# AC 2012-3946: DESIGNING A BOOLEAN ALGEBRA TOOL AND ITS USE IN THE CLASSROOM

**Mr. Howard Whitston, University of South Alabama**

Howard Ernest Whitston is an Instructor at the University of South Alabama, School of Computing, having taught at several colleges and universities since 1983. He has two B.S. degrees, one in mathematics and one in chemistry. He has two M.S. degrees, one in biochemistry, the other one in CIS, specializing in Computer Science. He has been at the University of South Alabama since 2005. Whitston is a member of ACM (Association for Computing Machinery), ACS (American Chemical Society), IEEE-Computer Society, MAA (Mathematical Association of America), and ASEE (since Dec. 2011).

**Mr. Adam Thomas Moore, University of South Alabama**

Adam Moore is a computer science and mathematics major at the University of South Alabama. He is interested in artificial intelligence and bioinformatics.

©American Society for Engineering Education, 2012

# A Boolean Logic Simplification Tool for Students

## 1. Introduction

Computer Science (CS) students usually take a course in Digital Logic during the second year of their CS education. The study of Boolean Algebra, its theorems and their relationship to combinational logic circuit description is a large part of that course. These rules are used to algebraically simplify the equation of a circuit, which usually leads to a smaller circuit that will cost less to produce. A solid understanding of Boolean Algebra concepts is likewise needed to understand the more complicated aspects of combinational logic circuits. These rules are found in Appendix A at the end of this paper for reference.

However, the process of simplifying expressions using Boolean Theorems is not always straightforward, especially to students who have little experience with a Boolean Algebra formula. Furthermore, Mirmotahari, et. al[1] found that Computer Architecture students often do not "master practical exercises in the relation between Boolean logic and gates." A program that assists students by performing the task of simplification and explains the process would facilitate their understanding.

The organization of this paper has Section 2 giving the background of Boolean Algebra, while Section 3 discusses the initial solution, followed by the current solution and the steps used to write the program along with the reasons for the programming language selected. The user experience with the current version of this program is discussed in Section 4 with the authors' conclusion in Section 5. The paper concludes with Section 6 with the authors' plans for the future. Appendix A contains the Boolean Theorems used.

## 2. Background: What is Boolean Algebra?

Students uninitiated to the concepts of Boolean Algebra are often shocked to discover that, in Boolean Algebra, one plus one is not two. It is shown to be 1 + 1 = 1. In fact, Boolean variables and constants may only have one of two possible values, either one or zero[2].

Similarly, a circuit in a digital system can be in one of two states, HIGH or LOW. This corresponds with the idea that computers operate in 1s and 0s, with 1 being HIGH and 0 being LOW. Digital logic students will often begin the study of circuits by learning two logic gates, the AND gate and OR gate. The AND gate can take multiple inputs, and will go HIGH when all of its inputs are HIGH , and be LOW the rest of the time. The OR gate will go HIGH when any of its inputs are high and LOW if and only if all of its inputs are LOW.

In Boolean Algebra, OR is represented by '+' and AND is represented by '*'. AND is also represented by symbol concatenation, so XY = X * Y. In some contexts, other symbols are used, but '+' and '*' will be the convention used in this paper.

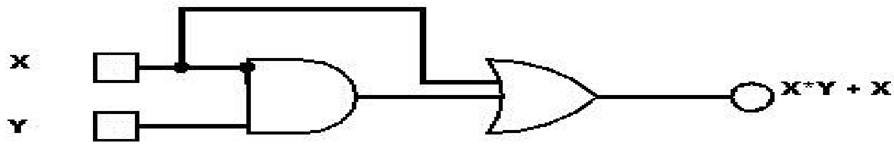


Figure 1

Consider the circuit in **Figure 1**. There are two inputs, X and Y, with X and Y connected to an AND gate, whose output is connected to an OR gate. X is also connected to this OR gate. The formula in **Figure 2** is a Boolean Algebra expression that describes the operation of this circuit.

