Design Goals

In this section we give an overview of the main components of our design and how they work together to generate our simulation. The program overall is split into two primary packages, the controller classes, and the simulation object classes.

The controller class include a Controller, Compiler, Handler and User Interface. To achieve the functionality described in the introduction and the demo of the Logo interface, we need the Controller and the User Interface to communicate with each other.

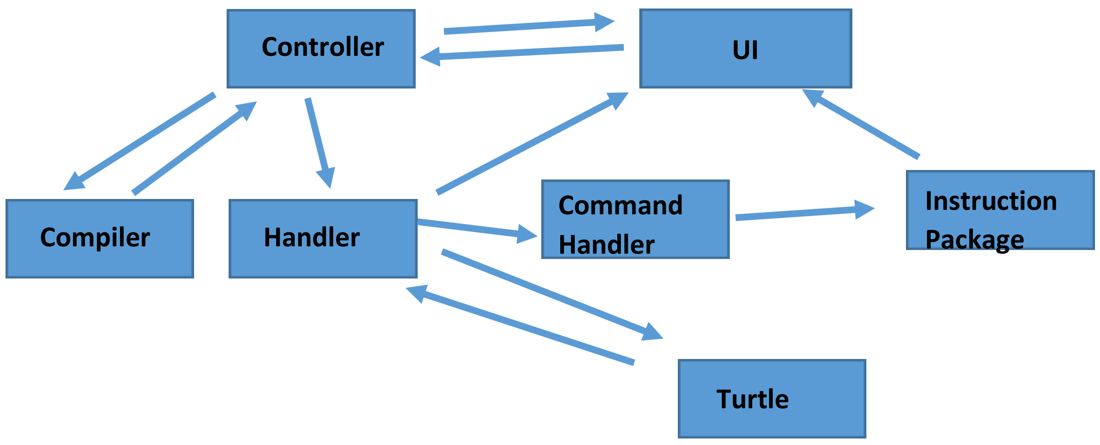


Figure 1: Program Abstraction

Controller

Our controller execute the whole control for the program, it calls the creation of the UI with a start method:

```
public void start(Stage s) throws Exception {

    myUserInterface = new UserInterface(s, this);

  }
```

Then it reacts to the User input sent from the UI and calls required components such as compiler or handler, etc.

