Shraddha Patel
Project 1 Write-up

<u>Project Approach:</u>

For this project, I approached the problem by first reading the requirements and then creating a general layout on paper. I started by remembering the logic I used to create a prefix expression calculator in Java, last semester. I know we used popping from a stack and saving expressions, however in Haskell it would have to be different since it is a functional language. We cannot use variables in the same way as in Java or Python. I started to break down the project into its main parts: Parsing the expression, Evaluate expression, Receive user input, and a main function.

<u>Project Organization:</u>

First, I imported the necessary modules and defined the important data types.

1. data Expr – these are for all the different data types that will be using our expressions from input.

2. data Error – the different error types that we may come across. This is a beginning list and I believe there would be even more error handling scenarios I have not yet tackled. I wanted to create separate error messages for all situations, but that was too ambitious of me.

3. History – a tuple with the first value being an Int (the ID) and the second value being a Double, the value of result.

   **deriving(Show) is for the ability to convert of values of these data types into strings.

   The project is then organized into a couple of main functions:

1. <u>parse:</u> this function converts a prefix expression into an abstract syntax tree of expressions. Each character has a different meaning and is handled according to the specification in the project description. If the input is empty or encounters an unrecognized character, it returns an appropriate error.
   a. Arguments: It takes a history of previous calculations (a list of tuples) and the string to parse.
   b. Return Type: It returns an Either Error (Expr, String) where Expr is the parsed expression, and String is the remaining unparsed part of the input string.
2. <u>evalExpr:</u> The function recursively evaluates the expression tree. It handles different expression types (Number, Add, Multiply, Divide, Negate) by performing the corresponding arithmetic operations.
   a. Arguments: It takes an expression (Expr) as input.
   b. Return Type: It returns a Double, which is the evaluated result of the expression.
3. <u>evaluate:</u> The function first checks if the input string is "exit" to exit the program. Otherwise, it parses the expression using the parse function and evaluates it using the evalExpr function. It updates the history with the new calculation result and returns the result along with the updated history.
   a. Arguments: It takes a string representing the prefix expression and a history of previous calculations.
   b. Return Type: It returns an Either Error (Double, History), where Double is the result of the evaluation, and History is the updated history of calculations.
4. <u>trim</u> : I added this one later on, so it is its own little function. This is to take care of cases where the user accidentally inputs an extra space before or after their expression.
   a. Arguments: It takes a string as input.
   b. Return Type: It returns a string with leading and trailing spaces removed.
5. <u>mainProcess:</u> The function prompts the user for a prefix expression, reads the input, evaluates the expression using the evaluate function, displays the result, and handles errors or user exit commands. It then recursively calls itself with the updated history to continue processing.
   a. Arguments: It takes a history of previous calculations.
   b. Return Type: It returns IO (), indicating it performs IO actions.

6. main: The function prints an initial message, then starts the main processing loop (mainProcess) with an empty history, effectively starting the calculator program.
    a. Arguments: It takes no arguments.
    b. Return Type: It returns IO (), indicating it performs IO actions.

Problems Encountered and Resolution:

The main problem I had was the parse function's functionality. I was getting many errors, or sometimes that the calculator was unable to handle doing the calculation with spaces in between the tokens. To solve this, the first thing I changed was adding the line to skip spaces. Furthermore, I had to change the actual arithmetic several times so that all the cases are condensed into one function. Breaking each step down was useful in solving this issue.

Another issue I had was in Error handling! I originally wanted to have separate, unique, specific error messages that can pinpoint an issue. The Error data type idea came to mind while discussing with some classmates, and we used this to avoid any confusion when handling errors down the line. This way it was easy to change certain error messages, etc.

Lastly, input validation was also something important I struggled with. This is where the "trim" function came in. I was having trouble handling the case in which there were trailing or leading spaces. I was unsure if this was even supposed to be a feature in the program, but I went ahead and took care of this edge case.

What did I learn:

- I have learned a lot about Haskell, and about how to learn a coding language in a very short amount of time! The suggestion to implement the calculator in a language we are familiar with is very useful. I tried to imagine it out in Java or Python and then was able to translate it into the functional paradigm.
- Concepts such as pattern matching, recursion, and monads have become clearer to me through this project.
- I learned different ways to think about possible errors, and how to handle different errors.
- Parsing! I learned a lot about the careful way to parse and evaluate expressions in Haskell. Parsing prefix expressions with multiple operators taught me a lot on the intricacies.

Incomplete features:

I don't believe there are any incomplete features in my program based on the project description. It does everything that is asked of the calculator! If I were to make the calculator more advanced, I would include:

- Robust History Management: Being able to call the last result, or navigate through history commands with 'prev' or 'next'
- Including more operations: square root, exponentiation, etc.
- I am not sure if I covered all the possible errors. I still am nervous I might have forgotten a certain error to handle. I would continue playing around with my calculator to catch any possible errors or have someone else look at it to find possible errors.
- Improve User Interface. We can always improve this aspect.