**Haskell Assignment 3**

For full credit, do enough of the following to earn 10 points.

1. [2 points] Write a function to compute the volume of a sphere, given its radius.

**Answer:**

**volSphere1 r =**

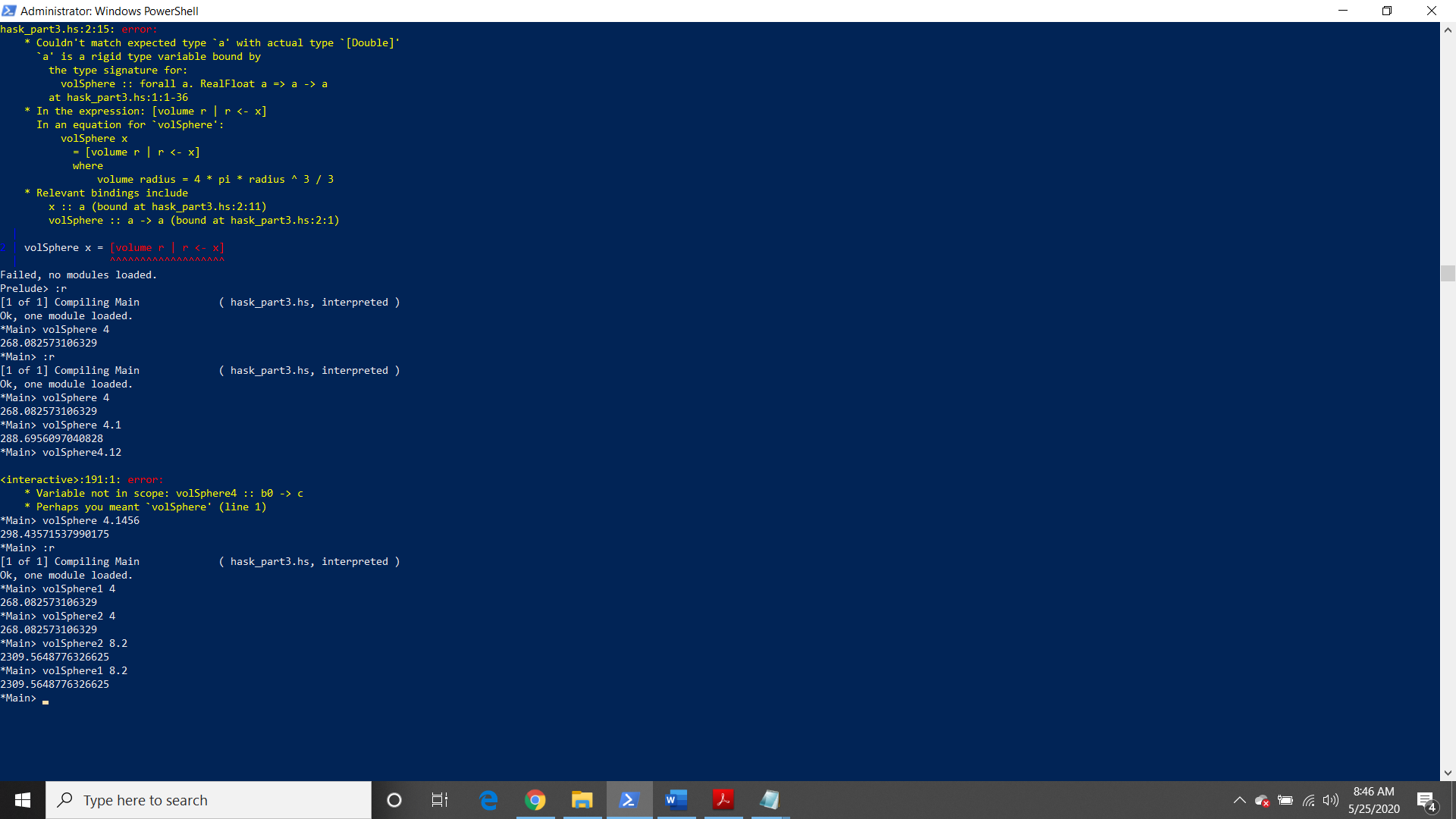
**let fHalf = 4 \* pi**

**lHalf = r^3 / 3**

**in fHalf \* lHalf**

**volSphere2 r = 4 \* pi \* r^3 / 3**

The two functions reside in “hask\_part3.hs” The two function will take the inputted value of r as the radius and compute the volume of a sphere if its radius. volSphere1 splits the equation into two halves and then multiples the results of each half with each other to obtain the result/value of a sphere. volSphere2 is the straightforward approach and simply place the entire formula on the other end of the function. I did these two approaches that I read in learnyouahaskell.org to verify if these two approaches will produce the same result and to make sure the order of operations were done correctly if volSphere2 somehow didn’t behave as expected. Both functions return the same results when given the same radius and verified the results on a calculator.

**Screenshot for** **volSphere1 and volSphere2**

1. [2 points] Write a recursive function to raise a number to a power.

