Chapter 3

Intuition of the Algebriac Mathematical Program

3.1 What Type of Algebra

- **Plus, Multiply and Power as Operations:** We only will be talking about solving algebra, where the allowed operations will be addition, multiplication and power. Also, when the power operation appear, in the exponent what it will have is an integer equal to or greater than two. This should make everything simple. That is because, all possible numbers could be allowed inside this operations without the fear falling outside the domain of defined. For example if we have included division in this, then we would have to handle division by zero for example. It will need additional handling of situation which we might address how to do in the next book.

- **Variables:** We only have one type of variable, variable variables. Constant variables was not needed and we will need them only when functions and calculus are introduced.

In next book you might get to see things like trigonometric functions along with some calculus being implemented.

3.2 Equations As Trees

We will represent equations as trees. You can observe how it is done from the tree diagram of the equation $x \cdot x + x^2$. This is a very sophisticated way of representating equations and this is done popularly in subjects like symbolic computation or computer algebra. We will have lots of similarity with those subjects even though the idea of TC will set us apart. But anyway, for some reason math equations perfectly gets into tree form and shows all the necessary prooperties which it should. Which includes -

- The sub equations (the equations in bracket) is an valid equation too, just like the whole equation itself

- Changing a certain sub equation leaves the other parts of the equation unchanged

Also, the order of the nodes in the tree will matter, unless something like commutative property would be needed be to be handled.

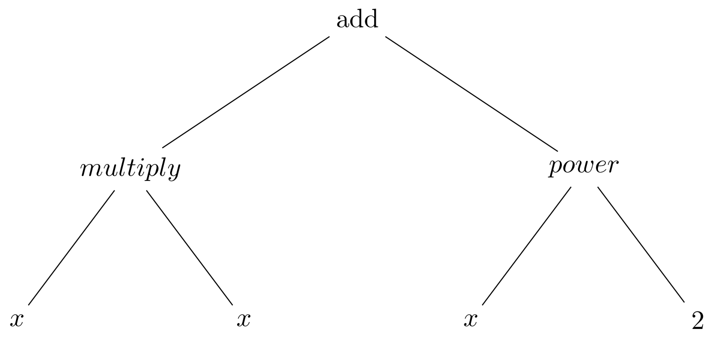

Figure 3.1: $x \cdot x + x^2$

3.3 Brackets are nodes in the trees

Brackets are important in mathematical equations. That what make equations trees, the brackets. Mathematical equations rely on recursiveness. Each bracket returns a value to the parent bracket, them to their own parents. And this process should go on. But sometimes people tend to not write the brackets. They say they rely on the BODMAS rules. That is a valid approach too, but doesn't change the fact. We are just ignoring the bracket because we know we can put them back taking consideration of the operator precedence.

3.4 Atmost two elements in a bracket

We will allow only two elements in a bracket, in most of the cases. We could have allowed more elements to come in a single bracket. But it was chosen for the following reasons -

- More number of elements mean we would need to do permutation, to rebracket in all possible ways. And that's the only way to do that. Using formulas will not suffice.

- After when we are ready to use maximum two elements in a bracket, we could rebracket using simple formulas like $x + y = y + x$ which would be enough to explore things like commutativity and associativity if it appears.

- We are simplifying and focusing on fundamental aspects, reducing the system. Because when two elements are enough in a bracket, adding more elements only mean sophistication but less relying on more fundamental principles.

- But we could allow for more than two elements when we need to consider the order of arguments. And commutativity and associativity does not seem relevant. That time it could be allowed.

3.5 Equation tree as recursive data structure or as string representation

We also are creating a string representation of the tree structure. Its not only for convinience but also has functional role in the software. Of-course we use can use string representation to create a text file of formulas or equations, but its application more than that. Even though, both are relevant purposes. Lets give an example on how we are going to do that. Say, we want to represent $(x + 2) + 3$.

```
f_add
 f_add
  v_0
  d_2
 d_3
```

Both the information of initial spaces and new lines are important to retain all information about a tree. And for functional purposes we can do something like is string "f_mul" in str_equation and the answer will be no because no multiplication was done. Or we check "v_" in str_equation, that will say whehter the equation contain variables. Which might be a useful function.

3.6 Equal equations are not equivalent

There are many equations which we are dealing with here. In mathematics, we usually mention the manipulation of equal equations in our steps and put less emphasis is put on differentiating within equations which are equal. We give a careful consideration on differentiation based on equality though. But in computer science we prefer to to distinction between both equivalence and equality. We can also see that from the tree analysis of equations. What we are going to do after all is, in our project as described in this book, generate equations which are not equivalent. Equivalent equations have the exact same tree structure and in string representation of trees it means equal string content. So we can just do set(equation_list) to ensure no pair of equations selected from the equation_list are equivalent. The next we perform is to categorize the equations into categories of equal equations. That's what will be the final output of the program after we build it.

