CSCE: 625 Introduction to Artificial Intelligence

Programming Assignment 1: Simplifying Mathematical Expressions via Search

Last Name: **Singh**
First Name: **Jasmeet**
UIN: **523005618**

"An Aggie doesn't lie, cheat or steal or tolerate those who do"

1 Task Environment

We have to design and agent which solves a mathematical expression for a given variable. The task environment for this problem can be defined by four parameters:

1) **Performance Measure**: Get the solution for the variable, Get the most simplified solution
2) **Environment**: Given equation, the variable to solve for
3) **Sensors**: Console Input, Keyboard
4) **Actuators**: Console Output, Computer Screen

The Environment has the following characteristics:

1) **Partially Observable**: The environment is partially observable as from the given state of equation we only know the next state which can be achieved by applying a mathematical identity. We have no information on how the goal state will look like
2) **Deterministic**: The next state is completely determined by the current state and the action we perform.
3) **Episodic**: The agent acts on each set of equation separately and once it has solved the equation, it starts afresh.
4) **Single Agent**: We have only one agent
5) **Semi Dynamic**: We will be running the agent with a clock, so if it is not able to look for an answer it should give the result for the best it could do.
6) **Discrete**: From a given equation there are only a finite set of states to which the current equation can be transformed using rules of mathematics.

2 State Space

I design the state space to include the following 4 parameters:

1) Current state of the equation: This is saved in the form of an expression tree which is returned by the PLY parser.
2) Distance: The distance the current state has travelled from the initial equation. The distance is measured in terms of number of transformations applied. Each atomic transformation counts for 1 in the distance.
3) Expected Distance from the Solution: This is described by a heuristic function, which tells the least possible distance between the current state and the goal state.

3 Goal Formulation

The goal state will be when the variable (sayx) we are solving for is on one side of the equation and the other side does not include x. For example

Input Equation: $x + y = 3$
Variable: x
Goal Equation: $x = 3 - y$

However, with this formulation there could be multiple goal states. For example

Input Equation: $x + y = 2 * 3$
Variable: x
Goal Equation1:$x = (2 * 3) - y$
or
Goal Equation2:$x = 6 - y$

Both of the above goal states are possible answers, but the second equation is much more simplified, so it is the desired output we would want.

4 Search Algorithm

We can use an informed search algorithm to reach the solution. The information we can supply is how far the current state is from the goal state. But, as there are multiple possible goals, I am dividing the problem into 2 parts:

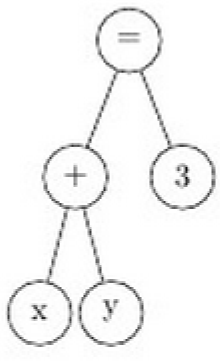
1) Reaching a correct solution
2) Reaching a simplified solution

The rationale behind the dividing the problem into 2 parts is to handle the situations where multiple goal states exist. Like we discussed in the example above both the equations are possible answers. So the first part of the solution works to get the correct solution and the second part can work on simplifying it. More reasoning on dividing the problem in 2 parts is given in the Heuristic section.

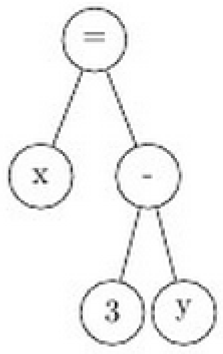
5 Searching a Correct Solution

I have used A* search to search for the possible solution for a given equation and a variable. The search problem is defined as follows:

1) **States**: A state is described in form of an expression tree. I also include the distance from the initial state and the expected distance (heuristic function) in the state itself. For example, the equation x+y = 3 will be represented in the following way.



2) **Initial State:** Initial state is the expression tree for the given equation, the distance = 0, and the expected distance to goal state calculated by the heuristic function
3) **Actions**: Actions are to apply one of the possible mathematical transformation from the set of mathematical identities which can be applied to the current state. For example, Commutative Rule, Distributive Rule, Integration by parts.
4) **Transition Model**: Transition model describe how the tree is modified when an identity is applied. For example, if we take the y from the LHS to the RHS, the new tree will look like the image shown below. The transition model describes how to modify the tree to carry out a transformation.