**Answer:**

**numPow1 x 0 = 1**

**numPow1 x y = x \* numPow1 x (y-1)**

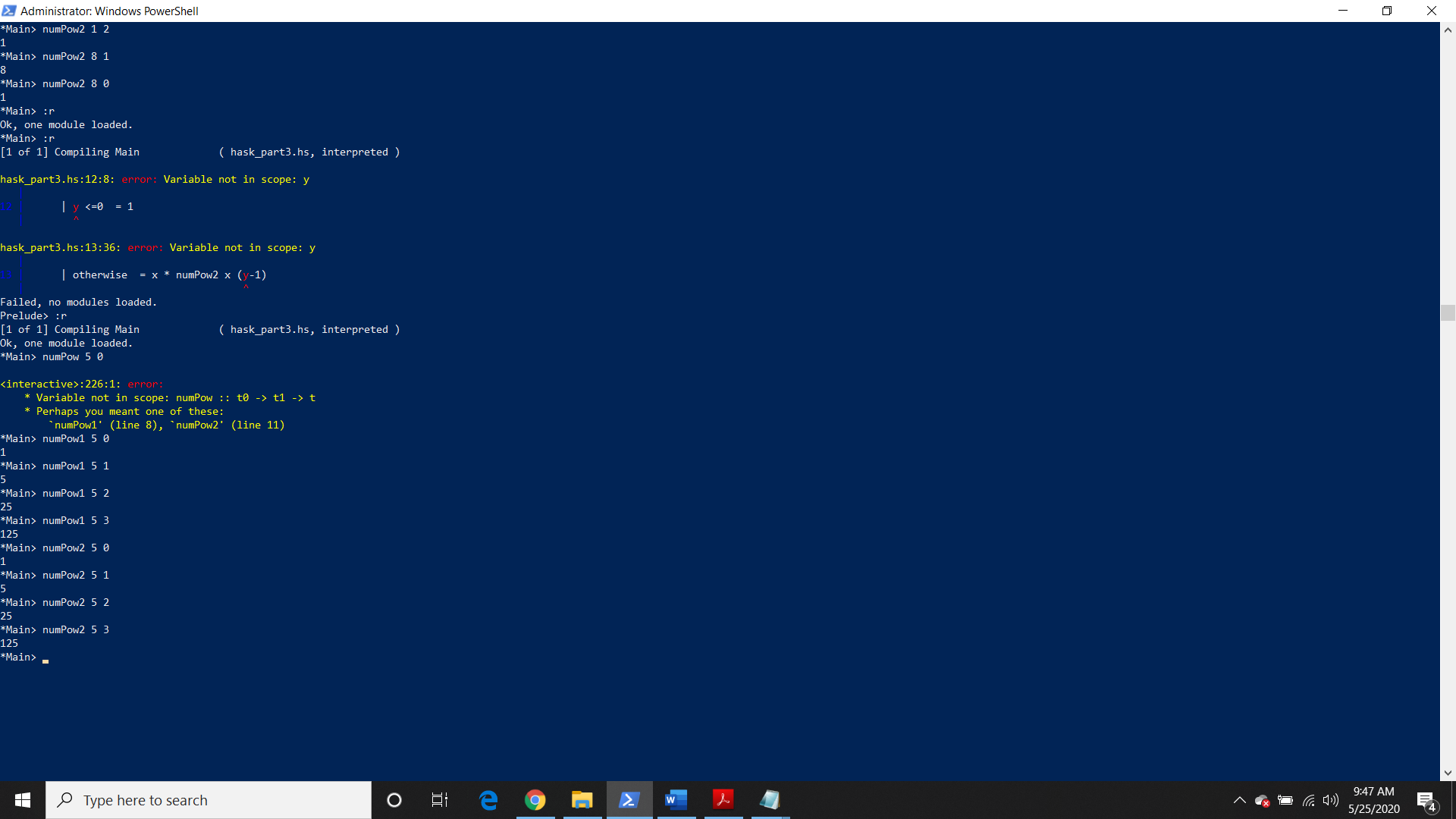
**numPow2 x y**

**| y = 0 = 1**

**| otherwise = x \* numPow2 x (y-1)**

The two functions above are contained within “hask\_part3.hs”. numPow1 breaks the function into two expressions where the first expression is a rule/condition that states whenever y is zero, always return a 1. While the second expression is the recursive function that states multiple x by a number of other x’s that are generated until y becomes 0. Once y equals zero, numPow1 finalizes the result and computes product of x multiplied by each x that was generated from the recursion expression. numPow2 functions exactly like numPow1 however the expression utilizes guards in the syntax to define the recursion function. Both functions are unable to take a negative number for either x or y and both functions will break or go into an endless loop in y is not a whole number. Thus the limitation of both functions is that the functions only raises positive numbers to a power that is a whole number equal to or greater than 0.

**Screenshot for numPow1 and numPow2**



1. [2 points] The implementation of the function **maxlist** below generally manages to find the largest element in a list of integers, but fails if all are negative. Fix it by adding one line.