While this circuit may seem quite simple, it can be reduced even further. One of the Boolean theorems (Appendix A 14) directly applies to this situation and reduces the expression to just X [2]. If the Boolean OR is commutative, meaning that X + Y = Y + X, then the original circuit can be modified. Fortunately, this is provided in theorem 9, which states just that. **Figure 2** depicts the simplification process, with the applied theorem listed.

X*Y + X
is equivalent to X + X*Y (Theorem 9)
is equivalent to X (Theorem 14)
Figure 2 – *Simplification of the formula X*Y + X*

The simplification of the first example was relatively straightforward, requiring only two theorems (and only one had the terms been ordered differently from the start). Usually, the process of simplification is not as straightforward, and may be performed in a variety of ways. This is illustrated in Example 4-2 from *Digital Systems* [2]. The equation from the example is z = A(~B)(~C) + A(~B)C + ABC.

The tilde (~) is used to denote NOT, or negation. The NOT gate is shown in **Figure 3.** The NOT gate "inverts" the signal, so a HIGH becomes a LOW and a LOW becomes a HIGH. In the equation above, this means that the first term, A(~B)(~C) is HIGH when A is HIGH, B is LOW and C is LOW.

Figure 3 – the NOT gate inverts its input

The simplification of this formula is not as obvious as the first example. The first observation is that the first two terms have the common factor A(~B). Using Theorem 13a, which states that X(Y + Z) = XY + XZ, the equation is now z = A(~B)(~C + C) + ABC. The term A(~B) has been factored out. According to Theorem 8, ~C + C would simplify to 1. The equation then becomes z = A(~B)(1) + ABC. Then, according to Theorem 2, A(~B)(1) becomes A(~B), so the theorem is now z = A(~B) + ABC.

Now A can be factored, resulting in z = A(~B + BC), which according to Theorem 15b simplifies to z = A(~B + C).

However, this is not the only method of simplification. Further observation of the original formula reveals that while the first two terms have the common factor A(~B), the last two terms share AC. These can *both* be employed by simply adding an extra A(~B)C to the equation. This idea is one of the strangest to students of traditional math, since normally adding a term to one side of the equation and not the other would change the expression. In Boolean Algebra, however, adding an additional unit of an existing term does not change the equation at all! By adding this term then allows for the factoring of *both* common factors, and the equation becomes z = A(~B)(~C) + A(~B)C + A(~B)C + ABC = A(~B)(C + ~C) + AC(~B + B).

Both C + ~C and B + ~B simplify to 1, resulting in z = A(~B)(1) + AC(1), which according to Theorem 2, simplifies to z = A(~B) + AC. The A is finally factored out, which gives the same result as before, z = A(~B + C).

This example illustrates two facts: (i) that Boolean Algebra operates by different rules than traditional algebra, and (ii) that there is often more than one way to simplify an expression. A simplification program would allow a student to experiment with many formulas to see these rules in action, while practicing their application in homework assignments. However, the above idea that multiple ways exist presents a problem in the implementation of such a program.

There are two common expressions possible in Boolean Algebra, namely Product of Sums (POS) and Sum of Products (SOP). The prior two examples are both SOP's. An example of POS would be (A + B)(B + C)(C + ~A) with the implied * (or AND) between the parentheses.

3.1 Initial Solution Attempts

The implementation of a program to demonstrate the concepts of simplification is far from trivial. As the second example illustrates, there are often multiple ways to simplify an equation, which makes a simple algorithmic solution difficult to achieve. The initial idea was to encode the theorems in the computer by defining *regular expressions* to match them in a Boolean Algebra expression. A regular expression is a way to define a pattern for the computer to match.

However, as Jeffrey Friedl explains in *Mastering Regular Expressions ,* a regular expression to match arbitrarily nested parentheses is not possible[3]. This is an obvious problem, since an expression may contain an arbitrary amount of nested parentheses. This led to the next idea, which was to build a parser that identifies the parts of the equation for further processing.

