2013

Mechanical Proof Checkers for First Order Logic

Russ Johnson
Grand Valley State University

ScholarWorks Citation
Johnson, Russ, "Mechanical Proof Checkers for First Order Logic" (2013). *Honors Projects*. 188.
https://scholarworks.gvsu.edu/honorsprojects/188

Mechanical Proof Checkers for First Order Logic

Russ Johnson
Grand Valley State University
Allendale, MI
`russjohnson09@gmail.com`
`https://github.com/russjohnson09/`

December 7, 2012

Abstract

This project has revolved around creating a proof checker in python. This proof checker checks proofs that are written in the Hilbert-style as seen in the book *Understanding Symbolic Logic* by Klenk. It does a reasonably good job, but could also be expanded upon a great deal. Also, because this was only for a two credit class for one semester there were certain things that I was not able to do simply due to the fact that my time was limited. I will be going over what I did during this semester, some of the research that I did, and explaining some of the basics of formal languages and logic.

1 Introduction

Over the course of the semester I have read from many different books and articles on the subject of logic and mechanical proof checking. It has been my goal to create my own proof checker written in my programming language of choice, python. I first would like to say what motivated me in choosing this topic and what I had hoped to be able to achieve. I will then go over the basics of propositional and predicate logic, how these two logics are connected and some things about them that I found interesting. Finally, I will go over the development of the python proof checker and some screenshots showing what it can be used for.

2 Motivation

My main motivation comes from having read *The QED Manifesto*. *The QED Manifesto* is an anonymous pamphlet written in 1994 and argues for the standardization of mathematical proofs. This standardization would allow for proofs to be checked for validity in an algorithmic manner and if some algorithm existed for checking proofs it would follow that they could be checked mechanically. It has been eighteen years since this paper was written and we are still nowhere close to having a single standard for mathematical proofs. This is probably due to the lack of interest in this area. The argument made in *The QED Manifesto Revisited* is that even if this could be done it would serve no practical benefit to the mathematical community.

I, however, feel that having a standard that could be used for all proofs could be of great benefit to the mathematical community. As proofs become more and more complex it becomes more and more necessary to employ computers. However, programs do not have the same rigour that is associated with mathematical proofs. The way that programs are now developed revolves around test cases, which are good enough in most situations, but in some cases bugs still appear. In order to prove that a program will act in the way that you expect it to you need to provide proof of this fact. In order to accept the validity of the computer based proof of the four-color theorem we need to be assured that the program that is running the proof has no bugs. I feel that the solution to this problem revolves around creating some standard for proofs and using this standard with proof checkers and proof assistants.

Without a standardization of mathematical proofs and a proof checker that can verify the validity of these proofs, advances in computer science will be underemployed in solving some of the tougher problems in mathematics. In the next two sections we will look at how propositional and predicate logic are defined.

3 Propositional Logic

Before we talk about what propositional logic is, we must first introduce some new terms. Propositional logic is an example of a formal system. We usually denote formal systems with

$\mathcal{F}$ and because propositional logic is also referred to as zeroth order logic we denote it as $\mathcal{F}_0$. Propositional logic, like all formal systems, contains a formal language $\mathcal{L}_0$ and a deductive system $\mathcal{D}_0$. The formal language of propositional logic is propositional calculus and the deductive system is defined by the axioms within $\mathcal{D}_0$. Because propositional logic is made up of these two parts, we say that $\mathcal{F}_0 = (\mathcal{L}_0, \mathcal{D}_0)$.

3.1 Propositional Calculus

Propositional calculus is the formal language of propositional logic and is denoted $\mathcal{L}_0$ as we have said before. Because $\mathcal{L}_0$ is just one element of the set of all formal languages $\mathcal{L}_0 \in \mathcal{L}$. Propositional calculus is made up of an alphabet and a grammar. This alphabet's elements are called letters and these letters are divided up into statement variables, statement constants, and signs. Statement variables are typically lower case letters such as p, q, and r, while statement constants are typically written with upper case letters such as A, B, and C. It is also possible to use something like Px to mean that x is president. The signs in $\mathcal{L}_0's$ alphabet are either connectives or punctuation. The following is a list of connectives in $\mathcal{L}_0$.

$\wedge$	conjunction
$\vee$	disjunction
$\supset$	conditional
$::$	biconditional
$\sim$	negation
$\top$	tautology
$\perp$	contradiction

The binary connectives are $\wedge$, $\vee$, $\supset$, and $::$. The unary connective is $\sim$. The two nullary connectives are $\top$ and $\perp$. There are two punctuation signs in propositional calculus's alphabet. These two are (and).

In addition to an alphabet, every formal language must have a formal grammar. There are different ways of expressing the same formal grammar. The only necessary condition for a formal grammar is that every string of letters is decidedly valid or invalid according to the formal grammar. Here is the formal grammar of propositional calculus written in Backus-Naur Form where p is an element the set of statement variables.

$$\begin{aligned} < formula > &::= p|\top|\perp \\ < formula > &::= \sim < formula > \\ < formula > &::= (< formula >< op >< formula >) \\ < op > &::= \wedge \mid \vee \mid \supset \mid :: \end{aligned}$$

3.2 Propositional Logic's Deductive System

The deductive apparatus used in propositional logic is split into a group of axioms, also called an axiom schemata, and rules of inference for deriving theorems in this logic. In the case of the book *Understanding Symbolic Logic* we are given eight rules of inference and ten replacement rules. This does not represent a minimal set, because some of these rules of inference can be derived from others and all of the replacement rules can be proven to be a result of the rules of inference. An example of a minimal set would be the three axioms Simplification, Frege, and

Transposition from page 49 of Margaris' book *First order mathematical logic* and Hamilton's postulated inference rule of propositional calculus from page 73 of *Logic for mathematicians.* Margaris' three axioms can be thought of as assumed tautologies. These three tautologies make up the following set where φ, ψ, and χ are formulas as defined in propositional calculus. $\{(\varphi \supset (\psi \supset \varphi)), ((\varphi \supset (\psi \supset \chi))((\varphi \supset \psi) \supset (\varphi \supset \chi))), ((\sim \varphi \supset \sim \psi) \supset (\psi \supset \varphi)\}$ Hamilton's postulate is represented in this way on `us.metamath.org`.

Hypotheses
φ
$\varphi \supset \psi$
Assertion
ψ

Margaris' third axiom is similar to the contraposition replacement rule, but is not a bijection. One direction of the bijection can be proven from the other and so it is redundant. The other two axioms are not directly connected to the rules of inference or the replacement rules. Hamilton's postulate is the same as the law of inference called Modus Ponens used in Klenk's book. It is the only inference rule needed to be a complete deductive system for propositional logic.

