appear before the decimal point. Examples of valid double constants include:

```
0.345
0.0
1.0793211
3.141592654
```

…but do not include sequences like:

```
.314
.010
0.0.0
..0
.02.2
2..9
```

Floating numbers can also be written in the standard scientific form using e notation. For example `6.023e23` is a valid floating constant.

Boolean constants

Boolean constants consist of the keywords true and false. A valid example of their use is:

```
boolean b = true;
if(b==true)
{
#code
}
```

3.2.5 Operators

An operator is a symbol that denotes a specific operation that is to be performed. Below is a list of operators supported:

Symbol	Explanation
+	Performs addition operation
-	Performs subtraction operation
*	Performs multiplication operation
^	Performs exponeniation
/	Performs division operation
=	Performs value assignment

3.2.6 Punctuators

A punctuator is a symbol that does not specify a specific operation to be performed. A punctuator is primarily used in formatting code and so it does have a special meaning (i.e. significance). A punctuator can be only one of the symbols below:

Symbol	Explanation
:	Used in defining a section in a draw function
;	Statement terminator
{ }	Used for grouping code, example in function declarations.

3.3 Meaning of Identifiers

3.3.1 Scoping

The region of a program in which a certain identifier is visible and thus accessible, is called its scope. The scoping in our language is **local only** and no global scope exists. That is, all identifiers declared are visible only in that specific function and nowhere else. Also, identifiers become visible only after being declared and thus, it is illegal to refer to identifiers that have not yet been declared. For example, the following is not allowed and the compiler will produce error messages in such cases:

```
int foo = 5;
int bar = 10;
int x = 25;
int sum = 0;
sum = foo + bar + x + y;  # This is illegal. You cannot access  undeclared
value y.
int y = 15;
```

Function Scope

Similar to the scoping of identifiers, you cannot refer to a function that has not yet been seen. For example:

```
def compute sqrt(double x) {
  ...
}

def compute twice_square_root(double x) {
  double root = 0;
  root =  sqrt(x);
  # This is illegal. Add(…) must be declared before it’s used.
  return add(root, root);
}

def compute add(double x, double y){
 ...
}
```

3.3.2 Object types

Our language supports only the following four fundamental types of objects:

1. integers
2. floating point numbers
3. strings
4. booleans

Character Types

The only supported character type is the string type. This can store a string of potentially unlimited length and does not have an upper bound limit. The length is only limited by the amount of computing resources (such as memory) available.

Integer and Floating Point types

The only supported integer type is int which can store 32-bits worth of data and the only supported floating point type is double which can store 64-bits of data. Both of these data types are signed.

Boolean Type

This data type is essentially a truth value and can only store a single bit of information. Specifically, it may only take a value of either “true” or “false”.

3.4 Type Conversions

3.4.1 Conversion from int to double

Integer typed variables (also known as ints) can be converted into double-typed variables with automatic casting. Note that the reverse operation is not (and cannot be) performed as this can potentially result in a loss of information. The way to invoke this automated conversion is simply like so:

```
int x = 45;
double y = x; # convert x from int type to double type
```

Note that the converted variable must have a different identifying name than the original variable. So writing the following will cause an error:

```
int x = 45;
double x =  x;
```

The automated conversion from int to double is the only type of casting that is permitted and specifically it is not possible to manually convert any one data type to another. For example, the following will cause an error:

```
double x = 45.0;
int y = (int) x;
```

3.5 Expressions and Operators

3.5.1 Precedence and Associativity Rules

The language follows classical mathematical order of operations, prioritizing multiplication and division over addition and subtraction. The logical AND operator takes precedence over the logical OR operator. Also, expressions inside parentheses have top precedence and are therefore evaluated first.

Expression	Results	Comments
3 + 2 * 6	15	Multiplication occurs before addition
3 + (2 * 6)	15	Expression within parentheses is evaluated first, though the answer doesn't change from the above expression
(3+2) * 6	30	Expression within parentheses is evaluated first
FALSE || TRUE && FALSE	FALSE	The logical AND operator takes precedence over the logical OR operator.
(FALSE || TRUE) && TRUE	TRUE	The expression within the parentheses is evaluated first.
FALSE || (TRUE && FALSE)	FALSE	The expression within the parentheses is evaluated first.

The following table is a list of operator precedence and associativity for our computational functions, adapted from the C language reference manual.

Tokens (from high to low priority)	Operators	Class	Associativity
Identifiers, constants, string literal, parenthesized expression	Primary expression	Primary	
()	Function calls	Postfix	L-R
^	Exponentiation	Binary	R-L
* /	Multiplicative	Binary	L-R
+ -	Additive	Binary	L-R
< <= >= >	Relational comparisons	Binary	L-R
== !=	Equality comparisons	Binary	L-R
&&	Logical AND	Binary	L-R
||	Logical OR	Binary	L-R
=	Assignment	Binary	R-L
,	Comma	Binary	L-R

3.5.2 Primary expressions

Identifiers

An identifier refers to either a variable or a function. An example is `int x` or `def draw hilbert`, where x and hilbert are the respective identifiers.

Constants

A constant's type is defined by its form and value. See section 2d for examples of the use of constants.

String literal

String literals are translated directly to Java strings by our compiler, and are treated accordingly.

Parenthesized expressions

A parenthesized expression's type and value are equal to those of the un-parenthesized expression. The presence of parentheses is used to identify an expression's precedence and its evaluation priority.

3.5.3 Function calls

Our language supports two kinds of functions: Compute Functions and Drawing Functions. To be able to call a function, it must have been declared and implemented before. That is, it is a syntax error to call a function which has not yet been seen by the compiler. Also, recursive function calls are not supported at the moment.

The call to a compute function must match and agree with the signature of a previously declared compute function. The syntax of a compute function call is:

```
def compute Function_Name(Argument_Parameter_List);
```

The grammar for the compute function call, where here *identifier* refers to the name of a user-defined function, is:

identifier (argument-expression-list)
argument-expression-list: *argument-expression*
argument-expression-list, argument-expression

Example :

Assume there is a compute function with the signature below:

```
def compute sum(int a, int b) { … }
```

It can be called like so:

```
sum(1,3);
```

On the other hand, the drawing function can only take a single parameter that is an integer.

Example:

Assume there is a draw function with the signature below:

```
def draw DragonCurve (int level) {...}
```

It must be called like so:

```
DragonCurve(10);
```

The purpose of a draw function is to paint an L-System onscreen. All draw functions take a single integer parameter which describes how many times the L-system rules will be applied in painting the system.

The grammar for the draw function call is:

identifier (argument-expression)

Note, that the argument-expression here should be exactly a single integer. The semantic analysis section of the compiler will check this and compilation will fail if that actual parameter is not an integer.

3.5.4 Arithmetic operators

Arithmetic operators involve manipulating arithmetic expressions. The grammar that defines what is an arithmetic expression is given below:

arithmetic expression : *NumberLiteral*
VariableIdentifier

The operators * (multiply), / (divide) are what we call multiplicative operators and they group from left to right. The ^ operator is also part of this group but it is right-associative. These operators form what we call multiplicative expressions.

The grammar for these expressions is:
multiplicative expression : *arithmetic expression*
*multiplicative-expression * arithmetic expression*
multiplicative-expression / arithmetic expression
multiplicative-expression ^ arithmetic expression

The * operator performs arithmetic multiplication and its operands must have arithmetic type (i.e. be numbers of either type int or double)

The ^ operator performs arithmetic exponentiation and thus its operands must have arithmetic type of either int or double. The result of this operation is a double.

The / operator performs arithmetic division and its operands must also have arithmetic type.The result of this operation is similar to the type of the operands. E.g. if both the operands are integers, then so is the result. If they both, doubles, then so is the result. If the types don't match up, the result is a double.

The operators + and - are known as additive operators and they associate from left to right. The grammar syntax for additive operators is given below :

additive-expression: *multiplicative-expression*
additive-expression + multiplicative-expression
additive-expression - multiplicative expression

The operator + performs arithmetic addition and so the operands must have arithmetic type. The value returned by this operator is the sum of the operands. Note that it is **not** possible to use the + operator to perform string concatenation.

Similarly, the operator - performs arithmetic subtraction and so the operands must also have arithmetic type. The value returned by this operator is the difference between the two operands.

3.5.5 Relational operators

Relational operators are used to compare variables to one another in relational expressions. The two variables on either side of a relational operator must be of a numeric type - for instance, `"I don't know" < 3` would cause a compiler error. A relational expression will evaluate to a boolean constant- that is, to true or false. Thus, only ints and doubles are the valid types for comparison when using this type of operator. Specifically, if `a` and `b` are number variables, `a < b` checks to see if `a` is smaller than `b`; `a > b` checks to see if `a` is bigger than `b; a <= b` checks if a is less than or equal to b, and `a >= b` checks whether `a` is greater or equal to `b`. The relational expression returns true if the condition specified holds and false if it does not.

Relational expression:
Variable rel_operator variable
Variable:
Arithmetic expression
constant or literal (e.g., previously instantiated int `d`, `6`)
Rel_operator:
<
<=
>
>=

3.5.6 Equality operators

The equality operators work the same way as the relational operators - however, they check to see if a variable is equal or not equal to another variable. Again, the two variables in an equality expression must be of numeric type. Thus, Ints and doubles are the only data types that can be used with these operators.

Equality expression:

Variable eq_operator variable

Variable:

Arithmetic expression

constant expression (see below section on Constant Expressions)

Eq_operator:

==
!=

3.5.7 Boolean operators

The language features support for the logical AND and logical OR operations. For logical AND, if both values being ANDed together are true, then the expression evaluates to true. Otherwise, the expression evaluates to false. For logical OR, if one or both values being ORed together is true, the expression evaluates to true. Otherwise, the expression evaluates to false.

A boolean expression takes the form of (boolean boolean_operator boolean). The expressions on each side of the boolean_operator (&& or ||) must evaluate to a boolean type. As a result, the expressions on each side of the boolean operator must be boolean constants (`true` or `false`), relational expressions (`a < b`), or equality expressions (`g == h`).

Boolean expression:

Boolean expression && Boolean expression

Boolean expression || Boolean expression

Boolean

Boolean:

Equality expression

Relational expression

Boolean_constant

Boolean_constant:

true

false

3.5.8 Assignment operator

The assignment operator, =, is used to associate an identifier with a value. For instance, the following statement will cause the identifier d to be associated with a value of 4, and to have the type int.

```
int d = 4;
```

As such, the next time d is used in an expression, it will evaluate to 4. The type declared on the left hand side of the assignment must match the expression on the right hand side of the expression. For example, an identifier declared as a boolean cannot be assigned a value that is a double literal.

3.5.9 Constant expressions

Constant expressions are expressions that evaluate to string and number literals. For instance, "test", 4, and 4.266 are all constant expressions. Note that constant expressions are a subset of primary expressions. Constant expressions can be assigned to variables in a variable declaration. For instance:

```
int d = 4; # 4 is a constant expression and it's been assigned to d
string s = "hi"; # hi is a const expression and it's been saved in s
```

3.6 Declarations

3.6.1 Function declarations

Our language supports two kinds of functions: *Compute Functions* and *Drawing Functions*. All functions must be preceded with a **def** keyword. Functions are both declared and implemented at the same time. Thus if one compute function calls another compute function, for example, then that other function must have been declared and implemented before. Otherwise, this is a compile-time error.

Compute Functions

Compute functions are your normal functions and they typically do some kind of computation like finding the square root of a number. All compute functions must have the "compute" specifier after the "def" specifier and they must return a value of type double (i.e. there is no need to specify a return type). If a function does not need to return any specific value, then it may simply return the value 0 to indicate that it has completed successfully. Compute function can have any number of named argument parameters. The argument parameter passing mechanism in our language is always pass-by-value. An example of a compute function declaration and implementation is below :

```
def compute sum (int a , int b){
      int sum = 0;
      sum = a + b;
      return sum ;
}
```

Note that all variable declarations in a compute function must occur first at the top of the function implementation. For example, the following is illegal and causes a compile time error.

```
def compute avg (int a , int b, int c){
      int tot = 0;
      tot = a + b + c;
      double avg = tot / 3; # compile time error
}
```

Drawing Functions

Drawing Functions are the functions that specify the structure of an L system that will be eventually be drawn. These functions only take in a single parameter that is an integer. This parameter specifies how many times the L system rules will be applied in drawing the L system.

The structure of the body of a drawing function is as follows :

alphabet : (List of Alphabet Letters separated by commas);

lambda -> (start letters);

Rules of how to transform or expand a certain alphabet character.

The syntax for the rules is:

Alphabet_Letter -> Result_of_Transformation

Alphabet_Letter = draw_function_call

The letters ‘r’ ‘l’ ‘f’ by default have the meaning, turn right by 90 degress, turn left by 90 degrees and move forward by a unit amount respectively. However, these defaults can be overriden like the example below shows.

An Example of a drawing function:

```
def draw hilbert(int n){
      alphabet: (A,B;f,r,l,s);
      rules:{
            lambda -> A;
            A -> l B f r A f A r f B l;
            B -> r A f l B f B l f B r;
            A = ;
            B = ;
            f = forward(15);
            r = rotate(-60);
            l = rotate(60);
      }
}
```

3.6.2 Variable declarations

Variable declarations are used to initial variables equal to constant values. The type used in a declaration must match with the type that the expression on the right hand side of the assignment operator returns. A variable declaration consists of the following grammar:

Variable declaration:

Type identifier = variable_expr

Type:

int

double

bool

string

variable_expr:

string_literal

number_literal

bool_value

Variable declarations are only permitted in compute functions and cannot be used in draw functions. Also, as previously mentioned, all variable declarations in a compute function must occur first at the top of the function.

Lastly, the variable declaration only support and allow for simple literals/constants to be assigned and any more complex expressions such as function calls are not supported at the moment.

```
def compute sum(int a, int b){...}
def compute test() {
      int x =  sum (1,2); # not allowed.
}
```

3.7 Statements

3.7.1 Expression statement

An expression statement is composed of primary statements with a semicolon at the end of the line.

3.7.2 If statement

If statements come in the following two varieties:

```
    if (expression)
        statement1
```

and

```
    if  (expression)
        statement1
    else
        statement2
```

For both if statements, “expression” must be of Boolean type, and statement1 executes if expression evaluates to true, while statement2 in the second variety executes if expression evaluates to false.

3.7.3 While loops

The While loop control flow construct allows for executing a statement any number of times.

```
        while (expression) statement
```

As with if statements, “expression” must be of boolean type, and statement executes until the expression evaluates to false. The evaluation of expression comes before the execution of the statement.

3.7.4 Return statements

A compute function must return a value to the caller through return statements. The only exception is the “main” function which does not return anything.

```
        return number
```

The value returned by a function, “number” in this instance, must be of double type. In the case where no specific value needs to be returned, a value of 0 can be specified to indicate that the function ran correctly.