User Interface

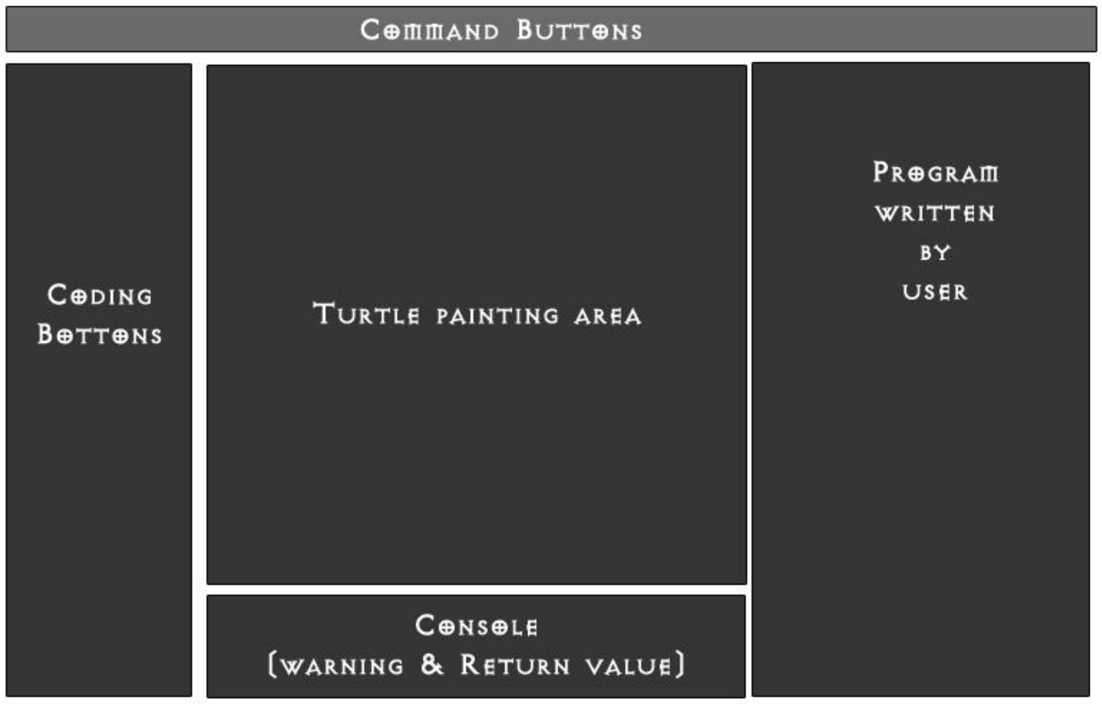


Figure 2: Turtle GUI sketch

The UI class will serve as the controller class that will bridge our User Interface and our Controller classes. As in following the MVC, it will own the Model and View objects and use their public getter methods to update the view and propagate logic instructions. The UI is broken into five basic parts as shown in above Figure 1.

The top section is a tool bar contains required control options for us to implement, such as “Reset”, “Parse”, or “Run simulation”, or maybe open a program file to run, etc.

The Left section contains the coding block buttons of the Turtle languages that allows user to enter program commands interactively.

The middle section is the painting section which shows the program simulation result.

The right section contains the code written by user, either using the left section buttons or typed in directly.

The bottom section is the console part that throws warning or gives return values for the user program.

Handler

Handler takes the syntax tree created and returned by the compiler (Acquire by Controller), and traverse the tree to execute the program user provided. It performs all the computational commands and send all the painting commands to UI to execute.

Turtle

The turtle class holds the turtle object and its properties. How much or how little encapsulation will depend on specific implementation needs once the project begin.

Since we use the handler to control and move the turtle object, the only thing turtle class needs to do is to hold its pen state, show state and x Coordinate, y Coordinate.

It also holds the setter and getter functions that allow handler to execute commands that modifies the turtle properties or return the properties.

Compiler

This class will be used to compile and interpret user input (in the form of strings) into runnable and usable code. The parser mainly contains three parts, a scanner, an interpreter and a syntax tree constructor.

Scanner

Given a user input program, the first task is to identify all the parts (tokens and commands) in the input before attempting to interpret or execute it. The purpose of the scanner is to find the next token in your program, create a symbol table and enter each first seen token into the symbol table and if it is seen next time return its location in the table.

Thus, a scanner method gets the input and stores it in a buffer. Then it scans through the buffer until it identifies a token. Once we encounter a token while processing, first scan the symbol table to see if the token exist. If so, return a reference to its location; if not, insert and return its location. Once processed a token, it stores its reference in the order of processing for further use. For each symbol table entry, it has a Type, a String to hold its name and an integer to hold its value, which will be used in actually processing.

If a variable foo is set to value 5. The symbol table will be like

Type: v	Name: foo	Value: 5

Variables and integers are entered into the symbol table but the Keyword such as commands of the SLogo languages are not entered. We have a separate table for keywords since they are all predetermined and there is no use for entering them multiple times into another table.

The scanner is called by the interpreter when it requires a new token until reaches error or EOF.

The scanner gives a JPanel alert for programs not grammatically correct.

For example, a program is given in examples/loops_with_variables/flower.logo

```
repeat 11 [
  dotimes [ :t 360 ]
  [
    fd 1
    rt / sin :t 2
  ]
]
```

The above code going through the scanner will give a symbol table like below:

Type	Name	Value
i	11	11
v	t	0
i	360	360
i	1	1
i	2	2

And the program return is as below:

r i [d [v I] [f i r / s v I]] $ //"$"added as a EOF marker for interpreter use

Interpreter

The interpreter receives the return value from the scanner and parse the data with a SLR(1) parser and identify if it is syntactically correct. If it is, then produce a right most derivation of the program.