**mymax a b**

**| a > b = a**

**| b > a = b**

**| otherwise = a**

**maxlist [] = 0**

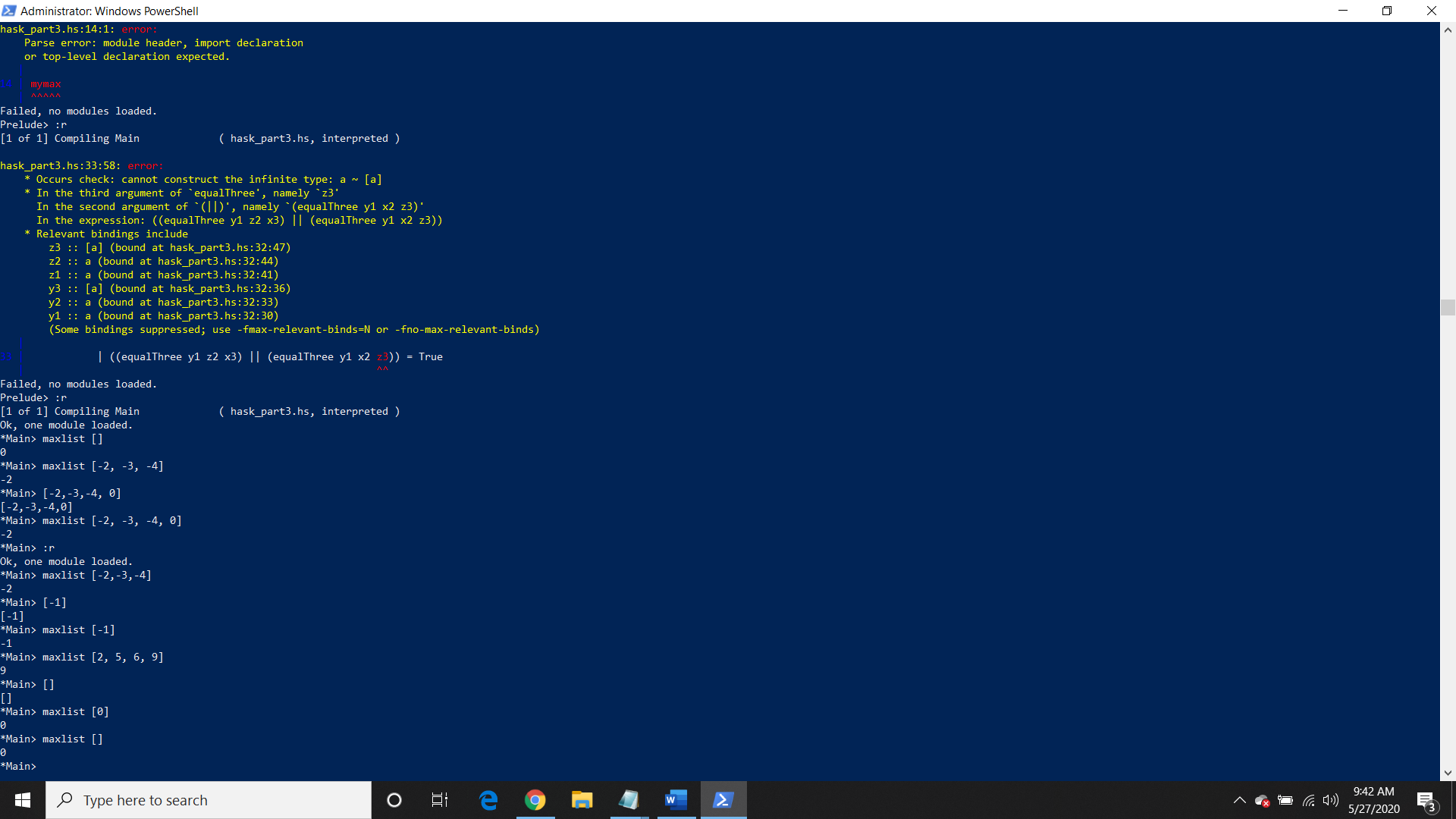
**maxlist (head:tail) = mymax head (maxlist tail)**

**Answer:**

**mymax a 0 = a**

By simply adding the line above in the code, the function will ensure that during the mymax comparison recursion expression, mymax will catch when the list is NOT an empty list and will ignore the 0 since that is the added value given to the list by maxlist [] = 0. This is needed because when given a list of negative numbers to the function, the function will always return 0 because when maxlist adds a 0 to the mymax comparison expression when maxlist reaches the end of the list. This affects a list of negative numbers because 0 will always be the max in that list of negative numbers. So one must add a rule to where the function will ignore that 0 if that zero is the last element/number in the mymax comparison expression and there are other element(s)/number(s) in front of the 0 that are also being compared.

**Screenshot for maxlist**



1. [2 points] Use the built-in function **zipWith** to write a function **wsum** that adds up two lists, while double-weighting the elements of the first list:

**> wsum [2,4] [3,5]**

**[7, 13]**

**> wsum [1, 1, 1, 1] [1, 1, 1, 1]**

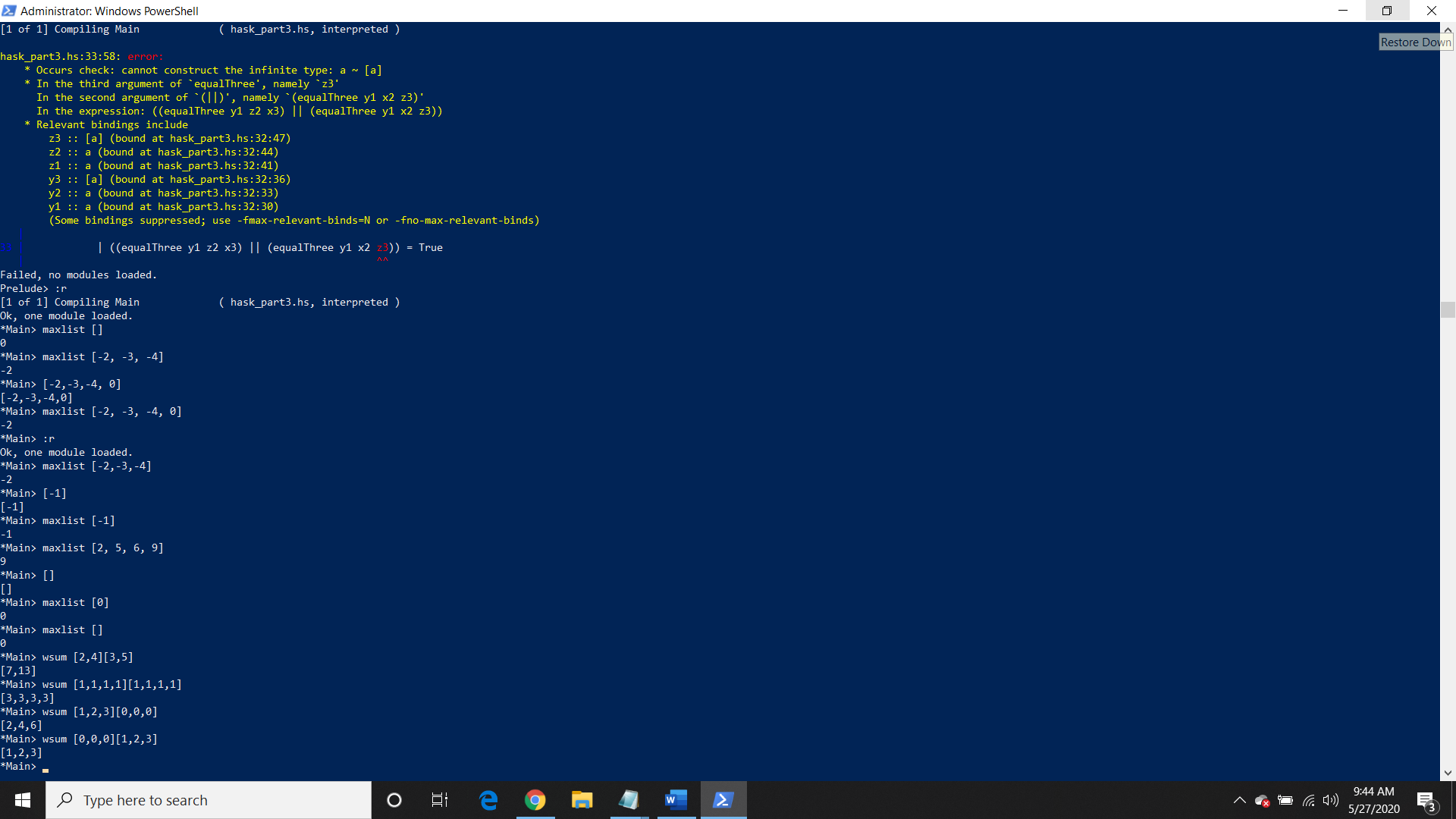
**[3, 3, 3, 3]**

**Answer:**

**wsum = zipWith (\x y -> 2\*x+y)**

The function above is contained in the file “hask\_part3.hs”. zipWith requires two lists of integers and will apply the function of 2\*x+y to each pair of same index elements from both lists.

**Screenshot for wsum**



1. [2 points] Write an expression to compute the sum of all the natural numbers less than one thousand that are multiples of 3 or 5.

