**Python Project**

**Boolean Expression Simplifier**

**Joshua Waters**

**N0916167**

Contents

[1. Analysis 3](#_Toc58204871)

[Aims 3](#_Toc58204872)

[Problem Identification 3](#_Toc58204873)

[Research of Existing Problem Solutions 4](#_Toc58204874)

[Program Requirements 4](#_Toc58204875)

[2. Design 5](#_Toc58204876)

[Problem Decomposition 5](#_Toc58204877)

[Class Layout 12](#_Toc58204878)

[Parser 12](#_Toc58204879)

[GenerateContext 13](#_Toc58204880)

[QM 14](#_Toc58204881)

[Pseudocode 15](#_Toc58204882)

[Parser 15](#_Toc58204883)

[GenerateContext 19](#_Toc58204884)

[QM 20](#_Toc58204885)

[Instantiation of Classes and Main Loop 26](#_Toc58204886)

[3. Final Testing 27](#_Toc58204887)

[4. Evaluation 29](#_Toc58204888)

[What worked and what did not 29](#_Toc58204889)

[Future improvements 29](#_Toc58204890)

# 1. Analysis

## Aims

I would like to create a program that converts a Boolean expression as an input and returns the expression in its most simplified form. This program would not do the usual of applying a bunch of Boolean algebraic rules to the expression, but rather using the Quine-McCluskey algorithm.

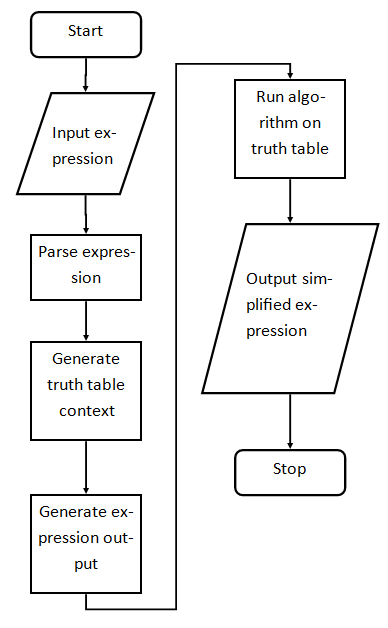
## Problem Identification

Many online Boolean expression simplifiers use the slow method of attempting to apply all the different rules to the current state of the expression. This is good for visualising the steps of the method a human would go about simplifying an expression themselves, however it does not work so well when there are more than maybe 4 or 5 unique variables in the expression.

Therefore, I have come up with the following method to achieve my aims:

1. Parse the inputted expression into an abstract syntax tree
2. Generate a truth table context from the variables in the expression
3. Generate the output of each row of the truth table based on the expression
4. Generate a simplified expression from the truth table
5. Further simplify expression where possible

This can be visualised with the following diagram.



## Research of Existing Problem Solutions

A common feature of all other expression simplifiers is being able to use different mathematical terms for the inputted expression. For example, this could be using “and” or “.” or “&”, for the AND operator. I this should be a feature I need to implement into my program so that people who have learnt any type of syntax should be able to use it. However, some other programs do not allow interchangeability i.e. you cannot input an expression like “(A and B) . C”, which could be annoying for longer expressions. Therefore, I will try to implement this into my program as well.

In addition to operator interchangeability, I think that my program should also allow variables to be used in whatever way they are inputted, and it should save those variables for the simplified expression. I have noticed that a few of the more complicated web simplifiers will output an expression using variables different to those that I entered. On top of this, allowing any word as a variable, if there is no space separating them, is also good. E.g. you could input something like “(apples and bananas) and carrots” (which would output applesbananascarrots)

Many other programs may not provide a reason *why* the user’s inputted expression is not valid. This should not be a too difficult task, so therefore I would also like to include this feature. I think this should be something as simple as the console telling the user which part has gone wrong e.g. if they wrote something like “(A and)” the program should output “Error: Variable not detected” or if they wrote “((A and B).C” the program should output “Error: Missing ‘)’”.