A *parser* is used to process string input into a form that will be evaluated by a program which for this project could be an array containing the tokens of interest[4]. The initial application for the use of a parser to this problem would be to define a *grammar*, which is a description of a language, that would match the Boolean theorems and replace them with the simplified input. The parsing approach was attractive because grammars can be recursively defined, which would solve the problem posed by a regular expression-based solution. This is an unusual use of a parser, since it is generally not the purpose of the parser to manipulate the data, but rather to put it into a form for manipulation by some other part of the program[4].

3.2 An Object-Oriented Solution

As mentioned earlier, the result of parsing could be an array. One of the approaches was to parse each part of the equation into an array whose first member would be the operator. The operator would be either a '+' or '*' for OR and AND, respectively. The operator would then determine what should be done with the operands in the rest of the array. This idea is already implemented in object-oriented programming languages, and is known as *operator overloading*[5].

In object-oriented programming (OOP), data "types" are defined, and their data members and operations that can be performed on that data are packaged into a *class* for convenience and reuse[5]. For example, a Bank Account class can be defined with a Balance and an Owner. Operations on a Bank Account would include Withdrawal and Deposit. These characteristics and operations can be defined or re-defined by the programmer.

When used in an application program, *instances* of classes are created and manipulated. A Bank patron would have his/her own Bank Account, which itself would be an *instance* of a Bank Account. The *class* is the description or definition of something, while an *instance* is a concrete manifestation of that description.

OOP also allows for classes to *inherit* from other classes. In the Bank Account example, a general Bank Account may be too general. Most banks offer Checking and Savings accounts. These classes would *inherit* from the Bank Account class. These classes themselves can be extended to have their own properties, such as an Interest Rate for a savings account. The Checking Account and Savings account *are* bank accounts, but are more specifically Checking Accounts or Savings Accounts.

Python is an scripting OOP language that will be used to build our program. As there is a symbolic computing module called SymPy for doing calculus-based operations along with other related operators, this module was analyzed and was used as a pattern for our module[7].

The Boolean logic problem must be framed in this manner. The base class for all Boolean objects is **BoolType**. The **Symbol** class, used for variables, will inherit from this class. **BoolExpr,** the class used to describe the **And** and **Or** operations, will also inherit from **BoolType. Or** and **And** inherit from **BoolExpr**. A diagram of these relationships is shown in **Figure 4.**

The object-oriented paradigm is useful for this problem, because of the aforementioned data encapsulation and *operator overloading* capabilities. One of the first tasks of object-oriented design is to define how classes interact and what operations they can perform[6]. This has already been done in this problem in the list of Boolean theorems. *Operator overloading*, then, is used to enforce the rules described in the Boolean Theorems.

As mentioned earlier, data attributes like "balance" or "owner name" for Bank Accounts are a crucial part of OOP. An object can be "composed of" other objects. This is known as *composition* or *aggregation*[6]. In the Bank Account example, a Bank Account is composed of a “Balance” and an "Owner Name," which are *float (in Python, float is the double precision*

*floating-point number)* and *strings,* respectively. These *attributes* or *data members* can be accessed by the program or other programs using an instance of this class.