While being able to represent propositional logic's deductive system in a minimalistic manner has some appeal, it is much harder to work with than what we normally think of as the replacement rules and the rules of inference. Even though using these axioms means that we do not have a minimal set, these axioms are much more intuitional and easier to use. That is why I chose to work with these instead of the ones used on the metamath site.

3.3 Example Propositional Logic Proof

Here is an example proof coming from pages 118 and 119 of *Understanding Symbolic Logic.* It illustrates how sentences are symbolized and how a step by step Hilbert-style proof works.

John is weak with a stomach ache. If John is weak, has a stomach ache, and has a rash or a fever, then he either has food poisoning or mono. If John has food poisoning, then he must have eaten bean sandwiches at Joe's. John did not eat bean sandwiches at Joe's, but instead had lobster at Max's. John does not have a rash, but does have a fever. From these statements we can prove that John has mono.

We start this proof by first assigning letters to each of the simple sentences.

Symbol	**Simple Sentence**
W	John is weak.
S	John has a stomach ache.
R	John has a rash.
F	John has a fever.
P	John has food poisoning.
M	John has mono.
B	John ate bean sandwiches at Joe's.
L	John had a lobster at Max's.

Next we list all of our premises numbering each along the way.

Step	**Statement**	**Reasoning**
1.	$W \cdot S$	Premise
2.	$((W \cdot S) \cdot (R \vee F)) \supset (P \vee M)$	Premise
3.	$P \supset B$	Premise
4.	$\sim B \cdot L$	Premise
5.	$\sim R \cdot F / \therefore M$	Premise

Each premise is justified simply by being a premise. For the last premise it is common to also say what it is we are trying to prove which will be our last statement. Next we will use the rules of inference to prove that John has mono.

Step	**Statement**	**Reasoning**
6.	F	Simplification 5
7.	$R \vee F$	Addition 6
8.	$(W \cdot S) \cdot (R \vee F)$	Conjunction 1, 7
9.	$P \vee M$	Modus Ponens 2, 8
10.	$\sim B$	Simplification 4
11.	$\sim P$	Modus Tollens 3, 10
12.	M	Disjunctive Syllogism 9, 11

Each step is justified by one of the rules of inference and so under reasoning we state the rule of inference being used and the line numbers that are used to satisfy its hypotheses. The rules of inference are in the appendix which is taken from *Understanding Symbolic Logic*. For step 9 the hypotheses are satisfied because $((W \cdot S) \cdot (R \vee F)) \supset (P \vee M)$ (line 2) is a form instance of $p \supset q$ and $(W \cdot S) \cdot (R \vee F)$ is a form instance of p where p and q are simple variables. I used this fact later when putting together a second version of my proof checker. The last statement is M, which follows directly from the preceding statements which follow from the premises and so we have shown that John having mono follows from the premises. All together the Hilbert-style proof is

Step	**Statement**	**Reasoning**
1.	$W \cdot S$	Premise
2.	$((W \cdot S) \cdot (R \vee F)) \supset (P \vee M)$	Premise
3.	$P \supset B$	Premise
4.	$\sim B \cdot L$	Premise
5.	$\sim R \cdot F$	Premise
6.	F	Simplification
6.	F	Simplification 5
7.	$R \vee F$	Addition 6
8.	$(W \cdot S) \cdot (R \vee F)$	Conjunction 1, 7
9.	$P \vee M$	Modus Ponens 2, 8
10.	$\sim B$	Simplification 4
11.	$\sim P$	Modus Tollens 3, 10
12.	M	Disjunctive Syllogism 9, 11

4 Predicate Logic

Predicate logic is also a formal system. It is denoted as $\mathcal{F}_1$, because it is also called first order logic. It inherits all of the axioms from propositional calculus, which means that anything that has been proven for propositional logic is also true for predicate logic. We call the formal language for $\mathcal{F}_1$ predicate calculus. We do not go into the details of it here, but it is important to know that $\mathcal{L}_0 \subset \mathcal{L}_1$ and so everything that is a letter in propositional calculus is also a letter in predicate calculus. In addition to the letters in $\mathcal{L}_0$, predicate calculus has the quantifiers $\forall$ and $\exists$. The grammar for predicate calculus is very similar to propositional calculus, but with additional rules for $\forall$ and $\exists$.

4.1 Deductive Apparatus

With predicate logic we have all of the axioms and rules of inference from $\mathcal{D}_0$. In other words, $\mathcal{D}_0 \subset \mathcal{D}_1$. In addition to the axioms and rules of inference from propositional calculus, we have Universal Instantiation, Existential Instantiation, Universal Generalization, and Existential Generalization.

4.2 Example Predicate Logic Proof

A proof in predicate logic works by first taking the terms that are bound by quantifiers and making them instances. Then we can use the same rules that we used in propositional logic to make changes to the statements. Finally, we apply the generalization rules to reach our conclusion. Here is an example proof for some Quakers are ambitious from *Understanding Symbolic Logic*.

Step	**Statement**	**Reasoning**
1.	$(x)(Px \supset Ax)$	Premise
2.	$(\exists x)(Px \cdot Qx)$	Premise
3.	$Pa \cdot Qa$	Existential Instantiation 2
4.	$Pa \supset Aa$	Universal Instantiation 1
5.	Pa	Simplification 2
6.	Aa	Modus Ponens 4,5
7.	Qa	Simplification 3
8.	$Qa \cdot Aa$	Conjunction 7,6
9.	$(\exists x)(Qx \cdot Ax)$	Existential Generalization 8

5 Proof Checking Program

This was what I spent the majority of my time on this semester. I have made three different versions of this program.

5.1 Development

My first attempt in checking proofs followed this approach. First, I would break up the text document that contains a proof into individual lines and put each of these lines a dictionary. The key for each element in the dictionary would be the line number and the value would be a list containing the rest of the line broken up by the statement, the reasoning, and the referenced lines. I would then go through this dictionary line by line, calling the appropriate method according to the rule being used, and check if this line followed logically from the preceding lines. Although it worked for the first few rules, this ended up making the code much too difficult to read and so I scrapped most of it and started over.