3.7 Representing integers and variables

We can represent integers by adding a suffix d_ and for variables suffix v_. We will start counting from zero in variables case, v_0, v_1, v_2, so on. And for integers we should represent them like this d_-1, d_0, d_0.

3.8 Applying Fomulas on Equations

Formulas are a pair of equations. Having one lhs and one rhs. Each of the equation can be desrcibed in the tree format as we discussed before. For example, let's take the formula $x + y = y + x$. The two equations in it are, $x + y$ and $y + x$, the lhs and rhs respectively. The purpose of making a pair of equation was to apply them in a third "given equation" to transform it in many ways. Its quite a brief process how we can apply a formula (lhs and rhs) to the third given equation. I will write down the steps here below.

- **Matching given equation structure with LHS:** The lhs will contain variables in it. That is the reason we treat and call it as a structure. Those variables can take in equations within in. But there could be any equation going into the variable, we are free about that. The only thing is the operation order, tree depth, and all that should match. But we have the freedom to fit in anything within the variable. So, if the rest of the things except the variables matches, the structure is the same and we verfied it, we go on the next step.

- **Extracting Variable Data from the Given Equation:** The structure of the lhs and the given equation has matched. That's why we are here in this step. We take the equations which the variables are carrying (it could be a unknown number too, but an equation is also a possiblity, unlike traditional mathematics) and store them in the dictionary variable_list. We will use the information present in the dictionary to complete our next step.

- **Fill the data into the RHS:** We will take the contents of the dictionary and fill them in the variables of the rhs equation. Once its done, we are ready to call the transformed rhs of the formula to be one transformed equation.

Remember that the variables present in the rhs should be equal or a subset of the variables present in the lhs. This is because we can't leave a variable lone in the rhs and we need some extracted data according to the lhs formula structure to fill it.

There one another step zero which we need to take care about. The given equation which had chose, should be chosen as per our requirement. The given equation question may be a complete tree (applying at the root). But we can just extract a part of it, a sub tree and then treat it as the given equation to process it further. This is because not only the whole equation behaves as a equation but its recursive too. Each and every bracket is an equation in itself and we can apply formula to any of them. As a result of this a single formula to single question will have multiple outputs to it. Not necesary but possibly. The lower limit will be 0 (the formula lhs structure didn't matched) and the upper limit is the number of nodes present in the question equation tree.

Check out the example diagrams illustrating all this.

3.9 Formula List And Arithmetic

Here we were only dealing with a single formulas and its application on a given equation. But not only a single formula would be enough, we need many of them in order to do something really mathematical. And hence we do that. We keeping collecting the transformations generated by each formula while we are iterating through the formula list. After iterating through all the formulas we are not done yet though. We have a lot of equation in the collection though, because of our formula list knowlegde. But there is another thing which is missing here. We also need to take care about the arithmetic. Doing that is pretty easy. It would be simple calculator needed for doing that. We can use python operation in order to do so. We will pick a pair of numbers and solve them. How to solve them is dependent on the operation between them, but we are going to only do one pair at a time. We pick a pair of integers in a single bracket, do the operation, destroy the bracket and then include it in our list of equation collection.

3.10 Deduction

We now have the tool with us to be able to generate transformations of a given equation. Give me one equation and the formula list and a calculator, we will be able to generate various transformations of the equation. When doing that is possible, we have covered a lot of steps already in the matter of computing algebra. The next step which remain is create more transformations of the transformed equations further. And keep doing it. Sometimes we might see repeating equations coming but there is no harm in that. We could ignore the equation if it was previously seen given the nature of the equation generator remains same. In computer science, we approach this problem of exploration using search algorithms. The two major algorithms are breadth-first seearch and depth-first search. The breadth-first search completes every possible search before moving on to the next depth. The depth-first search completes all the depth and then eventually searches everything. A cool thing about breadth-first search algorithm in here is that, we can generate the shortest possible path from equation to the other, because we are giving less pirority to the depth (the number of the steps). This exploration is what I called here deduction. Deduction in short is finding equal but non-equivalent to a given equation in a chain form.

3.11 Iterating over All Possible Mathematical Equations

This is the algorithm which is unique and find promising, expecially in the context of TC. It is one of the ways on how the processing power available can be exploited. The earlier method of deduction too exploits the processing power from our TC too but this one does that more neatly in my opinion. This method of iteration isn't a replacement of the earlier one, but both can be engineered together to create a more powerful mathematics software. You will gain insights on how it is possible to, once we list down the details of how the iteration algorithm functions.

The parameters it take.