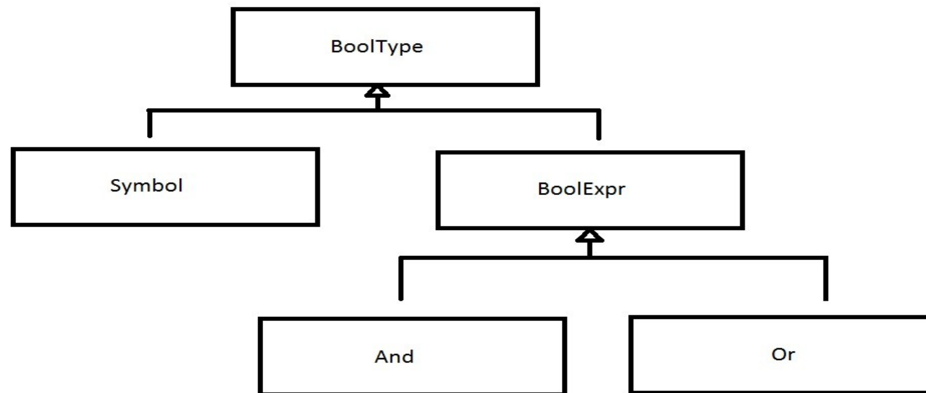


Figure 4

The attributes serve several purposes in the solution to this problem. The **BoolType** class does not have any data attributes, and is used for naming convenience. The **Symbol** class has name, which is simply the name of the symbol. The name for the symbol 'x', for example, is 'x'. The **Symbol, Or,** and **Add** objects of a set of Boolean (the **bool** type in Python) values that are used for type checking. These are 'isAnd,' 'isOr' and 'isSingle.' These need to be implemented in the base class, which will be done in the future. The **BoolExpr** class has a list of arguments, named 'args,' which holds the operands for a Boolean expression. For example, the expression x + y would be an **Or** with an 'args' list that holds x and y. The 'args' list may also hold other **BoolExpr** objects, which is how nested expressions are handled. These data attributes are used in the algorithms of the program to determine how to handle cases during the program's main objective, which is simplification of an expression.

3.3 Program Implementation

OOP concepts have now been introduced. The previous section explained how Boolean Algebra expressions will be framed in an Object-Oriented program. Python can run on multiple computer platforms, making it an ideal choice for implementation.

The program will be described here using *pseudocode*, which is a mixture of English and actual Python code as shown below. Each step of the pseudocode will be explained.

1) Initialize variables
2) Convert the Boolean Expression into a BoolExpr Object
3) Convert the expression into Sum of Product form
4) Find the Greatest Common Factors among the terms

Step 1: Initialize Variables

Before a Boolean expression can be evaluated, the variables used in the expression must be created. This is done in 4 steps, illustrated below. Consider an expression that involves an 'x' variable. First, the x variable is created and then the negated version of x , ~x, is created. The 'True' value is passed as an argument to the **Symbol()** method to indicate that the Symbol is negated. Each of these variables has a 'neg' attribute, which will point to the negated version of the other. The third and fourth steps are to set these attributes. These attributes "point" to the negated version of the object itself. This is how negation of variables is implemented in the program. The negated version of 'x' is retrieved by calling 'x.neg'. 'x.neg' can be negated again by calling 'x.neg.neg', which equals 'x'. The negated version of 'x' would typically be named 'not_x'. Thus, 'not_x.neg ' is equivalent to 'x.neg.neg'.

```
x = Symbol('x')                  # initialize the variable
not_x = Symbol('~x', True)       # initialize negated variable
not_x.neg = x                    # set the 'neg' attribute of negated variable
x.neg = not_x                    # set the 'neg' attribute of the variable
```

**Symbol** objects have an attribute named 'negated'. If this attribute holds a 'True' value, then the variable is considered negated. If the attribute holds 'False', it is considered to be not negated. Thus, not_x.negated is True, and x.negated is False.

Step 2: Convert the Boolean Expression into a BoolExpr Object

Once the variables and their negated forms are initialized, the expression can be converted to a BoolExpr object. The Python interpreter does this at runtime by calling the appropriate *special methods*. All of the variables in the expression have been initialized at this point. This means they are instances of the **Symbol** class. Since the **Symbol** class inherits from the **BoolType** class, this means that **Symbol** objects are also **BoolType** objects. *Operator overloading* is the method by which Boolean multiplication and addition are implemented in this program. In Python, this involves overloading these *special methods*: **__mul__()** and **__rmul__**() for AND's, **__add__()** and **__radd__()** for OR's in the definition of the **BoolType** class.

When addition or subtraction of a **BoolType** object is required at runtime, the Python interpreter searches first the class definition of the object and subsequently the parent classes until the appropriate version of the function is found. In OOP, this process is known as *Dynamic Binding*[6]. These special methods are defined in the **BoolType** class, so its versions of the special methods are invoked when addition or multiplication is required.

