Computer Science 331 – Assignment 3

Methodology:

For my program I decided to split the task into three main sections, a method which finds all sub-expressions in an expression, denoted by getSubExpressions(String). Another method denoted by evaluateExpression() that takes an expression as a parameter and calculates the truth value, and a third method denoted by getTruthTable() that takes an expression as a parameter and generates a table for it through the global 2d-array table. The getSubExpressions(String) method traverses the expression until it reaches the end. Upon finding an open bracket it will loop through the sub-expression and save it through a sub-loop and return back to index after the open bracket to check the next character. A stack is used to determine if the sub-expression has been fully traversed by counting open and closed brackets. The evaluateExpression() method similarly traverses the expression but only saves the most immediate sub-expressions to an array temp. Upon reading an operator, the algorithm will update the form string, and use it to determine the truth of the expression. The truth of the immediate sub-expressions are determined through the truthValue hash-map. The getTruthTable() method traverses the table 2d-array row by row. The main loop is divided into three sub-loops where the first one sets up the first row with the names and numbers of the expressions and variables. The second loop assigns differentiated truth values to the variables so each row is different and accounts for all possible combinations. It does this by converting binary form of the index; 0’s and 1’s into false and true strings respectively. The final loop handles the sub-Expressions by evaluating the truth of each expression based on the corresponding rows variables truth values. I also created a findExpression() method as a search algorithm to use for the case where a sub-expression already exists in the dictionary. I implemented a linear search for this portion of the task because there is no partial order in the dictionary and it seemed more efficient to stop the loop upon finding an identical element as opposed to traversing the entire dictionary.

Proof for getSubExpressions(String) method:

**While statement:**

The loop invariant for the parent while loop is that the dictionary subExpressions accurately reflects the sub-expressions of the expression, for all i.

Base Property: Before the first iteration of the loop the dictionary is empty, this is accurate since no character have been read from the expression, thus no sub-expressions have been evaluated.

Inductive property: Assume that the loop body is executed I > 0 times and that R is satisfied at the beginning of the I-th execution.

Case 1: Character.isLetter(character) & !variables.contains(character + ""): Dictionary is not changed so dictionary is still correct

Case 2: character != '(': At the end of the execution no new sub-expressions are added to the dictionary so dictionary Is still correct

Case 3: character == '(': At the end of the execution the sub-expression added to the dictionary depends on the inner while loop, so in order to prove that the dictionary is correct at the end of each execution we must prove that the inner while loop is always correct.

**Inner while loop:**

The loop invariant for the sub-loop is that all characters added to the sub-expression maintain that the sub-expression is correct for all j.

Base Property: Assume that the loop body is executed I > 0 times and that R is satisfied at the beginning of the I-th execution:

An open bracket is added to the start of sub-expression. All characters that are read henceforth are added to subexpression string, the sub-expression is correct so long as the loop terminates correctly

Proof of correct termination: The loop terminates when the counter is empty. When a ‘(‘ is read count size is increased by 1, when a ‘(‘ is read, count size is decreased by 1. Since the amount of open and closed brackets must be equivalent for any sub-expression by the end of the sub-expression the size of the array will be: amount of open – amount of closed brackets = 0. Thus the loop terminates at the end of each sub-expression.

If the sub-expression generated is already in the dictionary, the sub-expression is not added. Since the dictionary is not changed it is still correct.

Since all characters added to the sub-expression string are correct, the loop is partially correct and since it terminates the loop must be correct.

Since the inner loop is correct, all sub-expression added to the dictionary are correct thus the dictionary is correct for case 3. Since the dictionary is correct for all cases, the outer loop is partially correct.

To prove termination: The loop continues while I < pExpression.length(), since I = 0 < pExpression.length() and pExpression.length() is finite, and because I is increased through each loop I will eventually reach pExpression.length() and terminate. Thus, since the outer loop is partially correct and terminates, the algorithm is correct.

Proof for evaluateExpression() method:

**While statement:**

The loop invariant for the parent while loop is that the temp array accurately reflects the operands of the expression, for all i.