3.8 System functions

3.8.1 Turtle Functions

For L-system drawings, we use turtle graphics, which draws images based on a user supplying relative positioning commands on a cursor. One can imagine the cursor being a turtle with a pen on its tail, and the user telling the turtle to go forward, turn, or lift its tail. In the following functions, r, theta, x, and y are all of double type. Furthermore, let X, Y, and Theta be the position and orientation of the cursor, respectively, and let “down” be the current state of the “tail.”

forward(r)

The turtle moves “forward” by r pixels. Formally, the change in X and Y is cos(Theta)r and sin(Theta)r, respectively. If “down” is set to true, then the line between (X,Y) and (X+cos(Theta)r,Y+sin(Theta)r) is drawn.

turn(theta)

The turtle turns counterclockwise by theta (in degrees). Formally, the change in Theta is theta.

up() - pen up

This function sets pen “down” to false.

down()

This function sets “down” to true.

setx(x)

This function sets X:=x

sety(y)

This function sets Y:=y

3.8.2 Input / Output

At the moment, user input, such as command line arguments, are not supported but it is still possible to print out to a JTextArea using a builtin library command print. Print supports taking as input all 4 data types in the language. These are namely boolean, int, double and strings.

For example:

```
def compute test() {
      int i = 0;
      while ( i< 10){
      print(i); # prints numbers 0 through 9
      i = i + 1;
}
```

Section 4: Project Plan

4.1 Team Responsibilities

Figure 4.1.1 summarizes the distribution of responsibilities for the project. Though there was some overlap with the compiler modules, each team member created the vast majority of the compiler components listed next to their name. Aside from Ethan attempting to provide a consistent format style across the document and typo fixes, there was no overlap in the creation of the with the final report sections. Section 7 is the only exception to this as each member obviously wrote their own lessons learned and advice for future teams. Section 7 is not listed in figure 4.1.1 for this reason.

Team Member	Compiler	Final Report
Jervis	Scanner Parser AST Generation Semantic Analysis	Section 3: Language Reference Manual
Michael	Test Script and Cases	Section 1: Language White Paper Section 6: Test Plan
Timothy	Standard Library Optimization Standardization of Formatting	
Ethan	Command Line Interface Code Generation Standard Library	Section 2: Language Tutorial Section 4: Project Plan Section 5: Architectural Design Section 8: Appendix Editing, formatting, and polishing.

Figure 4.1.1 – Team Member Responsibilities

4.2 Software Development Environment

The LSystem compiler is written in O'Caml on Linux and Mac. Though the compiler itself is written in O'Caml, it requires the Java JDK 1.6 to compile the intermediate Java code, and thus to test the compiler during development. Team members used the VIM text editor and the Eclipse (with the OcaIDE plugin) integrated development environment. Though a makefile has been included with the compiler source code, some of our team members found it easier to develop using the ocamlbuild tool as it, combined with OcaIDE, provided for automatic compiling when source files were modified and saved.

The compiler was tested with a custom shell script, called test.sh, and a collection of LSystem source files. The source code for the compiler was managed in a subversion repository on Google code: http://code.google.com/p/plt-lsystem/.

4.3 Project Processes

4.3.1 Planning

After submitting the language reference manual the team met each Sunday to discuss the project, assess our progress, and decide what tasks to perform during the upcoming week. This method worked pretty. It kept the team on track. As a result almost all of the functionality originally specified in the language reference manual was included in the compiler.

4.3.2 Specification

During the Sunday group meeting, immediately following the Wednesday the LRM was due, the group discussed how to build the compiler given the LRM and the example MicroC compiler. The specification of the compiler changed as we realized the difficulty of implementing certain features given the allotted amount of time. These changes have been noted in section 3 of this document.

4.3.3 Development

Initially we imagined an iterative approach to development where the team added a feature tested it, then added another feature and tested it, and so on. What actually happened was the scanner, parser, and AST modules were created before the code generation module was created. The code generation module was intern fully functional by the time the semantic analysis module was started. Testing did happen incrementally, by feature. That went as originally planned.

The semantic analysis module was the last major component to be created. It was created and tested during about the last 2 weeks of the semester. Our initial assumption was that it would be difficult to do semantic analysis with a partially complete AST module. This proved to be true, as it was easier to implement the semantic analysis module after the rest of the compiler front-end (i.e. scanner, parser, and AST modules) was close to being complete. In hindsight, the insight we gained while doing the semantic analysis would have informed the development of the rest of the compiler had it been created in tandem with the other components.

4.3.4 Testing

The testing process was managed by Michael. Test case files were written by Michael, and contributed by other team members, as features were implemented. The testing process is described in greater detail in section 6 of this document.

4.4 Programming Style Guide

4.4.1 General Principles

The team had a very informal approach to programming style. We attempted to follow the coding style of the example Microc compiler provided during the course of the term. This worked well at first as we divided up the modules very cleanly. We learned during the last few weeks of the semester that this informal approach was a mistake. We began to trip over each when more fully testing and patching the modules. The largest problem was that each team member was using a different text editor or IDE with different settings. The tab width in one team members editor might have been eight whitespace characters, in another two whitespace characters, and another a one four character wide tab. This caused a huge formatting headache and could have been avoided if we had standardized our tab width to start with.

4.4.2 Documentation Comments

We found that the Professor Edwards thoughts regarding commenting O'Caml code held true throughout the development of the compiler. Namely that O'Caml's succinct syntax makes it very understandable, and thus a large quantity of verbose comments was not needed to explain the functionality of the various modules.

4.5 Project timeline

The following target dates were set to for the various project milestones

Date	Milestone
09-28-2011	Language proposal, with core language feature complete
10-31-2011	The language reference manual complete
11-13-2011	Compiler 0 – Able to print "Hello, World!"
11-19-2011	Scanner and Parser complete
11-26-2011	Code generation complete
12-16-2011	Semantic analysis complete
12-18-2011	Compiler fully complete

4.6 Project Log

The project log, shown below, is essentially the milestones and highlights from the project's Subversion repository commit log. It has been rearranged by feature, and then sorted chronologically. Minor commits and bugfixes were left out, as well as those lacking sufficient detail to discern the purpose of the commit. The full commit log can be viewed at: http://code.google.com/p/plt-lsystem/source/list

4.6.1 Scanner/Parser/AST

Date	Team Member	Milestone/Feature
11/6	Jervis	Implemented a basic working scanner & parser based on MicroC.
11/14	Jervis	Added the compute functions. Implemented the type double and boolean in parser and AST.
11/15	Jervis	Completed implementing the def_draw function that's used to describe lsystem. Tested out with hilbert function described in LRF to make sure that the program is parsed successfully.
11/16	Jervis	Added the POW (^) operator and updated make file to produce verbose output when parser is compiled.

4.6.2 Command-Line Interface

Date	Team Member	Milestone/Feature
11/13	Ethan	Create basic top-level based on Microc.
11/14	Jervis	Added a -a switch for AST printing.
11/20	Ethan	Top-level has been converted to a command line interface. Added improvements such as usage instructions as well as exception handling for invalid arguments.
12/4	Jervis	Added a -s option do that the semantic analysis stage can be run.
12/18	Ethan	Added a -t switch for "testmode"
12/19	Jervis	Reconfigured the CLI so that the semantic analysis stage runs before the code generation stage, and not indepedantly.

4.6.3 Code Generation

Date	Team Member	Milestone/Feature
11/13	Ethan	Compiler 0: The compiler produces compilable Java source code that prints "Hello, World!" to the console. Main and other compute functions are translated.
11/27	Ethan	Draw function code generation complete. Compiler able to produce Java code that renders images from L-systems.
12/14	Ethan	Compiler now produces Java bytecode (not just Java source code) with the –c command.
12/17	Ethan	The start rule of a draw function comes from its lambda rule, and not a function argument.

12/17	Ethan	Added slider that scales the rendered L-system image.
12/18	Timothy	Made standard library functions assignable to constants in L-system. Broke slider control. Image no longer scales.

4.6.4 Standard Library

Date	Team Member	Milestone/Feature
11/6	Timothy	Created Java Turtle class.
11/14	Ethan	Standard library module created. Java code integrated.
12/17	Ethan	Added slider that scales the rendered L-system image.
12/18	Timothy	Made standard library functions assignable to constants in L-system.
12/18	Timothy	Made the image scale with Java program resizable.
12/19	Timothy	The -t flag now outputs a bitstring of the rendered image to a text file for testing.

4.6.5 Testing

Date	Team Member	Milestone/Feature
11/6	Michael	Created arithmetic tests.
11/20	Michael	Updated test.sh bash script to iterate through all test cases with updated syntax, listing test cases that failed at the end of execution.
12/4	Michael	Updated test.sh bash script to attempt to compile all test cases from .ls to java byte code (.ls -> .java -> .class).
12/15	Michael	Began adding semantic tests.
12/15	Ethan	Added a variety of draw tests.
12/19	Michael	Finished testing to validate drawing programs. Test script should essentially be finished. Added a directory with expected bitstring result files for each drawing test file.

4.6.6 Semantic Analysis

Date	**Team Member**	**Milestone/Feature**
12/4	Jervis	Outlined basic functions that will be used to semantic analysis.
12/14	Jervis	Implemented data type checking.
12/17	Jervis	Added checks for compute functions, control flow functions, and other constructs
12/19	Jervis	Completed semantic analysis for draw functions.
12/19	Jervis	Connecting semantic analysis with the normal compilation process.

4.6.7 Commit Statistics

The graphic in figure 4.6.0.1 depicts the commit statistics for the life time of the project. It was generated using project management software called Redmine. In the graphs, **Revisions** refers to a count of repository commits and is displayed in reddish/orange. **Changes** refers to a count of the number of files that have changed overall and is displayed in blue. Note that “engiskahn09” refers to Michael in the lower graph.

The commit statistics highlight that even though we began developing the compiler in early November, and worked on it steadily, more commits were done in December. This is due to the bug fixes and code reformatting that took place at the end of the project.

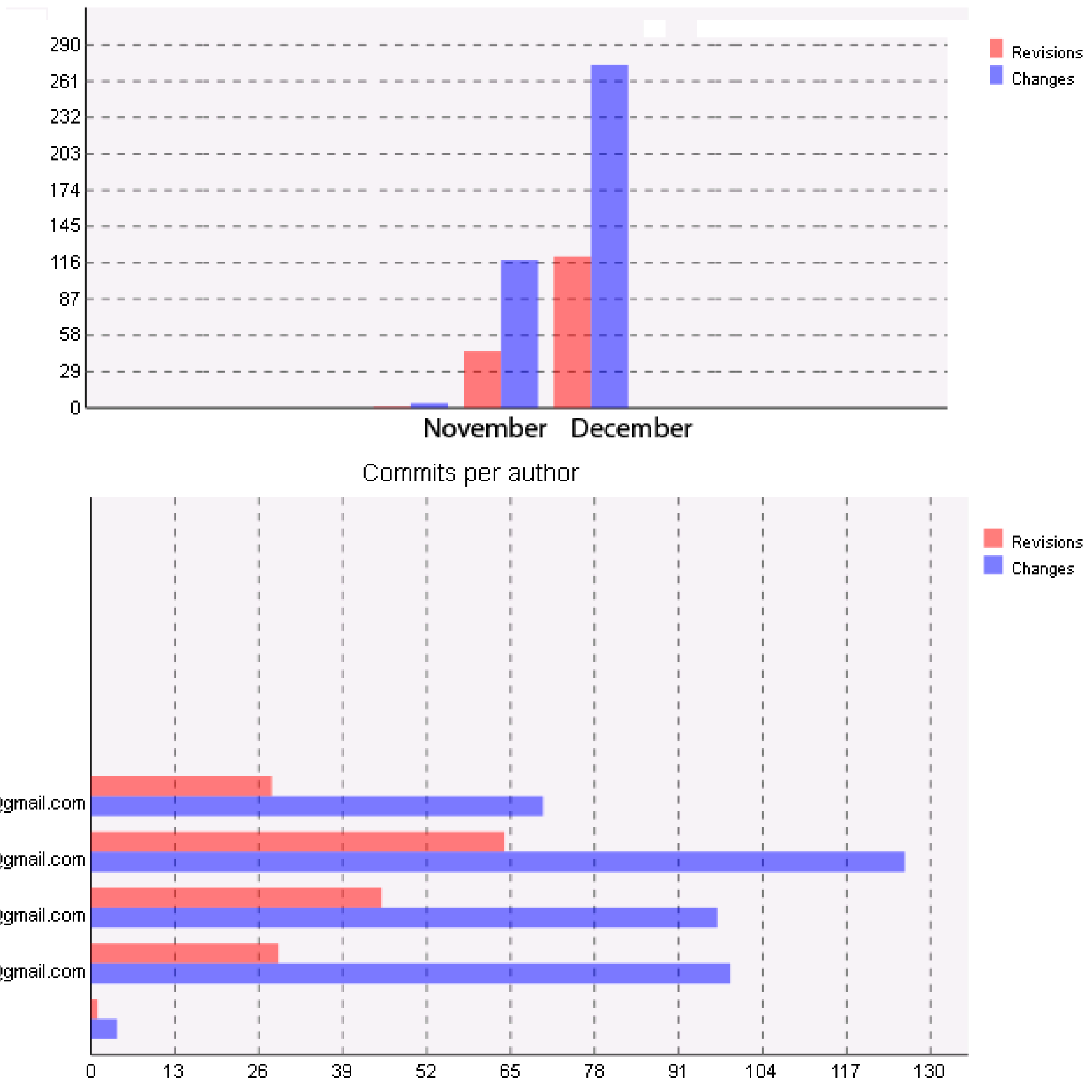

Figure 4.6.0.1 – Subversion Repository Commit Statistics

Section 5: Architectural Design

5.1 High Level Architectural Design

The LSystem compiler consists of seven modules. The modules are depicted in figure 5.1.1, and their purposes are explained in the subsequent subsections of section 5.