Thus, whenever **BoolType** objects are added together, **__add__()** or **__radd()__** is invoked. This method creates an **Or** object with the operands of the '+' sign. Similarly, when **BoolType** objects are multiplied, **__mul__()** or **__rmul__()** is invoked. This method creates an **And** object with the operands of the '*' sign.

Boolean theorems one through twelve are implemented in the object creation process. For example, 'X * X.neg,' when entered into the program, will return 0 rather than 'And(X, X.neg).' This enforces Boolean Theorem 4. Similarly, 'X + X.neg' will result in 1 rather than Or(X,

X.neg). This enforces Boolean Theorem 8. The results when entering any of the remaining theorems of the twelve will give the proper results.

Theorems 16 and 17, known as DeMorgan's theorems, are implemented in the **neg()** function of the program. The **neg()** function takes a **BoolExpr** and returns the result of negating the **BoolExpr** using DeMorgan's theorems. For example, 'neg(x + y)' results in x.neg*y.neg, which is equivalent to (~x) * (~y). This process aides in converting the expression to SOP form in the next step. This is because in SOP form, one negation sign cannot apply to more than one variable.

To allow the student to enter the expression using the '~' sign, the program uses regular expressions to format the input. A parser would also be able to decide which variables need to be created so the user can simply enter the expression to be evaluated without needing to specify the variables first. This functionality will be included in future versions.

The result of the object creation process will be a **BoolType** object containing the entire expression. This object can be processed further. The processing will simplify the object further, if possible, according to the Boolean theorems.

Step 3: Convert the Boolean Expression to Sum of Products (SOP) Form

Once the expression is converted to a **BoolType** object and stored in a variable, the variable can be sent to the **SOP()** function. The **SOP()** function takes a **BoolType** object and converts it to SOP form. A Boolean expression in SOP form consists of *"two or more AND terms that are ORed together "* [2]. Each AND term contains one or more variables occurring only once[2].

Converting an expression to SOP form is accomplished by distributing terms. For example, the expression A*(B + A) will become A*B + A*A, according to Theorem 13a. Converting to this form is desirable because factors among the terms in an SOP expression can be found by an algorithm.

Step 4: Find Greatest Common Factors (GCF) Among the Terms

Once the expression is in SOP form, factors among the terms can be found. Factoring will result in the elimination of one or more terms, if possible. The **factor()** function finds all of the common factors among the terms in the SOP equation.

Step 5: Factor the GCFs

After finding the factors, the **factor()** function reconstructs the expression with the common terms factored out. When the **factor()** function reconstructs the expression, any terms that can evaluate to one or zero will do so. These values will often eliminate one or more terms, resulting in a simpler expression.

**Figure 8** shows the object creation and factoring process. The equation is already in SOP form, so this step is not shown. This is the same equation presented earlier in this paper.

