Mathematical Expression Evaluator (Part 2)
Foundations 2 (F29FB)

Jack Webster

H00154950

Introduction

The aim of this task to create a program that extends the functionality that was implemented in part 1 of the coursework. The program will be written in python using PyCharm IDE, extending the program from part 1. Added functionality will consist of checking whether or not an expression is a function and applying that function, producing the inverse and domain of a function or list of ordered pairs, producing the union, difference and intersection of sets and implementing diagonalisation.

Extended Functionality

As this program will extend the program for part 1 it will include all past functionality that was implemented, allowing the program to satisfy the specification from both parts of the coursework.

Only functionality implemented in part 2 will be discussed here, and is as follows:

evalIsFunction()

This method checks whether or not the expression inputed is a function, returning 1 if it is and 0 otherwise. A temporary dictionary is used to store information about tuples in the expression. For a pair (x,y) if x does not exist as a key in the dictionary then it is added with y as a value. If x is already a key in the dictionary then the value associated with it is compared to the y in the pair, if they are not equal then the expression does not obey the laws of a function. The expression is a function if each pair obeys the laws of a function and there are no other pairs to check.

evalApplyFunction()

This method applies an expression to a function, returning the result on successful execution. Like the previous method, a temporary dictionary is used to store information about pairs in the function.The first argument must be a function and contain at least one valid pair that corresponds to the second argument, if a valid pair is not found or the first argument is not a function then the result is undefined. The method follows similar rules to the previous function in checking that pairs obey the laws of a function, returning undefined otherwise. A value is returned once each pair in the function is checked and does not terminate.

evalUnion()

This method takes in two sets and returns the union of the two sets. A blank set, ‘union’, is used to hold the results. Each element in the two sets is checked if they exist in ‘union’, if they do not then they are added. The method terminates once all elements have been checked and the resulting union is returned.

evalDifference()

This method takes in two sets and returns the difference of the two sets. A blank set, ‘diff’, is used to hold the results. Each element in the first set is checked if they exist in the second set, if they do not then they are added to ‘diff’. The method terminates once all elements have been checked and the resulting difference is returned.

evalIntersection()

This method takes in two sets and returns the Interection of the two sets. A blank set, ‘inter’, is used to hold the results. Each element in the first set is checked if they are equal to each element in the second set, if they are equal then the element from the first set is added to ‘inter’. The method terminates once all elements have been checked and the resulting intersection is returned.

evalDomain()

This method takes in a function or a set of ordered pairs and returns the domain of that set. A blank set, ‘dom’, is used to hold the results. The left side of each pair in the function is checked if they exist in ‘dom’, if they don’t exist then they are added to ‘dom’. The method terminates once all elements have been checked and the resulting domain is returned.

evalInverse()

This method takes in a function and returns the inverse of that function. A blank set, 'inv', is used to hold the results. The first argument of 'value' is checked if it is a function, if it is not then the method returns undefined. For each pair in the function the elements are switched and added to 'inv'. The method terminates once all elements have been checked and the inverse of the function is returned.

evalDiagonalize()

This method takes in four subexpressions and returns a function through the process of diagonalization. The method begins by checking that there are exactly four parameters passed, returning undefined otherwise.

The four subexpressions are as follows:

- V1 can be represented as a set {V0, V1, V2, …, Vi}, a list of integers forming the domain of V2.

- V2 can be represented as a set of pairs {(V0, F0), (V1, F1), (V2, F2), …, (Vi, Fi)}, for each element in V1 there is a function that corresponds to it, each function can be represented as a set of pairs in the form of {(V0, N0), (V1, N1), (V2, N2), …, (Vi, Ni)} where N is an integer.

- V3 is a set of pairs that are used to modify elements once they have been selected in the diagonalization process, e.g. for a selected pair (Vi, Z) and a pair in V3 (X,Y), if Z is equal to X then set Z equal to Y

- V4 is a value that given the selected pair (Vi, Z), Z is replaced by V4 given that in a pair (Vi, Fi), the function Fi does not contain a pair where Vi is the first element.

V2 can be visualised as a table, with {V0, V1, V2, …, Vi} listed vertically, creating rows, and {0, F1, F2, …, Fi} listed horizontally, with each Fi corresponding to a Vi.

Each element in V1 contributes to a step in the diagonalisation process, for example; Let V1 equal {0,1} and V2 equal {(0, {(0, 1), (1, 2)}), (1, {(0, 3), (1, 4)})}.

V2 would look like this:

Vi	Fi
0	(0,1), (1,2)
1	(0,3), (1, 4)

V1 is used to iterate through the Vi column and at each step check there is a pair in Fi that has a first argument equal to Vi. This allows pairs to be selected in the same way diagonalization does, resulting in these pairs being highlighted:

Vi	Fi
0	(0,1), (1,2)
1	(0,3), (1, 4)

The next step is to modify the pairs, this is done by using V3. Let V3 equal {(1, 2), (2, 3), (3, 4), (4, 1)}, applying V3 to the selected pairs gives this function:

{(0,1), (1, 4)} -> {(0, 2), (1, 1)}

This result is then returned by the method and the method terminates.

In the program each step is repeated for each iteration through V1, resulting in the pair being selected, modified and inserted before moving onto the next element in V1. The method terminates once the last iteration of V1 has been completed.

Testing

Through testing the program with my own test cases as well as those available to me I have found that the program returns the correct expressions for each function implemented.

Although I believe that more testing would be appropriate and would aid in creating a stronger implementation, further lowering the chance of receiving faulty outputs and generating errors.

Conclusion

I believe that the program satisfies the specification and correctly implements all functions. The program successfully handles the usage of each function in the program and returns the correct expressions. The program would benefit from further testing but due to time constraints it is not possible, however I have enjoyed working on this project and learning how to implement mathematical expressions and functions into programming languages.