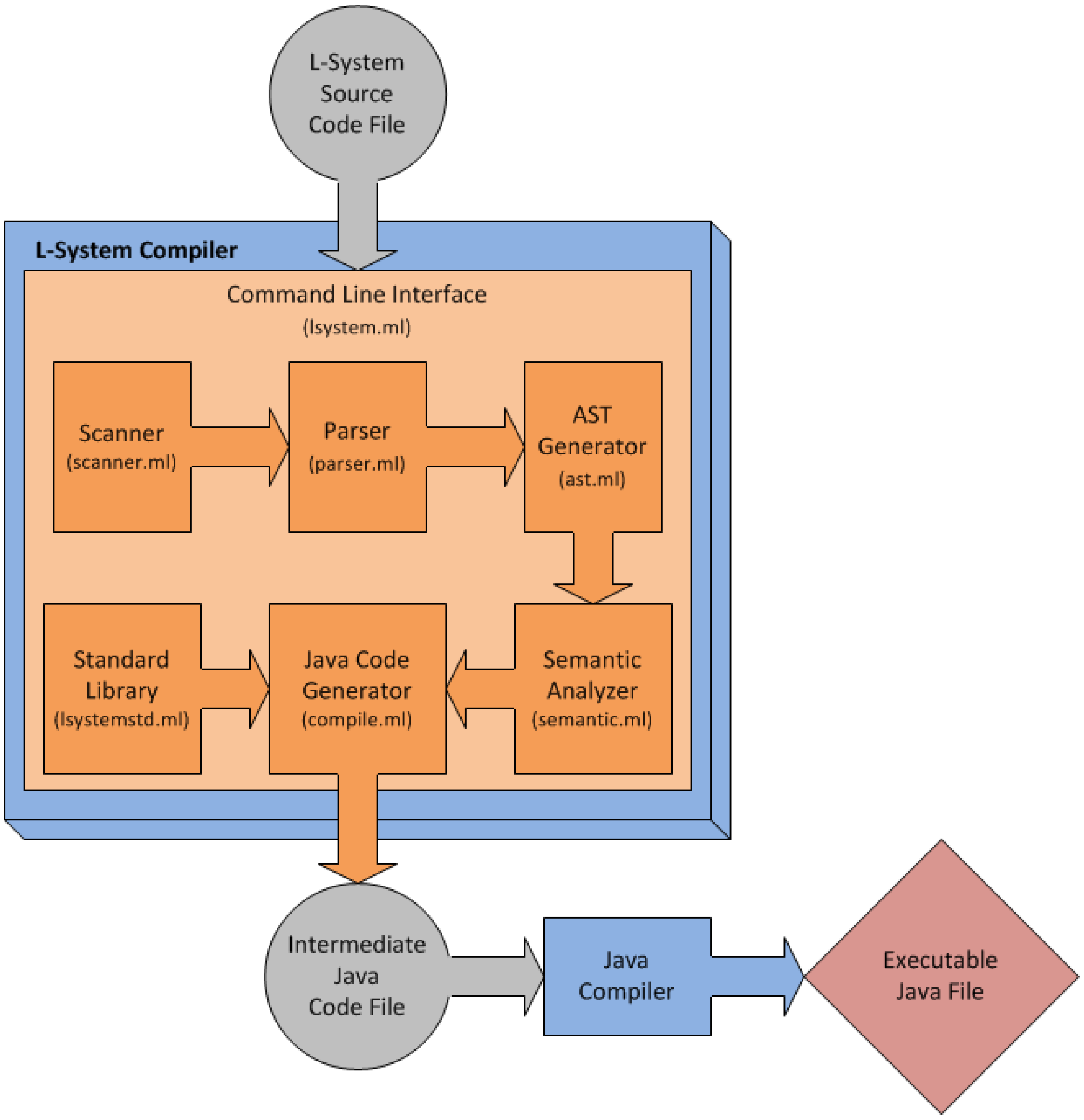
L-System Source Code File
L-System Compiler
Command Line Interface (lsystem.ml)
Scanner (scanner.ml)
Parser (parser.ml)
AST Generator (ast.ml)
Standard Library (lsystemstd.ml)
Java Code Generator (compile.ml)
Semantic Analyzer (semantic.ml)
Intermediate Java Code File
Java Compiler
Executable Java File

Figure 5.1.1 – Compiler Architecture and Related Components

5.2 Component Interface Interaction

5.2.1 Command Line Interface (lsystem.ml - Author: Ethan Hann)

The LSystem CLI evolved out of the Microc top-level. It, however, is not a top-level, but a switch-centric command line interface for the LSystem compiler. The CLI is used to invoke the various stages of the compiler. The ordered list of these stages includes:

1. AST generation
2. Semantic analysis checking
3. Intermediate (Java) code generation

This list is ordered to indicate that each stage requires that the previous stages in the list are invoked in sequence before the desired stage is itself invoked. For example, if the **-s** switch is passed to the CLI (see "CLI Usage" below) to perform semantic analysis, stage 1 (AST generation) is executed before stage 2 (semantic analysis). This is logical, as semantic analysis cannot be performed unless the AST has been generated.

```
lsystem [-a|-s|-c] SOURCE_FILE [-t|-v]
```

Figure 5.2.1.1 CLI Usage

The compiler accepts a switch that corresponds to a stage in the above list as its first argument. This is either -a (AST generation), -s (semantic analysis), -c (generate Java code/program). The second argument is the file system location of the target LSystem source file. The third argument is either –t (test mode), or –v (verbose mode). More complete explanations of these arguments are contained in figure 5.2.1.2.

Switch	**Explanation**
-a	Generates the AST and prints an exact copy of the input source file if AST generation was successful.
-s	Performs semantic analysis on the AST, generated from the input source file.
-c	Generates an intermediate representation of the source file in the form of Java code from the AST.
SOURCE_FILE	The file path of the target LSystem source file.
-v	Verbose option. Prints intermediate code, and other information useful for debugging.
-t	Test mode option. Allows the compiler to perform additional testing of draw functions.

Figure 5.2.1.2 – CLI Argument Descriptions

5.2.2 Scanner (scanner.mll - Author: Jervis Muindi)

The purpose of the scanner is to specify what tokens are recognizable in our language. That is, the scanner goes through the source file and transforms the stream of character that are present in the source into a stream of token which are specified in this. This process is necessary in reject all source programs that do not use the syntax of our language.

5.2.3 Parser (parser.mly - Author: Jervis Muindi)

The role of the Parser is to take the stream of tokens obtained from the scanner and try to deduce whether they are in the language specified by the context-free grammar. The context-free grammar, in our case was written in the parser.mly file. In the process of parsing the source file, an abstract syntax is also generated.

5.2.4 AST (ast.ml - Author: Jervis Muindi)

The abstract syntax tree defines the core structure of a program in a language and this file (ast.ml) contains the precision definition of the structure of our abstract syntax tree for our language. The parser.mly file references the ast.ml file so that during the process of parsing the program, the abstract syntax tree is also generated.

5.2.5 Semantic Analyzer (semantic.ml - Author: Jervis Muindi)

The semantic analyzer examines the abstract syntax tree structure produced by the AST module. If it completes its analysis without throwing any exceptions the compiler proceeds to the code generation phase of the compilation process.

5.2.6 LSystem Standard Library (lsystemstd.ml - Author: Timothy Sun)

The LSystem standard library (STL) functions are written in Java. They are concatenated to the intermediate Java code produced by the compiler in the Java Code Generator component. It provides the Java drawing and console output functions. It also dynamically scales the rendered image so that it fits on screen.

5.2.7 Java Code Generator (compiler.ml - Author: Ethan Hann)

The code generator transforms the AST into Java source code and combines it with the Java code in the LSystem standard library (STL). The resultant code is written to a Java source file that shares the same name as the LSystem source file. For example, if the LSystem source file is called “hilbert.ls” (sans quotes) then the Java source code file would be called “hilbert.java” (again, sans quotes). After producing the intermediate Java source file the code generator then attempts to compile the intermediate Java code into a Java bytecode program by using an external Java compiler located in the user’s path.

Section 6: Test Plan

Formal testing began as soon as a rudimentary compiler was constructed. The test suite on this project was designed to be built alongside the L-system compiler. As soon as features were implemented in the compiler, corresponding test classes were created to verify that these features were in working order. This agile approach allowed for features to be tested immediately, leading to quick verification and easing any necessary debugging. Furthermore, these tests were carried over from build to build, constructing a suite of regression tests. This further reinforced build integrity, as the test suite could be executed with each new build to ensure that new changes did not break any existing functionality. The net result was a project developed with no emergency code rollbacks and a very fast quality assurance process.

The test suite can be divided into three parts that reflect the three main phases of development experienced by the project. Owing to the agile nature of testing, these parts were developed in parallel with their corresponding phases.

6.1 Phase 1: Rudimentary Compiler

Phase 1 of the development process was the initial development phase, where a rudimentary compiler had been created. The development goal at the end of this phase was to have a compiler that could handle purely computational programs. As such, functions such as arithmetic, printing, and basic program structure were tested during this phase.

6.2 Phase 2: L-system Drawing

Phase 2 of the development process corresponded to the development of portions of the compiler dedicated to handling L-system drawing programs, the main objective of the project. The goal at the end of this phase was to have a compiler that could handle drawing and computational programs, outputting a visual representation of L-systems on-screen in the case of the former. All functionality pertaining to creating L-system drawing programs, such as defining custom terminal variables or customizing the number of iterations to use in expanding an L-system grammar, were tested during this phase.

6.3 Phase 3: Semantic Analysis

Phase 3 of the development process corresponded to the construction of the semantic analyzer, designed to catch syntactic and programmatic errors in files before beginning formal compilation into intermediate Java source code. The goal at the end of this phase was to develop functionality to catch errors in input programs that would trigger errors or warnings from the Java compiler if the intermediate Java source code were to be compiled, in addition to catching syntactic errors pertaining to the L-system grammar itself. Common errors and edge cases, such as defining a non-boolean condition in a conditional statement, attempting to create a program without a *main* method, and incorrectly instantiating variables were tested during this phase.

6.4 Tools

A bash shell script was used to automate the execution of all tests from each phase. Individual test programs were written in the L-system programming language.

6.5 Implementation

6.5.1 Implementation Phase 1

Since this phase was concerned with processing computational programs, test programs were created to test basic computational functionality. Each of these test programs would focus on a piece of functionality- for instance, a test program was created to test the subtraction of two integers, and another test program was used to test the subtraction of two floating-point doubles. The common thread amongst the tests in this phase was that they all were expected to run successfully with a specific end result, akin to making an assertion during unit testing that a manipulated piece of data in a test method was in fact manipulated as expected. To that end, a data file was created with values of expected output for each test file made during this phase. When testing automation was implemented, each of these programs was compiled and executed, with the output being checked against the corresponding expected output in the data file.

An example test program from phase 1:

```
def compute test(double a, double b)
{
	return (a + b);
}

def compute main()
{
	int x = 3;
	double y = 2.0;
	double r = 0;
	r = test(x, y); # int can be used in a place of a double.
	print(r);
}
```

Figure 6.5.1.1 - validcall1.ls test

6.5.2 Implementation Phase 2

The goal in this phase was to ensure that drawing programs would be executed correctly and could be customized according to the specifications laid out in the language reference manual. The test programs created during this phase can be divided into two subgroups- one consisting of well-known L-systems that are likely to be programmed in the language, allowing for easy visual verification of correctness, and the other consisting of programs that customize some aspect of the L-system grammar- for instance, a program that explicitly maps all of an L-system's terminal symbols to specific drawing functions.

Image data from the visual representation of the L-systems was utilized to verify the test programs in this phase. Automation implementation of this phase was similar to phase 1; an image for each test program was produced that was known to be correct. This image was then transformed into pixel data, consisting of a bit for every pixel in the image that determined whether the pixel was black or white. This pixel data effectively formed a large bitstring that was stored in a data file, one for each test program. When a test program was executed in subsequent builds, the image bitstring was extracted from its generated image and compared to the corresponding data file to verify image correctness.

An example test program from phase 2:

```
def draw levycCurve(int level){
      alphabet:  (X);
      rules:{
      lambda -> X;
      X -> r f X l l f X r;
      l = turn(-45); #Mapping custom variables to built-in drawing methods
      r = turn(45);
      }
}

def compute main(){
    levycCurve(12);
}
```

Figure 6.5.2.1 – levyc.ls test

6.5.3 Implementation Phase 3

This phase centered around ferreting out issues that could arise from improperly coded programs- for instance, rejecting an input L-system program if the user did not define a *main* method in the program body. The goal was to ensure that programs not adhering to the syntax laid out in the language reference manual would be appropriately rejected by the compiler before compilation was actually attempted. Automation of this process focused on ensuring that for each test program, the compiler prematurely exited due to an error in the program and did not generate an intermediate Java source code file.

Due to time constraints, the determination of which errors to check during this phase was based mostly on what the developers believed would be common mistakes or likely attempts at breaking the built-in grammar.

An example test program from phase 3:

```
def draw hilbertDerivation(int level){
	alphabet:  (A , B, C, D);
	rules:{
	lambda -> A;
	A  ->  l B f r A f A r f B l;
	B  ->  r C f l B f B l f C r;
	C  ->  l D f r C f C r f D l;
	D  ->  r A f l D f D l f A r;
	f = forward(1);
	l = turn(-80);
	r = turn(80);
	}
}

def compute main()
{
	hilbertDerivation(7, 3);
}
```

Figure 6.5.3.1 – toomanydrawfunctionparameters.ls

6.6 Automation:

As previously stated, a bash shell script was written to automate all testing. For the computational and drawing groups of test programs, the tests were compiled, the script verified that compilation from the L-system language to Java occurred and that compilation from Java to a Java class file occurred, and then the programs were executed and their output compared appropriately to expected values, as detailed above. For the semantic group of test programs, the programs were fed into the compiler, and the script verified that no intermediate Java file was generated for each test, indicating that the compilation had failed due to an error present in each test.

Section 7: Lessons Learned

7.1 Jervis

7.1.1 Lessons Learned

Taking this PLT class has been a lot of fun and I have learned so much. I had previously approached compilers as black boxes that magically did their work. However, after taking this class, I have come to know that there is actually no black magic and that the process of compilation is well structured one that involves stages of scanning, parsing and eventual code generation. For example, the simplicity of the Donald Knuth algorithm to build a LR(0) automaton which is in turn used to build the SLR parsing table is something that I find to be amazing.

Additionally, I have also come around to the functional paradigm of programming. In particular, when I first starting learning OCaml, I was annoyed with how picky the compiler was - it seemed to me that it would complain even about the slightest of problems. However, in writing the compiler in OCaml, I have grown to actually appreciate the error checking that the compiler does. In every single case where I got an error message, it was to an actual problem in the code. The benefit of this strict error checking by the compiler is that when it compiles actually successfully, it will always work.

Moreover, working and collaborating together in a group has been a good learning experience. I got to see the usefulness of using code control tools such as SVN in code management as well as the importance of having a good plan of division of labor. In particular, in my group, we are able to divide the tasks in way that we were all able to work on different parts of the compiler simultaneously. We did this as much as possible to avoid the problem of having to wait on a certain team member to complete a certain module before work on another could begin.

Furthermore, in developing the compiler, I also saw how useful and crucial it is to integrate testing as part of the development process. Having a good testing suite helped ensure that as we added more features to the compiler or perhaps after rectifying a known bug, we did not inadvertently introduce extra bugs in the process.

Lastly, I have tremendously enjoyed the class lectures which were always interesting and educative. Of particular note are discussions on different language paradigms such as logic programming and lambda calculus. It was very refreshing to see the mathematical and theoretical underpinnings on which functional programming languages such as OCaml are actually built upon.

7.1.2 Advice for Future Teams

My advice to future team is to first choose your team members well since you will need to collaborate very closely with them over the course of the semester. It is also important to figure out your working style and how you are going to work. Of crucial importance, is having weekly progress meeting so that you can track your progress and discuss future goals for the coming week. If you're in a team where you're unable to agree and adhere on a weekly meeting time, I'd strongly suggest joining another team.