## Program Requirements

Based on my problem identification and research of existing solutions to the problem, I have come up with this specification for my program:

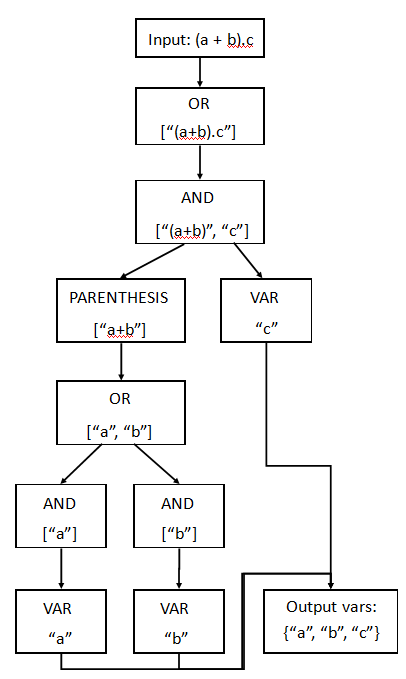
|  |  |  |
| --- | --- | --- |
| Success Criteria | Proposed Solution | Necessity for program? |
| Simplify expressions | Using the sum of product simplification algorithm called the Quine-McClusky algorithm. It is deterministic in nature and so will be more efficient with many variables. | Yes |
| Interchangeability of operators | Parser will be able to interchange between any of the operators if the user has inputted one of the allowed ones. | Yes |
| Any variable name can be used if it contains no whitespace | Store variables as a string rather than character and writing the simplifier and parser to allow for this | No |
| Abstraction of simplification process | The program will not print any tables of the simplification process because the program should abstract as much as this information from the user as possible. The program should just get expression input and output the simplified expression. | Yes |
| Error outputs | If the user inputs an invalid expression, the program should stop running and output an exception message, so they know why it did not work. | No |
| Execution time output | The program should output the overall time it took to simplify the expression so that the user can see how long it took for much longer expressions with many more variables. | No |
| Be able to input literals | The program should be able to take in expressions with literals (0 and 1) and still simplify them. | Yes |

# 2. Design

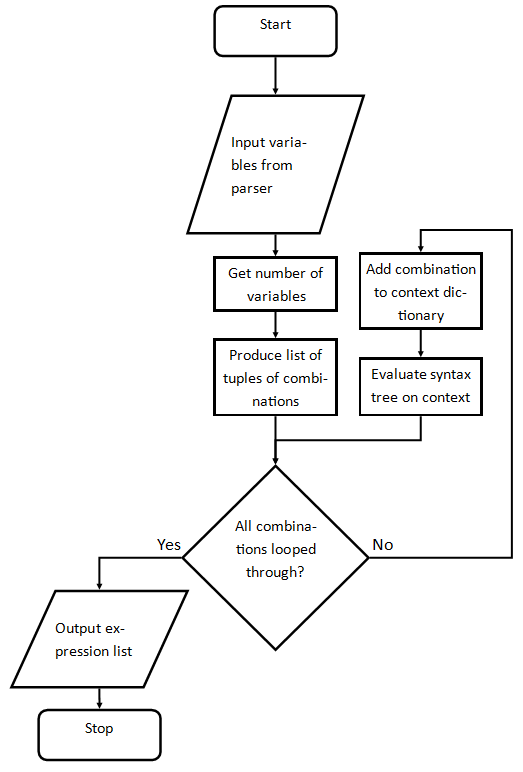
## Problem Decomposition

Each of the steps in my method shown in analysis can be decomposed further into smaller steps.

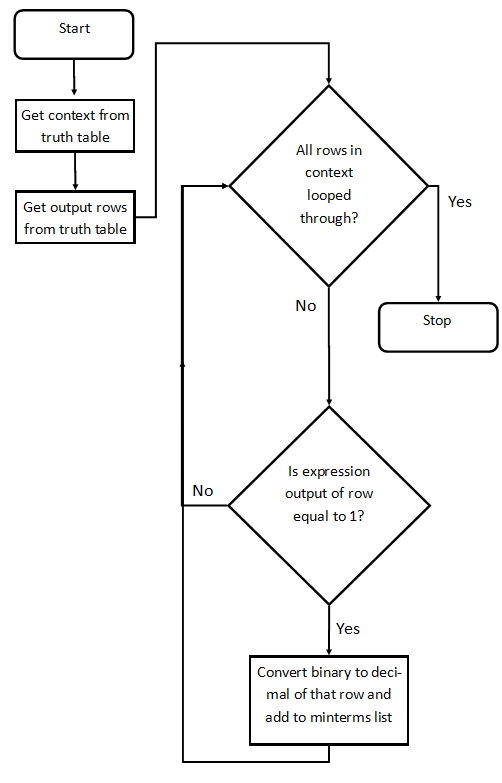
1. **Parse the inputted expression into an abstract syntax tree**
   1. Take in the inputted expression as text
   2. Remove all whitespace from text
   3. Descend into the syntax tree by
   4. Parsing the entire expression
      1. Consumes token position
   5. Parses the OR symbol of the expression
      1. Peeks and consumes token position
   6. Parses the AND term of the OR symbol
      1. Peeks and consumes token position
   7. Parses the symbols of the AND term
      1. Peeks token position
   8. Parses the NOT of the symbols
      1. Consumes token position
   9. Parses the parenthesis of the expression
      1. Consumes token position for inner and outer brackets
   10. Parses the variables in the expression
       1. Consumes token position
   11. Parses the literals of the expression
       1. Consumes token position
   12. Store all the variables in a set