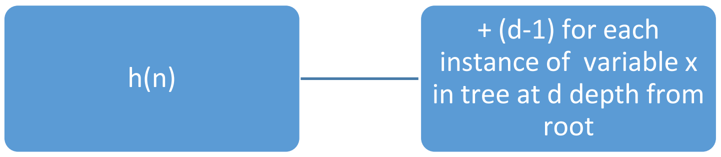


5) **Goal Test**: When the variable we want to solve for is on one side of the equation and all other terms on the other side, we assume we have reached the goal state. The goal test for simplification part will be different.
6) **Path Cost**: We define only atomic transformations, so the path cost for each transformation will be assumed to be 1.

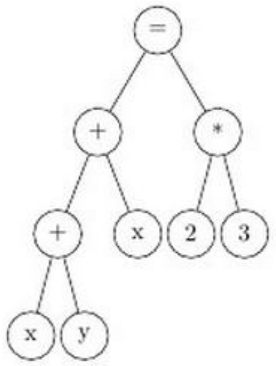
6 Algorithm

To describe the algorithm for searching for the correct solution, we first look into the possible heuristic functions which give us the estimate of the distance from goal state. The heuristic function which I have used is:

h(n) = For each occurrence of variable (say x) in the expression tree add (d-1) to the h(n) where d is the depth of the variable x from the root node.


h(n)
+ (d-1) for each instance of variable x in tree at d depth from root

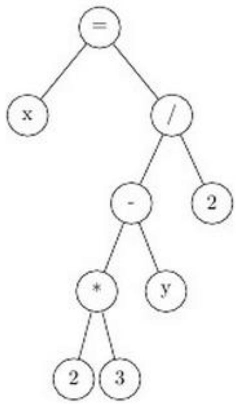
So the value of this heuristic function for a tree which looks like this will be 5



$$h(n) = (2-1) + (3-1) = 3$$

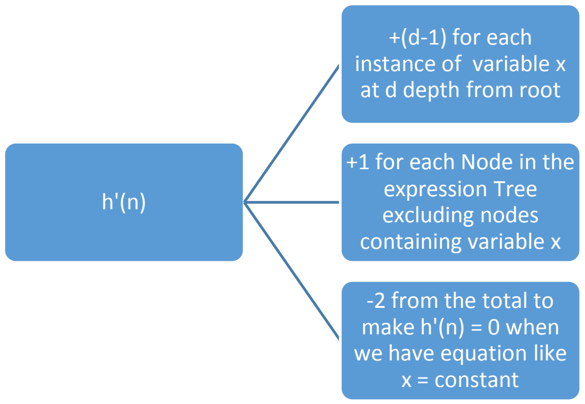
The idea behind the heuristic is that if our target variable (say x) is at a depth of 2 from the root node, it will take at least one transformation to bring it to the depth 1. Similarly if x is at depth 5 from the root node, it will take at least 4 transformations to bring it to the depth of 1. When the state reaches the goal state x will be at depth of 1 and there will be no other instance of x in the tree. So the h(n) will compute to 0. This ensures admissibility of the heuristic because $h(n) <= c(n)$ for every state n, where c(n) is the cost or distance from the initial state. (Distance is measured in form number of transformations applied).

So for the tree above, when the equation reached this state, h(n) will be equal to 0.



Though this is a correct solution, but it is not simplified. If we use the above heuristic, the A* search algorithm terminates here as it has reached the goal state.

Another possible heuristic function can be in addition to the (d-1) added to h(n) for each instance of x, we also add 1for every node in tree and subtract 2 from the total.


h'(n)
+(d-1) for each instance of variable x at d depth from root
+1 for each Node in the expression Tree excluding nodes containing variable x
-2 from the total to make h'(n) = 0 when we have equation like x = constant

For the tree example in previous heuristic, the value of h'(n) in the initial state will be (3 + 7 – 2) = 8 in the initial state and (0 + 8 – 2) = 6 in the final state. In this case h'(n) can be further reduced to 4 when two and three are multiplied together to a single node 6.

The goal state in this case is assumed to be $x = constant$, in which case h'(n) will be (0 + 2 -2) = 0.

This heuristic takes care of some simplification since it tells that in a goal state all the variables (excluding x) , all the constants and all the operators will be quashed to one nodein at least one step each. However this heuristic is not admissible, for example:

x = (a+b+c+2+4+a)^0

The minimum cost for such simplification is 1, as anything raised to power 0 will be 1. However h'(n) will predict that it will take (0 + 13 –2) = 11 steps to reach a simplification, which is clearly greater than the actual cost. So this heuristic is not admissible.