- **List of operations and the number of arguments each operation should be given:** In our case it is [addition: 2, mutliplication: 2, power: 2] with two arugments each (maximum two elements in a bracket)
- **The depth limit of the equation trees to be generated:** The more the depth limit is the larger the equations tree will be maximally created and hence more number of equations. We can increase depth limit in order to make the software handle more and more complicated problems.
- **The leaf nodes:** The leaf nodes will usually include variables and integers. We will include usually a single variable, v_0 and integers in whatever ranges we want to: ... -3, -2, -1, 0, 1, 2, 3 ...

Given some valid parameters we know it will be output unequivalent equations as we discuss earlier. If the paramters are set high, probably a lots of them. But the explaination remains how does the algorithm in our current implementation. We can explain.

- **Adding operations nodes to the current tree:** Adding operations to the tree is the crucial part when growing the tree. Leaf nodes would end the tree growing at once, so here, we add operations (addition, multiplication and power). But without adding any leaf nodes, the tree will remain invalid.
- **Adding leafs nodes**: So other than adding operation nodes (other than putting brackets), we should add leafs nodes too. We will add leaves one by one. Not all the trees will be valid, but they will be we can collect them. Here, adding leaves mean adding numbers and variables, the main components of equation.
- **Exploring addition of operations nodes and leaf nodes:** We can keep adding operations and leaf nodes in all possible orders and keep collecting the trees which are valid. We can do this exploration again by a search algorithm.

So, after using this function, we have a list of unequivalent equations we can work on further. We will categorize them. Next section explains how.

3.12 Equal Categories

We will create as many as categories as many as equations. Then, if we find both elements from the pairs of each category to be equal, then we can merge those two categories. We can give an example. Let two categories. First category has $x + 1$ and $1 + x$, and second category, $x + 0 + 1$ and $x + 1$. We merge both of this categories as $x + 1$ is common. To create better equality check we will can use deduction and our knowlegde of the formula list. And like that, we need to keep merging the categories, until the number of categories is minimized.

So, after discussion all this, we can check the code of how this is implemented in python programming language.