My second piece of advice is to start learning OCaml early on especially if it is the first time that you are encountering functional paradigm of programming. Yes, it will be challenging and difficult at first – which is why you should start early – but the final compiler has to be written in OCaml and not having a good grasp of OCaml by the time you actually need to start implement the compiler is a situation you would not want to be in.

While still on the topic of OCaml, I should mention that in OCaml, there are only two possible types of errors that you can get : syntax errors and type-mismatch errors. You are will be seeing and dealing with these a lot so it's also important to understand how to read and resolve the errors. When starting out you may feel that the OCaml compiler very picky – I certainly did – but rest assured that the error messages are valid ones. Indeed, every single time I have had to resolve such errors, there was indeed a problem with the code. What this means, is that when you are able to get your code to compile successfully, it really is going to work.

Also, I know everyone says this (and you probably already know it yourself) but I feel that it is important enough that it bears repeating again: You should start your work early. Actually, don't start early, start *earlier*. Seriously, you should figure out your project as soon as possible so that this way you can get more time to work on it. Starting early also gives you the nice luxury of having some buffer time should you even need it.

With regards to actually implementing the compiler an excellent starting point is the MicroC compiler that Professor Edwards puts up on the website. Note that if your language deviates significantly from a C-style syntax then, it would probably be best to start from scratch. That said, it is still a good idea to look at the MicroC compiler for nothing else than seeing how the various parts of a compiler can be built using OCaml.

7.2 Ethan

7.2.1 Lessons Learned

The most important lesson I learned was that adding constraints to the syntax of the language in order to simplify the compiler often makes implementing the compiler more difficult than it would otherwise be. For example, specifying that the language has a main function and that function has to be the last function defined in a source file means that you have to add extra semantic test cases to make sure that the main function is actually the last function defined in a source file. If the order of the functions does not matter (and it should not, as the code generation module can work however you want it to… reordering the generated code as it needs for instance) during intermediate code generation, then that extra work is a waste of time.

7.2.2 Advice for Future Teams

Working steadily on the project over time is beneficial. It is good to start early, but continuing to work on the project steadily every week over the course of the semester is better than having marathon coding sessions every three weeks. This goes for the final report as well. Thankfully my group started the final report right after the LRM was due. This made the last few weeks of the semester a lot easier.

Write your coding standard before you start coding. We had team members using Eclipse, vim, or gedit at any given time with all different tab settings. This caused formatting problems. Standardize the tab width among team members at least.

7.3 Michael

7.3.1 Lessons Learned

Building up a test suite as the compiler is developed is optimal- it saves time in the long run and serves as a suite of regression tests that can be used to ensure that new builds didn't break any existing functionality.

There's a reason why version control systems are used in virtually all corporate software development projects, especially since a lot of them have built-in bug tracking systems. Having everything integrated into one application like that makes the development process a lot faster and easier.

7.3.2 Advice for Future Teams

Start early! At the very least, plan out how you're going to grow your project- figure out what to get working first, and then build off of that. Having that sequence of things to implement makes the process a lot smoother.

Start the report early as well- even if you just copy/paste the outline and fill in bullet points as you go. Memories of stuff you did on a given day will be fresher and reports will be more accurate the sooner you log what you did.

7.4 Timothy

7.4.1 Lessons Learned

Working on the code regularly allowed me to actually know what was going on. I didn't understand all of the code my teammates wrote all at once, but had I not looked at the code periodically, I would not have really gotten anywhere with the code.

Functional programming is cool. As a math-oriented person, a lot of it made a lot of intuitive sense after I figured out the syntax. I can't imagine having to write a parser in Java. Higher-order functions and list-manipulating functions were probably the best features for me.

I have no idea how we would have gotten through the semester without SVN. "svn update" is so much easier than e-mails, etc.

7.4.2 Advice for Future Teams

Make sure you'll enjoy whatever language you're working on. Even though it might not turn out exactly as you wanted it to (our language is missing some features I had envisioned at the beginning), but trust me: seeing something close to the final product is a great feeling. When we had just finished the drawing code generation, I was plugging in every L-system specification I could find online just so I could see the program spit out some beautiful fractal.

Starting early. Even though some people might work better under pressure, you still want to spread everything out. We probably missed all our projected milestones, but oh well.

Get used to reading/writing O'Caml code. It's perhaps not as immediately accessible as languages like Java or Python, but it's a surprisingly clean language to work in. Agree on coding conventions early on; it's hard to read different styles.

On a less serious note: don't use Eclipse. I have never seen more random whitespace in my life.

Appendix A: Code Listing

lsystem.ml

```
(* Primary Author: Ethan Hann (eh2413) *)
(* Command Line Interface *)

(* Possible actions for the compiler. *)
type action = Ast | Compile | SA

(* Custom exceptions. *)
exception NoInputFile
exception InvalidArgument

(* Compiler usage instructions. *)
let usage = Printf.sprintf "Usage: lsystem [-a|-s|-c] SOURCE_FILE [-t|-v]"

(* Get the name of the program from the file name. *)
let get_prog_name source_file_path =
      let split_path = (Str.split (Str.regexp_string "/") source_file_path)
in
      let file_name = List.nth split_path ((List.length split_path) - 1) in
      let split_name = (Str.split (Str.regexp_string ".") file_name) in
            List.nth split_name ((List.length split_name) - 2)

(* Main entry point *)
let _ =
      try
            let action = if Array.length Sys.argv > 1 then
                  match Sys.argv.(1) with
                              | "-a" -> Ast
                              | "-s" -> SA (*semantic analysis testing*)
                              | "-c" -> Compile
                              | _ -> raise InvalidArgument
                  else raise InvalidArgument in
            let prog_name =
                  if Array.length Sys.argv > 2 then
                        get_prog_name Sys.argv.(2)
                  else raise NoInputFile in
            let verbose =
                  if Array.length Sys.argv > 3 then
                        match Sys.argv.(3) with
                        | "-v" -> true
                        | _ -> false
                  else false in
            let testmode =
                  if Array.length Sys.argv > 3 then
                        match Sys.argv.(3) with
                        | "-t" -> true
                        | _ -> false
                  else false in
            let input_chan = open_in Sys.argv.(2) in
      let lexbuf = Lexing.from_channel input_chan in
      let reversed_program = Parser.program Scanner.token lexbuf in
            let program = List.rev reversed_program in
            match action with
                  | Ast -> let listing = Ast.string_of_program program in
print_string listing
                  | SA -> ignore (Semantic.check_program program);
                  | Compile -> if Semantic.check_program program then
                                          let listing = Compile.translate
program prog_name verbose testmode in
                                          print_string listing
                                     else raise(Failure("\nInvalid
program.\n"))
      with
            | InvalidArgument -> ignore (Printf.printf "InvalidArgument\n
%s\n" usage)
            | NoInputFile -> ignore (Printf.printf "The second argument must
be the name of an l-system file\n %s\n" usage)
```

scanner.mll

```
(* Primary Author: Jervis Muindi (jjm2190) *)
{ open Parser }

let letter = ['a'-'z' 'A'-'Z']
let digit = ['0'-'9']
let punc = ['~' '`' '!' '@' '#' '$' '%' '^' '&' '*' '(' ')' '-' '+' '=' ','
'.' '?' '/' '<' '>' ':' '''  ';' '{' '}' '[' ']' '|' ' ']
(*Escape character sequences
  "\\\"" -> "[ \" ]" -> a single double quote
      "\\\\" -> '\\' -> a backslash
      "\\n" -> \n -> new line
      "\\t" -> \t -> tab char
*)
let esp =   "\\\"" | "\\\\" | "\\n" | "\\t" (*Escape characters : see comment
above*)
let exp = 'e'('+'|'-')?['0'-'9']+
let float = '-'? (digit)+ ('.' (digit)* exp?|exp)
let stringlit = '"' (letter | digit | punc | esp)*  '"'
let negative_int = '-'(digit)+

rule token = parse
  [' ' '\t' '\r' '\n'] { token lexbuf } (* Whitespace *)
| '#'     { comment lexbuf }            (* Comments *)
| '('      { LPAREN }
| ')'      { RPAREN }
| '{'      { LBRACE }
| '}'      { RBRACE }
| ';'      { SEMI }
| ','      { COMMA }
| ':'      { COLON }
| '+'      { PLUS }
| '-'      { MINUS }
| '*'      { TIMES }
| '/'      { DIVIDE }
| '='       { ASSIGN }
| "&&"      { AND }
| "||"      { OR }
| '^'       { POW }
| "=="      { EQ }
| "!="      { NEQ }
| '<'       { LT }
| "<="      { LEQ }
| ">"       { GT }
| ">="      { GEQ }
| "->"      { ARROW }
| "alphabet" { ALPHABET }
| "boolean" { BOOLEAN }
| "def"      { DEF }
| "compute" { COMPUTE }
| "draw" { DRAW }
| "double" { DOUBLE }
| "false"   { FALSE }
| "true"    { TRUE }
| "if"      { IF }
| "else"    { ELSE }
| "int"     { INT }
| "lambda" { LAMBDA}
| "return" { RETURN }
| "rules"   { RULES }
| "string" { STRING }
| "while"   { WHILE }
| (digit)+ as lxm { LITERAL(int_of_string lxm) }
| negative_int as lxm { LITERAL(int_of_string lxm) } (*negative integer*)
| letter as lxm { LETTER(String.make 1 lxm)  } (*converts lxm to a string*)
| letter (letter | digit | '_')* as lxm { ID(lxm) }
| float as lxm { FLOAT(float_of_string lxm) }
| stringlit as lxm { STRINGLIT(lxm) }
| eof { EOF }
| _ as char { raise (Failure("Illegal character: " ^ Char.escaped char)) }

and comment = parse
  '\n'  { token lexbuf } (*Comments are in effect until the end of the line*)
| _     { comment lexbuf }
```

parser.mly

```
%{
(* Primary Author: Jervis Muindi (jjm2190) *)
open Ast
let parse_error s = (* Called by the parser function on error *)
  print_endline s;
  flush stdout
%}

%token SEMI LPAREN RPAREN LBRACE RBRACE COMMA COLON
%token PLUS MINUS TIMES DIVIDE POW ASSIGN
%token EQ NEQ LT LEQ GT GEQ
%token AND OR
```

```
%token BOOLEAN DOUBLE STRING INT
%token FALSE TRUE
%token ALPHABET LAMBDA RULES
%token RETURN IF ELSE WHILE
%token DEF COMPUTE DRAW
%token ARROW
%token LETTER
%token <int> LITERAL
%token <float> FLOAT
%token <string> STRINGLIT
%token <string> ID
%token <string> LETTER
%token EOF

%nonassoc NOELSE
%nonassoc ELSE
%right ASSIGN
%left OR
%left AND
%left EQ NEQ
%left LT GT LEQ GEQ
%left PLUS MINUS
%left TIMES DIVIDE
%right POW

%start program
%type <Ast.program> program


%%

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

fdecl:
        DEF COMPUTE id LPAREN formals_opt RPAREN LBRACE vdecl_list stmt_list
RBRACE
                  {
                        CFunc({
                              fname = $3;
                              formals = $5;
                              locals = List.rev $8;
                              body = List.rev $9
                        })
                  }
      | DEF DRAW id LPAREN formals_opt RPAREN LBRACE rules RBRACE
                  {
                        DFunc({
                              name = $3;
                              formal = $5;
                              rules = $8;
                        })
                  }

id:
        ID     { $1 }
      | LETTER { $1 }

alphabet_list:
        LETTER                     { [$1] }
      | alphabet_list COMMA LETTER { $3 :: $1 }

alphabet:
      ALPHABET COLON LPAREN alphabet_list RPAREN SEMI { Alphabet($4) }

production_list: /*the RHS of a production rule*/
        LETTER                 { [$1] }
      | production_list LETTER { $2 :: $1 }

turtle_func_paramlist_opt:
        /* nothing */         { [] }
      | turtle_func_paramlist { List.rev $1 }

turtle_func_paramlist:
        expr                             { [$1] }
      | turtle_func_paramlist COMMA expr { $3 :: $1 }

erule: /*expansion rule*/
      LETTER ARROW production_list SEMI { ERule($1, List.rev $3) } /*Reverse
it so that we read in list in the right order going from left to right*/

frule: /*rule that specifies */
        LETTER ASSIGN SEMI                                          {
EmptyFRule($1) } /*the empty rule. e.g A = ;*/
      | LETTER ASSIGN ID LPAREN turtle_func_paramlist_opt RPAREN SEMI {
FRule($1, $3, $5)    }

lambdarule:
      LAMBDA ARROW production_list SEMI {Lambda(List.rev $3) } /*Reverse it
so that we read in list in the right order going from left to right*/

rule:
        erule { $1 }
      | frule { $1}

rule_list:
        rule             { [$1] }
      | rule_list rule   { $2 :: $1 }

rules:
      alphabet RULES COLON LBRACE lambdarule rule_list RBRACE { LSystem($1,
$5, List.rev $6) } /*Apply List.rev so that the rules are printed in the
right order going from top to bottom as they were originally entered.*/

datatype:
        BOOLEAN { BooleanType }
      | INT     { IntType }
      | DOUBLE  { DoubleType }
      | STRING  { StringType }

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

formal_list:
        datatype id                   { [FParam($1, $2)] }
      | formal_list COMMA datatype id { FParam($3, $4) :: $1 }

vdecl_list:
        /* nothing */    { [] }
      | vdecl_list vdecl { $2 :: $1 }

vdecl:
      datatype id ASSIGN expr SEMI { VDecl($1, $2, string_of_expr $4) }

stmt_list:
        /* No empty block allowed */ { [] }
      | stmt_list stmt               { $2 :: $1 }

stmt:
        expr SEMI                                  { Expr($1) }
      | RETURN expr SEMI                           { Return($2) }
      | 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) }

expr: /*a primary expression*/
        LITERAL                      { Literal($1) }
      | STRINGLIT                    { String($1) }
      | FLOAT                        { Float($1) }
      | id                           { Id($1) }
      | expr PLUS  expr              { Binop($1, Add,   $3) }
      | expr MINUS  expr             { Binop($1, Sub,   $3) }
      | expr TIMES  expr             { Binop($1, Mult,  $3) }
      | expr DIVIDE expr             { Binop($1, Div,   $3) }
      | expr POW expr                { Binop($1, Pow,   $3) }
/* Boolean expression part*/
      | TRUE                         { BVal(True) }
      | FALSE                        { BVal(False) }
      | expr EQ expr                 { EExpr($1, BEqual, $3) }
      | expr NEQ expr                { EExpr($1, BNeq, $3) }
      | expr GT expr                 { RExpr($1, BGreater, $3) }
      | expr GEQ expr                { RExpr($1, BGeq, $3) }
      | expr LT expr                 { RExpr($1, BLess, $3) }
      | expr LEQ expr                { RExpr($1, BLeq, $3) }
      | expr AND expr                { BExpr($1, And, $3) }
      | expr OR expr                 { BExpr($1, Or, $3) }
      | id ASSIGN expr               { Assign($1, $3) }
  | id LPAREN actuals_opt RPAREN { Call($1, $3) }
  | LPAREN expr RPAREN           { Bracket($2) }

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

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

ast.ml

