Project 2 ReadMe

Methods we wrote and how we tested them:

LineNumber Class:

Constructors: The constructor tests could potentially show the improper initialization of LineNumber objects. The first constructor, being a no argument constructor, simply works as it should. Our tests confirm this, testing that a newly initialized LineNumber with no arguments would start at a LineNumber of just 1. In addition, our class includes a LineNumber constructor that takes in a string, which converts a string (ex. "1.4.4") into a corresponding LineNumber object. Our tests for this constructor are very comprehensive, testing small numbers, large numbers, long line numbers (1.1.1.1.1.1.1.1, etc.) and short line numbers alike. We cannot see any way that this code would break unless an invalid input is put in, which shouldn't happen.

Next method: This method takes our LineNumber ArrayList and adds one to the last number in the ArrayList. Our tests attempt to test all borderline cases, including large numbers in the triple digits and small numbers close to 1. In addition, we tested if the next method would function properly after calling the inner and outer methods to move in and out of subproofs, which it did. Some potential bugs here would be arraylist out of bounds exceptions, but we are sure those won't happen, as the LineNumbers will always have a size of at least one, and the ArrayList get is only called on (size – 1), which should not cause any null pointer errors or out of bounds exceptions.

Inner method: This method takes our LineNumber ArrayList and moves into a new subproof (ex. LineNumber 1.5 to 1.5.1). The implementation is simple, you simply add a new "1" value into the ArrayList. Because of the simplicity of this code, we failed to see how any bugs would be induced here. With the mutable ArrayList sizing, the innerNumber method should work even when the subproof goes in 100 levels deep. The size of the numbers and the depth of the ArrayList shouldn't matter, but we tested several cases too and they were all okay.

Outer method: This method takes our LineNumber ArrayList and moves one level out of the current subproof (ex. LineNumber 1.4.4 back out to 1.5). In cases where the LineNumber is in a subproof, the method has no problems, correctly moving out one level and moving to the next LineNumber. However, there was still the question of how the code would react if we called this on a LineNumber where there was no outer proof to move out to, which could potentially cause out of bounds and null pointer errors elsewhere in the class. Although we first flirted with the idea of making this method not do anything at all in that case and return the same LineNumber, we decided instead to just let this go, as our code, first of all, should not even ever call this code on a LineNumber that is not in a subproof, and secondly, creates a new LineNumber object anyway, which ultimately should not corrupt our data structures and cause any issues.

Subproof method: When this method is called on a LineNumber in a subproof, it returns the LineNumber that started the subproof (ex. LineNumber 1.4.4 to LineNumber 1.4) to help us check when a subproof is complete. This is simply done by removing the last object in the ArrayList, but NOT adding one to the next number like the outerNumber() method does. Like the outer method, there could potentially be issues caused when the LineNumber is currently not in a subproof, but that case is handled by our "if" statement in the beginning that will just return the same Linenumber and not cause

any bugs. Otherwise, our program handles both long and short LineNumbers well, as well as the borderline case of having a LineNumber without any subproofs.

toString method: Our toString method iterates through our ArrayList and takes each number and puts it in a string, adding a "." In between each number. Our whole test file is based on this working properly, and as all our tests have been passing and working fine, we fail to see a reason why this would ever have any bugs given a properly constructed LineNumber.

Expression Class:

For the testing of our Expression class, we looked at two different things. First of all, what expression statements would be correct? After this, we tried statements that would force the class to throw exceptions to make sure no incorrect Expression objects were ever created, instead throwing an error when user input was invalid. We had our testing file create expressions various input strings and made sure that our Expression class would correctly create an Expression object if the input was valid. We tackled the basic case of only a variable, a (E1 operand E2) case, the ~E1 case where the numbers of ~ could vary, and more complex nested expressions. If our Expression class made these objects without any errors, we would know that it could handle correct user input.

In testing invalid output we chose to test this by having our testing cases throwing a fail if an Expression object that took in an incorrect string was ever created. We looked at strings that should force the Expression class to throw an exception, and started off with smaller cases such as the string including a space of being empty. From here, we moved on to cases such as the expression not being correctly parenthesized with either a missing or extra opening or closing parenthesis. Other things that were tested included having incorrectly nested sub-expressions and missing or invalid operands.

TheoremSet Class:

Constructor: TheoremSet has one potential constructor; one with no operand. It simply makes an instance of the object with a TreeMap, named TheoremMap. This map is constructed empty.

Methods: There are three methods to provide access to the TheoremMap that are called in different parts of the code. These methods are put, get, and contains. These just call the methods associated with TreeMap, making testing very easy.

Testing: For testing we used a file called theorems.txt file, which is attached to the submission, and proceeded to used tests that used that file's theorems. Or rather, we tested with a command line prompt, and confirmed that it was passing in correctly, but did not use the file for our posted JUnit tests. The tests made sure all the methods were working properly.

LineStatement Class:

Constructor: The constructor for Linestatement must take in many different sets of arguments, depending on the inference type. Thus, there are several overloaded constructors. It also checks if the statements are logically valid by running them through a series of if checks before finishing the constructor. It creates an "inferred" statement, which is what the statements imply, and checks that against the inputted expression that is supposed to be inferred. If the if checks fail, the constructor will

throw the proper exception. It makes some assumptions about what extendProof will put in, but will throw the requisite exception if not the case. The theorem handling is done by adding a theoremequals method in expression, which utilizes a string to string map to store and compare expressions to see if they match to the theorem framework. This is an extension of the statement constructor that was easier to implement in expression, which is why it is mentioned here instead of the expression class. It was tested the same way as the rest of the statement constructor (see what follows).