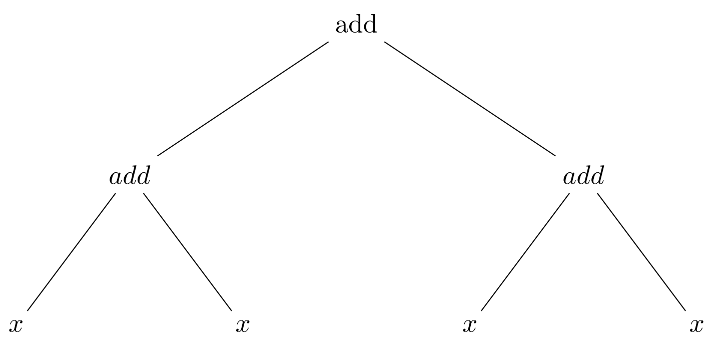

Figure 3.2: Given equation, formula $x + x = 2 \times x$ to be applied

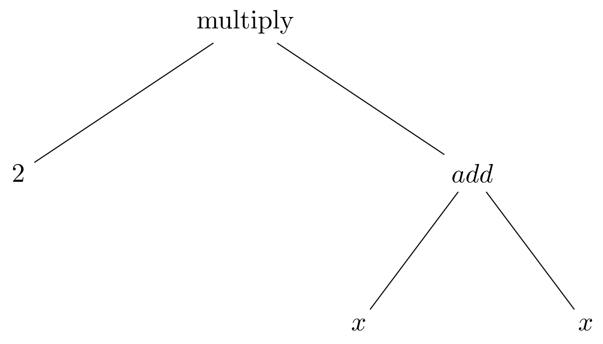

Figure 3.3: Formula applied at root

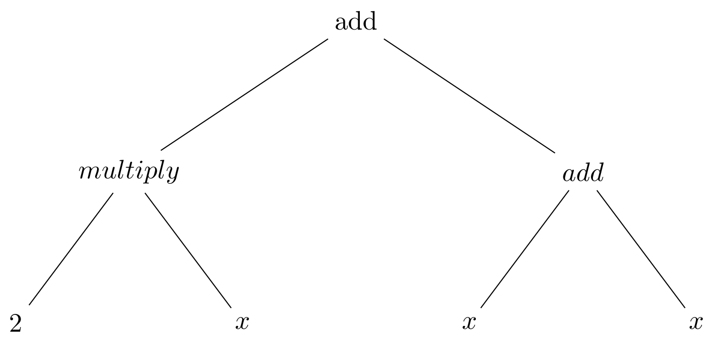

Figure 3.4: Formula applied to a certain part only

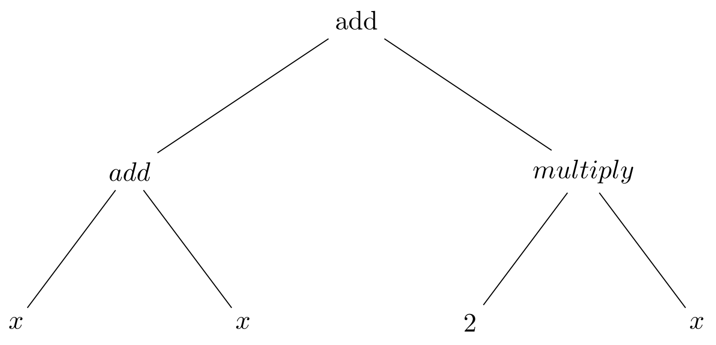

Figure 3.5: Formula applied to some another part

Chapter 4

Brief on Relating the Program Code and Previous Chapter

4.1 class TreeNode

Buliding blocks for buliding equation trees. Refer 3.2

4.2 tree_form()

Converting string representation of equation to tree form. Example of a tree represented as a string. The equation $x \cdot x + x^2$ will be used as example.

```
f_add
 f_mul
  v_0
  v_0
 f_pow
  v_0
  d_2
```

Where, f_add is addition, f_mul is multiplication and f_pow power. v_0 is the variable x and d_2 is the integer 2.

Refer 3.5

4.3 str_form()

Converting the tree representation to string. Refer 3.5

4.4 apply_individual_formula_on_given_equation()

Takes in a single formula, a single equation, and returns equals to that equations return by the formula. In case do_only_arithmetic is true, it will perform artihmetic on the equation and ignore all other things.

Refer 3.8

4.4.1 does_given_equation_satisfy_forumla_lhs_structure()

Returns true if the formula lhs satisfies the given equation structure. Also stores the extracted variable data from the given equation to the variable_list dictionary if its true in statisfying.

Refer first and second point

4.4.2 formula_apply_root()

Apply formula on the root. The equation will transformed from the data in variable_list after it gets filled into the formula rhs.

Refer third point

4.4.3 formula_apply_various_sub_equation()

Apply formula in sub equations, leaving rest of the equation unchanged. Also keep track where to make the change using the count_target_location value.

Refer rest

4.5 return_formula_file()

Formulas are alternatively lhs and rhs in the text file. There can be so many formulas in a file and lhs and rhs pair of each of them.

Refer 3.9

4.6 generate_transformation()

Collect all the formulas from a file and then apply it to equation using the function apply_individual_formula_on_given_equation() and creates transformation list.

Refer 3.9

4.7 search()

Recursively generates all equals to the given equation using generate_transformation()

Refer 3.10

4.8 fx_nest()

This function helps iterate through all mathematical equation given a certain depth limit of equation to form.

Refer 3.11

4.8.1 neighboring_math_equation()

Some function which generates both unequal and unequivalent given a single equations.

Refer the second and third points from the last

4.8.2 append_at_last()

We append the element at the leaf, and check if its a valid equation. A valid equation have all the arguments of the operation filled and also, the leaf node should not be a operator.

Refer the second and third points from the last

4.8.3 bfs()

simple breadth-first search to generate mathematical equations using neighboring_math_equation(). That function will produce the neighboring nodes given a node.

Refer last point

4.9 break_equation(), spot_invalid_equation() and print_equation()

break_equation(), break all the sub equations and return a list filled with them. spot_invalid_equation() ensures power should be integer greater than 2. Discard the equation otherwise.

3.1 mentions why is power being limited

4.10 Driver Code

Makes leaf nodes and operations-arugment dictionary and then use fx_next() to generate only to categorize them based on equality. And then print each category.

Refer 3.12

4.11 Formula List

We mention commutation, association, identity, for all operations power, addition and multiplication. We also write the relationship between power and multiplication as a formula, just to remove all powers to make them multiplication. Also to be said that, we are representing lhs and rhs of each formula alternatively in the string representation of tree in the file.