Below page is a CFG for the parsing rules, I will later construct a parse table and let the program import for parsing.

The interpreter will give us a stack of production rules used during parsing the program.

It will return a Boolean value if the program is syntactically correct.

The interpreter gives a JPanel alert for programs not syntactically correct.

Context Free Grammar for parsing

1. <Program>→<List>
2. <List>→<Statement>
3. <List>→<List><Statement>
4. <List>→REPEAT<Type>[<List>]
5. <List>→DOTIMES [Variable <Type>] [<List>]
6. <List>→FOR [Variable <Type><Type><Type>] [List]
7. <List>→IF <Type> [<List>]
8. <List>→IFELSE <Type>[<List>][<List>]
9. <List>→TO Variable [<Parameters>] [<List>]
10. <Parameters>→<Type>
11. <Parameters>→<Parameters><Type>
12. <Statement>→<Command>
13. <Statement>→<Queries>
14. <Command>→SET Variable<Type>
15. <Command>→<Move><Type>
16. <Command>→<Turn><Type>
17. <Command>→SETXY<Type><Type>
18. <Command>→TOWARDS<Type><Type>
19. <Command>→<Property>
20. <Command>→HOME h
21. <Command>→CS c
22. <Queries>→XCOR x
23. <Queries>→YCOR y
24. <Queries>→HEADING e
25. <Queries>→PENDOWNP
26. <Queries>→SHOWINGP
27. <Move>→FD
28. <Move>→BK
29. <Turn>→LT
30. <Turn>→RT
31. <Turn>→SETH
32. <Property>→PD
33. <Property>→PU
34. <Property>→ST
35. <Property>→HT
36. <Type>→Constant
37. <Type>→Variable
38. <Type>→<Math>
39. <Type>→<Boolean>

40. <Math>→+<Type><Type>
41. <Math>→-<Type><Type>
42. <Math>→*<Type><Type>
43. <Math>→/<Type><Type>
44. <Math>→%<Type><Type>
45. <Math>→~<Type>
46. <Math>→RANDOM<Type>
47. <Math>→<Tri><Type>
48. <Math>→LOG<Type>
49. <Math>→POW<Type><Type>
50. <Tri>→SIN
51. <Tri>→COS
52. <Tri>→TAN
53. <Tri>→ATAN
54. <Boolean>→LESSP<Type><Type>
55. <Boolean>→GREATERP<Type><Type>
56. <Boolean>→EQUALP<Type><Type>
57. <Boolean>→NOTEQUALP<Type><Type>
58. <Boolean>→AND<Type><Type>
59. <Boolean>→NOT<Type>
60. <Boolean>→OR<Type><Type>