```
(* Primary Author: Jervis Muindi (jjm2190) *)

type op = Add | Sub | Mult | Div | Equal | Neq | Less | Leq | Greater | Geq |
Pow
type nop = NAdd | NSub | NMult | NDiv (*the four normal operators*)

type bv = True | False
type bop = And| Or
type eop = BEqual | BNeq
type rop = BLess | BLeq | BGreater | BGeq

type mop = MTimes | MDivide | MMod (*multiplicative expr ops*)
type aop = AAdd | ASub (*additve expr ops*)

type vop = VAdd | VSub | VMult | VDiv
type dt = StringType | DoubleType | IntType | BooleanType (*Data types in our
language*)

type fparam = FParam of dt * string (*Type to hold a Formal parameter, e.g.
int x*)

type vdecl = VDecl of dt * string * string  (*DataType, Name, Value*)

type arithexpr = (*key*)
        | ALiteral of int
        | AId of string
        | AFloat of float

type varexpr = (*key*)
        | VLiteral of int
        | VId of string
        | VFloat of float
        | VStringLit of string
        | VBoolLit of bool
        | VBinop of varexpr *  vop * varexpr

type expr =
          Literal of int
        | Float of float
        | Boolean of bool
        | String of string
        | Id of string
        | Bracket of expr
        | Binop of expr * op * expr
        | Assign of string * expr
        | Call of string * expr list
        | Noexpr
        | BVal of bv (*boolean value : true/false*)
        | RExpr of expr * rop * expr (*relational expresion : < <= > >=*)
```

```
      | EExpr of expr * eop * expr (*equality expression : == !=*)
      | BExpr of expr * bop * expr (*boolean compound expression : && || *)

type stmt =
        Block of stmt list
      | Expr of expr
      (*| Decl of dt * string * string*)
      | Return of expr
      | If of expr * stmt * stmt
      | For of expr * expr * expr * stmt
      | While of expr * stmt

type alphabet =
      | Alphabet of string list

type turtle_param =
      TurtleParam of varexpr

type rule =
      | Lambda of string list (*start rule: lambda ->  production_rule *)
      | ERule of string * string list (*Expansion rule : alphabet_symbol ->
Expansion.*)
      | FRule of string * string * expr list (*Function rule : name | turtle
function name | parameters. E.g f = turtle_move(100)*)
      | EmptyFRule of string (*the empty function rule. E.g. A = ;*)

type lsystem =
      LSystem of alphabet * rule * rule list (*Alphabet | Lambda rule | The
other rules*)

type lfunc_decl = {
      name : string;
      formal : fparam list;
      rules : lsystem;
}

type func_decl = {
      fname : string;
      formals : fparam list;
      locals : vdecl list;
      body : stmt list;
}

type func =
      | CFunc of func_decl (*compute function*)
      | DFunc of lfunc_decl (*draw function*)

type program = func list

let string_of_var_dec (a,b,c) = a ^ b ^ c

let string_of_vop  = function
      | VAdd -> "+"
      | VSub -> "-"
      | VMult-> "*"
      | VDiv -> "/"
```

```
let string_of_arithexpr  = function
      | ALiteral(i) -> string_of_int i
      | AId(s) -> s
      | AFloat(f) -> string_of_float f

let rec string_of_varexpr = function
      | VLiteral(i) -> string_of_int i
      | VId(s) -> s
      | VFloat(f) -> string_of_float f
      | VStringLit(s) -> s
      | VBoolLit(b) -> string_of_bool b
      | VBinop(v1,op,v2) -> string_of_varexpr v1 ^ " " ^ string_of_vop op ^ "
" ^ string_of_varexpr v2

let string_of_dt = function
        StringType -> "string"
      | DoubleType -> "double"
      | IntType -> "int"
      | BooleanType -> "boolean"

let string_of_bop = function
      | And -> "&&"
      | Or -> "||"

let string_of_rop = function
      | BLess -> "<"
      | BLeq -> "<="
      | BGreater -> ">"
      | BGeq -> ">="

let string_of_eop = function
      | BEqual -> "=="
      | BNeq -> "!="

let string_of_bv = function
      | True -> "true"
      | False -> "false"

let string_of_op = function
        Add -> "+"
      | Sub -> "-"
      | Mult -> "*"
      | Div -> "/"
      | Equal -> "=="
      | Neq -> "!="
      | Less -> "<"
      | Leq -> "<="
      | Greater -> ">"
      | Geq -> ">="
      | Pow -> "^"

let rec string_of_expr = function
        Literal(l) -> string_of_int l
      | Boolean(b) -> string_of_bool b
      | Float(f) -> string_of_float f
```

```
       | String(s) -> s
       | Id(s) -> s
       | Binop(e1, o, e2) ->
         begin
          match o with
          | Pow -> "Math.pow(" ^ string_of_expr e1 ^ " , " ^ string_of_expr e2
^ ")"
          | _ -> string_of_expr e1 ^ " " ^
            (match o with
             Add -> "+"
             | Sub -> "-"
             | Mult -> "*"
             | Div -> "/"
             | Equal -> "=="
             | Neq -> "!="
             | Less -> "<"
             | Leq -> "<="
             | Greater -> ">"
             | Geq -> ">="
             | Pow -> "^")
            ^ " " ^ string_of_expr e2
         end
       | Assign(v, e) -> v ^ " = " ^ string_of_expr e
       | Call(f, el) -> f ^ "(" ^ String.concat ", " (List.map string_of_expr
el) ^ ")"
       | Noexpr -> ""
       | BVal(v) -> string_of_bv v
       | RExpr(e1,o,e2) -> string_of_expr e1 ^ " " ^ string_of_rop o ^ " " ^
string_of_expr e2
       | EExpr(e1,o,e2) -> string_of_expr e1 ^ " " ^ string_of_eop o ^ " " ^
string_of_expr e2
       | BExpr(e1,o,e2) -> string_of_expr e1 ^ " " ^ string_of_bop o ^ " " ^
string_of_expr e2
       | Bracket(e1) -> " ( " ^ string_of_expr e1 ^ " ) "

let rec string_of_stmt = function
         Block(stmts) -> "{\n" ^ String.concat "" (List.map string_of_stmt
stmts) ^ "}\n"
       | Expr(expr) -> string_of_expr expr ^ ";\n";
       | Return(expr) -> "return " ^ string_of_expr expr ^ ";\n";
       | If(e, s, Block([])) -> "if (" ^ string_of_expr e ^ ")\n" ^
string_of_stmt s
       | If(e, s1, s2) ->  "if (" ^ string_of_expr e ^ ")\n" ^ string_of_stmt
s1 ^ "else\n" ^ string_of_stmt s2
       | For(e1, e2, e3, s) -> "for (" ^ string_of_expr e1  ^ " ; " ^
string_of_expr e2 ^ " ; " ^ string_of_expr e3  ^ ") " ^ string_of_stmt s
       | While(e, s) -> "while (" ^ string_of_expr e ^ ") " ^ string_of_stmt s

let string_of_vdecl = function
      VDecl(dtt, nm, v) ->  string_of_dt dtt ^ " " ^ nm ^ " = " ^ v ^ ";\n"

let string_of_alphabet = function
      Alphabet(string_list) -> String.concat " " string_list

let string_of_lambdarule = function
       | Lambda(string_list) ->  "lambda -> " ^ String.concat " " string_list
      | _ -> ""  (*output nothing if not a lambda rule*)

let string_of_lambdarule_value = function
      |  Lambda(string_list) ->  String.concat " " string_list
      | _ -> ""  (*output nothing if not a lambda rule*)

let string_of_rule = function
      | Lambda(string_list) ->  "lambda -> " ^ String.concat " " string_list
      | ERule(name, string_list) -> name ^ " -> " ^ String.concat " "
string_list ^ "\n"
      | FRule(name, fname, params) -> name ^ " = " ^ fname ^ "(" ^
String.concat "," (List.map string_of_expr params)  ^ ")" ^ "\n"
      | EmptyFRule(s) -> s ^ " = " ^ "\n"

let string_of_fparam = function
      FParam(dt,s) -> string_of_dt dt ^ " " ^ s

let string_of_lsystem  = function
      LSystem(a,s,rl) ->  string_of_alphabet a ^ "\n" ^string_of_lambdarule
s ^ "\n" ^ String.concat "" (List.map string_of_rule rl)

let string_of_dfunc (func) =
      "Function name : " ^ func.name ^ "\n" ^
      "Formal Parameter(s) : " ^ String.concat "," (List.map string_of_fparam
func.formal) ^ "\n" ^
      "LSystem: " ^ "\n" ^ string_of_lsystem func.rules

let string_of_fdecl  = function
      | CFunc(fdecl) ->
                  "\ndef compute " ^ fdecl.fname ^ "(" ^ String.concat ", "
(List.map string_of_fparam fdecl.formals) ^ ") {\n" ^
                  String.concat "" (List.map string_of_vdecl fdecl.locals) ^
                  String.concat "" (List.map string_of_stmt fdecl.body) ^
                  "}\n"
      | DFunc(fdecl) -> "\ndef draw " ^ fdecl.name ^ "(" ^ String.concat ", "
(List.map string_of_fparam fdecl.formal) ^ ") {\n" ^
            string_of_lsystem fdecl.rules ^ "}\n"

let string_of_program (funcs) = String.concat "\n" (List.map string_of_fdecl
funcs)
```

semantic.ml

```
open Ast
open Str
open LSystemstd

type var_table = {
      variables : Ast.vdecl list;
      }

type env = {
      mutable functions : func list ;
      variables : vdecl list;
}

(*determines if the given function exists*)
let exists_function func env =
      match func with
       DFunc(func) -> begin
        try
          let _ = List.find ( fun(f) ->
            match f with
            | DFunc(f) -> f.name = func.name
            | CFunc(f) -> f.fname = func.name) env.functions  in
              let e = "Duplicate function name : " ^ func.name ^ "\n" in
                (*throw error on duplicate func.*)
                raise(Failure e) with Not_found -> false
       end
      | CFunc(func) -> try let _ =
        List.find ( fun(f) ->
          match f with
          | DFunc(f) -> f.name = func.fname
          | CFunc(f) -> f.fname = func.fname) env.functions  in
            let e = "Duplicate function name : " ^ func.fname ^ "\n" in
              (*throw error on duplicate func.*)
              raise(Failure e) with Not_found -> false

let print_function_list flist =
      List.map(fun(f) ->
                  match f with
                                  | DFunc(f) -> let nm = f.name in
print_endline ("DFunc:" ^nm)
                                  | CFunc(f) -> let nm = f.fname in
print_endline ("CFunc:" ^nm)
                        ) flist

(*Determine if a function with given name exists*)
let exists_function_name name env =
    try
        let _ = List.find ( fun(f) -> match f with
            | DFunc(f) -> f.name = name
            | CFunc(f) -> f.fname = name
            ) env.functions  in
            true (*Found a function with name like that*)
     with Not_found -> false

(*Returns the function that has the given name*)
let get_function_name name env =
    try
        let afunc = List.find ( fun(f) -> match f with
            | DFunc(f) -> f.name = name
            | CFunc(f) -> f.fname = name
            ) env.functions  in
            afunc (*Found a function with name like that*)
     with Not_found -> raise(Failure("Function " ^ name ^ "has not yet been 
declared" ) )

(*Exists function when func == CFunc *)
let cexists_function func env =
      try
            let _ = List.find ( fun(f) -> match f with
                                  | DFunc(f) -> f.name = func.fname
                                        | CFunc(f) -> f.fname = func.fname
                                        ) env.functions  in
                                        let e = "Duplicate function name : " ^
func.fname ^ "\n" in
                                                      raise(Failure e) (*throw
error on duplicate func.*)
   with Not_found -> false


(*Determines if a formal paramter with the given name 'fpname' exits in the
given function*)
let exists_formal_param func fpname =
      match func with
      | DFunc(func) -> false(*to be implemented*)
      | CFunc(func) -> try
                                                                        let _ =
List.find( fun(fp) -> let FParam(_,cn) = fp

                                                                                    in cn
= fpname

) func.formals in

                                          true (*we're able to find a formal paramter*)
                              with Not_found -> false (*no formal parameter found in
the function*)



(*this is for compute functions only*)
let cexists_formal_param func fpname =
            try
                              let _ = List.find( fun(fp) -> let FParam(_,cn)
= fp
                                                      in cn = fpname
                              ) func.formals in
                                                      true (*we're able to
find a formal paramter*)
                        with Not_found -> false (*no formal parameter found in the
function*)


(*for computing functions only*)
let cexists_variable_decl func vname =
                  try
                        let _ = List.find( fun(fp) -> let VDecl(_,vn,_) = fp
                                          in vn = vname
                        ) func.locals in
                                                true (*we're able to find a
variable*)
          with Not_found -> false (*no variable declaration - found in the
function*)
```

```
(*Determines if a variable declaration with the given name 'vname' exists in
the given functioin*)
let exists_variable_decl func vname =
      match func with
      | DFunc(func) -> false(*to be implemented*)
      | CFunc(func) -> try
                                                            let _ =
List.find( fun(fp) -> let VDecl(_,vn,_) = fp

                                                                      in vn
= vname

) func.locals in

                              true (*we're able to find a variable*)
                      with Not_found -> false (*no variable declaration -
found in the function*)


(*this gets formal paramters for COMPUTE function*)
let get_cfparam_type func fpname =
      try
                  let fparam = List.find( fun(fp) -> let FParam(_,cn) = fp

                              in cn = fpname
                                                                        )
func.formals in
                                                   let FParam(dt,_) = fparam
                                                   in dt (*return the data
type*)
  with Not_found -> raise (Failure ("Formal Parameter " ^ fpname ^ " should
exist but was not found in compute function " ^ func.fname)) (*this shouldn't
not happen*)

(*gets the variable type - only for COMPUTE functions*)

let get_var_type func vname =
            try
                  let var = List.find( fun(v) -> let VDecl(_,vn,_) = v

                              in vn = vname
                                     ) func.locals in
                  let VDecl(dt,_,_) = var
                  in dt (*return the data type*)
            with Not_found -> raise (Failure ("Variable " ^ vname ^ " should
exist but was not found in compute function " ^ func.fname)) (*this shouldn't
not happen*)



(*Returns the type of a given variable name *)
let get_type func name =
      if( cexists_variable_decl func name ) (*It's a variable*)
            then get_var_type func name
      else
```

```
            if (cexists_formal_param func name) then
                  get_cfparam_type func name
            else (*Variable has not been declared as it was not found*)
                          let e = "Variable " ^ name ^ " is being used without
being declared in function " ^ func.fname in
                          raise (Failure e)
                  (* not needed
                        | DFunc(func) -> let e = "function get_type called on
a draw function function " ^ func.name  ^ " which is not allowed\n" in

raise (Failure e)*)