Refer 3.9

Chapter 6

The Entire Mathematical Program Code

Listing 6.1: main.py

```python
# Copyright (c) 2024 Swastik Majumder
# All rights reserved.

# Part of the book Transcendental Computing with Python: Applications
    in Mathematics - Edition 1

from collections import deque
import copy

# Basic data structure, which can nest to represent math equations
class TreeNode:
    def __init__(self, name, children=None):
        self.name = name
        self.children = children or []

# convert string representation into tree
def tree_form(tabbed_strings):
    lines = tabbed_strings.split("\n")
    root = TreeNode("Root") # add a dummy node
    current_level_nodes = {0: root}
    stack = [root]
    for line in lines:
        level = line.count(' ') # count the spaces, which is crucial
            information in a string representation
        node_name = line.strip() # remove spaces, when putting it in
            the tree form
        node = TreeNode(node_name)
        while len(stack) > level + 1:
            stack.pop()
        parent_node = stack[-1]
        parent_node.children.append(node)
        current_level_nodes[level] = node
        stack.append(node)
    return root.children[0] # remove dummy node

# convert tree into string representation
def str_form(node):
    def recursive_str(node, depth=0):
        result = "{}{}".format(' ' * depth, node.name) # spacings
        for child in node.children:
            result += "\n" + recursive_str(child, depth + 1) # one node in one line
        return result
    return recursive_str(node)

# Generate transformations of a given equation provided only one formula to do so
# We can call this function multiple times with different formulas, in case we want to use more than one
# This function is also responsible for computing arithmetic, pass do_only_arithmetic as True (others param it would ignore), to do so
def apply_individual_formula_on_given_equation(equation, formula_lhs, formula_rhs, do_only_arithmetic=False):
    variable_list = {}
    def node_type(s):
        if s[:2] == "f_":
            return s
        else:
            return s[:2]
    def does_given_equation_satisfy_forumla_lhs_structure(equation, formula_lhs):
        nonlocal variable_list
        # u can accept anything and p is expecting only integers
        # if there is variable in the formula
        if node_type(formula_lhs.name) in {"u_", "p_"}:
            if formula_lhs.name in variable_list.keys(): # check if that variable has previously appeared or not
                return str_form(variable_list[formula_lhs.name]) == str_form(equation) # if yes, then the contents should be same
            else: # otherwise, extract the data from the given equation
                if node_type(formula_lhs.name) == "p_" and "v_" in str_form(equation): # if formula has a p type variable, it only accepts integers
                    return False
                variable_list[formula_lhs.name] = copy.deepcopy(equation)
                return True
        if equation.name != formula_lhs.name or len(equation.children) != len(formula_lhs.children): # the formula structure should match with given equation
            return False
        for i in range(len(equation.children)): # go through every children and explore the whole formula / equation
            if does_given_equation_satisfy_forumla_lhs_structure(equation.children[i], formula_lhs.children[i]) is False:
                return False
        return True
    # transform the equation as a whole aka perform the transformation operation on the entire thing and not only on a certain part of the equation
    def formula_apply_root(formula):
        nonlocal variable_list
        if formula.name in variable_list.keys():
            return variable_list[formula.name] # fill the extracted data on the formula rhs structure
        data_to_return = TreeNode(formula.name, None) # produce nodes for the new transformed equation
        for child in formula.children:
            data_to_return.children.append(formula_apply_root(copy.deepcopy(child))) # slowly build the transformed equation
        return data_to_return
    count_target_node = 1
    # try applying formula on various parts of the equation
    def formula_apply_various_sub_equation(equation, formula_lhs, formula_rhs, do_only_arithmetic):
        nonlocal variable_list
        nonlocal count_target_node
        data_to_return = TreeNode(equation.name, children=[])
        variable_list = {}
        if do_only_arithmetic == False:
            if does_given_equation_satisfy_forumla_lhs_structure(equation, copy.deepcopy(formula_lhs)) is True: # if formula lhs structure is satisfied by the equation given
                count_target_node -= 1
                if count_target_node == 0: # and its the location we want to do the transformation on
                    return formula_apply_root(copy.deepcopy(formula_rhs)) # transform
        else: # perform arithmetic
