Methodology

Section 1: getInput()

This section handles retrieving data from the user. Input is taken using a Scanner object and stored into a string.

Section 2: parse()

This sections handles finding independent variables and sub-expressions. It does this by going through the user’s input string character by character and handling them. If a character is a letter, then it is added if it is not already in the list of variables. If the character is a ‘(‘, then it’s index is added to a stack, when a corresponding ‘)’ is found the previous index from the stack is taken and a substring between the brackets is found, if this string is not in the expressions map already it gets added. Finally, the total number of rows needed for the variables truth values is found to be 2 ^ (number of variables).

Section 3: evaluate()

This section calculates truth values for each variable and expression and puts them in the truth table. It does this by iterating through the table by column and then row. If the column should hold a variable then the variable is stored at the first row of that column, otherwise the expression’s key is stored at the top of this column. Next the values of the variable/expression are obtained, finally these values are iterated through and put into the rows under the current column.

Section 4: printVariables()

This section prints out the variables by using an iterator to go through the elements of the list and printing each.

Section 5: printExpressions()

This section prints out the expressions by iterating through the length of the expressions and finding a key using the index and then prints the key and expression for that key.

Section 6: printTruthTable()

This section prints out the truth table by iterating through column and rows. If the row is the title (first) row then the title is simply printed, otherwise, the rows are padded to be in the middle of the title.

Helper Section 1: indexOf(list, object)

This section finds the index of an object in a list. It does this by using an iterator and iterating through the list, when the object is found the index is returned, when the object is not in the list the loop will finish and return -1, signifying that the object has not been found.

Helper Section 2: contains(map, object)

This section checks whether an object is in a map or not. It does this by using an iterator over the keys of the map and checking the value mapped to each key, if the values is equal to the object then true is returned. When the object is not in the map the loop will finish and false will be returned.

Helper Section 3: toPostfix(input)

This section converts an input into it’s postfix notation. It does this by iterating through the input character by character. When the character is a letter it is added to the output, if it is a ‘)’ then an operator is taken from the operators stack and added to the output, finally if the character is not ‘)’ it must be an operator so it gets added to the stack; all ‘)’ are simply skipped. After iterating through the input any operators still on the stack will be added to the end of the output, this solves the problem of missing outer brackets for negations, i.e., -(A+B) will come out as AB+- instead of AB+. Lastly, the output is returned.

Helper Section 4: getTruthValues(expression)

This section calculates truth values for an expression by first converting the expression to post fix notation. This new expression is then iterated through, when a letter is found the truth values of that variable are pushed to the stack of truth values. When a symbol (+, -, \*) is encountered then either 1or 2 values are popped from the stack and either negated, anded or ored, then the singular result is put back on the stack. After iterating through all the expression, the stack should only contain a single value, which is then returned.

Helper Section 5: getTruthColumn(variable)

This section finds the truth values for a single variable. This is done by first finding how many trues the variable should have in a row before switching to false. This is just total values / 2 ^ (index of variable + 1). Next a loop goes through the total number of values needed and checks if enough trues have been added, signalling that false should be added, and just appending what values is needed and incrementing the counter. Finally, the values are returned.

Helper Section 6: negate(input)

This section negates an input by iterating through character by character and adding the logical opposite of the character to a result string, finally the result is returned.

Helper Section 7: or(input1, input2)

This section ors 2 strings by iterating through and checking if either character is equal to ‘T’, when they are then a ‘T’ is added to the result string otherwise a ‘F’ is added. Finally, the result is returned.

Helper Section 8: and(input1, input2)

This section ands 2 strings by iterating through and checking if both characters are equal to ‘T’, when they are then a ‘T’ is added to the result string otherwise a ‘F’ is added. Finally, the result is returned.

Proof

indexOf(list, object)

Lines 291-312

Pre-Condition

1. list is a List
2. object is the same type as list’s values

Post-Condition

1. list, object are unchanged
2. index of object is returned if found, -1 is returned otherwise

Proof

Proof by parts below

Lines 291-296

Pre-Condition

1. list is a List

Post-Condition

1. iterator of list is obtained
2. index is -1

Proof

Trivial, section precondition guarantees that list is a List and all Lists have iterators so one can be obtained from list, index is just set.

Lines 299-308

Pre-Condition

1. iterator, object have values
2. index has been initialized

Post-Condition

1. index is of the object is found and returned
2. loop has stopped when index is not found

Proof

Upon entering the loop the index is incremented and the next element of the iterator is checked, if the element matches then the index is returned, otherwise the loop will continue. Once all elements are checked and a return has not occurred the loop will end.

Proof: Invariant

The invariant is iterator.hasNext() and element not found.

Base Case: list is empty, then iterator.hasNext() is false so the loop will not start.

Inductive Step: the loop has run k times for k < length of list. If the loop starts then the end of the list has not been reached and all elements at index < k are not equal to object. If the loop has started, then the index of object has not been found and the iterator still has elements, otherwise the loop will not continue as the invariant is not satisfied.

Proof: Variant

The variant is f(list.length(), index + 1) = list.length() – (index + 1). After every iteration index increases so the variant decreases by 1. When the variant is less than or equal to 0 index + 1 must be equal to list.length(), this prevents the loop from running once more and ensures termination.

Line 311

Pre-condition

1. index of object has not been found

Post-condition

1. -1 is returned

Proof

Trivial, when this line is reached the loop must have finished and the object has not been found, or else the method would have returned, in this case -1 is simply returned.

Analysis

indexOf(list, object) = 5n+5 units

This method is O(n) for C = 10 and N0 = 1 because 5n+5 is always less than or equal to 10n