**Answer:**

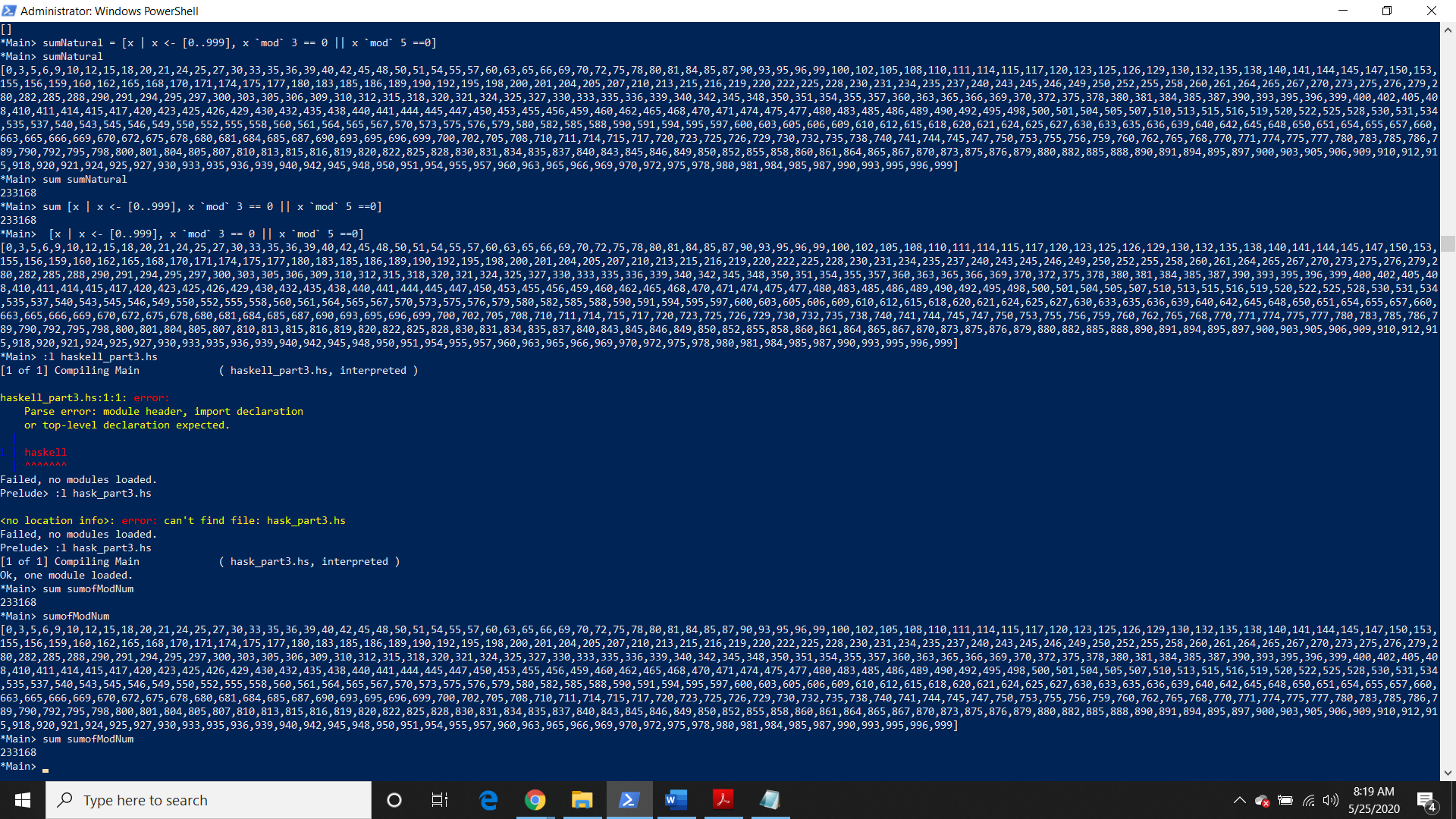
**sumofModNum = [x | x <- [0..999], x `mod` 3 ==0 || x `mod` 5 == 0]**

The expression above is contained within the file “hask\_part3.hs” will give the entire list of all natural numbers that are multiples of either 3 or 5 and that are within the range of 0 to 999. Then to simply sum all the numbers that have been identified and are contained in the list called sumofModNum, we would simply apply the expression below to return the sum of all the numbers from 0 to 999 that are multiples of 3 or 5.

**sum sumofModNum**

**Result of sum sumofModNum: 233168**

**Screenshot for sumofModNum**



1. [8+ pts] Ict-Act-Oet is a new game. It's played on a three-by-three board. One player is X and the other is O. The players take turns placing markers Xs or Os, until one wins. The winning condition is to have three markers in a configuration where no two appear in the same row or column.

Thus in XO-

XXO

OXO

player O is the winner, since there are 3 Os in a valid configuration. Her fourth O, in the lower right, is not part of the configuration,and neither counts for or against her.

Similarly in XOX

OX-

XO-

player X is the winner, since the 3 Xs on the diagonal are a valid configuration. If you play chess, you can think of a valid configuration as one where three rooks could be placed and none of them could attack the others.

1. [10 points] Write a Haskell function wonInThreeByX that will return true iff X has won the game, given as input a Ict-Act-Oet board where both X and O have taken exactly three moves. The board can be represented as you please.

**Answer:**

**equal3X 'X' 'X' 'X' = True**

**equal3X x y z = False**

**checkWin (x1:x2:x3:[]) (y1:y2:y3:[]) (z1:z2:z3:[])**

**|(equal3X y1 z2 x3) || (equal3X y1 x2 z3) = True**

**|(equal3X x1 z2 y3) || (equal3X x1 y2 z3) = True**

**|(equal3X z1 x2 y3) || (equal3X z1 y2 x3) = True**

**|otherwise = False**

**checkMove3 \_ [] = 0**

**checkMove3 a (x:xs)**

**| a == x = 1 + checkMove3 a xs**

**|otherwise = 0 + checkMove3 a xs**

**checkBoard3 a b c**

**| ((checkMove3 'X' a + checkMove3 'X' b + checkMove3 'X' c) == 3) && ((checkMove3 'O' a + checkMove3 'O' b + checkMove3 'O' c) == 3) = True**