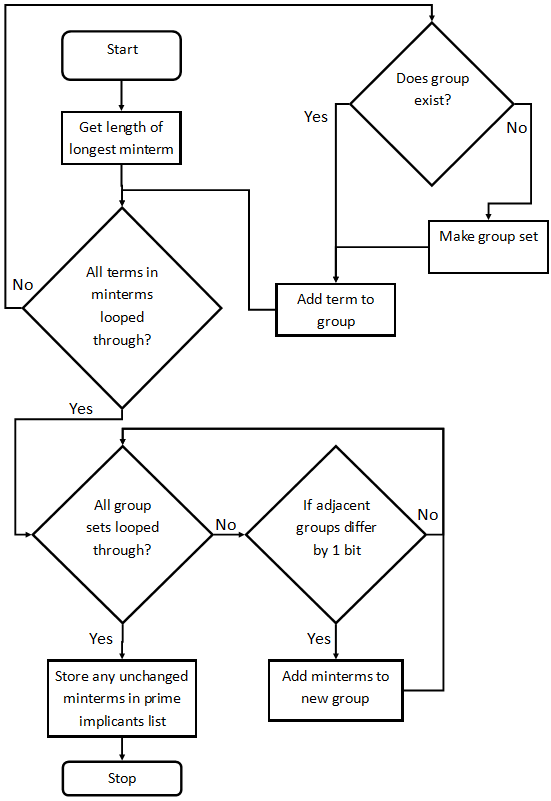
1. **Generate a truth table context from the variables in the expression (combined with step 3)**
2. **Generate the output of each row of the truth table based on the expression**
   1. Take in the variables outputted from the parser
   2. Get the number of variables
   3. Produce a list of tuples of all the combinations of the variables
   4. Loop through each of the tuples and add each one to a dictionary with its representing variable
   5. Perform syntax tree evaluation on the context dictionary
   6. Make new context dictionary with new combination
   7. Evaluate context
   8. Repeat until all combinations have been exhausted
   9. Output the true or false for expression in list



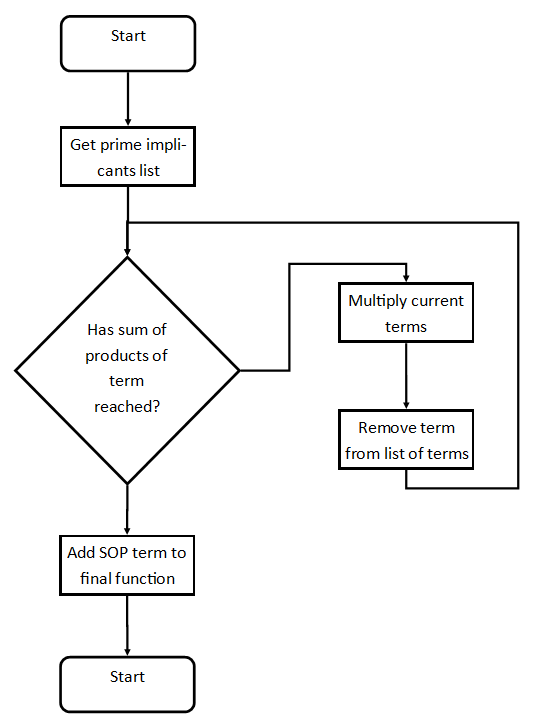
1. **Generate minterms from the expression**
   1. Generate minterms from expression
      1. Loop through expression output
      2. When expression output is 1
      3. Add context values of that row to list
      4. Loop through list, convert binary to decimal for each row to generate minterms