My second attempt started with making sure that the methods for applying the rules of inference were as simple as possible. The simplest way that I could think of was having each method compare two or three strings, depending on the rule of inference being used, and return a true or false. The strings themselves would have to confirmed to be well formed formulas using another method. So, I worked on making some methods that would apply the rules of inference to strings. I then decided that I needed a quicker way of checking these methods, so I made a python unittest to go through cases for each of the rules. At this point I still did not have a method for confirming proofs found in text files so I decided that since I had enough rules of inference I should probably try that next. The text files are in the following format.

```
4.\tB\t\tMP 1,2
```

The first part is the line number followed by a period and separated from the next element by at least one tab. This part is actually discarded (its position in the list containing the proof is sufficient), but it is still a necessary part in writing out the proofs. The next part is the formula. The third part is separated from the second by at least one tab. The last element is a list of referenced lines and is separated from the third by a single space. Eventually the references are replaced by the strings that they reference, the rule of inference becomes the first element, and the conclusion becomes the last element. The resulting list follows.

```
[\mp","A->B","A","B"]
```

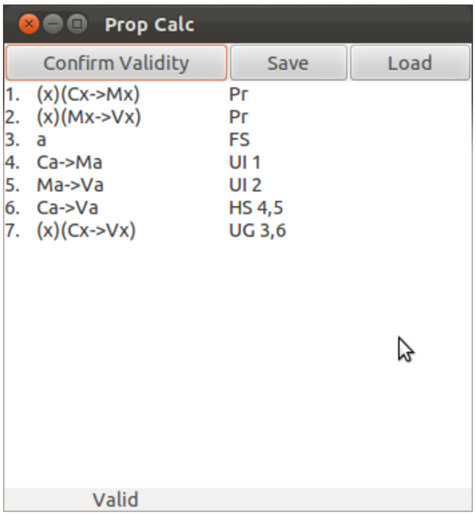
The first element of the list is the name of the method called and the last three elements are arguments for this method. After creating the methods for taking a proof text file, I took some examples from the book and used them as test cases. I used this same process for the replacement rules.

I then worked with conditional proofs and indirect proofs (proofs by contradiction). Conditional proofs can be nested, but one cannot start inside one conditional proof and end outside another. So, I wrote a method to make sure that the structure of the conditional proof is valid as well as a method to check the conclusion made by the conditional proof.

Working with predicate logic was much more challenging than propositional logic. For the universal and existential instantiation and generalization rules, you have to keep track of flagged variables. I did this using a dictionary to make sure that flagged variables always referred to the same thing. If a single flagged variable references to different things, you can easily make invalid proofs such as all integers are odd and even.

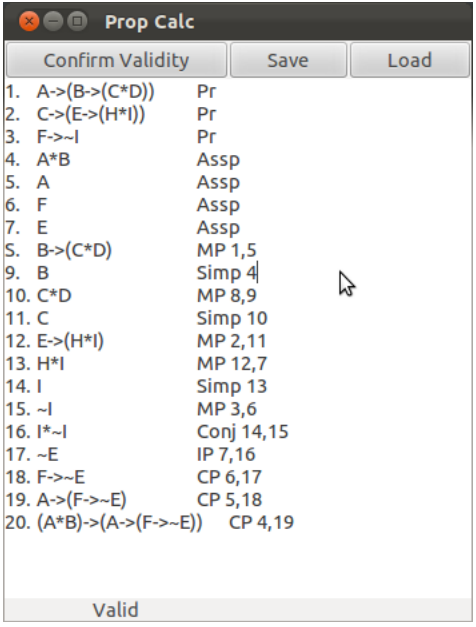
After making my second version of the proof checker, I realized that the programs code could be shortened a great deal if I made a single general method for all of the inference and replacement rules. What this general method does is check that each statement being referenced and the conclusion is a form instance of the hypotheses and conclusion of the rule of inference or replacement rule being used. With this one method (named confirm) I was able to reduce each of the methods for the reference rules to a single call to the confirm method. For this third version of the program, I also split the program into classes to form a more logical structure. There is a predicate class which inherited from the propositional class. There is also a class that stores the parts of each wff. Its attributes include the main operator, the wff to the left of the main operator, and the wff to the right of the main operator. There is a class that checks the validity of the proof as well as stores all of the problems with it. Unfortunately, I had not yet refactored all of the parts for the predicate logic before the deadline and so the third version is still incomplete. It is however much more concise and easier to read than the second version.

5.2 Screenshots



```
1.  (x)(Cx->Mx)     Pr
2.  (x)(Mx->Vx)     Pr
3.  a               FS
4.  Ca->Ma          UI 1
5.  Ma->Va          UI 2
6.  Ca->Va          HS 4,5
7.  (x)(Cx->Vx)     UG 3,6
```

Valid



```
1.  A->(B->(C*D))      Pr
2.  C->(E->(H*I))      Pr
3.  F->~I              Pr
4.  A*B                Assp
5.  A                  Assp
6.  F                  Assp
7.  E                  Assp
S.  B->(C*D)           MP 1,5
9.  B                  Simp 4
10. C*D                MP 8,9
11. C                  Simp 10
12. E->(H*I)           MP 2,11
13. H*I                MP 12,7
14. I                  Simp 13
15. ~I                 MP 3,6
16. I*~I               Conj 14,15
17. ~E                 IP 7,16
18. F->~E              CP 6,17
19. A->(F->~E)         CP 5,18
20. (A*B)->(A->(F->~E))   CP 4,19
```

Valid

6 Conclusion

When I first started this project I had hoped that I would be able to come up with some useful application for it in Mathematics. Computer algebra systems may be the closest fit for this. A logic system embedded into one of these programs that could confirm that a method will produce the correct output could be useful for a project like Mathics which attempts to mimic some of Mathematica's functionality.

References

[1] Georges Gonthier. Formal proof – the Four-Color theorem. *Notices of the American Mathematical Society.*, page 1382, 2008.

[2] A. G Hamilton. *Logic for mathematicians.* Cambridge University Press, Cambridge [Eng.]; New York, 1978.

[3] Michael Huth and Mark Ryan. *Logic in computer science : modelling and reasoning about systems.* Cambridge University Press, Cambridge, [England]; New York, 2004.

[4] Angelo Margaris. *First order mathematical logic.* Blaisdell Pub. Co., Waltham, Mass., 1967.

[5] Norman Megill. Metamath. http://users.shore.net/%7Endm/java/mm.html.

[6] Norman D Megill. A finitely axiomatized formalization of predicate calculus with equality. *Notre Dame Journal of Formal Logic*, 36(3):435–453, 1995.

[7] F. Wiedijk, Roman Matuszewski, and Anna Zalewska. The QED manifesto revisited, 2007.