These were tested by constructing a proof in the LineStatementTest file, with lines targeted at confirming or breaking the code. Statements designed to fail were put into try/catch blocks. Theorem handling was done in a proof file, as the theorems.txt file could not be imported into junit.

Methods:

The statement class has many simple methods, most of which are getter methods for private variables. The non-getter method s are proved(), which allows access to a show statement after it is called. It just sets a private variable. The ones above are trivial. They were implicitly tested over the course of normal operation.

A toString() method was added for printing the proof. It accesses the information stored in private variables and converts them to strings if necessary, then concatenates them. It was tested in the proof test by making a proof and printing it.

Proof Class:

Parser: Our parser was simple, it split up the string passed in between the whitespaces. We tested every reason that could possibly have been passed into this method and checked if the linenumbers and expressions were stored in the proper place for each separate reason. We also made sure to catch several other potential bugs, such as extra arguments, and in one special case, any additional arguments after the print statement. With every case accounted for, we are confident that our code shouldn't have any bugs unless completely invalid strings are passed in.

extendProof: Our extendProof class takes the parsed input and places it into the Statement constructor with the proper arguments. Because we were already sure that our parser did the work properly, all we had to do was make sure we passed in the right get calls from the parser Array into our Statement constructor, where the rest of the logic is dealt with. The only special case is print, where the previous lines of the proof will be printed out and nothing else will happen. Our extendProof tests add proof steps into the Proof (we made sure to pass in every single reason possible) and at the end, we pass in "print" to the proof to make sure everything passed in correctly, and it worked fine.

Constructor: The Proof class has two potential constructors. The first, no operand, constructor is used for when there is no Theorem.txt file, and therefore no TheoremSet can be passed in. The second constructor passes in a TheoremSet as a single operand and creates and instance with a new LineNumber object a myStatments<Statements> ArrayList.

Methods:

The toString method for Proof takes the Statement toString and iterates though the different myStatments. The associated JUnit test made sure the toString printed the whole proof in a predictable manor.

The isComplete method for Proof returns a Boolean for if the proof is complete. It works by checking if the expression in the last line is equal to the first expression, and it makes sure the line number is only one long, to indicate it is not in a subproof.

For the testing of our Expression class, we looked at two different things. First of all, what expression statements would be correct? After this, we tried statements that would force the class to throw exceptions to make sure no incorrect Expression objects were ever created, instead throwing an error when user input was invalid. We had our testing file create expressions various input strings and made sure that our Expression class would correctly create an Expression object if the input was valid. We tackled the basic case of only a variable, a (E1 operand E2) case, the ~E1 case where the numbers of ~ could vary, and more complex nested expressions. If our Expression class made these objects without any errors, we would know that it could handle further correct user input. .

In testing invalid output we chose to test this by having our testing cases throwing a fail if an Expression object that took in an incorrect string was ever created. We looked at strings that should force the Expression class to throw an exception, and started off with smaller cases such as the string including a space, an uppercase letter, or just being empty string. From here, we moved on to cases such as the expression not being correctly parenthesized with either a missing or extra opening or closing parenthesis. Other things that were tested included having incorrectly nested sub-expressions and missing or invalid operands. We know that as long as every incorrect input failed to create an expression, our expression class would be working correctly as every expression can be seen as just a combination of the basic variable and operand case (a, (a&q)). This meant that our expression would never create an invalid string expression and combined with the testing of the correctly made expressions, our test would have no bugs as long as we touched upon all the possible basic invalid expressions.

In testing our not, getRight, getLeft, and equals method, we decided that the best way was to just create expressions and directly compare them to one another. In this way we went through each one and tested multiple expressions in order to see if our methods would work. We tested to see if getRight and getLeft correctly returned Expression objects using our knowledge of hour we stored our data in the tree structures. With not, we made sure to account for a normal case involving just a variable as well as an expression that contained an operator. In theory if it works for these two than it should work for every case that calls upon the Not method.

Individual contributions:

Anthony Sun: I wrote the logic handling. I created a class statement that originally had all the logic info, but ended up containing everything for a line. The statement constructor handles the bulk of the logic checks, and will throw the proper errors. Some minor logic handling is in the extendsProof method, which feeds the requisite information into the statement constructor. If the required information is not given, the extendsProof method will throw the proper error. Theorem handling is done using a theoremEquals method in Expression that will return true if an expression fits the theorem equals framework. Some small parts were added to statement to enable other member's work, such as a statement.toString() method.

I coordinated with Michael to write tests for Proof, as the logic handling is just a part of the proof handling, albeit a major one. Thus, it made sense to test the proof functionality as a whole, with a few tests targeted at logic functions.

Dickson Lui: I came up with our LineNumber class, and the ArrayList storage methods. I also helped out with some Expression methods and Expression tends, the extendProof method, parsing methods, and the readme for all LineNumber and the Proof parts I worked on.

Jesse Luo: I wrote the Expression class as well as its related methods. Furthermore, I wrote the entry in the readme for Expression.

Michael Vredenburgh: I structured and edited the readme and Wrote toString and isComplete for proof. I had to restructure parser, to allow its use for theorem parsing. Originally, it was only useful for parsing the user inputs. Wrote TheoremSet method. Created the Proof constructor. Did debugging all around.