            if len(equation.children) == 2 and all(node_type(item.name) == "d_" for item in equation.children): # if only numbers
                x = []
                for item in equation.children:
                    x.append(int(item.name[2:])) # convert string into a number
                if equation.name == "f_add":
                    count_target_node -= 1
                    if count_target_node == 0: # if its the location we want to perform arithmetic on
                        return TreeNode("d_" + str(sum(x))) # add all
                elif equation.name == "f_mul":
                    count_target_node -= 1
                    if count_target_node == 0:
                        p = 1
                        for item in x:
                            p *= item # multiply all
                        return TreeNode("d_" + str(p))
                elif equation.name == "f_pow" and x[1]>=2: # power should be two or a natural number more than two
                    count_target_node -= 1
                    if count_target_node == 0:
                        return TreeNode("d_"+str(int(x[0]**x[1])))
        if node_type(equation.name) in {"d_", "v_"}: # reached a leaf node
            return equation
        for child in equation.children: # slowly build the transformed equation
            data_to_return.children.append(formula_apply_various_sub_equation(copy.deepcopy(child), formula_lhs, formula_rhs, do_only_arithmetic))
        return data_to_return
    cn = 0
    # count how many locations are present in the given equation
    def count_nodes(equation):
        nonlocal cn
        cn += 1
        for child in equation.children:
                count_nodes(child)
    transformed_equation_list = []
    count_nodes(equation)
    for i in range(1, cn + 1): # iterate over all location in the
        equation tree
        count_target_node = i
        orig_len = len(transformed_equation_list)
        tmp = formula_apply_various_sub_equation(equation, formula_lhs,
             formula_rhs, do_only_arithmetic)
        if str_form(tmp) != str_form(equation): # don't produce
            duplication, or don't if nothing changed because of
            transformation impossbility in that location
            transformed_equation_list.append(tmp) # add this
                transformation to our list
    return transformed_equation_list

# Function to read formula file
def return_formula_file(file_name):
    content = None
    with open(file_name, 'r') as file:
        content = file.read()
    x = content.split("\n\n")
    input_f = [x[i] for i in range(0, len(x), 2)] # alternative formula
         lhs and then formula rhs
    output_f = [x[i] for i in range(1, len(x), 2)]
    input_f = [tree_form(item) for item in input_f] # convert into tree
         form
    output_f = [tree_form(item) for item in output_f]
    return [input_f, output_f] # return

# Function to generate neighbor equations
def generate_transformation(equation):
    input_f, output_f = return_formula_file("formula_list.txt") # load
        formula file
    transformed_equation_list = []
    transformed_equation_list +=
        apply_individual_formula_on_given_equation(tree_form(equation),
         None, None, True) # perform arithmetic
    for i in range(len(input_f)): # go through all formulas and collect
         if they can possibly transform
        transformed_equation_list +=
            apply_individual_formula_on_given_equation(tree_form(
            equation), copy.deepcopy(input_f[i]), copy.deepcopy(
            output_f[i]))
    return list(set(transformed_equation_list)) # set list to remove
        duplications

# Function to recursively transform equation
def search(equation, depth):
    if depth == 0: # limit the search
        return None
    output = generate_transformation(equation) # generate equals to the
         asked one
    for i in range(len(output)):
        result = search(str_form(output[i]), depth-1) # recursively
            find even more equals
        if result is not None:
            output += result # hoard them
    return output

# Generate all possible equations in mathematics !!!
```

```python
# Depth is how much complex equation we allow. It can be made as complicated as desired.
def fx_nest(terminal, fx, depth):
    def neighboring_math_equation(curr_tree, depth=depth): # Generate neighbouring equation trees
        def is_terminal(name):
            return not (name in fx.keys()) # Operations are not leaf nodes
        element = None # What to a append to create something new
        def append_at_last(curr_node, depth): # Append something to generate new equation
            if (is_terminal(element) and depth == 0) or (not is_terminal(element) and depth == 1): # The leaf nodes can't be operations
                return None
            if not is_terminal(curr_node.name):
                if len(curr_node.children) < fx[curr_node.name]: # An operation can take only a mentioned number of arugments
