Lab1 – Nathaniel Foster

My code uses two stacks for processing. It assumes that the input file contains a list of prefix equations separated by new lines and a carriage return. It reads the file line by line. For each line, it pushes each character onto a stack and ignores any white space. The stack is then passed to a converter along with an output variable. The actual conversion returns only true/false for the success or failure of the conversion since validation of the input is done within the converter class. The converter class then uses another stack to create the converted equation. The original equation gets destroyed in the process as characters move from the original stack to the converted stack.

The Stack class that I implemented is array based. A default size of 30 is used for the array. However, a different implementation that uses a String to read the input file allows for the Stacks to be created using variable array length to better fit the data. A variable length is used to indicate the top of the stack.

For my implementation, I initially decided it made the most sense to only prep the input prefix equations for conversion in the main driver. All other verification of the inputs and actual processing would be performed by a separate converter that handled one equation at a time. While it would have been possible to read directly from the input file each line into a stack and then pass that to the converter, I found it to be a much more elegant solution to instead separate the file by line and then traverse the line from right to left.

However, in order to be in alignment with the project parameters, a stack was used to avoid parsing through a String. A stack also naturally allowed the file to be read from left to right since the stack would reverse the order of the input characters due to its LIFO properties. This change had a negative effect on space since the default size of the array supporting the stack was always being used instead of being fitted to a smaller value. Both implementations have the same time efficiency of O(n). A stack will lend itself much better to a recursive solution.

In this example, a stack makes sense because of the nature of the conversion algorithm. Because we are processing the input equations from right to left, it is logical to load the contents of the input onto a stack and then begin processing so that the last item placed on the stack is the right most part of the equation and the first item processed by the conversion algorithm. Using Stacks vice Strings also allowed for much easier use of return variables since a String cannot be altered, a variable is simply just referenced to a new String object.

A recursive solution would require to recursive methods. One for reading the file and another for performing the algorithm which is where while loops are currently implemented. Instead, recursive functions would be implemented that would capture the stopping case for the while loop as well as stopping cases aligning with any break or return statements. The recursive solution may be easier to read as numerous break/return statements were needed to capture all the various error cases.

For my Stack class, I would have liked to have implemented a copy function. This could have avoided the input equation into the converter from being destroyed to do the conversion in a simple and elegant manner. As chance would have it, the variable was not needed again but that may not be true in all cases. The peak function was really only useful for debugging. I would also like to add better validation feedback. The current code only indicates an invalid input. While the error is fairly obvious for smaller equations with invalid characters, it would be nice to go another step and indicate the actual error perhaps by returning a String error message or null if successful.