Base Property: Before the first iteration of the loop the temp array is empty, this is accurate since no character have been read from the expression, thus no operands have been evaluated.

Inductive property: Assume that the loop body is executed I > 0 times and that R is satisfied at the beginning of the I-th execution.

Case 1: character == '(' & !count.empty() if this case is true, we haven’t read the first open bracket yet so no operands have be read or added to the temp array. Thus the temp array is still correct

Case 2: Character.isLetter(character)) temp.add(character + "": If this case is executed, the operand read is a letter and it is added to the temp array. Since it is not within a sub-expression temp is accurate

Case 3: character == '+' | character == '\*' | character == '-': if this case is executed, the character read is an operator and temp is unchanged thus temp is correct.

Case 4: character == '(' & count.empty(): In this case we have entered the expression and a sub-expression is anticipated, to prove that temp is correct at the end of the loop we must prove that the inter while loop of this case is correct for all loops.

**Inner while loop:**

The loop invariant for the sub-loop is that all characters added to the sub-expression maintain that the sub-expression is correct for all j.

Base Property: Assume that the loop body is executed I > 0 times and that R is satisfied at the beginning of the I-th execution:

An open bracket is added to the start of sub-expression. All characters that are read henceforth are added to subexpression string, the sub-expression is correct so long as the loop terminates correctly

Proof of correct termination: The loop terminates when the counter is empty. When a ‘(‘ is read count size is increased by 1, when a ‘(‘ is read, count size is decreased by 1. Since the amount of open and closed brackets must be equivalent for any sub-expression by the end of the sub-expression the size of the array will be: amount of open – amount of closed brackets = 0. Thus the loop terminates at the end of each sub-expression.

Since all characters added to the sub-expression string are correct, the loop is partially correct and since it terminates the loop must be correct.

Since the inner loop is correct, all operands added to temp are correct thus the temp is correct for case 4. Since temp is correct for all cases, the outer loop is partially correct.

To prove termination: The loop continues while I < pExpression.length(), since I = 0 < pExpression.length() and pExpression.length() is finite, and because I is increased through each loop I will eventually reach pExpression.length() and terminate. Thus, since the outer loop is partially correct and terminates, the while loop is correct.

To prove that the algorithm is correct we need to show that the method returns the correct truth value.

Case 1: form.equals("-") & getTruthValue().get(temp.get(0)).equals("F")) result = "T": since the while loop is correct we know that the operands that are evaluated are correct. If the form is negation and the operand is false, the expression result is true, this is correct.

Case 2: form.equals("+") && (getTruthValue().get(temp.get(0)).equals("T")|getTruthValue().get(temp.get(1)).equals("T"))  
If the form is +, and one of the two operands are true set result to true. This is correct

Case 3: form.equals("\*") && (getTruthValue().get(temp.get(0)).equals("T") & getTruthValue().get(temp.get(1)).equals("T"))  
If the form is \*, and the two operands are true set result to true. This is correct

If none of the cases hold, the result is false, this is correct.

Thus in all cases the algorithm is correct.

Proof for getTruthTable() method:

The loop invariant for the parent while loop is that the table is correct for all executions of the while loop, i.e. all rows are correct in each iteration.

Base Property: Before the first iteration of the loop the table is empty, this is accurate since no string has been added to the table yet

Inductive property: Assume that the loop body is executed I > 0 times and that R is satisfied at the beginning of the I-th execution.

**Inner while loop 1:**

The loop invariant for the first sub while loop is that each string added to the table is correct

Base Property: Before the first iteration of the loop the table is empty, this is accurate since no string has been added to the table yet

Inductive property: Assume that the loop body is executed I > 0 times and that R is satisfied at the beginning of the I-th execution.

Case 1: m - 1 >= getVariables().size(): In this case we are iterating through the portion of the table that handles the sub-expressions. This part of the table is also the same size as the subexpression dictionary so all sub-expressions are placed in their respective index correctly.