1. **Generate a simplified expression from the truth table**
   1. Generate first groups for minterms
      1. If number of 1s in binary minterm is 1, put into group 1
      2. If number of 1s in binary minterm is 2, put into group 2
      3. Repeat
   2. Generate second groups for minterms
      1. Combine all minterms in adjacent groups that differ by only 1 literal
      2. Replace that literal with a ‘-‘
      3. Put combined minterms from first 2 adjacent groups into group 1
      4. Put combined minterms from next 2 adjacent groups into group 2
      5. Store any minterms that were not combined
   3. Generate next set of groups for minterms and repeat everything in the above set of steps
   4. Repeat each set of groups until the groups are left with minterms that cannot be combined any further
   5. Convert minterms from previous step and stored minterms from group combining into prime implicants
   6. Work out which prime implicants are essential prime implicants then remove the rest
   7. Work out the function from that



1. **Perform extra simplification using Petrik’s method if possible**
   1. Multiply terms until the sum of product of the terms is reached
   2. Choose the terms with the least number of variables
   3. The add all the essential prime implicants to the final function



## Class Layout

In order to implement my very linear program (with only one path for the user to choose), I will use Object Oriented Programming. This will allow me to break down my problem into multiple classes, which will in turn, have their own functions. This encourages abstraction of the problem through encapsulation, meaning that I do not need to worry what happens in one class when I write another.

I will implement 3 main classes and 6 child classes:

1. Parser
   1. NotExpression
   2. AndExpression
   3. OrExpression
   4. ParenthesizedSymbol
   5. VariableSymbol
   6. LiteralSymbol
2. GenerateContext
3. QM

### Parser

For the parser, I am going to go by my own written grammar. Here it is, in Backus-Naur Form:

<expression> ::= <term> ( <or\_symbol> <term> )...

<or\_symbol> ::= "+" | "or"

<term> ::= <symbol> ( ("." | "and") <symbol> )...

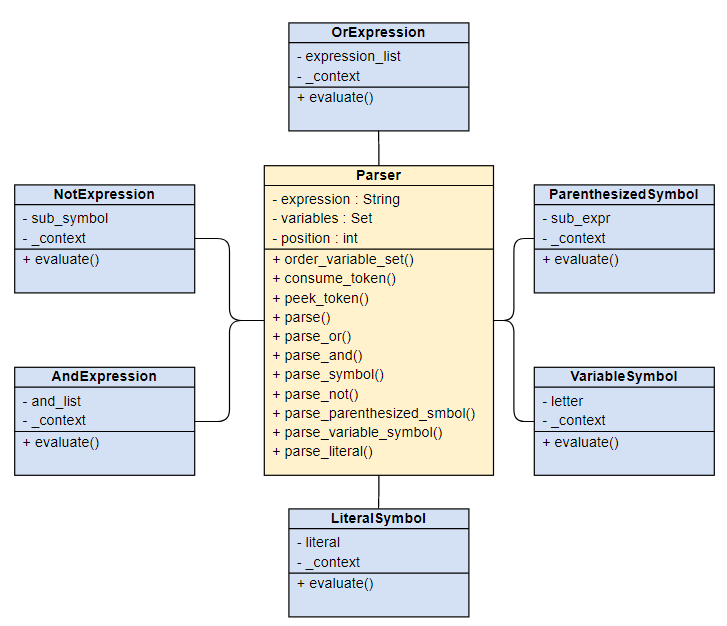
<symbol> ::= <not\_symbol> | <parenthesized\_symbol> | <variable> | <literal>

<not\_symbol> ::= "!" <symbol>

<parenthesized\_symbol> ::= "(" <expression> ")"

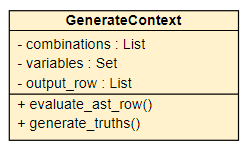
<variable> ::= a letter

<literal> ::= "0" | "1"



As you can see, the evaluate() function is the same in every sub-class. This is because they all take in \_context, which is polymorphemic with sometimes evaluate() is called and sometimes not.