We can use a Greedy Best first search algorithm also, but such algorithm will keep on trying to simplify the solution even if it cannot be simplified more. Though it can be terminated if it exceeds certain time limit.However for problems where no further simplification is possible the A* search solution is the best solution given the goal state definition. This is the main reason I split the process into two steps of reaching a solution and simplifying it.

So I use standard A* search algorithm for the first step, i.e. reaching a solution, and then a slightly greedy version of A* search where the goal is defined as reaching (x = constant) and the second heuristic h'(n) is used. Though the goal may be unreachable in many cases where the algorithm terminates the simplification if it exceeds certain pre-decided time limit. Since the heuristic h'(n) is not admissible, the simplification step may not give the shortest path to the goal, but it will always give the result which is closest to the goal (being closest to the goal is the aim of simplification)

7 Algorithm for step 1 (A* Search):

A recursive algorithm to traverse the expression tree and return the heuristic function value has been implemented.

```
Enqueue(queue, 0, h(root), root)
// Enqueue the distance from root, its expected distance from
// goal and root node to priority queue. Nodes in priority queue
// are sorted in increasing order of ( distance + h(n) )

closestSolution = (0, h(root), root)
timeLimit = 1000
A_Star_Search(variable):
    Time = 0
    While queue is not empty AND time< timeLimit:
        curState = Dequeue(queue)
        distance = curState.distance
        Apply all possible transformations and get a newState for each:
            If h(newState) == 0:
                // Goal Test: h(state) == 0 means whether the variable
                // has reached the minimum depth and has one
                // instance only
                correctSolution = newState
                break
            if newState is not already visited:
                Enqueue(queue, distance+1, h(newState), newState.root)
                If h(newState) < h(closestSolution):
                    // I also maintain a closestSolution in case no goal
                    // is reached within the time
                    closestSolution = newState
            Time += 1

    if correctSolution == None:
        return closestSolution
    else:
        return correctSolution

Heuristic function h(n) is defined as:
h(State):
    h+=0
    for every Node in the expression tree:
        if Node.Value == variable
            h+=d-1          // Where d is the depth of the current Node
    return h
```

8 Algorithm for step 2 (Greedy version of A*): The only difference is in the goal test and the definition of the heuristic function h’(n) in this case. We have the solution for A* algorithm in first step in variable answer. While programming I figured out that diff, integrate, sin, cos and other complex functions need to be given higher weights in order for algorithm to simplify quickly. In my program I have given weight of 10 to every integration and differentiation node and weight of 4 to every sin and cos node.

```
Enqueue(queue, 0, h’(answer), answer.root)
// Enqueue result of Step 1 in an empty queue
// Nodes in priority queue // are sorted in increasing
// order of ( distance + h’(n) )

closestSolution = answer
timeLimit = 1000
GREEDY_A_Star_Search(variable):
    Time = 0
    While queue is not empty AND time< timeLimit:
        curState = Dequeue(queue)
        distance = curState.distance
        Apply all possible transformations and get a newState for each:
            If h’(newState) == 0:
                // Goal Test: h’(state) == 0 means whether the
                // equation has reached the expected best solution
                // of ( variable = Constant) in which case h’(n) == 0
                correctSolution = newState
                break
            if newState is not already visited:
                Enqueue(queue, distance+1, h’(newState), newState.root)
                If h’(newState) < h’(closestSolution):
                    // I also maintain a closestSolution in case no goal
                    // is reached within the timeLimit
                    closestSolution = newState
            Time += 1
    if correctSolution == None:
        return closestSolution
    else:
        return correctSolution

Heuristic function h'(n) is defined as:
h’(State):
    h+=0
    for every Node in the expression tree:
        if Node.Value == variable
            h+=d-1  // Where d is the depth of the current Node
        else:
            h+=1
    h-=2
    return h
```

9 Syntax Grammar Changes Made in PLY

The following syntax grammar changes have been done:

1) Unary Minus has been transformed to (0 – expression)
   So –x is parsed as (0 – x)
2) New Type BINARYFUNCTION has been added to support differentiation and integration. The syntax for differentiation and integration is like this
3) diff (A(x) , x) is stored as BINARYFUNCTION Node 'diff' with first children as A(x) and second children as x.
4) integrate(A(x) , x) is also stored as BINARYFUNCTION Node 'integrate' with first children as A(x) and second children as x.