(*Determines if the given identiifier exists*)
let exists_id name func =
      if( cexists_variable_decl func name ) (*It's a variable*)
            then true
      else
            if (cexists_formal_param func name) then
                  true
            else (*Variable has not been declared as it was not found*)
                  false

(*determines if the given compute function collides with a number of another
function*)
(*let exists_CFunction func env =
            try
                        let _ = List.find ( fun(f) -> match f with
                              | DFunc(f) -> f.name = func.fname
                              | CFunc(f) -> f.fname = func.fname
                              ) env.functions  in true (*return true on
success*)
             with Not_found -> print_endline "notfound\n"; false (*return
false on failure*)
*)


(*see if there is a function with given name "func"*)
let find_function func env =
      try
            let _ = List.find ( fun(f) -> match f with
                                                                          |
DFunc(f) -> f.name = func
                                                                          |
CFunc(f) -> f.fname = func
            ) env.functions in true (*return true on success*)
       with Not_found -> raise Not_found


let dup_fparam func =
      match func with
            | DFunc(func) -> let length = List.length func.formal in
                                  if(length = 1) then (*must have 1
arguments*)
                                     let _isvalid = List.map(
                                                 fun(x) -> let
```

```
FParam(t,_) = x in match t with
                                                                                | IntType -> false
                                                                                | _ ->
raise(Failure("Formal parameter type for draw function must be an int"))


                                                            ) func.formal in false
                                                                                  else

      raise(Failure("Draw function '"^ func.name ^"' must have only 1 formal
parameters but it has " ^ string_of_int length ^ " params"))

             | CFunc(func) -> let isdup f = List.fold_left(

              fun c x ->

                         let FParam(_,my_name) = f and FParam(_,curr_name) = x
in

                         if ( c = 0 && my_name = curr_name ) then c + 1



                         else
                                     if ( c = 1 &&  my_name = curr_name) then
(*found a 2nd dup match*)

                                     let e = "Duplicate formal parameter in
function : " ^ func.fname ^ "\n" in
                                 raise(Failure e) (*throw error on duplicate
formal parameter.*)

                                     else c



                                                              ) 0 func.formals
                                                                              in let _ =
List.map(isdup) func.formals
                                                                              in false

(*This check for duplicate formal parametersin COMPUTE function*)
let cdup_fparam func =
      let isdup f = List.fold_left(

              fun c x ->

                         let FParam(_,my_name) = f and FParam(_,curr_name) = x
in

                         if ( c = 0 && my_name = curr_name ) then c + 1
```

```
                        else

                                    if ( c = 1 &&  my_name = curr_name) then
(*found a 2nd dup match*)

                                    let e = "Duplicate formal parameter in
function : " ^ func.fname ^ "\n" in

                                raise(Failure e) (*throw error on duplicate
formal parameter.*)

                                    else c



                                                ) 0 func.formals
      in let _ = List.map(isdup) func.formals
      in false

(*checks if there is a duplicate variable declaration for COMPUTE functions*)
let dup_vdecl func =
      match func with
      | DFunc(func) -> false
      | CFunc(func)->
      let isdup var = List.fold_left(

                   fun c x ->

                              let VDecl(mdt,mn,_) = var

                              and VDecl(tdt,tn,_) = x in

                              if ( c = 0 && (mn) = (tn)  ) then c + 1



                              else

                                          if ( c = 1 &&  (mn) = (tn) ) then
(*found a 2nd dup match*)

                                                                  let e =
"Duplicate variable declaration '"^ mn ^"' in compute function : " ^
func.fname  in

                                      raise(Failure e) (*throw error on
duplicate formal parameter.*)

                                          else c



                        ) 0 func.locals
```

```
                                                                   (*check if
the given variable decl. name has already been declared in the formal
paramters*)
                                                                  in let _ =
List.map(

      fun(x) -> List.map(

                                                                  fun(y) ->
let FParam(_,formal_nm) = y

                   and VDecl(_,varname,_) = x

                   in if (formal_nm = varname) then

                                          let e = "Redeclaration of formal
parameter '" ^ formal_nm ^"' not allowed in function : " ^ func.fname ^"\n"

                                          in raise(Failure e)

                          else false

                                                                       )
func.formals
                                                                   )
func.locals
                                                                  in
                                                                  let _ =
List.map(isdup) func.locals (*see if we have duplicate var names*)
                                                                  in false

let is_int s =
      try ignore (int_of_string s); true
      with _ -> false

let is_float s =
      try ignore (float_of_string s); true
      with _ -> false
```

```
let is_letter s =
    let regex = regexp  "[A-Za-z]" in (*Make any string that starts with a
double quotes and ends with one*)
    let str = "|" ^s ^"|" in
    print_endline str; string_match regex s 0

(*Function that checks if given input is a string. Used to make sure that an
expression is indeed of type string
It does this by just checking if the first character and the last character
are all the same and that they equal
a single double quote. This check is sufficient because the parse will reject
any streams of character which do not
make a valid string literal.*)
let is_string s =
      let l = String.length s in
      let last_idx = l - 1 in
      let first_char = String.sub s 0 1 and
      last_char = String.sub s last_idx 1 in
      match first_char,last_char with
            | "\"" , "\""-> true (*check that 1st char = last char = double
quote*)
            | _          -> false


(*continue from here
let rec is_bool s =
      match s with
            | BVal(s) ->
            | RExpr(s) ->
            | EEXpr(s) ->
            | BExpr(s) ->
            | _ -> false


*)



let is_string_bool s =
            match s with
                  | "true" -> true
                  | "false" -> true
                  | _ -> false

(*check if variable declation is valid*)
let valid_vdecl func =
      match func with
            | DFunc(func) -> false
            | CFunc(func) ->
      let _ = List.map(
                                                     fun(v) ->

                                                           let
VDecl(dt,nm,value) = v in
```

```
                                                                      let e = "Invalid
variable declaration for '" ^ nm ^ "' in compute function " ^ func.fname ^
"\n" in
                                                                      let be = e ^ "The
Only allowed values for initializing boolean values is 'true' or 'false' \n"
                                                                            in match dt
with
                                                                                      |
StringType -> if (is_string value) then true

else raise (Failure e )

                                                                                      |
DoubleType -> if ( (is_float value)) then true

                                          else raise (Failure e)


                                                                                      |
IntType    -> if (is_int value) then true

else raise(Failure e)


                                                                                      |
BooleanType -> if (is_string_bool value) then true

else raise (Failure be)

                      ) func.locals
                   in true


let rec is_num func expr =
      match expr with
            | Literal(i) -> true
            | Float(f) -> true
            | Id(s) -> let dt  = get_type func s in
                                          begin
                             match dt with
                                                                      | IntType -> true
                                                                      | DoubleType ->
true
                                                                      | _ -> false
                                                end
    | Binop(e1,op,e2) ->      let   b1 = is_num func e1 and

      b2 = is_num func e1  in

      b1 && b2
            | Call(name,expr) -> raise (Failure "TBI") (*to be implemented*)


            | _ -> false
```

```
let rec get_expr_type e func   =
      match e with
            | String(s) -> StringType
    | Id(s) -> get_type func s
            | Literal(i) -> IntType
            | Float(f) -> DoubleType
            | Boolean(b) -> BooleanType
            | Binop(e1,op,e2) -> let t1 = get_expr_type e1 func  and
                                 t2 = get_expr_type e2 func  in
                                                                           begin

match t1,t2 with

      | DoubleType,DoubleType -> DoubleType

      | DoubleType,IntType -> DoubleType (*Upconvert to double type*)

      | IntType,DoubleType -> DoubleType (*Upconvert to double type*)

      | IntType,IntType -> IntType

      | _,_ -> raise (Failure ("Invalid Types used in a binop expression"))
                                                                           end
            | Assign(id,expr) -> get_expr_type expr func
            | Call(fname,expr) -> DoubleType (*function calls return double*)
            | BVal(b) -> BooleanType
            | RExpr(e1,rop,e2) -> let t1 = get_expr_type e1 func and
                                  t2 = get_expr_type e2 func in

begin

      match t1,t2 with

            | DoubleType,DoubleType -> BooleanType

            | DoubleType,IntType -> BooleanType

            | IntType,DoubleType -> BooleanType

            | IntType,IntType -> BooleanType

            | _,_ -> raise(Failure("Invalid Types used in a relational
expression"))

      end
            | EExpr(e1,eop,e2) -> let t1 = get_expr_type e1 func and
                                  t2 = get_expr_type e2 func  in

      begin

      match t1,t2 with

            | DoubleType,DoubleType -> BooleanType

            | DoubleType,IntType -> BooleanType
```

```
             | IntType,DoubleType -> BooleanType

             | IntType,IntType -> BooleanType

             | StringType,StringType -> BooleanType (*can do string
comparisons*)

             | BooleanType,BooleanType -> BooleanType (*can compare bool
values*)

             | _,_ -> raise(Failure("Invalid Types used in a equality
expression"))

      end
             | BExpr(e1,bop,e2) -> let t1 = get_expr_type e1 func and
                                   t2 = get_expr_type e2 func  in

      begin

      match t1,t2 with

             | BooleanType,BooleanType -> BooleanType

             | _,_ -> raise(Failure("Invalid Types used in a boolean compound
expression"))

      end
         | _ -> IntType (*should not happen - added this to turn off compiler
warnings about incomplete matching for Noexpr*)



(*Checks if the given expression is a valid  assignment / call expression*)
let is_assign_call func expr =
      match expr with
             | Assign(_,_) -> true
             | Call(_,_) -> true
             | _ -> false




(*Makes sure that the given arguments in a function call match the function
signature*)
(*fname of function being called*)
(*exprlist - list of expr in funcation call*)
(*cfucn- compute function*)
(*env - the enviroment*)
let check_types fname exprlist cfunc env =
  let func = get_function_name fname env in
      begin

            match func with
      | DFunc(func) -> 0 (*still to be implemented*)
      | CFunc(func) ->
```

```
                        let arg_types = List.map(fun(e) -> get_expr_type e
cfunc) exprlist in
                          if((List.length arg_types) != (List.length
func.formals) ) then (*number of args don't match up*)

      raise(Failure("Number of arguments in a function call don't match up in
compute function " ^ func.fname))
                                                                      else

let check_arg c arg_type = (*c is the counter, arg_type is type of actual
parameters. meant to be used in the list.foldleft *)

           let formal_param = List.nth func.formals c in

                   let FParam(formal_type,_) = formal_param in


                   begin

                        match formal_type,arg_type with

                        | DoubleType, DoubleType -> c + 1

                                                                          |
DoubleType, IntType -> c + 1


                        | IntType, IntType -> c + 1

                        | StringType, StringType -> c+1

                        | BooleanType, BooleanType -> c+1

                        | _,_ -> raise(Failure("Types don't match in call
expression " ^ fname ^ " in the compute function " ^ cfunc.fname))

end

      in

List.fold_left (check_arg) 0 arg_types

  end

let rec valid_expr (func : Ast.func_decl) expr env =

      match expr with
            | Literal(i) -> true
            | Float(f) ->  true
            | Boolean(b) -> true
            | String(s) -> true
            | Id(s) -> if(exists_id s func) then true else raise( Failure
("Undeclared identifier " ^ s ^ " is used" ))
            | Binop(e1,op,e2) -> let r1 = is_num func e1
                                                                       and
r2 = is_num func e2 in
```

```
                                                                        r1
&& r2
          | Assign(id, e1) -> if(exists_id id func) then
     let dt = get_type func id and
     _ = valid_expr func e1 env and
     exprtype = get_expr_type e1 func
     in match dt,exprtype with
          | StringType,StringType -> true
          | IntType,IntType -> true
          | DoubleType,DoubleType -> true
          | DoubleType,IntType -> true (*allow int to double conversion*)
          | BooleanType,BooleanType -> true
          | IntType,DoubleType -> raise(Failure ("Cannot assign a double to
an int"))
          | _,_ -> raise(Failure ("DataTypes do not match up in an
assignment expression to variable " ^ id))

                                                                   else
     raise( Failure ("Undeclared identifier " ^ id ^ " is used" ))
     (*Call check has not yet been fully implemented*)
          | Call(fname, exprlist) ->  if(exists_function_name fname env)
then
              let _has_valid_exprs = List.map(fun(e) -> valid_expr func e
env) exprlist in
                     let _checktypes = check_types fname exprlist func
env (*check that the types match up otherwise throws an error *)
                     in
                     true
                                    else
              (if List.mem fname LSystemstd.func_names then (*It's a
standard library function call*)
                  true (*STILL TO DO:  checking of std lib functions *)
              else
```

```
                  raise( Failure ("Undefined function : " ^ fname ^ " is
used" ))

                      )
            | BVal(b) -> true
            | RExpr(e1,rop,e2) -> let t1 = get_expr_type e1 func and
                                      t2 = get_expr_type e2 func in

      begin

      match t1,t2 with

            | DoubleType,DoubleType -> true

            | DoubleType,IntType -> true

            | IntType,IntType -> true

            | IntType,DoubleType -> true

            | _,_ -> raise(Failure("Invalid Types used in a relational
expression"))

      end
            | EExpr(e1,eop,e2) -> let t1 = get_expr_type e1 func and
                                      t2 = get_expr_type e2 func in

      begin

      match t1,t2 with

            | DoubleType,DoubleType -> true

            | DoubleType,IntType -> true

            | IntType,IntType -> true

            | IntType,DoubleType -> true

            | StringType,StringType -> true

            | BooleanType,BooleanType -> true

            | _,_ -> raise(Failure("Invalid Types used in an equality
expression"))

      end
            | BExpr(e1,bop,e2) -> let t1 = get_expr_type e1 func and
                                      t2 = get_expr_type e2 func in

      begin

      match t1,t2 with

            | BooleanType,BooleanType -> true
```

```
             | _,_ -> raise(Failure("Invalid Types used in a boolean compound 
expression"))

      end
        | _ -> false(*should not happen - added this to turn off compiler 
warnings about incomplete matching for Noexpr*)


(*Returns alphabet list from the draw function*)
let get_alphabet_list func =
      let LSystem(alphabet,lambda,rlist) = func.rules in
      let Alphabet(alphabet_list) = alphabet in
      alphabet_list

(*Check to make sure that alphabet has no repeating letters*)
let valid_alphabet alphabet func =
          let Alphabet(alphabet_list) = alphabet in
      let isdup letter = List.fold_left(
                                 fun c curr_letter ->

if ( c = 0 && letter = curr_letter) then c + 1
                              else
                                 if ( c = 1 &&  letter = curr_letter) then 
(*found a 2nd duplicate match*)
                                    let e = "Duplicate alphabet letters '" 
^ letter ^ "' in function : " ^ func.name ^ "\n" in
                                    raise(Failure e)
                                 else c
                                  ) 0 alphabet_list
                  in
                  let _ensure_no_dups  = List.map(isdup) alphabet_list in
                  let valid_letters = List.for_all (is_letter) alphabet_list 
in
                  match valid_letters with
                        | true -> true
                        | false -> raise(Failure("Invalid letters used in 
alphabet of drawing function " ^ func.name))