                    curr_node.children.append(TreeNode(element))
                    return curr_node
                for i in range(len(curr_node.children)):
                    output = append_at_last(copy.deepcopy(curr_node.children[i]), depth - 1)
                    if output is not None: # Check if the sub tree has already filled with arugments
                        curr_node.children[i] = copy.deepcopy(output)
                        return curr_node
            return None
        new_math_equation_list = []
        for item in terminal + list(fx.keys()): # Create new math equations with given elements
            element = item # set the element we want to use to create new math equation
            tmp = copy.deepcopy(curr_tree)
            result = append_at_last(tmp, depth)
            if result is not None:
                new_math_equation_list.append(result)
        return new_math_equation_list
    all_possibility = []
    # explore mathematics itself with given elements
    # breadth first search, a widely used algorithm
    def bfs(start_node):
        nonlocal all_possibility
        queue = deque()
        visited = set()
        queue.append(start_node)
        while queue:
            current_node = queue.popleft()
            if current_node not in visited:
                visited.add(current_node)
                neighbors = neighboring_math_equation(current_node)
                if neighbors == []:
                    all_possibility.append(str_form(current_node))
                    all_possibility = list(set(all_possibility)) # remove duplicates
                for neighbor in neighbors:
                    if neighbor not in visited:
                        queue.append(neighbor)
    for item in fx.keys(): # use all the elements
        bfs(TreeNode(item))
    return all_possibility # return mathematical equations produce
```

```

# break a equation into parts
def break_equation(equation):
    sub_equation_list = [equation]
    equation = tree_form(equation)
    for child in equation.children: # breaking equation by accessing children
        sub_equation_list += break_equation(str_form(child)) # collect broken equations
    return sub_equation_list

# spot mathematical equations which are poorly formed
def spot_invalid_equation(equation):
    equation = tree_form(equation)
    if equation.name == "f_pow": # power should only have integer on the exponent and it should be two or more than two
        return equation.children[1].name[:2] == "d_" and int(equation.children[1].name[2:]) >= 2
    return True

# fancy print
def print_equation_helper(equation_tree):
    if equation_tree.children == []:
        return equation_tree.name # leaf node
    s = "(" # bracket
    sign = {"f_add": "+", "f_mul": "*", "f_pow": "^"} # operation symbols
    for child in equation_tree.children:
        s+= print_equation_helper(child) + sign[equation_tree.name]
    s = s[:-1] + ")"
    return s

# fancy print main function
def print_equation(eq):
    eq = eq.replace("v_0", "x")
    eq = eq.replace("v_1", "y")
    eq = eq.replace("v_2", "z")
    eq = eq.replace("d_", "")
    return print_equation_helper(tree_form(eq))

# integers start with d and variables start with v
element_list = ["d_" + str(i) for i in range(1, 3)] + ["v_" + str(i)
    for i in range(0, 1)] # allowed integers and variable in our
    mathematics

formed_math = fx_nest(element_list, {"f_add": 2, "f_mul": 2, "f_pow":
    2}, 2) # scoop out a part of mathematics

formed_math = [equation for equation in formed_math if all(
    spot_invalid_equation(item) for item in break_equation(equation))]
    + element_list # remove poorly form math

equal_category = [[item] for item in formed_math] # categories of equal
     equations

# iterate through all possible equations and categorize equal ones
for equation in formed_math:
    output_list = search(equation, 1) # generate equal ones
    for output in output_list: # check if they are in present in some equality category
        output = str_form(output)
        output_loc = -1
```

```python
        equation_loc = -1
        for j in range(len(equal_category)):
            if equation in equal_category[j]:
                equation_loc = j
            if output in equal_category[j]:
                output_loc = j
        if equation_loc != -1 and output_loc != -1 and equation_loc != output_loc: # if found two categories with atleast one equation in common
            equal_category.append(equal_category[output_loc]+equal_category[equation_loc]) # merge the two categories
            equal_category.pop(max(output_loc, equation_loc))
            equal_category.pop(min(output_loc, equation_loc))

# print all the equal equation categories
for item in equal_category:
    cat = list(set([print_equation(sub_item) for sub_item in item])) # remove duplicate fancy prints
    for sub_item in cat:
        print(sub_item)
    print("----------")
```