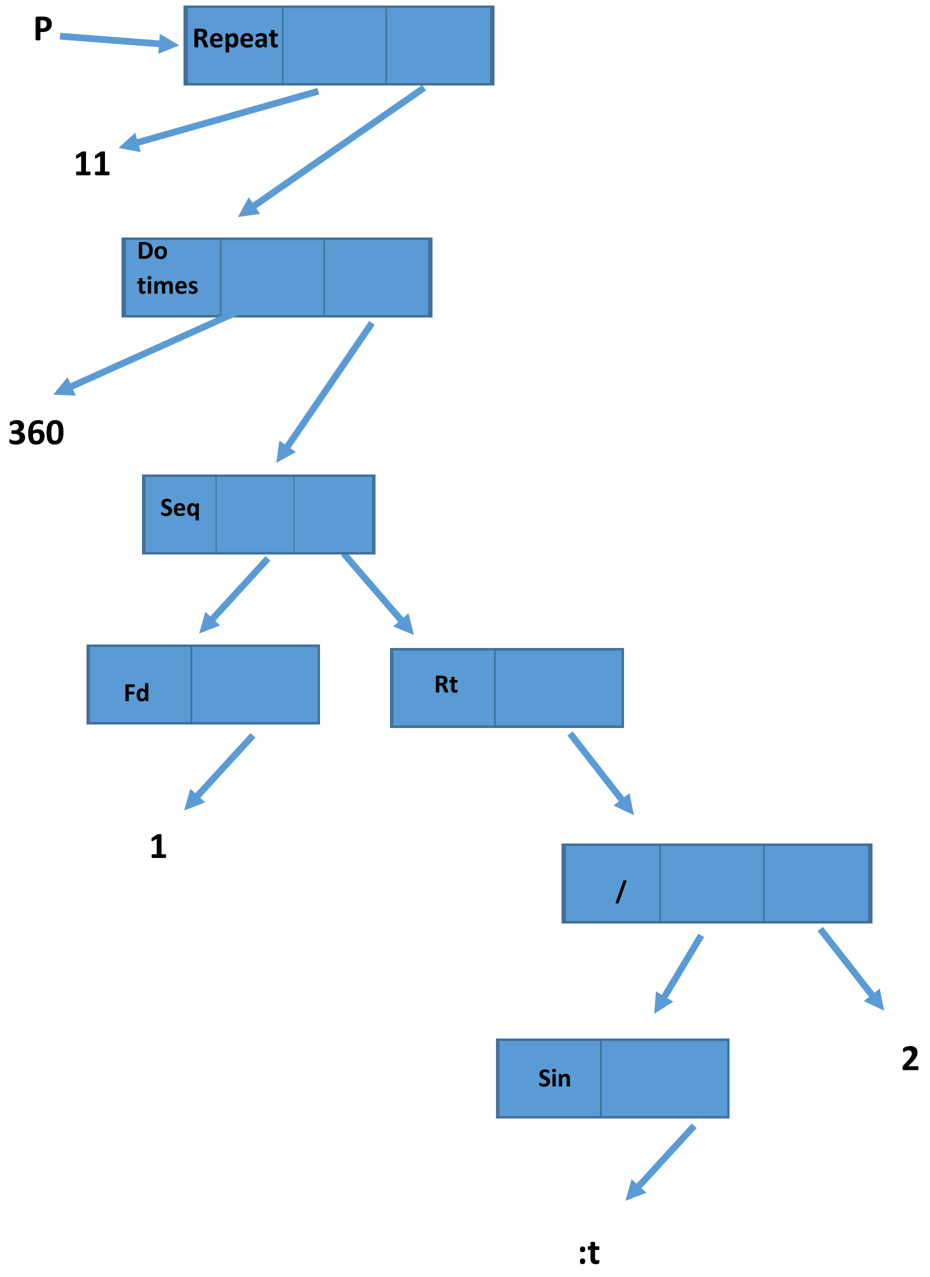
(Just a rough draft, I will further debug the above grammar while implementing the project)

Syntax Tree Constructor

This part mainly functions to identify if the program is syntactically correct. Since semantically correct programs can be syntactically wrong.

```
repeat 11 [
  dotimes [ :t 360 ]
  [
    fd 1
    rt / sin :t 2
  ]
]
```

Still take the above flowers example, the tree constructed will be as follows:



If we are able to construct a syntax tree as above based on the program given, the program is syntactically correct.

Then the compiling procedure is over and we sent the syntax tree back to the handler for executing.

Instruction Package

The instruction package all have a basic hierarchy from the instruction interface:

It will be divide into Math Command, Queries Command or Move Command packages.