Case 2: m - 1 < getVariables().size(): In this case we are iterating through the portion of the table that handles the variables. This part of the table is also the same size as the variables array so all variables are placed in their respective index correctly.

Since in all cases the table is correct, the loop is partially correct, in order to prove full correctness we need to prove termination. Since m = getVariables().size() + getSubExpressions().size() – 1 > -1 and m is decremented through each loop, me will eventually reach -1 and the program will terminate thus the loop Is correct.

**Inner while loop 2:**

The loop invariant for the second sub while loop is that each truth assignment for each variable is correct for each loop

Base Property: Before the first iteration of the loop the table has not been modified, this is accurate since no string has been added to the table yet

Inductive property: Assume that the loop body is executed I > 0 times and that R is satisfied at the beginning of the I-th execution.

Case 1: n - 1 < 0: In this case the full string has been traversed, since we are reading the number end to front any further characters can be interpreted as a 0 and evaluated as F. Thus this case is correct

Case 2: binary.charAt(n) == '1': If the character read is a 1 then the character added id T, this is correct

Case 3: binary.charAt(n) == '0': If the character read is a 0 then the character added id F, this is correct

In all cases the character is correct, and is correctly added to the table, so the loop is partially correct. To prove that it is full correct we must prove termination. Since m = getVariables().size() – 1 > -1 and m is decremented with each iteration m will eventually reach -1, thus the while loop terminates and therefore the loop is correct.

**Inner while loop 3:**

The loop invariant for the third sub while loop is that each truth assignment for each sub-expression is correctly placed for each loop

Base Property: Before the first iteration of the loop the table has not been modified, this is accurate since no string has been added to the table yet

Inductive property: Assume that the loop body is executed I > 0 times and that R is satisfied at the beginning of the I-th execution.

We find the corresponding sub-expression at row 0, in the same column and evaluate its truth, we then put this value in the I – 1th row at the m – 1th after the execution. This is correct for all sub-expressions

To prove full correctness we need to prove termination. Since m = getVariables().size() + getSubExpressions().size() – 1 > getVariables().size() – 1 and m is decremented through each iteration, m eventually reaches getVariables().size() – 1. Thus the loop terminates and this loop is fully correct.

All while loops are correct so the table Is correct in all cases. We need to prove termination. Since i = 0 < Math.pow(2, getVariables().size()) + 1 and I is incremented, I will eventually reach Math.pow(2, getVariables().size()) + 1 thus the outer loop terminates and is fully correct. Therefore the getTuthTable algorithm is correct.

Proof for findExpression() method:

**While loop:**

The loop invariant for the while loop is that the loop terminates correctly, in other words, if an identical element is found in the dictionary the loop should terminate.

Base Property: Before the first iteration of the loop none of the dictionary elements have been checked yet so the loop shouldn’t terminate and doesn’t, this is correct

Inductive property: Assume that the loop body is executed I > 0 times and that R is satisfied at the beginning of the I-th execution.

Case 1: getSubExpressions().get("LEO" + i).equals(pExpression)) result = "LEO" + i: In this case, an expression within the dictionary is equivalent to the expression inputted as a parameter. If this case is true the loop terminates and the number for that expression in the dictionary is assigned to return, this is correct

Case 2: If the expression inputted does not equal the expression checked at “LEO” + I in the dictionary and I < n then the loop does not terminate. This is correct

Case 3: If the expression inputted does not equal the expression checked at “LEO” + I in the dictionary and I >= n then the loop terminates. This is correct because all the elements in the dictionary have been checked

Thus, since the loop is correct in all cases, the loop is partially correct. In order to prove full correctness we need to prove termination. We know I = 0 < getSubExpressions().size() and since I is incremented with each loop I will eventually reach getSubExpressions().size() thus the loop terminates so it is fully correct.

After the while loop an if statement checks if I < n. If it is then the loop ended prematurely thus there is an identical element in the dictionary and return is set to that element. If I = n then no such element was found and the result remains none.