10 Actions developed

1) Inverse operator

Sr. No.	**Initial State**	**Next State**
1	A + B = C	A = C – B
2	A – B = C	A = C + B
3	A * B = C	A = C / B
4	A / B = C	A = C * B
5	Sqrt(A) = B	A = B ^ 2
6	A ^ B = C	A = C ^ (1/B)
7	A = e ^ B	Ln(A) = B
8	A = 2 ^ B	Log(A) = B

2) Arithmetic

Sr. No.	**Initial State**	**Next State**
1	a+b, a-b, a*b, a/b	For integers and floating points get the value
2	A + 0	A
3	A – 0	A
4	A * 0	0
5	A * 1	A
6	0 / A	0
7	A / 1	A
8	A / 0	Undefined
9	A ^ 0	1
10	A ^ 1	A
11	1 ^ A	1
12	0 ^ A	0
13	Ln(1)	0
14	Ln(e)	1
15	Log(1)	0
16	Log(2)	1

3) Commutative

Sr. No.	Initial State	Next State
1	A + B	B + A
2	A * B	B * A

4) Normal Associative

Sr. No.	Initial State	Next State
1	A + (B + C)	(A + B) + C
2	A *(B * C)	(A * B) * C

5) Complex Associative (Addition – Subtraction)

Sr. No.	Initial State	Next State
1	A + (B – C)	(A + B) – C
2	A – (B + C)	(A – B) + C
3	A – (B – C)	(A – B) + C
4	(A + B) + C	A + (B + C)
5	(A – B) + C	A – (B – C)
6	(A + B) – C	A + (B – C)
7	(A – B) – C	A – (B + C)

6) Complex Associative (Multiply – Divide)

Sr. No.	Initial State	Next State
1	A * (B / C)	(A * B) / C
2	A / (B * C)	(A / B) * C
3	A / (B / C)	(A / B) * C
4	(A * B) * C	A * (B * C)
5	(A / B) * C	A / (B / C)
6	(A * B) /C	A * (B / C)
7	(A / B) /C	A / (B * C)

7) Distributive

Sr. No.	Initial State	Next State
1	(A + B) * C	(A * C) + (B * C)
2	(A – B) * C	(A * C) – (B * C)
3	(A + B) / C	(A / C) + (B / C)
4	(A – B) / C	(A / C) – (B / C)
5	(A*a) + or - (A*b)	((1*a) + or –(1*b)) * A
6	(A*a) + or - (A/b)	((1*a) + or – (1/b)) * A
7	(A/a) + or - (A/b)	((1/a) + or – (1/b)) * A
8	(a/A) + or – (b/A)	(1/A) * ((a/1) + or – (b/1))

8) Trigonometry

Sr. No.	**Initial State**	**Next State**
1	Sin(A)^2 + Cos(A)^2	1
2	Cos(A)^2 + Sin(A)^2	1

9) Differentiation

Sr. No.	**Initial State**	**Next State**
1	diff(A + B , x)	Diff(A , x) + Diff(B , x)
2	diff(A - B , x)	Diff(A , x) – Diff(B , x)
3	Diff(x , x)	1
4	Diff(B , x) {B is a variable or constant}	0
5	Diff(A^n, x)	(n * (A ^ (n – 1)) * Diff(A , x)
6	Diff(C*A, x) { C is a constant}	C * Diff(A, x)
7	**Diff(A * B, x)**	**(A * Diff(B)) + (Diff(A) * B)**
8	Diff(Sin(A) , x)	Cos(A) * Diff(A , x)
9	Diff(Cos(A), x)	0 – (Sin(A) * Diff(A , x))

10) Integration

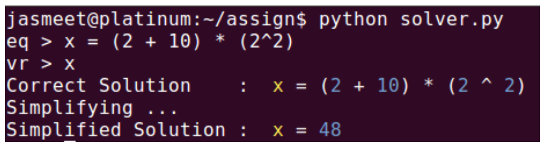
Sr. No.	**Initial State**	**Next State**
1	Integrate(A + B, x)	Integrate(A , x) + Integrate (B , x)
2	Integrate(A - B, x)	Integrate(A , x) - Integrate (B , x)
3	Integrate(x , x)	(x ^ 2) / 2
4	Integrate(x^n , x)	(x ^ (n+1)) / (n+1)
5	**Integrate(A * B , x)**	**A * Integrate(B,x) – Integrate(Diff(A,x) * Integrate(B,x))**

11 How to run

The how to run part of the section is very easy, because its python.