**|otherwise = False**

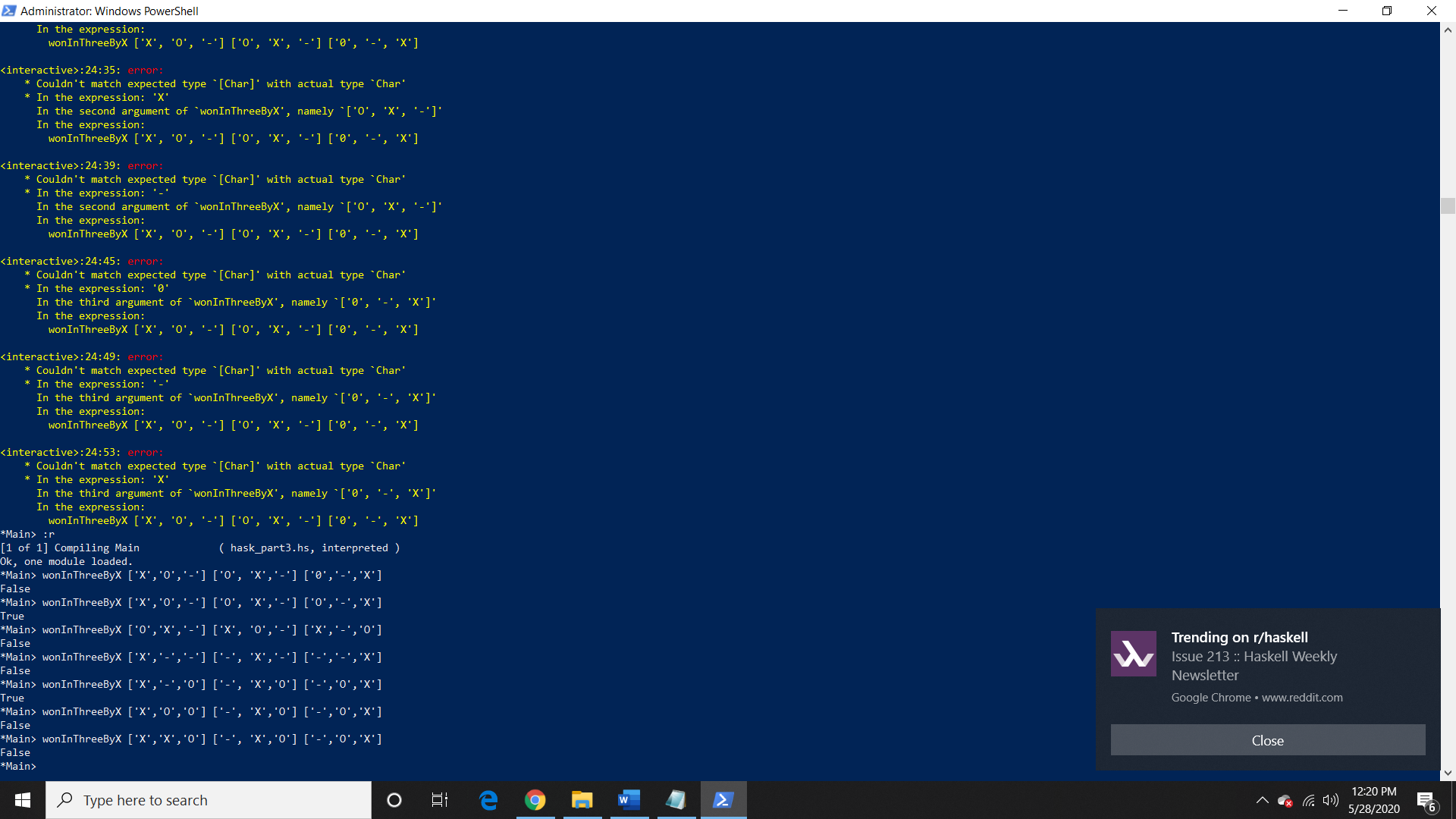
**wonInThreeByX x1 y1 z1**

**| checkBoard3 x1 y1 z1 = checkWin x1 y1 z1**

**| otherwise = False**

The above function is contained in the “hask\_part3.hs” file. “checkBoard3 makes sure there is exactly 3 X’s and 3 O’s within the board, if not return Boolean False which will cause “wonInThreeByX” to return False. Once “checkBoard3” returns true, then “checkWin” will go through all possible wins and if one possible wins contains 3 Chars of ‘X’ then it will return True which will also cause “wonInThreeByX” to return True as well. If none of the possible wins have 3 Chars of ‘X’, then “checkWin” will return False which will also result in “wonInThreeByX” to return False.

**Screenshot for wonInThree**



1. [2 more points] Modify or extend your solution to create a function wonInThree that will return X, O, or false, depending on who, if anyone, has won yet.

**Answer:**

**equal3 'X' 'X' 'X' = True**

**equal3 'O' 'O' 'O' = True**

**equal3 x y z = False**

**checkWinXorO (x1:x2:x3:[]) (y1:y2:y3:[]) (z1:z2:z3:[])**

**|(equal3 y1 z2 x3) || (equal3 y1 x2 z3) = y1:[]**

**|(equal3 x1 z2 y3) || (equal3 x1 y2 z3) = x1:[]**

**|(equal3 z1 x2 y3) || (equal3 z1 y2 x3) = z1:[]**

**|otherwise = "False"**

**checkMove3 \_ [] = 0**

**checkMove3 a (x:xs)**

**| a == x = 1 + checkMove3 a xs**

**|otherwise = 0 + checkMove3 a xs**

**checkBoard3 a b c**

**| ((checkMove3 'X' a + checkMove3 'X' b + checkMove3 'X' c) == 3) && ((checkMove3 'O' a + checkMove3 'O' b + checkMove3 'O' c) == 3) = True**