Thus return is correct at the end of each method call so the method is correct

Worst Case Cost:

Worst Case cost for getSubExpressions() method:

**Worst case for outer while loop:**

The test has 1 comparison plus 1 method call and is checked m + 1 times so the max is 2(m + 1) instructions executed

Worst case cost for execution of loop body (2 + 5 + 7 + 8 + worst-case-cost of inner loop)

**Worst case cost for inner loop:**

The test has 1 comparison plus 1 method call and is check n + 1 times so the max is 2(n + 1) instructions executed

Worst case cost for execution of loop body (2 + 3 + 5)

Upper bound on worst-case cost to execute the inner loop 2(n + 1) + 10n

Upper bound on worst-case cost to execute the outer loop 2(m + 1) + 22(2(n + 1) + 10n)

Upper bound on worst-case cost to execute the method 2(m + 1) + 22(2(n + 1) + 10n) + 6

Worst Case cost for evaluateExpression() method:

**Worst case for outer while loop:**

The test has 1 comparison plus 1 method call and is checked m + 1 times so the max is 2(m + 1) instructions executed

Worst case cost for execution of loop body (6 + 10 + worst-case-cost of inner loop)

**Worst case cost for inner loop:**

The test has 1 comparison plus 1 method call and is check n + 1 times so the max is 2(n + 1) instructions executed

Worst case cost for execution of loop body (2 + 3 + 5 + 6 + 10)

Upper bound on worst-case cost to execute the inner loop 2(n + 1) + 26n

Upper bound on worst-case cost to execute the outer loop 2(m + 1) + 16(2(n + 1) + 26n)

Upper bound on worst-case cost to execute the method 2(m + 1) + 16(2(n + 1) + 26n) + 8

Worst Case cost for getTruthTable() method:

**Worst case for outer while loop:**

The test has 1 comparison plus 3 method calls, 1 conversion and an addition and is check m + 1 times so the max is 5(m + 1) instructions executed

Worst case cost for execution of loop body (6 + 10 + 7 + 2 + worst-case-cost of inner loops)

**Worst case cost for inner loop 1:**

The test has 3 comparisons and is checked m + 1 times so the max is 3(m + 1) instructions executed

Worst case cost for execution of loop body (19 + 3 + 2)

Upper bound on worst-case cost to execute the inner loop 1 3(m + 1) + 24m

**Worst case cost for inner loop 2:**

The test has 3 comparisons and is checked p + 1 times so the max is 3(p + 1) instructions executed

Worst case cost for execution of loop body (16 + 4)

Upper bound on worst-case cost to execute the inner loop 2 3(p + 1) + 20p

**Worst case cost for inner loop 3:**

The test has 3 comparisons, 2 method calls and 1 arithmetic operation and is checked q + 1 times so the max is 6(q + 1) instructions executed

Worst case cost for execution of loop body (8 + 8 + 2)

Upper bound on worst-case cost to execute the inner loop 3 6(q + 1) + 16q

Upper bound on worst-case cost to execute the outer loop:

5(m + 1) + 25[(3(m + 1) + 24n) + (3(p + 1) + 20p) + (6(q + 1) + 16q)]

Upper bound on worst-case cost to execute the method:

5(m + 1) + 25[(3(m + 1) + 24n) + (3(p + 1) + 20p) + (6(q + 1) + 16q)] + 18

Worst Case cost for findExpression() method:

The test has 2 comparisons and 5 method calls and 1 string combination and is checked m + 1 times so the max is 8(m + 1) instructions executed

Worst case cost for execution of loop body (2)

Upper bound on worst-case cost to execute the loop 8(m + 1) + 2m

Upper bound on worst-case cost to execute the method: 8(m + 1) + 2m + 7

Big O growth Rate: O(n): Worst case iterations is n, loop body runs in constant time so worst case runtime is O(n)

Proof for big O:

8(m + 1) + 2m + 7 = 8m + 8 + 2m + 7 = 10m + 15 e O(n)

Proving: 10m + 15 e O(n): 10m + 15 < 10m + 15m = 25m, thus 10m + 15 e O(n): for c = 25 and n > 0