1) Make sure you are running a UNIX machine, otherwise the printing of expressions doesn't work correctly due to unknown reasons.
2) Go to the code directory and type the command "python solver.py" in a terminal window.
3) You will be asked to input an equation and then a variable
4) Just make sure you enter valid syntax, or the program will fail before searching anything.
5) The program will give 2 outputs, first a correct solution, but not simplified. Next its simplified version. **The Simplify stage will take 5 - 10 seconds, so try to be patient.**

Here is how it will look like:



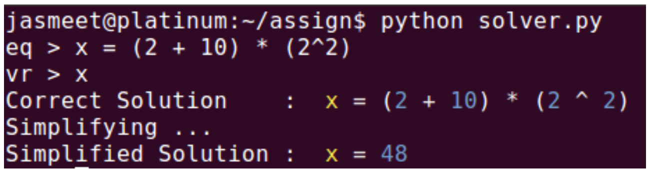
```
jasmeet@platinum:~/assign$ python solver.py
eq > x = (2 + 10) * (2^2)
vr > x
Correct Solution    :  x = (2 + 10) * (2 ^ 2)
Simplifying ...
Simplified Solution :  x = 48
```

12 Sample Test Cases

Arithmetic Evaluations

1) $x = (2 + 10) * (2^2)$

Solve for variable x

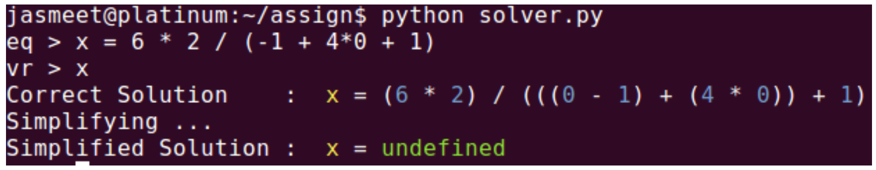


```
jasmeet@platinum:~/assign$ python solver.py
eq > x = (2 + 10) * (2^2)
vr > x
Correct Solution    :  x = (2 + 10) * (2 ^ 2)
Simplifying ...
Simplified Solution :  x = 48
```

The interesting part is that the initial state itself is the answer of the A* search as the variable x is in depth 1 and there is no other instance of x.

2) $x = (6 * 2)/(-1 + 4 * 0 + 1)$

Solve for variable x



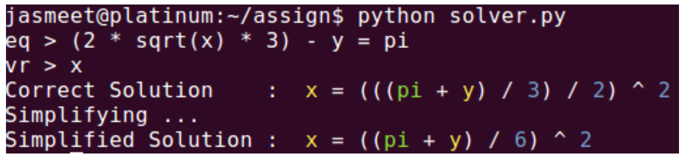
```
jasmeet@platinum:~/assign$ python solver.py
eq > x = 6 * 2 / (-1 + 4*0 + 1)
vr > x
Correct Solution    :  x = (6 * 2) / (((0 - 1) + (4 * 0)) + 1)
Simplifying ...
Simplified Solution :  x = undefined
```

The denominator of the division becomes 0 while simplification, hence the output is undefined.

Applying Inverse

3) $(2 * sqrt(x) * 3) - y = pi$

Solve for variable x



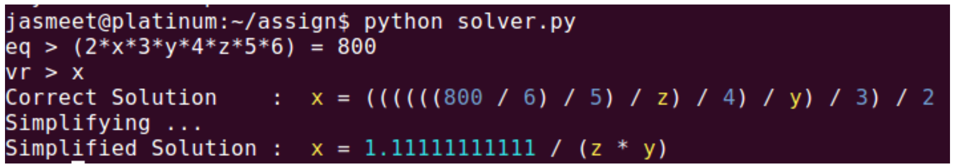
```
jasmeet@platinum:~/assign$ python solver.py
eq > (2 * sqrt(x) * 3) - y = pi
vr > x
Correct Solution    :  x = (((pi + y) / 3) / 2) ^ 2
Simplifying ...
Simplified Solution :  x = ((pi + y) / 6) ^ 2
```