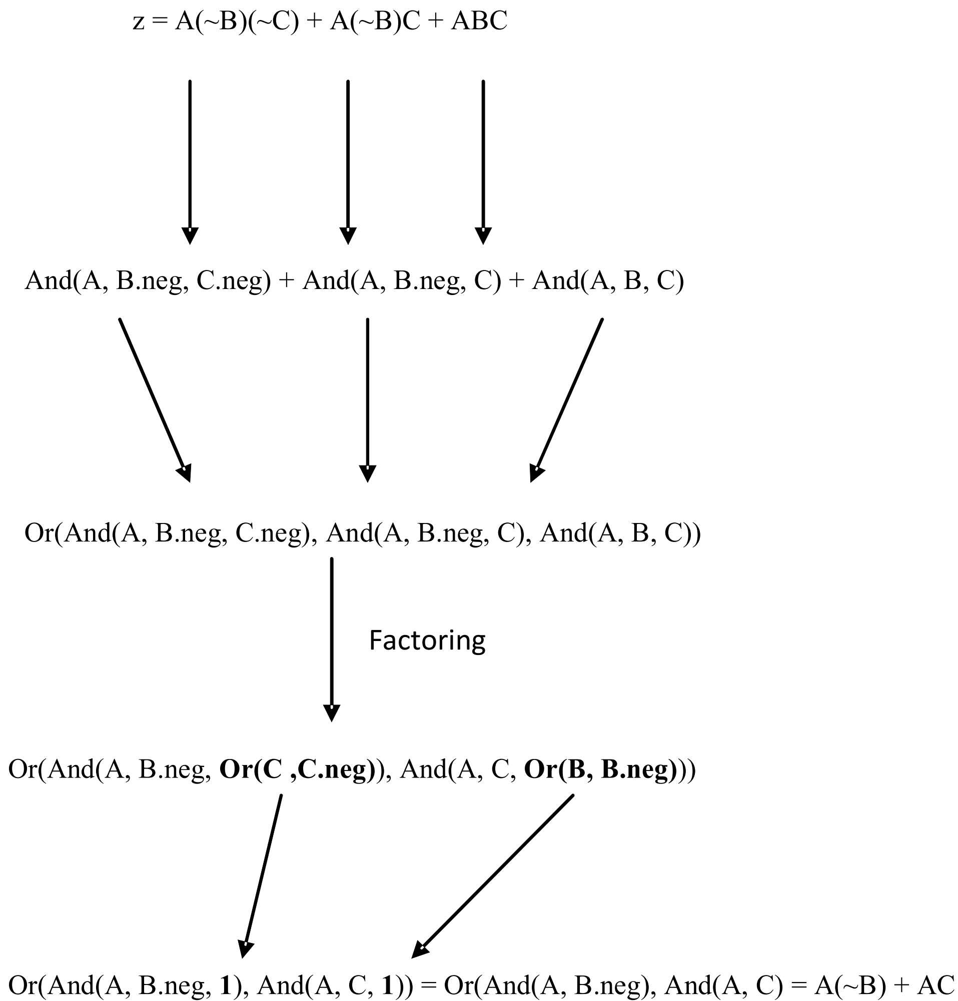


**Figure 8** Object creation and factoring process

It should be noted that the result is not exactly the same as the result earlier noted. This is because the factoring algorithm first finds the largest factors and then reconstructs the equation. If the **factor()** function was applied to the result in **Figure 8**, the result would be the same. This would suggest the addition of a *loop* to the factoring process, which will attempt to factor the equation again. This change may be made in the future.

## 4. The User Experience

The current goal is to have a command line version of the program, where a student can enter a Boolean expression and see the steps to simplification. This would be useful for both simple and more complex expressions. In observing the simplification of a relatively simple expression, the student can learn the principles of the strangest parts of Boolean Algebra (that 1 + 1 = 1, for example). For more complex functions, the student will learn the general heuristic of Boolean Algebra simplification suggested in *Digital Systems***,** which is described in the pseudocode for the algorithm above. This allows the student the opportunity to have examples worked out during study outside of class, or as a review once the material has been covered in class.

Thus, the next step in the development of this program is to find the most informative way to deliver the output to the student. **Figure 9** shows an example of ideal output. This output is ideal because it only shows one change in the expression at a time, and it displays the theorem that allowed for the change.

| x(y + x) | |
|---|---|
| is equivalent to xy + xx | Theorem 13a |
| which is equivalent to xy + x | Theorem 3 |
| which is equivalent to x(y + 1) | Theorem 13a |
| which is equivalent to x(1) | Theorem 6 |
| which is equivalent to x | Theorem 2 |

Figure 9 – Ideal Output

This output scheme allows the student to see the expression change by one theorem at a time, which will help the student understand the fundamentals of Boolean algebra.

The end goal is to create a Graphical User Interface (GUI) for the program. A GUI could greatly simplify the program design, as input can be correctly initialized by the GUI. This would greatly simplify error checking as well, since the programmer has more control over what input is given to the program. A GUI will also have the benefit of being compatible with accessibility devices for the blind.