Modus Ponens (M.P.)

$$\begin{array}{c} p \supset q \\ p \\ \hline / \therefore q \end{array}$$

Modus Tollens (M.T.)

$$\begin{array}{c} p \supset q \\ \sim q \\ \hline / \therefore \sim p \end{array}$$

Hypothetical Syllogism (H.S.)

$$\begin{array}{c} p \supset q \\ q \supset r \\ \hline / \therefore p \supset r \end{array}$$

Simplification (Simp.)

$$\begin{array}{c} p \cdot q \\ \hline / \therefore p \end{array} \qquad \begin{array}{c} p \cdot q \\ \hline / \therefore q \end{array}$$

Conjunction (Conj.)

$$\begin{array}{c} p \\ q \\ \hline / \therefore p \cdot q \end{array}$$

Dillemma (Dil.)

$$\begin{array}{c} p \supset q \\ r \supset s \\ p \lor r \\ \hline / \therefore q \lor s \end{array}$$

Disjunctive Syllogism (D.S.)

$$\begin{array}{c} p \lor q \\ \sim p \\ \hline / \therefore q \end{array} \qquad \begin{array}{c} p \lor q \\ \sim q \\ \hline / \therefore p \end{array}$$

Addition (Add.)

$$\begin{array}{c} p \\ \hline / \therefore p \lor q \end{array} \qquad \begin{array}{c} q \\ \hline / \therefore p \lor q \end{array}$$

Replacement Rules

Double Negation (D.N.)

$$p :: \sim\sim p$$

Duplication (Dup.)

$$p :: (p \lor p)$$
$$p :: (p \cdot p)$$

Commutation (Comm.

$$(p \lor q) :: (q \lor p)$$
$$(p \cdot q) :: (q \cdot p)$$

Association (Assoc.)

$$((p \lor q) \lor r) :: (p \lor (q \lor r))$$
$$((p \cdot q) \cdot r) :: (p \cdot (q \cdot r))$$

Contraposition (Contrap.)

$$(p \supset q) :: (\sim q \supset \sim p)$$

DeMorgan's (DeM.)

$$\sim (p \lor q) :: (\sim p \cdot \sim q)$$
$$\sim (p \cdot q) :: (\sim p \lor \sim q)$$

Biconditional Exchange (B.E.)

$$(p \equiv q) :: ((p \supset q) \cdot (q \supset p))$$

Conditional Exchange (C.E.)

$$(p \supset q) :: (\sim p \lor q)$$

Distribution (Dist.)

$$(p \cdot (q \lor r)) :: ((p \cdot q) \lor (p \cdot r))$$
$$(p \lor (q \cdot r)) :: ((p \lor q) \cdot (p \lor r))$$

Exportation (Exp.)

$$((p \cdot q) \supset r) :: (p \supset (q \supset r))$$

Necessary Restrictions

A. Preliminary Definitions

1. ϕx is a propositional function of x, simple or complex. If complex, it is assumed that it is enclosed in parentheses, so that the scope of any prefixed quantifier extends to the end of the formula.
2. ϕa is a formula just like ϕx, except that every occurrence of x in ϕx has been replaced by an a.
3. An instance of a general formula is the result of deleting the initial quantifier and replacing each variable bound by that quantifier uniformly with some name.
4. An a-flagged subproof is a subproofthat begins with the words ”flag a” and ends with some instance containing a.

B. The Four Quantifier Rules

Universal Instantiation (U.I.)

$$\frac{(x)\phi x}{/\therefore \phi a}$$

Existential Instantiation (E.I.)

$$\frac{(\exists x)\phi x}{/\therefore \phi a} \quad \text{provided we flag } a$$

Universal Generalization (U.G.)

$$\begin{array}{|l} \longrightarrow \text{flag } a \\ \quad . \\ \quad . \\ \quad . \\ \phi a \\ \hline \end{array}$$
$$/\therefore \ (x)\phi x$$

Existential Generalization (E.G.)

$$\frac{\phi a}{/\therefore (\exists x)\phi x}$$

C. Flagging Restrictions

1. A letter being flagged must be new to the proof; that is, it may not appear, either in a formula or as a letter being flagged, previous to the step in which it gets flagged.
2. A flagged letter may not appear either in the premises or in the conclusion of a proof.
3. A flagged letter may not appear outside the subproof in which it gets flagged.

7 Prop.py

The script `prop.py` contains the main program which is split up into three parts.