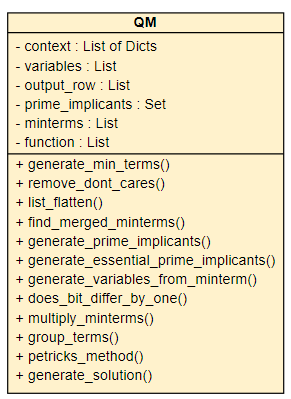
### GenerateContext

This class will quite simply, generate all the combinations of the truth table based on the variables set given from the parser. E.g. if the variables are a, b and c, the class will generate:

|  |  |  |
| --- | --- | --- |
| a | b | c |
| 0 | 0 | 0 |
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 0 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |
| 1 | 1 | 1 |

### QM

This class stands for Quine-McClusky, as it will be the part that runs the actual simplification algorithms.



## Pseudocode

### Parser

Class Parser

constructor(expression)

initialise class scoped variables

set class scoped expression to input expression

function order\_variable\_set()

return sorted variable set

function consume\_token()

while pos is less than length of expression and expression at pos is whitespace

increment pos

if pos is more than or or equal to length of expression

return nothing

elif text[pos] is a letter

while pos is less than length of expression and expression at pos is a letter

add expression at pos to ret

increment pos

return ret

else

set expression at pos to ret

increment pos

return ret

function peek\_token()

set previous pos to pos

set ret to consume\_token()

set pos to previous pos

return ret

function parse

ret = parse\_or()

if consume\_token() is not empty

raise Error

print(“Error: Parsing error”)

variables = order\_variable\_set()

return ret

function parse\_or()

set terms to parse\_and()

peek\_token()

while token is "+" or any alternatives

consume\_token()

add parse\_and() to terms

consume\_token()

ret = OrExpression(terms)

return ret

function parse\_and()

set terms to parse\_symbol()

peek\_token()

while token is "." or any alternatives

consume\_token()

add parse\_symbol() to terms

peek\_token()

ret = AndExpression(terms)

return ret

function parse\_symbol()

set token to peek\_token()

if token is "("

return parse\_parenthesized\_symbol()

elif token is "!" or any alternatives

return parse\_not()

elif token is a literal

return parse\_literal()

else

return parse\_variable\_symbol()

function parse\_not()

set token to consume\_token()

if token is not "!" or any alternatives

raise Error

print("Error: Invalid syntax")

set sub\_symbol to parse\_symbol()

set ret to NotExpression(sub\_symbol)

return ret

function parse\_parenthesized\_symbol()

set token to consume\_token()

if token not "("

raise Error

print("Error: Missing (")

set ret to ParenthesizedSymbol(parse\_or())

set token to consume\_token()

if token not ")"

raise Error

print("Error: Missing )")

return ret

function parse\_variable\_symbol()

set name to consume\_token()

if name does not contain anything

raise Error

print("Error: variable not detected")

if name does not contain letters

raise Error

print("Error: varibale not valid")

add name to variableSet

return VariableSymbol(name)

function parse\_literal()

set token to consume\_token()

if token is not "1" or "0"

raise Error

print("Error: incorrect syntax")

if token is "1"

set literal to truee

else

set literal to false

set ret to LiteralSymbol(literal)

return ret

### GenerateContext

I plan to use itertools, an external python module that is useful for optimal and easy iteration techniques. More specifically, I plan to use itertools.product() which helps to generate all combinations of an input. This replaces otherwise having to use a recursive top-down syntax tree to generate the combinations.

class GenerateContext

constructor(context, output\_row, variables)

initialise class scoped variables

set class scoped variables to input variables

function evaluate\_ast\_row()

for tuples in combinations

set context\_row to the dict of the zip of variables and tuples

append context\_row to context

return context

function generate\_truths()

set context to evaluate\_ast\_row()

for row in context

append ast.evaluate(row) to output\_row

return output\_row

### QM

class QM:

constructor(context, output\_row, variables)

initialise class scoped variables

set class scoped context, output\_row and variables to input vars

function generate\_min\_terms()

for row in context

for element in row

append string of row(element) to local minterm list

append int form of binary to minterms list

function remove\_dont\_cares()

for pos and row in the enumeration of output\_row

if row is false

append output\_row[row] to temp list

else

append the current pos to pos\_flags list

set output\_row to contents of temp list

set i to 0

for pos in pos\_flags

remove (pos - i) from minterms

increment i

function list\_flatten()

for i in input\_list

flatten input\_list

return flattened list

function find\_merged\_minterms()