**|otherwise = False**

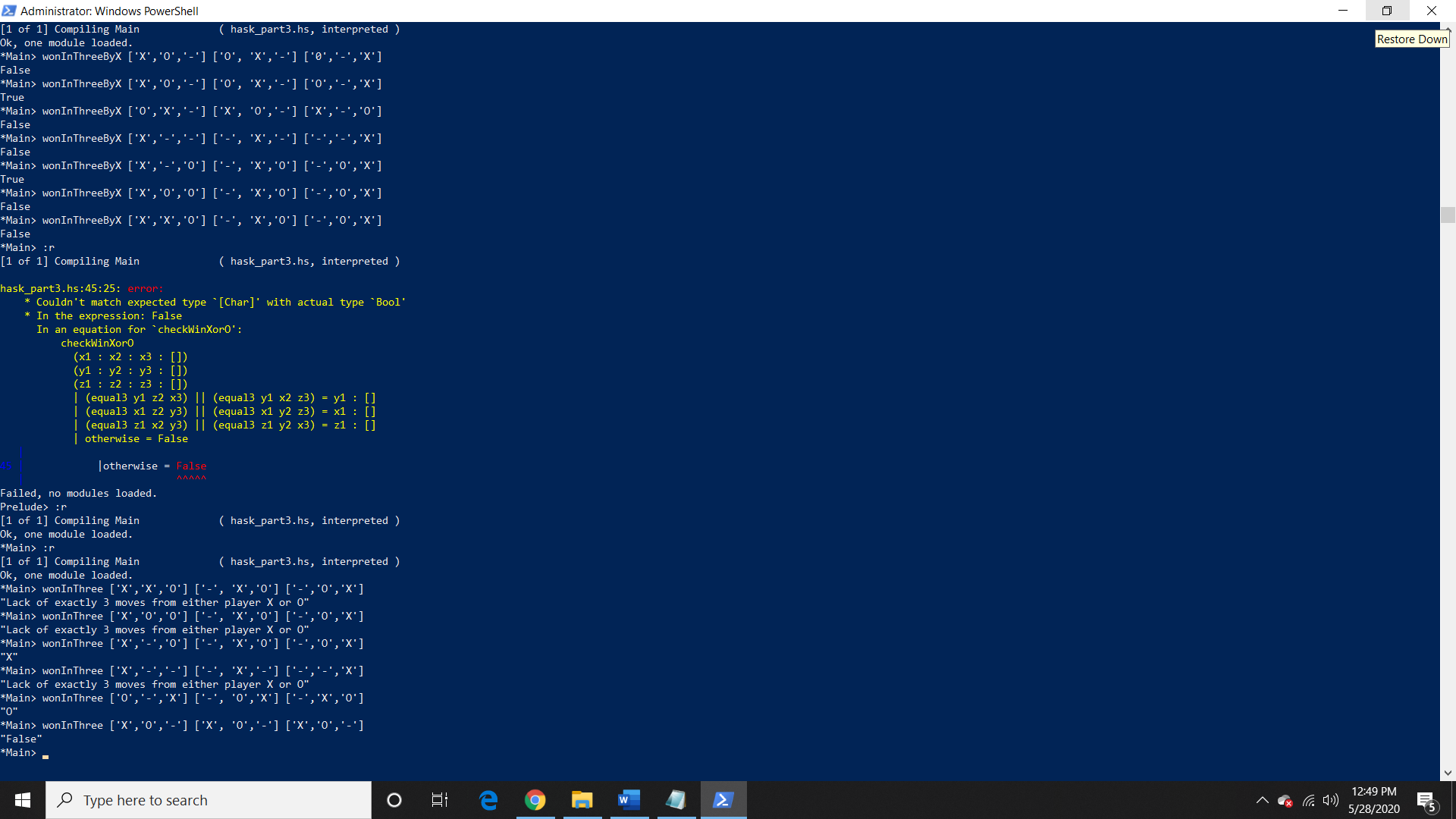
**wonInThree x1 y1 z1**

**| checkBoard3 x1 y1 z1 = checkWinXorO x1 y1 z1**

**| otherwise = "Lack of exactly 3 moves from either player X or O"**

The above function is contained in the “hask\_part3.hs” file. “wonInThree” differs from “wonInThreeByX” since it will return a String depending on the condition that was met. “wonInThree” will return a string of the letter of the player that won or the function will return a string stating “False” when none of the players won. However, these expected results are still entirely dependent on if each of the two players have exactly 3 moves within the board and if this doesn’t hold true then “wonInThree” will return a string stating "Lack of exactly 3 moves from either player X or O". The functions that “wonInThree” have that differ from “wonInThreeByX” are “wonInThree” which returns a string instead of boolean, “checkWinXorO” that will return a string of the winning player’s letter or “False, and “equal3” which will check if either of the two players satisfy one of the win conditions.

**Screenshot for wonInThree**



1. [5 more points] Write a function gameIsOver that will return winner information, namely X, O, or false, given a board with any number of Xs and Os.