7.1 Rules of Inference

The rules of inference make up the axioms of zeroth order logic and are implemented before everything else.

```
#!/usr/bin/python
import re
from pyparsing import Literal,Word,ZeroOrMore,Forward,nums,oneOf,Group,srange

class Prop():
    def __init__(self):
        pass
        self.flagset = set()

#The following two methods define wffs and check them in the proof.
    def syntax(self):
        op = oneOf( '\\/ -> * ::')
        lpar = Literal('(')
        rpar = Literal( ')' )
        statement = Word(srange('[A-Z]'),srange('[a-z]'))
        expr = Forward()
        atom = statement | lpar + expr + rpar
        expr << atom + ZeroOrMore( op + expr )
        expr.setResultsName("expr")
        return expr

    def confirm_wff(self, form1):
        expr = self.syntax()
        form1 = self.strip_form(form1)
        try:
            result = ''.join(list(expr.parseString(form1)))
        except:
            result = None
        return result == form1


#Rules of inference.
    def mp(self, form1, form2, form3):
        """
        Checks for the correct use of Modus Ponens.
        Both A->B,A,B and A,A->B,B are valid.
        """
        return (self.__mp_one_way(form1, form2, form3) or
                self.__mp_one_way(form2, form1, form3))



    def __mp_one_way(self, form1, form2, form3): #Modus Ponens
        """
        The first formula is split up and compared to the
    other two formulas.
    """

    a = self.split_form(form1)


    try:
        return (a[2] == 'imp' and
                self.strip_form(form2) == a[0] and
                self.strip_form(form3) == a[1])
    except:
        return False



def mt(self, form1, form2, form3):
    return (self.__mt_one_way(form1, form2, form3) or
            self.__mt_one_way(form2, form1, form3))

def __mt_one_way(self, form1, form2, form3): # Modus Tollens
    """
    The first formula is split and compared to the other
    two.
    """

    a = self.split_form(form1)

    strip2 = self.strip_form(form2)
    strip3 = self.strip_form(form3)

    try:
        return (a[2] == 'imp' and
                strip2[0] == '~' and
                strip3[0] == '~' and
                a[0] == self.strip_form(strip3[1:]) and
                a[1] == self.strip_form(strip2[1:])
                )
    except:
        return False

def hs(self, form1, form2, form3):
    return (self.__hs_one_way(form1, form2, form3) or
            self.__hs_one_way(form2, form1, form3))

def __hs_one_way(self, form1, form2, form3): #Hypothetical Syllogism
    """
    All three formulas are split and compared to one another.
    """
    a = self.split_form(form1)
    b = self.split_form(form2)
    c = self.split_form(form3)

    try:
        return (a[2] == 'imp' and
                b[2] == 'imp' and
                c[2] == 'imp' and
                a[0] == c[0] and
                a[1] == b[0] and
                b[1] == c[1])

    except:
        return False




def simp(self, form1, form2): #Simplification
    """
    The first formula is split and compared to the
    second.
    """

    a = self.split_form(form1)
    strip2 = self.strip_form(form2)

    try:
        return (a[2] == 'and' and
                (a[0] == strip2 or
                 a[1] == strip2)
                )

    except:
        return False

def conj(self, form1, form2, form3): #Conjunction
    """
    Conjunction uses the simplification method to
    validate that form1 is a simplification of form3
    and form2 is a simplification of form3.
    """
    return self.simp(form3,form1) and self.simp(form3,form2)


def dil(self, form1, form2, form3, form4):
    return (self.__dil_one_way(form1, form2, form3, form4) or
            self.__dil_one_way(form1, form3, form2, form4) or
            self.__dil_one_way(form2, form1, form3, form4) or
            self.__dil_one_way(form2, form3, form1, form4) or
            self.__dil_one_way(form3, form1, form2, form4) or
            self.__dil_one_way(form3, form2, form1, form4))


def __dil_one_way(self, form1, form2, form3, form4): #Dilemma
    tup1 = self.split_form(form1)
    tup2 = self.split_form(form2)
    tup3 = self.split_form(form3)
```

```
        tup4 = self.split_form(form4)

        return ((tup1[2], tup2[2], tup3[2], tup4[2]) == (’imp’,’imp’,’or’,’or’)
                and {tup3[0],tup3[1]} == {tup1[0],tup2[0]}
                and {tup4[0],tup4[1]} == {tup1[1],tup2[1]})

    def ds(self, form1, form2, form3):
        return (self.__ds_one_way(form1, form2, form3) or
                self.__ds_one_way(form2, form1, form3))

    def __ds_one_way(self, form1, form2, form3): #Disjunctive Syllogism

        try:

            a = self.split_form(form1)
            b = self.split_form(form2)
            c = self.strip_form(form3)

            return ((a[2] == ’or’ and
                    b[1] == ’neg’) and
                    ((a[0] == b[0] and
                    a[1] == c)
                     or
                    (a[1] == b[0] and
                    a[0] == c)
                     ))
        except:
            return False


    def add(self, form1, form2): #Addition

        a = self.split_form(form2)
        strip1 = self.strip_form(form1)

        try:
            return (a[2] == ’or’ and
                    (a[0] == strip1 or a[1] == strip1))

        except:
            return False
```