We can call from the commandHandler like below:

```
Instruction.math.add(input1,input2);


Public class Add {

Public int apply (int input1, int input2);

        Return input1+input2;
```

Program API description

When coming up with a design for our API, we really tried to create a system that would efficiently combine both the Model and the View aspects. We decided to utilize a controller class. With a controller class, we can remove the messy overlap and focus on proficiently managing communication. A controller class lets us accurately handle all interaction, including errors, outside of the confines of the Model and View. By disassociating the UI with the handler via the controller, one no longer need worry about how the other behaves. We thought this flexibility was pivotal to allow for possible extension to the handler. In this design, the UI and Controller interact frequently, as the UI passes whatever command or input it is given to the controller, and then waits until the Handler updates its visuals. The Handler & Compiler translate the commands and apply their logic to the scenario, sending these changes to the UI in order for the visuals to change. Distributing the power like this not only increases efficiency of the code, but also gives room for changes to be made and for new features to be implemented.

It can be shown clearly from the below example.

Input is passed in Strings from UI to controller,

The controller calls the compiler to compile the String to a syntax tree and pass back to Controller

The Controller calls the handler class and do a tree traversal and calls the instruction package hierarchy to perform operations needed.

Example Code

If the user types “fd 50” as an input string the sequence is as follow:

UserInterface

    If (inputString!=null) Controller.compile(inputString);

→

Controller

Private boolean compile (String inputString);

    syntaxTree=Compiler.compile(inputString)

→

Compiler

Private Tree<String> compile(inputString) throws IllegalArgumentException

    Return Interpreter(inputString);

private string scanner(String stringBuffer) throws IllegalArgumentException

private Tree<String> interpreter(String inputString) throws IllegalArgumentException

→

Controller

Private handle (Tree<String> syntaxTree);

    Traverse the tree and find the fd command

→

CommandHandler

Private int apply(String command,int input)

Instruction.move.forward(input);

→

Instruction.Move.Foward

Private int apply(int dist)

    Apply changes and show changes in GUI

Junit Test

For Compiler

```
public class compilerTest {

  @Rule
  public ExpectedException exception = ExpectedException.none();

  @Test
  public void testExpectedException() {
    exception.expect(IllegalArgumentException.class);
    exception.expectMessage(containsString("Grammatically Error"));
    //* is not an acceptable variable name
    String testInput=”fd *”;
    Compiler.compile(testInput);
  }
  @Test
  public void testExpectedException() {
    exception.expect(IllegalArgumentException.class);
    exception.expectMessage(containsString("Syntactically Error"));
    String testInput=”fd 12 13”;
    Compiler.compile(testInput);
  }
  @Test
  public void testExpectedException() {
    exception.expect(IllegalArgumentException.class);
    exception.expectMessage(containsString("Semantically Error"));
    //dist not initialized
    String testInput=”fd dist”;
    Compiler.compile(testInput);
  }


}
```

Alternative Designs

Before coming to conclusion to our final design, we explored many possibilities with regard to the possible implementation of our program. In our first implementation, we thought of having just two main classes running the bulk of the simulation. We thought we could maybe have one class running SLogo. One main class would take care of everything relating to running the model. Another main class would take care of running the UI and these two classes would refer to each other. It in theory can work but since all the functionalities are all in one class would make debugging harder, as we looked further into adding hierarchies it became even more obvious that extra functionalities were harder to implement when they all had to communicate with each other. As we added more classes our code's functionality was diminished as the program grew larger.

The advantages of our current design is that it supports extra functionalities. When we have a controller acting as a medium between the two classes makes for a much better and efficient code. Another design we explored implementing was having a main class that manages both the front end/GUI and the back end/model at the same time. The problem with that design was that we had to check on the status of both the GUI and the model every so often. It was a bottom up design, what we want to do is update the GUI and model when they need updating not all the time. This way we can avoid bottlenecks and other problems.

Team Responsibilities

The team will be split up into two subgroups. Zach and Wenjun will handle Controller and Compiler design and implementation, while Steven and Kevin work on the GUI and View design and implementation. The group will meet as a whole at least once a week and all group members are expected to attend. In addition we will meet in partial groups or teams as necessary.