**Answer:**

**equal3 'X' 'X' 'X' = True**

**equal3 'O' 'O' 'O' = True**

**equal3 x y z = False**

**gameIsOver (x1:x2:x3:[]) (y1:y2:y3:[]) (z1:z2:z3:[])**

**| ((equal3 y1 z2 x3) || (equal3 y1 x2 z3)) = y1:[]**

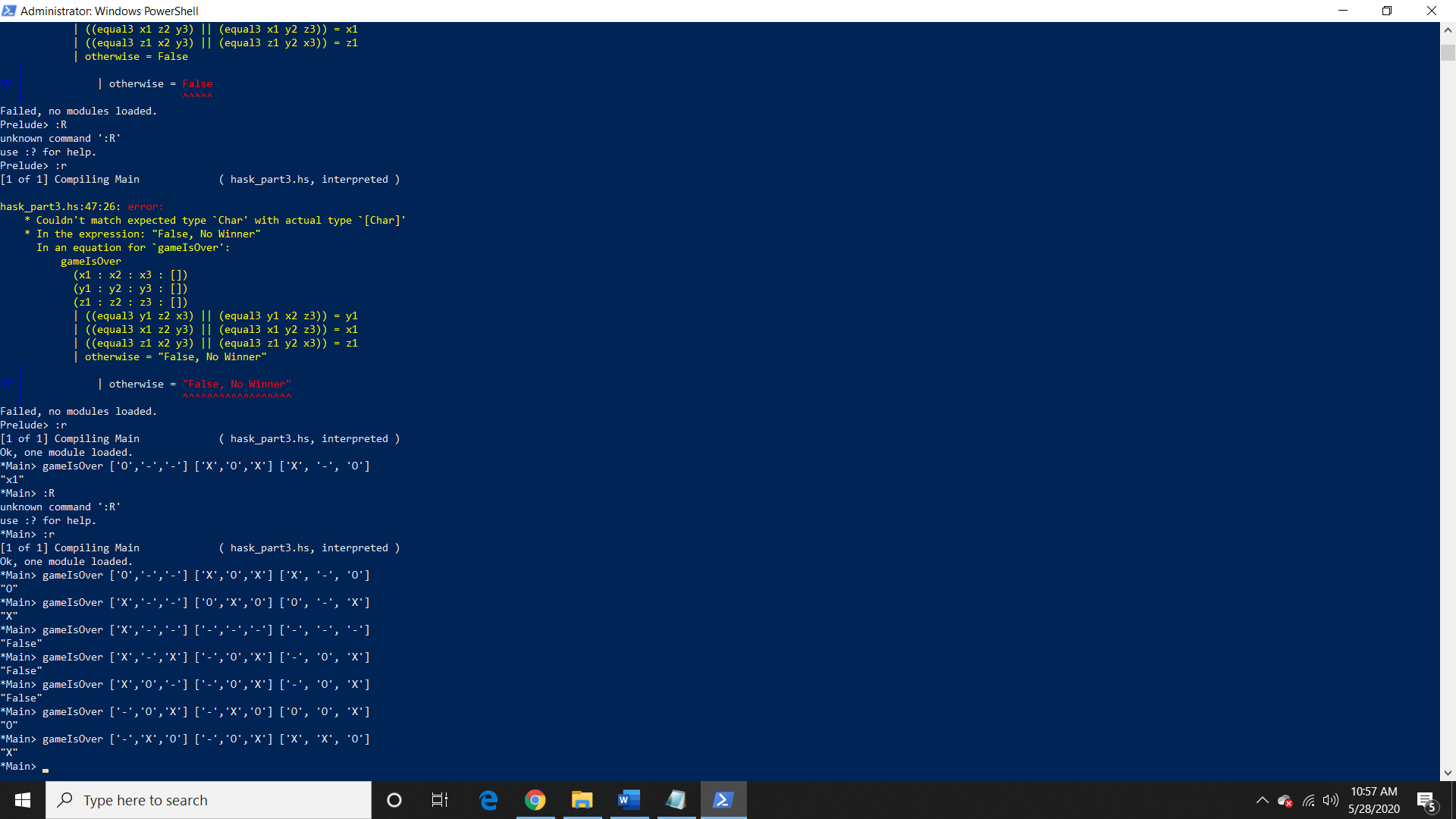
**| ((equal3 x1 z2 y3) || (equal3 x1 y2 z3)) = x1:[]**

**| ((equal3 z1 x2 y3) || (equal3 z1 y2 x3)) = z1:[]**

**| otherwise = "False"**

The above function is contained in the “hask\_part3.hs” file. “gameIsOver” function only concerns itself with the win conditions and if one of the win conditions holds true then it will return the winning player’s letter as a String NOT as a Char. If none of the win conditions return true than it will return “False” as a string but you can also replace it to say “No winner” as well.

**Screenshot for gameIsOver**



You may discuss algorithms and strategy with anyone, and discuss Haskell syntax and semantics with anyone. You may not share code with classmates, but you can use any code you find on the web, with acknowledgment of the source.

1. [2 points] For one of the above problems, list and describe the major differences between your Haskell solution and a typical iterative program for the same problem.

For question 4 regarding ‘wsum’ function, the major difference is that Haskell can apply the ‘wsum’ function with only one line of code using mapping unlike the typical iterative program. In order for the iterative to accomplish the same ‘wsum’ function to two lists/arrays, it would require more than one line of code that contains a conditional statements and a control structure. There would have to be a condition statement that will truncate the resulting array to have the same length as the list/array with the minimum length. Then there would also have to be a control structure that iterates through both lists/arrays given the same index, contains a statement that multiples the current number in the first array/list by 2 than add the resulting product with the number in the second list/array that shares the same index with the current number in the first list, and halts when the minimum length of either list/array is met to ensure truncation happens. In addition, we would have to declare another array/list with length of the minimum length between the two given lists/arrays in order to store all the results from applying the function to the two given arrays/list so that the program can return a single list/array that contains all the results obtained from the function. Conclusion, in terms of applying and defining functions and recursions, a Haskell program is more efficient than imperative program due to the less amount of code needed to define and apply a function unlike the imperative program.

Hand in your code and evidence that it ran correctly, for example screenshots on test cases.