7.2 Replacement Rules

The replacement rules can be derived from the axioms of zeroth order logic. We implement these rules in this next block of code.

```
#Replacement Rules
    def dn(self, form1, form2): #Double Negation
        return ((form1[:2] == ’~~’ and
                self.strip_form(form1[2:]) == self.strip_form(form2))
                or
                (form2[:2] == ’~~’ and
            self.strip_form(form2[2:]) == self.strip_form(form1))
            )


def dup(self, form1, form2):
    return (self.__dup1(form1, form2) or
            self.__dup1(form2, form1))

def __dup1(self, form1, form2): #Duplication
    a = self.split_form(form2)

    return (self.conj(form1, form1, form2) or(
            self.add(form1, form2) and
            a[0] == a[1]))

def comm(self, form1, form2): #Commutation

    a = (self.find_main_op(form1)[0], self.find_main_op(form1)[1],
         self.find_main_op(form2)[1])

    return ((a[1],a[2]) in [('or','or'),('and','and')] and
             form1[a[0]+1:] + form1[a[0]] + form1[:a[0]] == form2)


def assoc(self,form1, form2): #Association
    """
    First we will decide which way the association rule is applied.
    Then we will apply it and finally we will decide if
    it is valid.
    """

    a = self.split_form(form1)

    try:
        if a[2] == 'or':
            return self.assocor(form1,form2)
        else:
            return self.assocand(form1,form2)
    except:
        return False

def assocor(self, form1, form2):


    try:
        a = self.split_form(form1)
        b = self.split_form(form2)
        c = self.split_form(a[0])
        d = self.split_form(b[1])

        if (a[1] == d[1] and
            b[0] == c[0] and
            c[1] == d[0] and
            (a[2],b[2],c[2],d[2]) ==
            ('or','or','or','or')):

            return True

    except:
        pass

    try:
        a = self.split_form(form1)
        b = self.split_form(form2)
        c = self.split_form(a[1])
        d = self.split_form(b[0])


        if (a[0] == d[0] and
            b[1] == c[1] and
            c[0] == d[1] and
            (a[2],b[2],c[2],d[2]) ==
            ('or','or','or','or')):

            return True

    except:
        pass

    return False

def assocand(self, form1, form2):

    try:
        a = self.split_form(form1)
        b = self.split_form(form2)
        c = self.split_form(a[0])
        d = self.split_form(b[1])

        if (a[1] == d[1] and
            b[0] == c[0] and
            c[1] == d[0] and
            (a[2],b[2],c[2],d[2]) ==
            ('and','and','and','and')):

            return True

    except:
        pass

    try:
        a = self.split_form(form1)
        b = self.split_form(form2)
        c = self.split_form(a[1])
        d = self.split_form(b[0])
        if (a[0] == d[0] and
            b[1] == c[1] and
            c[0] == d[1] and
            (a[2],b[2],c[2],d[2]) ==
            ('and','and','and','and')):

            return True

    except:
        pass

    return False


def contra(self, form1, form2):
    return (self.__contra1(form1, form2) or
            self.__contra1(form2, form1))

def __contra1(self, form1, form2): #Contraposition

    a = self.split_form(form1)
    b = self.split_form(form2)

    try:
        return (self.strip_form(b[0][1:]) == a[1] and
                self.strip_form(b[1][1:]) == a[0] and
                a[2] == 'imp' and
                b[2] == 'imp')

    except:
        return False



def dem(self, form1, form2):

    return (self.__dem1(form1, form2) or
            self.__dem1(form2, form1))

def __dem1(self, form1, form2): #DeMorgan's
    try:
        split_form1 = self.split_form(form1)
        split_form2 = self.split_form(form2)
        if split_form1[1] != 'neg':
            return False
        split_form1 = self.split_form(split_form1[0])
        if split_form1[2] == 'and':
            return self.__demand(split_form1, split_form2)
        else:
            return self.__demor(split_form1, split_form2)

    except:
        return False

def __demor(self, split_form1, split_form2):
    a = split_form1
    b = split_form2

    try:
        return ('~' + a[0] == b[0] and
                 '~' + a[1] == b[1] and b[2] == 'and')

    except:
        return False

def __demand(self, split_form1, split_form2):

    a = split_form1
    b = split_form2

    try:
        return ('~' + a[0] == b[0] and
                 '~' + a[1] == b[1] and b[2] == 'or')

    except:
        return False


def be(self, form1, form2):
    try:
        return (self.__be1(form1, form2) or
            self.__be1(form2, form1))
    except:
        return False

def __be1(self, form1, form2):
    a = self.split_form(form1)
    b = self.split_form(form2)
    c = self.split_form(b[0])
    d = self.split_form(b[1])

    return (a[2] == 'equiv' and
            b[2] == 'and' and
            c[2] == 'imp' and
            d[2] == 'imp' and
            a[0] == c[0] and
            a[0] == d[1] and
            a[1] == c[1] and
            a[1] == d[0]
            )


def ce(self, form1, form2):

    try:
        return (self.__ce1(form1, form2) or
            self.__ce1(form2, form1))
    except:
        return False

def __ce1(self, form1, form2):
    a = self.split_form(form1)
    b = self.split_form(form2)

    return (a[2] == 'imp' and
            b[2] == 'or' and
            '~' + a[0] == b[0] and
            a[1] == b[1])



def dist(self, form1, form2):
    try:
        return (self.__dist1(form1, form2) or
                self.__dist1(form2, form1))
    except:
        return False

def __dist1(self, form1, form2):

    try:

        a = self.split_form(form1)
        b = self.split_form(form2)
        c = self.split_form(a[1])
        d = self.split_form(b[0])
        e = self.split_form(b[1])

        if a[2] == 'and':
            return self.__distand(a,b,c,d,e)
        else:
            return self.__distor(a, b, c, d, e)

    except:
        return False

def __distand(self,a,b,c,d,e):
    try:

        return (a[2] == 'and' and
                b[2] == 'or' and
                c[2] == 'or' and
                d[2] == 'and' and
                e[2] == 'and' and
                a[0] == d[0] and
                c[0] == d[1] and
                c[1] == e[1] and
                d[0] == e[0]
                )

    except:
        return False


def __distor(self,a,b,c,d,e):
    try:

        return (a[2] == 'or' and
                b[2] == 'and' and
                c[2] == 'and' and
                d[2] == 'or' and
                e[2] == 'or' and
                a[0] == d[0] and
                c[0] == d[1] and
                c[1] == e[1] and
                d[0] == e[0]
                )

    except:
        return False


def exp(self, form1, form2): #Exportation
    try:
        return (self.__exp1(form1, form2) or
                self.__exp1(form2, form1))

    except:
        return False

def __exp1(self, form1, form2):
    try:
        a = self.split_form(form1)
        b = self.split_form(form2)
        c = self.split_form(a[0])
        d = self.split_form(b[1])

        return (a[2] == 'imp' and
                b[2] == 'imp' and
                c[2] == 'and' and
                d[2] == 'imp' and
                a[1] == d[1] and
                b[0] == c[0] and
                c[1] == d[0])

    except:
        return False
```

7.3 Conditional Proof

This next block of code implements conditional proofs for zeroth order logic. It is the last thing we implement for zeroth order logic before moving on to first order logic.

```
#Conditional proof methods and structural checks.
    def cp(self, form1, form2, form3):
        a = self.split_form(form3)
        form1 = self.strip_form(form1)
        form2 = self.strip_form(form2)

        return (form1 == a[0] and
                form2 == a[1] and
                a[2] == ’imp’)

    def ip(self, form1, form2, form3):
        form1 = self.strip_form(form1)
        form3 = self.strip_form(form3)
        return (self.__is_contradiction(form2) and
                (form1 == ’˜’ + form3 or
                     form3 == ’˜’ + form1 or
                     form1 == ’˜(’ + form3 + ’)’ or
                     form3 == ’˜(’ + form1 + ’)’))


    def __is_contradiction(self,form1):
        a = self.split_form(form1)
        try:
            return (a[2] == ’and’ and
                    (a[0] == ’˜’ + a[1] or
                     a[1] == ’˜’ + a[0] or
                     a[0] == ’˜(’ + a[1] + ’)’ or
                     a[1] == ’˜(’ + a[0] + ’)’))

        except:
            return False

    def confirm_structure(self, ip, refs):
        for tuple1 in refs:
            if not len(tuple1) == 1:
                lst1 = []

                for tuple2 in ip:
                    # if the line number is outside the scope of
                    # an assumption we must be cautious
                    if tuple1[0] > tuple2[1]:
                        lst1.append(tuple2)
                for ref in tuple1[1:]:
                    if self.__is_between(ref,lst1):
                        return False
        return True


    def __is_between(self,ref,lst1):
        if lst1:
            for range1 in lst1:
                if (ref <= range1[1] and
                    ref >= range1[0]):
                    return True
        return False

    def ip_do_not_cross(self,lst1):
        lst2 = []
        for element in lst1:
            if (element[1] == 'assp' or
                element[1] == 'fs'):
                lst2.append(element[0])
            if element[1] in ('ip','cp', 'ug'):
                x = lst2.pop()
                if not element[2] == x:
                    return False
        return not bool(lst2)
```