This example clearly shows how my program works. First step finds a correct solution and second step simplifies it. This also shows how inverse of sqrt is applied by giving square of the RHS

Associative and Commutative

4) $(2 * x * 3 * y * 4 * z * 5 * 6) = 800$

Solve for variable x



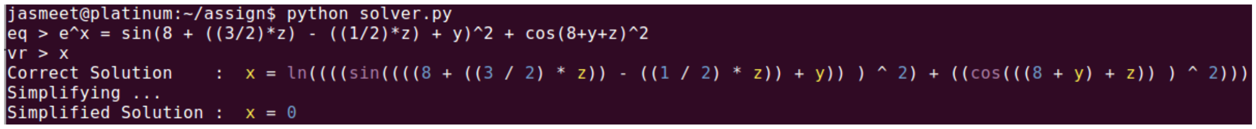
```
jasmeet@platinum:~/assign$ python solver.py
eq > (2*x*3*y*4*z*5*6) = 800
vr > x
Correct Solution    :  x = ((((((800 / 6) / 5) / z) / 4) / y) / 3) / 2
Simplifying ...
Simplified Solution :  x = 1.11111111111 / (z * y)
```

The divisions are not shown as fractions. They are converted to floats.

Identities

5) $e^x = sin(8 + ((3/2) * z) - ((1/2) * z) + y)^2 + cos(8 + y + z)^2$

Solve for variable x



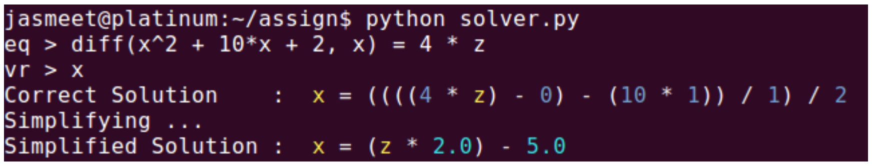
```
jasmeet@platinum:~/assign$ python solver.py
eq > e^x = sin(8 + ((3/2)*z) - ((1/2)*z) + y)^2 + cos(8+y+z)^2
vr > x
Correct Solution    :  x = ln((((sin((((8 + ((3 / 2) * z)) - ((1 / 2) * z)) + y)) ) ^ 2) + ((cos(((8 + y) + z)) ) ^ 2)))
Simplifying ...
Simplified Solution :  x = 0
```

This is a very complex equation. The simplify stage is able to solve but takes a lot of time.

Calculus

6) $diff(x^2 + 10 * x + 2, x) = 4 * z$

Solve for variable x

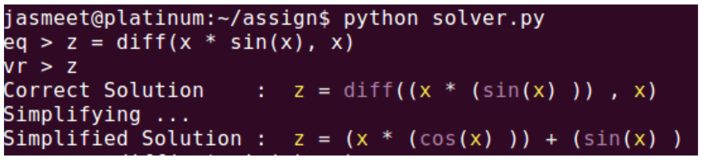


```
jasmeet@platinum:~/assign$ python solver.py
eq > diff(x^2 + 10*x + 2, x) = 4 * z
vr > x
Correct Solution    :  x = ((((4 * z) - 0) - (10 * 1)) / 1) / 2
Simplifying ...
Simplified Solution :  x = (z * 2.0) - 5.0
```

This example shows how diff operator is distributed among the additions. A more complex differentiation of multiplications is show below:

7) $z = diff(x * sin(x)), x)$

Solve for variable z

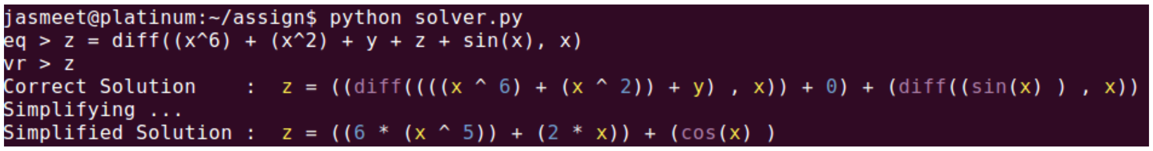


```
jasmeet@platinum:~/assign$ python solver.py
eq > z = diff(x * sin(x), x)
vr > z
Correct Solution    :  z = diff((x * (sin(x) )) , x)
Simplifying ...
Simplified Solution :  z = (x * (cos(x) )) + (sin(x) )
```

The multiplication identity of differentials (in bold in the actions table) is applied.