6.1 formula_list.txt

```
f_add
 u_0
 u_1

f_add
 u_1
 u_0

f_add
 u_0
 f_add
  u_1
  u_2

f_add
 u_1
 f_add
  u_0
  u_2

f_add
 u_0
 d_0

u_0

f_mul
 u_0
 u_1

f_mul
 u_1
 u_0

f_mul
 u_0
 f_mul
  u_1
  u_2

f_mul
 u_1
 f_mul
  u_0
  u_2

f_mul
 u_0
 d_0

d_0

f_mul
 u_0
 d_1

u_0

f_mul
 u_0
 f_add
  u_1
  u_2

f_add
 f_mul
  u_0
  u_1
 f_mul
  u_0
  u_2

f_pow
 u_0
 p_0

f_mul
 u_0
 f_pow
  u_0
  f_add
   p_0
   d_-1

f_pow
 u_0
 d_1

u_0

f_pow
 u_0
 d_2

f_mul
 u_0
 u_0
```

6.2 Output of the Program: Last 250 Lines

```
...
((2^2)+2)
((1+2)*2)
((2*2)+2)
((1*2)*(1+2))
((1*2)+(2*2))
((2*2)+(2*1))
((2*2)+(1+1))
((1+1)+(2*2))
((2*1)*(2+1))
(2+(2*2))
((2*1)*(1+2))
((1*2)+(2^2))
((1+1)+(2^2))
((2^2)+(2*1))
((2*2)+(1*2))
((2^2)+(1+1))
((2*1)+(2^2))
((1*2)*(2+1))
((1+1)*(1+2))
((2^2)+(1*2))
((1+2)*(1*2))
----------
((x*1)*(2+1))
((1*x)+(2*x))
(x*(2+1))
((2+1)*(1*x))
((1+2)*x)
((2*x)+(1*x))
((1+2)*(1*x))
(x+(2*x))
((x*2)+x)
((1*x)*(2+1))
((1*x)+(x*2))
((2*x)+x)
((1+2)*(x*1))
((2+1)*(x*1))
((x*1)+(2*x))
((2*x)+(x*1))
(x+(x*2))
((1*x)*(1+2))
((x*1)+(x*2))
(x*(1+2))
((x*1)*(1+2))
((x*2)+(1*x))
((2+1)*x)
((x*2)+(x*1))
----------
((1^2)+1)
(1*2)
(1+1)
((1*1)*(1*2))
((1^2)*2)
((2*1)*1)
((1*2)*(1^2))
(2*(1*1))
((1*1)+(1*1))
((1*2)*1)
((1*2)*(1*1))
((1+1)*(1^2))
((1^2)*(1+1))
((1^2)*(1*2))
(2*1)
((1^2)+(1*1))
(1+(1*1))
(2*(1^2))
(1*(1*2))
((2*1)*(1*1))
(1+(1^2))
((1*1)+1)
((1*1)*2)
((1*1)*(2*1))
2
(1*(2*1))
((1^2)*(2*1))
((2*1)*(1^2))
((1+1)*1)
((1+1)*(1*1))
((1*1)+(1^2))
((1^2)+(1^2))
((1*1)*(1+1))
(1*(1+1))
----------
1
((1*1)*(1*1))
((1^2)^2)
((1^2)*(1*1))
((1^2)*(1^2))
((1*1)^2)
((1*1)*(1^2))
(1*1)
(1^2)
(1*(1^2))
((1^2)*1)
(1*(1*1))
((1*1)*1)
```

```
----------
((1+1)*(2+2))
((1*2)*(2+2))
((2+2)*(1+1))
((2+2)*(2*1))
((2^2)+(2^2))
((2*2)+(2*2))
((2*2)+(2^2))
((2+2)*(1*2))
(2*(2+2))
((2^2)+(2*2))
((2*1)*(2+2))
((2+2)*2)
----------
((1*1)*(x*1))
((x*1)*1)
((1^2)*x)
((1*x)*(1^2))
((1^2)*(1*x))
(x*1)
(1*x)
(x*(1*1))
(x*(1^2))
((x*1)*(1^2))
x
(1*(1*x))
((1*1)*(1*x))
((1*1)*x)
((1^2)*(x*1))
((x*1)*(1*1))
((1*x)*(1*1))
(1*(x*1))
((1*x)*1)
----------
((x+x)*(x+x))
((x+x)^2)
----------
((x+x)+(1^2))
((x+x)+(1*1))
((1*x)+(x+1))
((1+x)+(1*x))
((x*1)+(1+x))
((x+1)+(1*x))
((1*1)+(x+x))
(x+(x+1))
(1+(x+x))
((1+x)+(x*1))
(x+(1+x))
((x+1)+x)
((x+1)+(x*1))
((x+x)+1)
((1+x)+x)
((x*1)+(x+1))
((1*x)+(1+x))
((1^2)+(x+x))
----------
((2*2)*(2*2))
((2^2)^2)
((2*2)^2)
((2*2)*(2^2))
((2^2)*(2*2))
((2^2)*(2^2))
----------
(1*(x*x))
((1^2)*(x^2))
((x^2)*(1^2))
((1*1)*(x^2))
(x*x)
((x^2)*1)
((1*x)^2)
((x*x)*(1^2))
((x*1)*x)
((1*x)*(1*x))
(x^2)
((x*1)*(1*x))
((x*x)*(1*1))
(x*(x*1))
((1*x)*x)
((x*1)^2)
((x*1)*(x*1))
((x^2)*(1*1))
(1*(x^2))
(x*(1*x))
((1*x)*(x*1))
((x*x)*1)
((1*1)*(x*x))
((1^2)*(x*x))
----------
((1+1)*(2*1))
(2+(1+1))
((1*2)+(1*2))
((1+1)+2)
((1+2)+(1^2))
((1+1)+(1+1))
((1*2)^2)
((2+2)*1)
((2*1)*(1*2))
((2^2)*1)
(1+(1+2))
((1^2)*(2+2))
```

```
(1*(2^2))
((2*1)*2)
((2^2)*(1*1))
((2*1)^2)
((2+2)*(1^2))
(2+(2*1))
((1+1)*(1*2))
((1*2)*2)
((2*1)+(1*2))
((2*1)+(1+1))
((1*1)+(2+1))
((2*1)+2)
(2+2)
(1*(2+2))
((1+1)*2)
((1*2)+2)
((2+1)+(1^2))
((2+2)*(1*1))
((2^2)*(1^2))
((2*1)*(2*1))
((1*2)*(1+1))
((2+1)+(1*1))
(2^2)
((1^2)*(2*2))
((1*2)*(2*1))
((1+1)^2)
((1*2)+(1+1))
((1*1)*(2*2))
(1+(2+1))
((1*1)*(2+2))
((2*2)*1)
((1^2)+(1+2))
((1+2)+1)
((1*2)*(1*2))
((1*1)+(1+2))
(2*(2*1))
((1+1)*(1+1))
(2*(1+1))
((1+1)+(1*2))
((1*2)+(2*1))
(2*(1*2))
((2*2)*(1*1))
((1^2)*(2^2))
((1*1)*(2^2))
((2*1)+(2*1))
(2+(1*2))
(1*(2*2))
((2*1)*(1+1))
(2*2)
((2*2)*(1^2))
((1+1)+(2*1))
((1^2)+(2+1))
((1+2)+(1*1))
((2+1)+1)
----------
```