(*Check if given symbol exists in alphabet*)
let exists_in_alphabet letter alphabet_list  =
      try
         let _ = List.find (fun(x) -> x = letter ) alphabet_list in
             true
      with Not_found -> false

(*check if given symbol is in the standary library symbol of 'l r f'*)
let is_std_symbol s =
      try
       let _ = List.find (fun(x) -> x = s ) LSystemstd.std_symbols in
         true
    with Not_found -> false

(*Check if the given letter exists in alphabet or is part of the 'l r f' 
standard library symbols*)
let valid_symbol letter func =
      let in_alphabet = exists_in_alphabet letter (get_alphabet_list func)
```

```
and
      is_std_symbol = is_std_symbol letter in
      match in_alphabet,is_std_symbol with
            | true,_ -> true
            | _,true -> true
            | false,false -> false


let valid_rule rule func =
    match rule with
        | Lambda(string_list) -> let ok = List.for_all(fun(x) -> valid_symbol
x func) string_list in
                                                      if(ok) then

                        let _ = print_endline "lambda OKAY" in true

                  else

                        raise(Failure("Lambda rule has an invalid character
that has not been declared in the alphabet"))
        | ERule(name, string_list) -> if(valid_symbol name func) then

                                                     let ok = List.for_all(fun(x)
-> valid_symbol x func) string_list in

                                                     if(ok) then

                                                           true

                                                else

                                                           raise(Failure("ERule
'"^ name ^"' has an invalid character that has not been declared in the
alphabet"))
                                                              else

                                              raise(Failure("ERule symbol '"^ name
^"' is not in the alphabet"))
                              | FRule(name, fname,string_list) ->true (*to do*)
                                  | EmptyFRule(name) -> if (valid_symbol name func)
then
                                                          true
                                                      else

               raise(Failure("Empty FRule symbol '"^ name ^"' is not in the
alphabet"))


(*validates the lsystem in a draw funciton*)
let validate_lsystem func env =
      let LSystem(alphabet,lambda,rlist) = func.rules in
      let _validate_alphabet = valid_alphabet alphabet func in
      let _valid_lambda = valid_rule lambda func in
      let _valid_rules = List.map(fun(x) -> valid_rule x func ) rlist
      in
      true
```

```
(*Checks the body of a compute function *)
let valid_body func  env =
      match func with
            | DFunc(func) -> validate_lsystem func env
            | CFunc(func) ->
                                    let rec check_stmt st =

      begin


            match st with

                  (*the 'block' will only occur in if and while condition
loop. *)

                  | Block(st_list) -> let _ = List.map(fun(x) -> check_stmt
x) st_list (*Check statements in the block. Err will be thrown for an invalid
stmt*)

                                                                  in true

                  | Expr(st) -> let vldexpr = valid_expr func st env and
(*make sure the expression is valid expression*)

                                                            assign_call  =
is_assign_call func st in

                                                            begin

                                                                  match
vldexpr,assign_call with (*The expression MUST be valid and also an
assignment/call expression. Can't have '1;' as a stmt expr alone *)

                                                                        |
true,true -> true

                                                                        |
true,false -> raise(Failure ("Invalid expression (No var assignment) in
function " ^func.fname ^ "\n"))

                                                                        |
false,_ -> raise(Failure ("Invalid assignment expression in function "
^func.fname ^ "\n"))

                                                            end



                  | Return(st) -> let ret = get_expr_type st func in


                  begin


                        match ret with
```

```
                              | DoubleType -> true

                              | IntType -> true

                              | _ -> raise(Failure("return type is not double
in compute function " ^ func.fname ^ ". It is of type :" ^ (string_of_dt
ret)))

                  end

                  | If(predicate,stmt1,stmt2) -> let pred_type =
get_expr_type predicate func and

                 ve1 = check_stmt stmt1 and

                 ve2 = check_stmt stmt2 in

             let _vpred = (*Check predicate*)

                                             begin

                                              match pred_type with

                                                  | BooleanType -> true

                                                  | _ ->
raise(Failure("predicate expression must be a valid boolean expression that
evaluates to true/false"))

                                             end

             in

             begin match ve1,ve2 with

                  | true, true -> true

                  | _ , _ -> raise(Failure("Invalid expression used in if
statement in compute function " ^ func.fname ^ "\n"))
```

```
              end


                    | For(_,_,_,_) -> let e = "For loop are not allowed in
function" ^ func.fname ^ "\n" in


      raise (Failure e) (*don't have to worrty about this case b'se parser
will give parse error for 'for loops'.*)

                    | While(pred,stmts) -> let isvalid = check_stmt stmts in
(*need to test*)
                                                                    begin

                                                match isvalid with

                                                      | true -> true


                                                      | false ->  raise
(Failure ("Invalid statement found inside while loop in compute function "
^func.fname ^"\n"))
                                                                    end

      end
                                                            in
                                                           let _ =
List.map(check_stmt) func.body
                                                           in true


(*Check a Compute Function. *)
(* The type of function 'f' passed however should be of type *)
(* Ast.func This is so that we can easily add functions to environment*)
(* and avoid having fields for Cfunctions and Dfunctions separately. *)
let check_cfunc f env =

      let dup_fname = exists_function f env
      in
      let dup_formals = dup_fparam f
      in
      let vlocals = (not (dup_vdecl f)) && (valid_vdecl f) (*make sure that
we've no dup variable names, and data types match up*)
      in
      let vbody = valid_body f env
      in
      let _ = env.functions <- (f) :: env.functions (*add function name to
environment *)
      in (not dup_fname) && (not dup_formals) && vlocals && vbody
```

```
let check_dfunc f env =
      let dup_fname = exists_function f env in
      let dup_formals = dup_fparam f in
      let vbody = valid_body f env in
      let _ = env.functions <- (f) :: env.functions (*add function name to
environment *)
      in
      (not dup_fname) && (not dup_formals) && vbody

let valid_func env f =  match f with
        CFunc(f) -> print_endline ("checking cf:" ^ f.fname); let afunc =
CFunc(f) in check_cfunc afunc env
      | DFunc(f) -> print_endline ("checking df:"^f.name); let afunc =
DFunc(f) in check_dfunc afunc env


(*
let valid_func a b = match b with
      | CFunc(x) ->  print_endline "hello\n"*)


(*Checks to make sure that the main function exists and is a compute
function*)
let exists_main env =
      if(exists_function_name "main" env) then
            let func_type = get_function_name "main" env in
            match func_type with
                  | CFunc(func_type) -> true
              | DFunc(func_type) -> raise(Failure("main function must be a
compute function. "))
      else
            raise(Failure("Compute function 'main' does not exist !"))

let check_program flist =
      let (environment : env) = { functions = [] ; variables = [] } in
      let _dovalidation = List.map ( fun(f) -> valid_func environment f)
flist in (*Do the semantic analysis*)
      let _mainexists = exists_main environment (*ensure that a main function
exists*)
            in
             "\nSuccess !\n"
```

compile.ml

```
(* Primary Author: Ethan Hann (eh2413) *)

open Ast
open LSystemstd
open Str

exception RedeclarationOfStandardFunctionNotAllowedError
```

```
module StringMap = Map.Make(String);;

let get_prod fname = function
        Lambda(symbols) -> "        " ^ fname ^ ".addProduction(\"lambda\",
\"" ^ (String.concat "," symbols) ^ "\");\n"
      | ERule(name, symbols) -> "        " ^ fname ^ ".addProduction(\"" ^
name ^ "\", \"" ^ (String.concat "," symbols) ^ "\");\n"
      | _ -> ""

let get_term fname = function
        FRule(symbol, command, param) -> "            " ^ fname ^
".addTerminal(\"" ^ symbol ^ "\", new Command(" ^ (String.uppercase command)
^ ", " ^ (string_of_expr (List.hd param)) ^ "));\n"
      | _ -> ""

let translate fname rule = match rule with
        Lambda(_) -> get_prod fname rule
      | ERule(_,_) -> get_prod fname rule
      | FRule(_,_,_) -> get_term fname rule
      | _ -> ""

let draw_fdecl fdecl =
      if List.mem fdecl.name LSystemstd.func_names then
            raise RedeclarationOfStandardFunctionNotAllowedError
      else
            let fname = fdecl.name in
                  let fun_sig = "         Function " ^ fname ^ " = new
Function(\"" ^ fname ^ "\");\n" in
                        let lsys = fdecl.rules in match lsys with
                              LSystem(alphabet, lambda, rules) -> fun_sig ^
(translate fname lambda) ^ (String.concat "" (List.map (function rule ->
translate fname rule) rules))

let translate_compute_fdecl fdecl =
      if List.mem fdecl.fname LSystemstd.func_names then
            raise RedeclarationOfStandardFunctionNotAllowedError
      else
            let fun_sig =
                  match fdecl.fname with
                    "main" -> LSystemstd.std_render_signature
                  | _ -> "    public double " ^ fdecl.fname ^ "(" ^
String.concat ", " (List.map string_of_fparam fdecl.formals) ^ "){\n" in
                  fun_sig ^ "        " ^
                        String.concat "         " (List.map string_of_vdecl
fdecl.locals) ^
                        String.concat "         " (List.map string_of_stmt
fdecl.body) ^
                        "     }\n"

(* Call the appropriate translation function depending on type of function.
*)
let translate_fdecl = function
  CFunc(fdecl) -> translate_compute_fdecl fdecl
| DFunc(fdecl) -> draw_fdecl fdecl

let get_dfuncs = function DFunc(fdecl) -> draw_fdecl fdecl | _ -> ""
let get_cfuncs = function CFunc(fdecl) -> translate_compute_fdecl fdecl | _ -
> ""

let get_dcalls = function DFunc(fdecl) -> let name = fdecl.name in "
      public void " ^ name ^ "(int depth){\n" ^ "            draw(\"" ^ name ^
"\", depth);\n    }\n" | _ -> ""

let translate funcs prog_name verbose testmode =
      let out_chan = open_out (prog_name ^ ".java") in
            let translated_prog =
                  LSystemstd.std_turtle1 ^ (if testmode then "true;" else
"false;") ^ LSystemstd.std_turtle2 ^ prog_name ^ LSystemstd.std_turtle3 ^
                  "public class " ^ prog_name ^ " extends Turtle {\n" ^
global_replace (Str.regexp "CLASSNAME") prog_name LSystemstd.std_main ^
                  "     public " ^ prog_name ^ "(){\n" ^ String.concat ""
(List.map get_dfuncs funcs) ^ "           execute();\n           scale(1);\n
      }\n" ^
                  String.concat "" (List.map get_cfuncs funcs) ^
                  String.concat "" (List.map get_dcalls funcs) ^ "}\n"
            in
                  let proc_status = ignore(Printf.fprintf out_chan "%s"
translated_prog);
                        close_out out_chan;
                        Sys.command (Printf.sprintf "javac %s.java"
prog_name) in
                              match proc_status with
                                0 -> if verbose
                                                            then
translated_prog ^ "\nCompilation successful\n"
                                                            else "Compilation
successful\n"
                              | _ -> "\nCompilation of Java bytecode
unsuccessful!\n" ^
                                          Printf.sprintf "Javac Process
Return Code: %i\n" proc_status ^
                                          Printf.sprintf "Compilation
Command: javac %s.java\n" prog_name
```

lsystemstd.ml

```
(* Primary Author: Timothy Sun (ts2578) *)

(* I'm going to say that most of the Java code's mine. :P -Tim *)

(* Standard java functions. *)
let std_main = "  public static void main(String[] args){
            JFrame j = new JFrame();
            CLASSNAME cn = new CLASSNAME();
            JScrollPane jsp = new JScrollPane(cn.jta);
            jsp.setPreferredSize(new Dimension(DEFAULT+2,100));
            j.add(cn, BorderLayout.CENTER);
            j.add(jsp, BorderLayout.PAGE_END);
            j.pack();
            j.setTitle(\"L-System: CLASSNAME\");
            j.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            j.setVisible(true);
      }
"

let std_render_signature = "  public void execute(){\n"

(* Standard Turtle functions for drawing. *)
let std_turtle1 =
"import java.awt.*;
import javax.swing.*;
import java.util.*;
import java.io.*;
import java.awt.image.BufferedImage;
class Turtle extends JPanel {
      public static final int EMPTY = -1;
      public static final int FORWARD = 0;
      public static final int TURN = 1;
      public static final int DOWN = 2;
      public static final int UP = 3;
      public static final int SETX = 4;
      public static final int SETY = 5;
      public static boolean testing = "

let std_turtle2 = "
      public static int DEFAULT = testing ? 100 : 400;
      private double x = 0;
      private double y = 0;
      private int height;
      private int width;
      public JTextArea jta;
      private double angle;
      private boolean down;
      private BufferedImage bi;
      private ArrayList<double[]> lines;
      public HashMap<String, Function> functions;
      public Turtle(){
            this(DEFAULT+2,DEFAULT+2,0);
      }
      public Turtle(int w, int h, double angle){
            setPreferredSize(new Dimension(w,h));
            height = h;
            width = w;
            this.angle = angle;
            this.down = true;
            jta = new JTextArea(5,20);
            jta.setEditable(false);
            setDoubleBuffered(true);
            lines = new ArrayList<double[]>();
            functions = new HashMap<String, Function>();
      }
      public class Function {
            public Function(String name){
                  functions.put(name, this);
```

```
                terms = new HashMap<String, Command>();
                prods = new HashMap<String, String[]>();
                terms.put(\"f\", new Command(FORWARD,1));
                terms.put(\"l\", new Command(TURN,-90));
                terms.put(\"r\", new Command(TURN,90));
        }
        public void addTerminal(String symbol, Command command){
                terms.put(symbol, command);
        }
        public void addProduction(String symbol, String expansion){
                prods.put(symbol, expansion.split(\",\"));
        }
        public boolean hasTerminal(String symbol){
                return terms.containsKey(symbol);
        }
        public boolean hasProduction(String symbol){
                return prods.containsKey(symbol);
        }
        public Command getTerminal(String symbol){
                return terms.get(symbol);
        }
        public String[] getProduction(String symbol){
                return prods.get(symbol);
        }
        HashMap<String, Command> terms;
        HashMap<String, String[]> prods;
}
public class Command {
        public Command(){
                this(EMPTY);
        }
        public Command(int command){
                this(command, 0);
        }
        public Command(int command, int param){
                this.command = command;
                this.param = param;
        }
        int command;
        double param;
}
public void turtle(Command c){
        switch (c.command){
                case FORWARD: forward(c.param); break;
                case TURN: turn(c.param); break;
                case DOWN: down(); break;
                case UP: up(); break;
                case SETX: setX(c.param); break;
                case SETY: setY(c.param); break;
                default: break;
        }
}
public double[] getDim(){
        double minx = Double.MAX_VALUE, miny = Double.MAX_VALUE;
        for (double[] line : lines){
                if (minx > line[0] || minx > line[2])
```