After the current fall class of students finished the chapters in the textbook directly associated with Boolean Algebra, they were given a laboratory exercise using this program. After the usual startup problems like getting Python correctly installed on their computers, they typed in six Boolean Algebra expressions and noted their results. One comment we didn't expect was, “Wow, this program is awesome! Why didn't we have this program during those earlier chapters?” Other common comments were:

1) Where's the Help for this program? (mostly done)
2) Could the program gracefully exit and show an error message instead of crashing? (being worked on)
3) Where's the GUI? (future work)
4) Why haven't you implemented all of the Theorems? (future work)

The Spring class used the same software package as the fall class did while studying Boolean Algebra simplification chapter and before the first test. Hence we had the same problems with set-up and the lack of help from the package. This time, though, the instructor included specific questions in the assignment to get a better understanding of the package's positive and negative points from the student's view.

Positive points were:

1) Faster than solving by hand
2) Individual steps were shown allowing the student to see the simplification (noted by most students in the class)

Negative points were:

1) Not all theorems implemented yet, hence answers were not in simplest form (noted by several students)
2) Package should allow both uppercase and lowercase letters
3) Answers were not in alphabetic order
4) Asterisks are required between terms rather than having implicit AND's

Neutral point was:
Did students' understanding of the material improve? Unknown as test grades were similar to fall semester when the package was introduced after the test. This may mean that the students will need additional practice using the package or an updated version of it before a difference will appear.

When the GUI version has be completed, this package will be made available for others to test. At this point, the authors are pleased with the results.

## 5. Conclusion

Given that students often have difficulty dealing with the oddities of Boolean Algebra concepts, an educational tool for Boolean simplification has been described. The program employs the OOP paradigm to define the rules as presented in the Boolean theorems. The general process of simplification has been discussed as well as the information about the ideal user experience.

In addition to tutoring students in Boolean Algebra, this program can serve as a lesson in the usefulness and flexibility of the OOP paradigm. Exposure to OOP solutions to complex problems will show the student the value of learning OOP.

## 6. Future Work

The program is currently functional in a test script. Our next step is to fix the various problems found and mentioned in the User Experience section and change the way the output is shown to the user. Then, the program can be made interactive by writing a parser and a GUI. During this time, extensive testing and debugging of the algorithms of the program will be ongoing to ensure correctness.

Acknowledgements:
The authors wish to thank the ten-week University of South Alabama's University Committee on Undergraduate Research (UCUR) program for the funds to start this project. The authors also thank the anonymous reviewers for their helpful suggestions for improving this paper.

References

1 O. Mirmotahari, C. Holmboe, J. Kaasboll (2003). "Difficulties Learning Computer Architecture" ItiCSE '03. New York, USA
2 G. Moss, R. Tocci, N. Widmer *et. al,* "Combinational Logic Circuits," in *Digital Systems: Principles and Applications.* Upper Saddle River, NJ: Pearson Prentice Hall, 2007
3 J. Freidl, *Mastering Regular Expressions: Powerful Techniques for Perl and Other Tools.* Cambridge: O'Reilly, 1997.
4 P. McGuire, "Getting Started with Pyparsing." O'Reilly Media, Inc., 2008.
5 M. Summerfield, *Programming in Python 3: a Complete Introduction to the Python Language.* Upper Saddle River, NJ: Addison-Wesley, 2009.
6 B. Eckel, "Introduction to Objects," in *Thinking In C++.* Upper Saddle River, NJ: Prentice Hall, 2000.
7 sympy.org

Appendix A: Boolean Theorems

1. $x * 0 = 0$
2. $x * 1 = x$
3. $x * x = x$
4. $x * (\sim x) = 0$
5. $x + 0 = x$
6. $x + 1 = 1$
7. $x + x = x$
8. $x + (\sim x) = 1$
9. $x + y = y + x$
10. $x * y = y * x$
11. $x + (y + z) = (x + y) + z = x + y + z$
12. $x(yz) = (xy)z = xyz$

13a. $x(y + z) = xy + xz$

13b. $(w + x)(y + z) = wy + xy + wz + xz$

14. $x + xy = x$

15a. $x + (\sim x)y = x + y$

15b. $(\sim x) + xy = (\sim x) + y$

16. $\sim(x + y) = (\sim x)(\sim y)$
17. $\sim(xy) = \sim x + \sim y$