8) $z = diff((x^6) + (x^2) + y + z + sin(x), x)$

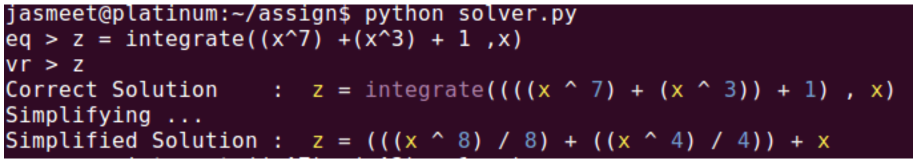
Solve for variable z



```
jasmeet@platinum:~/assign$ python solver.py
eq > z = diff((x^6) + (x^2) + y + z + sin(x), x)
vr > z
Correct Solution    :  z = ((diff((((x ^ 6) + (x ^ 2)) + y) , x)) + 0) + (diff((sin(x) ) , x))
Simplifying ...
Simplified Solution :  z = ((6 * (x ^ 5)) + (2 * x)) + (cos(x) )
```

9) $z = integrate((x^7) + (x^3) + 1, x)$

Solve for variable z



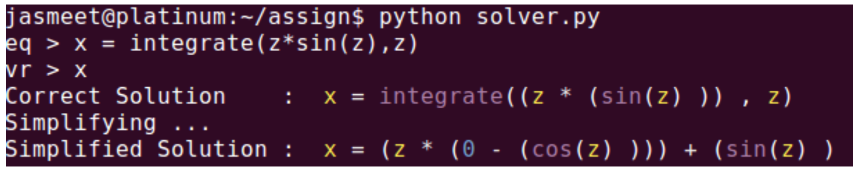
```
jasmeet@platinum:~/assign$ python solver.py
eq > z = integrate((x^7) +(x^3) + 1 ,x)
vr > z
Correct Solution    :  z = integrate((((x ^ 7) + (x ^ 3)) + 1) , x)
Simplifying ...
Simplified Solution :  z = (((x ^ 8) / 8) + ((x ^ 4) / 4)) + x
```

Another drawback integration is that my code doesn't add a constant C for every integral which is clearly seen in this example

Integration by parts

10) $x = integrate(z * sin(z), z))$

Solve for variable x



```
jasmeet@platinum:~/assign$ python solver.py
eq > x = integrate(z*sin(z),z)
vr > x
Correct Solution    :  x = integrate((z * (sin(z) )) , z)
Simplifying ...
Simplified Solution :  x = (z * (0 - (cos(z) ))) + (sin(z) )
```

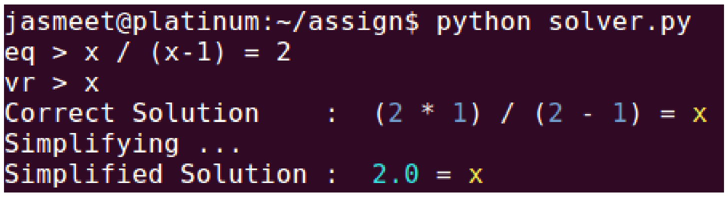
The main drawback of removing unary minus from the syntax grammar is seen here.

– Cos(z) is seen as (0 – Cos(z))

Multiple Instances of x

11) $x/(x-1) = 2$

Solve for variable x



```
jasmeet@platinum:~/assign$ python solver.py
eq > x / (x-1) = 2
vr > x
Correct Solution    :  (2 * 1) / (2 - 1) = x
Simplifying ...
Simplified Solution :  2.0 = x
```

This is a good example because it uses a lot of identities. The denominator has to be multiplied to the RHS, then the 2 on the RHS has to be distributed and the find x.

Some more examples (including these) are included in the screenshots folder.

13 References

1) Google search: https://www.google.com/?gws_rd=ssl
2) Wikipedia: http://www.wikipedia.org/
3) PLY (Python Lex-Yacc) an implementation of python lex and yacc parsing tools. Documentation and example codes http://www.dabeaz.com/ply/ply.html
4) Artificial Intelligence: A Modern Approach by Stuart Russel and Peter Norvig (Third Edition).
5) SymPy: Python library for symbolic mathematics, documentation and example codes.