```
				minx = Math.min(line[0],line[2]);
			if (miny > line[1] || miny > line[3])
				miny = Math.min(line[1],line[3]);
		}
		for (double[] line : lines){
			line[0] -= minx;
			line[1] -= miny;
			line[2] -= minx;
			line[3] -= miny;
		}
		double maxx = Double.MIN_VALUE, maxy = Double.MIN_VALUE;
		for (double[] line : lines){
			if (maxx < line[0] || maxx < line[2])
				maxx = Math.max(line[0],line[2]);
			if (maxy < line[1] || maxy < line[3])
				maxy = Math.max(line[1],line[3]);
		}
		return new double[]{maxx+1, maxy+1};
	}
	public void scale(double factor){
		double[] dim = getDim();
		double trueScale = factor*Math.min(width/dim[0],height/dim[1]);
		bi = new BufferedImage(width+1, height+1,
BufferedImage.TYPE_INT_RGB);
		Graphics g = bi.getGraphics();
		g.setColor(Color.WHITE);
		g.fillRect(0, 0, width, height);
		g.setColor(Color.BLACK);
		for (double[] line : lines)
			g.drawLine((int)(line[0]*trueScale),
(int)(line[1]*trueScale), (int)(line[2]*trueScale),
(int)(line[3]*trueScale));
		if (testing){
			try {
				String output = \"\";
				for (int x = 0; x < width; x++){
					for (int y = 0; y < height; y++)
						output += bi.getRGB(x,y) == -1 ? \"1\" :
\"0\";
					output += \"\\n\";
				}
				PrintWriter pw = new PrintWriter(new File(\""

let std_turtle3 =
".txt\"));
				pw.write(output);
				pw.close();
			}
			catch (Exception e){ e.printStackTrace(); }
			System.exit(1);
		}
	}
	public void paintComponent(Graphics g){
		if (height != getHeight() || width != getWidth()){
			height = getHeight();
			width = getWidth();
```

```
                    scale(1);
              }
              super.paintComponent(g);
              g.drawImage(bi, 1, 1, Color.WHITE, this);
       }
       public void draw(String name, int depth){
              draw(functions.get(name), depth, \"lambda\");
       }
       public void draw(Function f, int depth, String symbol){
              if (depth == -1){
                    if (f.hasTerminal(symbol))
                          turtle(f.getTerminal(symbol));
              }
              else {
                    String[] production = f.getProduction(symbol);
                    for (String term : production){
                          if (f.hasProduction(term))
                                draw(f, depth-1, term);
                          else if (f.hasTerminal(term))
                                turtle(f.getTerminal(term));
                    }
              }
       }
       public void down(){
              down = true;
       }
       public void up(){
              down = false;
       }
       public void forward(double t){
              t = t * 10;
              double nx = x + Math.cos(angle)*t;
              double ny = y + Math.sin(angle)*t;
              if (down)
                    lines.add(new double[]{x, y, nx, ny});
              x = nx;
              y = ny;
       }
       public void turn(double deg){
              angle += deg*Math.PI/180.0;
       }
       public void setX(double x){
              this.x = x;
       }
       public void setY(double y){
              this.y = y;
       }
       public void print(String args){
              jta.append(args);
              if (testing) System.out.println(args);
       }
       public void print(int args){
              print(args+\"\");
       }
       public void print(double args){
              print(args+\"\");
```

```
      }
      public void print(boolean args){
            print(args+\"\");
      }
}
"

(* A string list of reserved function names in the standard library. *)
let func_names = ["print"; "Turtle"; "down"; "up"; "forward"; "turn";
"paintComponent"; "resetPosition"; "setX"; "setY"]
let std_symbols = ["r"; "l"; "f"] (*Reserved symbols in draw functions*)
let std_lfunc = ["down"; "up"; "turn"; "forward"; "setX"; "setY"] (*Standard
drawing functions callable from draw/compute functions*)
```

Makefile

```
# Primary Author: Jervis Muindi
OBJS = ast.cmo parser.cmo scanner.cmo lsystemstd.cmo semantic.cmo compile.cmo
lsystem.cmo

TESTS = \
print

TARFILES = Makefile testall.sh scanner.mll parser.mly \
      ast.ml compile.ml lsystem.ml lsystemstd.ml \
      $(TESTS:%=tests/test-%.mc) \
      $(TESTS:%=tests/test-%.out)

lsystem : $(OBJS)
      ocamlc str.cma unix.cma -o lsystem $(OBJS)

.PHONY : test
test : lsystem testall.sh
      ./testall.sh

scanner.ml : scanner.mll
      ocamllex scanner.mll

parser.ml parser.mli : parser.mly
      ocamlyacc parser.mly

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

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

lsystem.tar.gz : $(TARFILES)
      cd .. && tar czf lsystem/lsystem.tar.gz $(TARFILES:%=lsystem/%)

.PHONY : clean
clean :
      rm -f parser.ml parser.mli scanner.ml testall.log \
      *.cmo *.cmi *.out *.diff *.java *.class *.txt lsystem
```

```
# Generated by ocamldep *.ml *.mli
ast.cmo:
ast.cmx:
compile.cmo: lsystemstd.cmo ast.cmo
compile.cmx: lsystemstd.cmx ast.cmx
lsystem.cmo: semantic.cmo scanner.cmo parser.cmi compile.cmo ast.cmo
lsystem.cmx: semantic.cmx scanner.cmx parser.cmx compile.cmx ast.cmx
lsystemstd.cmo:
lsystemstd.cmx:
parser.cmo: ast.cmo parser.cmi
parser.cmx: ast.cmx parser.cmi
scanner.cmo: parser.cmi
scanner.cmx: parser.cmx
semantic.cmo: lsystemstd.cmo ast.cmo
semantic.cmx: lsystemstd.cmx ast.cmx
parser.cmi: ast.cmo
```

test.sh

```
#!/bin/bash
#
# Primary Author: Michael Eng (mse2124)
#
#Run this file with the command:
#bash test.sh
#
#Three phases: Compiles and runs computational test files in Test/, attempts
to compile erroneous test files in Test/Semantic, and compiles, runs, then
validates image output data for test files in Test/Draw.
################################

make
FILES="Test/*.ls"
ACTION="-c"
TESTACTION="-t"
EXECUTABLE="./lsystem"
finalarr=()
echo "---------"
echo "Stage 1: Compiling computational programs in Test directory to Java"
echo "---------"
arr=()
for f in $FILES #Iterate through Test/, compile each
do
      #shortname= ${f:5}
      noex=${f%.ls}
      shortname=${noex:5}
      echo -ne "Compiling $shortname.ls..." #-ne means no newline
      $EXECUTABLE $ACTION $f $TESTACTION
      wait
      if [ -e "$shortname.java" ]
      then
            echo ""
      else
```

```
            arr+=($f)
            finalarr+=($f)
            #echo "Adding $shortname to array"
      fi
done
fails=${#arr[@]}
if [ $fails != 0 ]
then
      echo "${#arr[@]} test file(s) did not compile properly:" #Output list
of files that did not compile to Java as expected
      for var in "${arr[@]}"
            do
            echo "${var}"
            done
else
      echo "All test files compiled properly to Java."
fi

#####################################################
#Check that each output java file has a .class file#
#####################################################
echo "--------"
echo "Stage 2: Checking that each Java file has a corresponding class file"
echo "--------"
FILESF="./*.java" #Change to (pwd)/*.java later?
ACTION="javac "
arr2=()
for f in $FILESF
do
      shortname=${f:2}
      noex=${shortname%.java}
      echo -ne "Checking that $shortname has a corresponding .class file- "
      #$ACTION $shortname
      if [ -e "$noex.class" ]
      then
            echo "$noex.class exists"
      else
            arr2+=($f)
            finalarr+=($f)
            echo "Error compiling $shortname"
      fi
done
fails2=${#arr2[@]} #output list of files that did not compile from Java to
class files as expected
if [ $fails2 != 0 ]
then
      echo "${#arr2[@]} java file(s) did not compile properly:"
      for var in "${arr2[@]}"
            do
            echo "${var}"
            done
else
      echo "All compiled ls files were compiled to Java class files."
fi
```

```
############################################################################
#############
#Execute script to start comparing compute test .class files to expected
output for each.#
############################################################################
#############
echo "--------"
echo "Stage 3: Executing computational java files and comparing against
expected output:"
echo "--------"
declare -A expected
arr3=()
while read line
do
      IFS='~' read -ra ADDR <<< "$line"
#     expected["${ADDR[0]}"]="${ADDR[1]}"
      expected+=( ["${ADDR[0]}"]="${ADDR[1]}" )
done < Test/expected.txt
for x in "${!expected[@]}"
do
      #echo "$x: ${expected["$x"]}"
      if [ -e "$x.class" ]
      then
            compare=${expected["$x"]}
            if [ $x = "longprint" ] #Hacky fix.  Couldn't embed the newlines
into a line of text in the expected.txt file.
            then
                  compare="
n
n
n
n
n
s
w
e
r
t
y
Hello world"
            fi
            echo "----"
            echo "Running $x, expected output is $compare"
            actual=`java $x`
            wait
            rm -f $x.txt
            if [ "$actual" != "$compare" ]
            then
                  echo "Error comparing $x: $actual != $compare"
                  arr3+=(Test/$x.ls)
                  finalarr+=($x)
            else
                  echo "Match for $x: $actual = $compare"
            fi
      else
            echo "$x was not successfully compiled into Java byte code,
skipping it..." #Files in this state were already added to the report in
Stage 2
        fi
done


fails3=${#arr3[@]} #Output list of files that did not execute properly
if [ $fails3 != 0 ]
then
        echo "${#arr3[@]} java file(s) did not execute properly or did not
compile from Java into Java bytecode:"
        for var in "${arr3[@]}"
                do
                echo "${var}"
                done
else
        echo "All computational test files executed as expected."
fi

##############################################################################
##########################################################
#Attempt to compile files in Semantic subdirectory.  They should all generate
compiler errors and not create corresponding java files.#
##############################################################################
##########################################################
SEMANTICFILES="Test/Semantic/*.ls"
echo ""
echo ""
echo "--------"
echo "Stage 4: Compiling semantic test files, these should all cause compiler
errors and fail to create Java code:"
echo "--------"
SEMANTICACTION="./lsystem -c"
semanticarr=()
for s in $SEMANTICFILES #Iterate through Test/, compile each
do
        #shortname= ${f:5}
        noex=${s%.ls}
        shortname=${noex:14}
        echo -ne "Compiling $shortname.ls..." #-ne means no newline
        $SEMANTICACTION $s
        wait
        if [ -e "$shortname.java" ]
        then
                echo ""
                echo "$shortname.java exists- test program did not fail as
expected"
                echo ""
                semanticarr+=($s)
                finalarr+=($s)
        else
                echo "Compiler error encountered, program fails as expected."
                #echo "Adding $shortname to array"
        fi
        echo ""
done
```

```
semanticfails=${#semanticarr[@]} #Output list of files that did not fail to
compile as expected
if [ $semanticfails != 0 ]
then
	echo "${#semanticarr[@]} test file(s) did not fail properly:"
	for var in "${semanticarr[@]}"
		do
		echo "${var}"
		done
else
	echo "All files failed as expected."
fi

#Clean out generated java and class files
CLEAN="rm *.java"
$CLEAN
CLEAN="rm *.class"
$CLEAN

#######################################################################
#Compile files in Draw subdirectory.
#
#Then run each, get its resulting image bitstring.
#
#Then compare to an expected bitstring (get from file in subdirectory).#
#######################################################################
echo "----------"
echo "Stage 5: Compile and run drawing test classes, compare resulting image
data to expected results"
echo "----------"
FILESF="Test/Draw/*.ls"
arr5=()
arr6=()
arr7=()
for f in $FILESF
do
	shortname=${f:10}
	noex=${shortname%.ls}
	echo "Compiling $shortname..."
	./lsystem -c $f -t
	wait
	if [ -e "$noex.java" ]
	then
		echo "$shortname compiled successfully to Java file"
	else
		echo "$shortname failed to compile to a Java file"
		arr5+=($f)
	fi
	if [ -e "$noex.class" ]
	then
		echo "$shortname compiled successfully to an executable class
file"
	else
		echo "$shortname failed to compile into a class file"
		arr6+=($f)
	fi
	java $noex
	wait
	if [ -e "$noex.txt" ]
	then
		echo "Image bitstring output file generated, comparing to
expected result..."
		DIFF=$(diff -q $noex.txt Test/Draw/Expected/$noex.txt)
		wait
		if [ "$DIFF" != "" ]
		then
			echo "Error- differences found in image data.  Recompile
$noex.ls without the -t flag and run to visually verify correctness"
			arr7+=($f)
		else
			echo "Image bitstring output matches for $noex"
		fi
	else
		echo "An error has occurred and the bitstring output file
couldn't be found"
		arr7+=($f)
	fi
	wait
	rm $noex.txt
	echo ""
done
drawfails=${#arr5[@]} #List of drawing files that failed to compile to Java
let "drawfails += ${#arr6[@]}" #List of drawing files that failed to compile
from Java to a class file
let "drawfails += ${#arr7[@]}" #List of drawing files that ran and output
different image output than expected
if [ $drawfails != 0 ]
then
	echo "$drawfails test file(s) did not behave as expected in this
stage:"
	for var in "${arr5[@]}"
		do
		finalarr+=($var)
		echo "${var}"
		done
	for var in "${arr6[@]}"
		do
		finalarr+=($var)
		echo "${var}"
		done
	for var in "${arr7[@]}"
		do
		finalarr+=($var)
		echo "${var}"
		done
else
	echo "All draw test files compiled and ran as expected."
fi
```

```
numfails=${#finalarr[@]}
if [ $numfails == 0 ]
then
	echo "-------------"
	echo "All test cases passed."
else
	echo "-------------"
	echo "The following test cases did not perform as expected:"
	echo "-------------"
	for var in "${arr[@]}"
	do
		echo "$var failed to compile into a Java file."
	done
	for var in "${arr2[@]}"
	do
		echo "$var failed to compile from a Java file into a class file."
	done
	for var in "${arr3[@]}"
	do
		echo "$var did not execute as expected (either the output result 
was wrong or a runtime error occurred)."
	done
	for var in "${semanticarr[@]}"
	do
		echo "$var did not fail to compile, as expected."
	done
	for var in "${arr5[@]}"
	do
		echo "$var, a drawing test program, did not compile into a Java 
file."
	done
	for var in "${arr6[@]}"
	do
		echo "$var, a drawing test program, did not compile from a Java 
file into a class file."
	done
	for var in "${arr7[@]}"
	do
		echo "$var did not draw the expected output image, please 
recompile it without the -t flag and run to visually verify image integrity."
	done
fi

#####################################
#Clean generated java and class files#
#####################################
CLEAN="rm *.java"
$CLEAN
CLEAN="rm *.class"
$CLEAN
CLEAN="rm *.txt"
$CLEAN
wait
```