7.4 Predicate Logic

Next we implement the rules of first order logic UG, EG, UI, and EI.

```
#Predicate Logic Methods

    def ui(self, form1, form2):

        try:
            dict1 = {}
            form1 = self.strip_form(form1)
            form2 = self.strip_form(form2)
            var = form1[1]
            form1 = form1[4:-1]
            for i in range(len(form1)):
                if form1[i] == var:
                    if not dict1.has_key(var):
                        dict1[var] = form2[i]
                    else:
                        return re.sub(var,dict1[var],form1) == form2

        except:
            return False


    def eg(self, form1, form2):

        try:
            dict1 = {}
            form1 = self.strip_form(form1)
            form2 = self.strip_form(form2)
            var = form2[8]
            form2 = form2[11:-1]
            for i in range(len(form2)):
```

```
            if form2[i] == var:
                if not dict1.has_key(var):
                    dict1[var] = form1[i]
                else:
                    return re.sub(var,dict1[var],form2) == form1

    except:
        return False

def ei(self, form1, form2):
    try:
        dict1 = {}
        form1 = self.strip_form(form1)
        form2 = self.strip_form(form2)
        var = form1[8]
        form1 = form1[11:-1]
        for i in range(len(form1)):
            if form1[i] == var:
                if not dict1.has_key(var):
                    dict1[var] = form2[i]
                    break
        if dict1[var] not in self.flagset and re.sub(var,dict1[var],form1) == form2:
            self.flagset.add(dict1[var])
            return True

    except:
        return False

def ug(self, flag, form1, form2): #Universal Generalization
    """
    This method compares the flagged variable, the first
    formula in the Universal Generalization subproof and
    the conclusion to the subproof. The flagged variable
    is discarded if the proof is valid.
    """
    try: #Anything that goes wrong here means that something is incorrect.
        form1 = self.strip_form(form1) #Get rid of access space
        form2 = self.strip_form(form2)
        var = form2[1]
        form2 = form2[4:-1]
        bool1 = bool(re.sub(var,flag,form2) == form1)
        if bool1:
            self.flagset.discard(flag) #Once the ug subproof is complete flagged variable can be usec
            return True
        else:
            return False

    except:
        return False

def fs(self, flag):

    try:
            bool1 = bool(flag not in self.flagset)
            if bool1:
                self.flagset.add(flag)
                return True
            else:
                return False

        except:
            return False
```

7.5 Quantifier Negation

In this next block we implement the quantifier negation rules.

```
#QN Rules

    def qn1(self, form1, form2):
        return (self.__qn1oneway(form1, form2) or
                self.__qn1oneway(form2, form1))


    def __qn1oneway(self, form1, form2):

        try:

            form1 = self.strip_form(form1)
            form2 = self.strip_form(form2)
            var = form1[2]

            return (form1[0:4] == ’˜(’+var+’)’ and
                    form2[0:10] == "(\\exists" + var + ")" and
                    self.split_form(’˜’ + form1[4:]) == self.split_form(form2[10:]))

        except:
            return False


    def qn2(self, form1, form2):
        try:
            return (self.__qn2oneway(form1, form2) or
                    self.__qn2oneway(form2, form1))
        except:
            return False


    def __qn2oneway(self, form1, form2):

        try:
            form1 = self.strip_form(form1)
            form2 = self.strip_form(form2)
            var = form2[1]
        return (form1[0:11] == '~(\\exists'+var+')' and
                self.split_form('~('+form1[11:]+')') == self.split_form(form2[3:])
                and
                form2[0:3] == '('+var+')')
    except:
        return False


def qn3(self, form1, form2):
    return (self.__qn3oneway(form1, form2) or
            self.__qn3oneway(form2, form1))


def __qn3oneway(self, form1, form2):

    try:
        form1 = self.strip_form(form1)
        form2 = self.strip_form(form2)
        var = form1[2]

        return (form1[0:4] == '~('+var+')' and
                form2[0:10] == '(\\exists'+var+')' and
                self.split_form(form1[4:]) == self.split_form('~('+form2[10:]+')'))

    except:
        return False


def qn4(self, form1, form2):
    return (self.__qn4oneway(form1, form2) or
            self.__qn4oneway(form2, form1))


def __qn4oneway(self, form1, form2):

    try:
        form1 = self.strip_form(form1)
        form2 = self.strip_form(form2)
        var = form2[1]

        return (form1[0:11] == '~(\\exists'+var+')' and
                self.split_form(form1[11:]) == self.split_form('~('+form2[3:]+')')
                and
                form2[0:3] == '('+var+')')
    except:
        return False
```