set differed\_bit to the count of "-" in minterm

if differed\_bit is 0

return string of binary minterm

for i in range of differed\_bits

set diff to filled binary num of i

for j in diff[0]

if prev\_term is not -1

append temp\_min\_terms to prev\_term

else

set temp\_min\_terms to prev\_term

return temp\_min\_terms

function generate\_prime\_implicants()

for i in prime\_implicants

merged\_minterms = find\_merged\_minterms(i)

for j in merged\_minterms

if prime\_implicants\_list[j] exists

if i not in prime\_implicants\_list[j]

append i to prime\_implicants\_list[j]

else

set prime\_implicants\_list[j] to [i]

return prime\_implicants\_list

function generate\_essential\_prime\_implicants()

for i in prime\_implicants

if length of prime\_implicants[i] is 1

if prime\_implicants[i][0] is not in ret

append prime\_implicants[i][0] to ret

return ret

function generate\_variables\_from\_minterms()

for i in range of length of minterm

set current\_var to variables[i]

if minterm[i] is "0"

append current\_var + "'" to ret

else minterm[i] is "1"

append current\_var to ret

return ret

function does\_bit\_differ\_by\_one()

for i in range of length of first\_binary\_number

if first\_binary\_number[i] is not second\_binary\_number[i]

increment count\_diffs

set diff\_pos to i

if count\_diffs is more than 1

return false and none

if count\_diffs is 0

return false and none

return true and diff\_pos

function multiply\_minterms()

for x in exp1

if x + "'" in exp or length of x is 2 and x[0] in exp2 is true

return empty list

else

append x to ret

for x in exp2

if x not in ret

append x to ret

return ret

function multiply\_expressions()

for x in exp1

for y in exp2

set multi\_exp to multiply\_minterms(x and y)

if length of multi\_exp is not 0

append multi\_exp to ret

return ret

function group\_terms()

set bin\_length to the length of the binary of the final element of minterms minus 2

for term in minterms

set num\_ones\_in\_minterm to the binary of number of "1"s in term

if groups[num\_ones\_in\_minterm] exists

append binary of term filled with bin\_length to groups[num\_ones\_in\_minterm]

else

set groups[num\_ones\_in\_minterm] to binary of term filled with bin\_length

while true

copy groups into first\_set\_groups

empty groups

group\_elements = sorted list of keys in first\_set\_groups

for x in range of length of group\_elements - 1

for y in first\_set\_groups [group\_elements[x]]

for z in first\_set\_groups [group\_elements[x + 1]

bit\_differ = does\_bit\_differ\_by\_one(y and z)

if bit\_differ[0] is true

if groups exists

replace groups[group\_num] with bit\_differ position

else

set groups[group\_num] to bit\_differ position

else

create the group set

set break\_loop to false

add y to changed\_minterms

add z to changed\_minterms

increment group\_num by 1

store all unchanged minterms through list\_flatten()

set prime\_implicants to prime\_implicants union of unchanged\_minterms

if break\_loop is true

break out of loop

function petricks\_method

for x in prime\_implicants\_list

for y in prime\_implicants\_list[x]

append generate\_variables\_from\_minterm to petrick

while length of petrick is more than 1

set petrick[1] to multiply\_expressions(petrick[0] and petrick[1])

remove first element of petrick

set function to min of petrick[0]

add essential prime implicants to function

function generate\_solution

group\_terms()

prime\_implicants\_list = generate\_prime\_implicants()

essential\_prime\_implicants = generate\_essential\_prime\_implicants(prime\_implicants\_list)

for x in essential\_prime\_implicants

for y in find\_merged\_minterms(x)

if list exists

delete prime\_implicants\_list[y]

if length of prime\_implicants\_list is 0

set function to essential\_prime\_implicants

else

petricks\_method(prime\_implicants\_list, essential\_prime\_implicants)

if nothing in function

print("There is no solution to this expression")

else

print(function)

### Instantiation of Classes and Main Loop

while true

print(

"Boolean expression simplifier. Input your expression below or type 'quit' to quit. Supported tokens "

"are:\n'!', 'not', '¬', '-',\n'+', 'or', '|', 'v',\n'.', 'and', '^', '&'.\nE.g. '(A and B) or C' is "

"equivalent to '(A.B) + C'")

set expression to input

if expression is "quit"

exit program

else

instantiate all classes

# 3. Final Testing

Key:

* Green = Success
* Orange = Partial Success
* Red = Fail

|  |  |  |
| --- | --- | --- |
| Test (string input in) | Expected Output | Actual Output |
| quit | Program stops | Program stops |
| (A.B).C | A’ + B’ + C’ | Note: Variables output in different order to alphabetical is still the same solution |
| (A . B) . C | A’ + B’ + C’ |  |
| (a . b) .c | A’ + B’ + C’ |  |
| (a.b). C.D+E | F = C’E’ + E’a’ + D’E + E’b’ |  |
| (((A.B))) | F = A’ + B’ |  |
| (A and B) | F = A’ + B’ |  |
| hello and world | F = hello’ + world’ |  |
| hello or world | F = hello’ |  |
| !hello and world | F = hello + world’ |  |
| this and this and this and this | F = this’ |  |
| a and | Error: variable not detected |  |
| (a and b | Error: missing ) |  |
| (a and b)) | Error: missing ( | It gave an error, just not the correct type |
| a a a | Error: Parse error |  |
| 420 | Error: Parse error | It gave an error, just not the correct type |
| 420 and 500 | Error : Parse error | It gave an error, just not the correct type |
| (a v b) ^ c . -d | F = a’b’ + c’ + d’ |  |
| (a.b).c.(d.e)+f.g.(h.i) | F = d'g' + c'i' + e'f' + d'i' + e'g' + b'f' + c'h' + d'f' + a'h' + b'i' + a'g' + b'h' + d'h' + b'g' + a'f' + e'h' + c'g' + e'i' + a'i' + c'f' | Where |
| +++ | Error: variable not detected |  |
| (and and b) and or | F= and’ + b’ + or’ |  |
| (and and and) and and | F = and’ |  |
| a . 1 | F = a’ |  |
| a . 0 | There is no solution to this expression |  |
| (a.0)and(b.0)or(c.1) | F = c’ |  |
| (A+B).C+(D.E). !(F+A+C).G. !H.(C+B. (K.X+Y)) | AC’ + A’B’ + C’D’ + C’G’ + C’H + C’E’ + C’F’ + C’K’Y’ + C’X’Y | Execution time is almost 10 minutes, very high for only 10 variables. |

# 4. Evaluation

## What worked and what did not

Here is my success criteria table I made in the analysis section. I will highlight what worked in green and what did not work **as intended** in orange.

|  |  |  |
| --- | --- | --- |
| Success Criteria | Proposed Solution | Necessity for program? |
| Simplify expressions | Using the sum of product simplification algorithm called the Quine-McClusky algorithm. It is deterministic in nature and so will be more efficient with many variables. | Yes |
| Interchangeability of operators | Parser will be able to interchange between any of the operators if the user has inputted one of the allowed ones. | Yes |
| Any variable name can be used if it contains no whitespace | Store variables as a string rather than character and writing the simplifier and parser to allow for this | No |
| Abstraction of simplification process | The program will not print any tables of the simplification process because the program should abstract as much as this information from the user as possible. The program should just get expression input and output the simplified expression. | Yes |
| Error outputs | If the user inputs an invalid expression, the program should stop running and output an exception message, so they know why it did not work. | No |
| Execution time output | The program should output the overall time it took to simplify the expression so that the user can see how long it took for much longer expressions with many more variables. | No |
| Be able to input literals | The program should be able to take in expressions with literals (0 and 1) and still simplify them. | Yes |

The overall program is a great success, through 7 of 8 criteria being fully met, of which 3 of 4 are critical.

However, one part of the program, arguably the most important part, did not actually work as intended. The expected output of a typical Boolean expression simplifier is in the form of an actual expression. The output of my program is in sum of product (SOP) form without the and tokens because that is how the Quine McClusky algorithm works. So the user gets an expression that is the most simplified in a form that the user may not want.

## Future improvements

Because the output expression comes out in a possibly unexpected way, one obvious future improvement would be to take this SOP expression and turn it into a more readable expression in the sort of form that the user inputted. E.g. if the program currently outputs F = A’B’ + CD’ it should output F= A’.B’ + C.D’.

The second improvement, that would most likely have the largest effect, would be to optimise the code as much as possible. This is because, as seen in the testing table, the execution time of just 10 variables is 500 seconds. Although this is faster than a simplifier using K-maps as its algorithm (that would take hours), this does not hold true to the expected speed of almost instant until 20 variables.