7.6 Utilities

Finally, we have some methods that are used at various parts in the program to make things easier.

```
#Utilities used by above methods.
    def strip_form(self, form):
        form = re.sub(’ ’,’’,form)
        depth = 0
        for i,char in enumerate(form):
            if char == ’(’:
                depth += 1
            if char == ’)’:
                depth -= 1
            if depth == 0 and i == len(form) -1 and len(form) > 1:
                return self.strip_form(form[1:-1])
            elif depth == 0:
                break
        return form

    def find_main_op(self, form):
        """
        Takes a stripped formula as an argument. Not used
        directly. Used as a helper function to split_form.
        """
        subdepth = 0

        try:
            for i, char in enumerate(form):
                if char == ’(’:
                    subdepth += 1
                if char == ’)’:
                    subdepth -= 1
                if char == ’*’ and subdepth == 0:
                    return (i, ’and’)
                if char == ’\\’ and subdepth == 0:
                    if form[i+1] == ’/’:
                        return (i, ’or’)
                if char == ’:’ and subdepth == 0:
                    if form[i+1] == ’:’:
                        return (i, ’equiv’)
                if char == ’-’ and subdepth == 0:
                    if form[i+1] == ’>’:
                        return (i, ’imp’)

            if form[0] == ’~’:
                return (0, ’neg’)

        except:
            pass

    def split_form(self, form):
        """
        Splits a formula up into a tuple where the first element is the
        first part of the formula before the main operator, the second
        element is the second part of the formula after the main operator,
        and the third is the name of the main operator.
        """
        form = self.strip_form(form)
        a = self.find_main_op(form)

        #checks for None
        if not a:
            return None

        if a[1] == 'neg':
            return (self.strip_form(form[1:]), 'neg')


        if a[1] in ['or','imp','equiv']:
            tuple1 = (self.strip_form(form[:a[0]]), self.strip_form(form[a[0]+2:]),
                       a[1])

        else:
            tuple1 = (self.strip_form(form[:a[0]]), self.strip_form(form[a[0]+1:]),
                       a[1])

        return tuple1


#The following methods control the entire checking process.
    def confirm_validity(self, file1):
        lst1,ip,refs = self.proof_to_list(file1)
        lst2 = []

        for element in lst1:
            lst2.append(self.test(element))

        return (all(lst2) and
                self.confirm_structure(ip, refs)
                and
                self.ip_do_not_cross(lst1))


    def confirm_validity_string(self, file1):
        str1 = ("There is a problem with the " +
                "following lines: ")
        lst1,ip,refs = self.proof_to_list(file1)
        lst2 = []
        for element in lst1:
            lst2.append(self.test(element))
        if all(lst2):
            return "Proof is valid."
        else:
            for i,elem in enumerate(lst2):
                if elem == False:
                    str1 += str(i+1) + ", "
            return str1[:-2]

    def test(self, lst1):
```

```
    lst1[1]
    lst2 = []

    if lst1[0] == 'return False':
        return False

    if not (lst1[1] == 'pr' or lst1[1] == 'assp'
            or lst1[1] == 'fs'):
        str1 = "self." + lst1[1] + "(*lst2)"
        for x in lst1[2:]:
            lst2.append(x)
        lst2.append(lst1[0])
        try:
            return eval(str1)
        except:
            return False

    return True



def proof_to_list(self, file1):
    lst1 = []
    lst3 = []
    for line in file1:
        line = line.rstrip()
        line = re.sub(r"\t+","\t",line)
        line = re.sub(r"\.\t+","\t",line)
        lst2 = line.split("\t")
        if len(lst2) == 3:
            lst2 = lst2[1:]
            lst2 = self.convert1(lst2)
            lst1.append(lst2)
        elif re.sub(r"\s+","",lst2[0]):
            lst1.append(['return False','return False'])

    ip = self.__ip(lst1)
    refs = self.__refs(lst1)


    for element in lst1:
        lst2 = self.convert2(element, lst1)
        lst2[1] = lst2[1].lower()
        lst3.append(lst2)

    return (lst3,ip,refs)


def __refs(self,lst1):
    lst2 = []
    for i,element in enumerate(lst1):
        if (not isinstance(element[-1],int)
            or
            element[-3].lower() in ('cp','ip','ug')):

            lst2.append((i+1,))

        elif not isinstance(element[-2],int):
            lst2.append((i+1,element[-1]))
        else:
            lst2.append((i+1,element[-2],element[-1]))
    return lst2

def __ip(self,lst1):
    lst2 = []
    for element in lst1:
        try:
            if element[1].lower() in ('cp','ip','ug'):
                lst2.append((element[-2],element[-1]))
        except:
            pass
    return lst2


def flatten(self, x):
    result = []
    for el in x:
        if hasattr(el, "__iter__") and not isinstance(el, basestring):
            result.extend(self.flatten(el))
        else:
            result.append(el)
    return result


def convert1(self, lst1):
    lst1[1] = lst1[1].split(' ')
    try:
        lst1[1][1] = lst1[1][1].split(',')
    except:
        pass

    lst1 = self.flatten(lst1)

    if len(lst1) > 2:
        for i, x in enumerate(lst1[2:]):
            lst1[i + 2] = int(x)

    return lst1

def convert2(self, lst1, lst2):
    if not len(lst1) == 2: #Not a premise or assumption.
        for i, x in enumerate(lst1[2:]):
            lst1[i+2] = lst2[x - 1][0]

    return lst1
def prompt_for_file(self):
    filename = raw_input("Please enter the name of the file to be checked: ")
    return open(filename, ’r’)
```

8 Unit Tests

The unit tests are split up in a way similar to the main program `prop.py`. The order of the tests that are done is the same order as how they were implemented in `prop.py`.

```
#!/usr/bin/python
import unittest
from prop import Prop
class TestProp(unittest.TestCase):

    def setUp(self):
        self.prop = Prop()
        self.expr = self.prop.syntax()


#Tests for the Rules of Inference

    def test_mp(self):
        self.assertTrue(self.prop.mp("(A\\/B)->~C","A\\/B","~C"))
        self.assertTrue(self.prop.mp("A\\/B","(A\\/B)->~C","~C"))
        self.assertTrue(self.prop.mp("(A\\/B)->~C","A\\/B","~C"))
        self.assertFalse(self.prop.mp("(A\\/B)->C","A\\/B","~C"))
        self.assertFalse(self.prop.mp("A","A\\/B","~C"))

    def test_mt(self):
        self.assertTrue(self.prop.mt("Za->(Ha*Wa)","~(Ha*Wa)","~Za"))
        self.assertFalse(self.prop.mt("Za->(Ha*Wa)","Ha*Wa","~Za"))
        self.assertFalse(self.prop.mt("Za","Ha*Wa","~Za"))

    def test_hs(self):
        self.assertTrue(self.prop.hs("(A\\/B)->(C*D)","(C*D)->(~E*F)","(A\\/B)->(~E*F)"))
        self.assertTrue(self.prop.hs("(A\\/B)->(D)","(D)->(~E*F)","(A\\/B)->(~E*F)"))
        self.assertTrue(self.prop.hs("(A\\/B)->D","(D)->(~E*F)","(A\\/B)->(~E*F)"))
        self.assertFalse(self.prop.hs("(A\\/B)*(D)","(D)->(~E*F)","(A\\/B)->(~E*F)"))
        self.assertFalse(self.prop.hs("(A)","(D)->(~E*F)","(A\\/B)->(~E*F)"))

    def test_simp(self):
        self.assertFalse(self.prop.simp("(A\\/B)->~C", "~C"))
        self.assertFalse(self.prop.simp("(A\\/B)*~C", "C"))
        self.assertTrue(self.prop.simp("(A\\/B)*~C", "~C"))

    def test_conj(self):
        self.assertTrue(self.prop.conj("A\\/B","~(C->D)","(A\\/B)*~(C->D)"))
        self.assertFalse(self.prop.conj("A","B","A*B*C"))

    def test_dil(self):
        self.assertTrue(self.prop.dil("((A\\/B)->C)->(D\\/F)","(F::G)->(A->F)",
                                      "((A\\/B)->C)\\/(F::G)","(D\\/F)\\/(A->F)"))
        self.assertTrue(self.prop.dil("(F::G)->(A->F)","((A\\/B)->C)->(D\\/F)",
                                      "((A\\/B)->C)\\/(F::G)","(D\\/F)\\/(A->F)"))

    def test_ds(self):
        self.assertTrue(self.prop.ds("(~A\\/(B->C))\\/~D","~(~A\\/(B->C))","~D"))
        self.assertTrue(self.prop.ds("(~A\\/(B->C))\\/~D","~(~D)","(~A\\/(B->C))"))
        self.assertFalse(self.prop.ds("(~A\\/(B->C))\\/~D","(~D)","(~A\\/(B->C))"))
        self.assertFalse(self.prop.ds("A","(~D)","(~A\\/(B->C))"))
        self.assertTrue(self.prop.ds("~(B\\/C)\\/~(A*D)", "~~(A*D)", "~(B\\/C)"))
        self.assertTrue(self.prop.ds("~(B\\/C)\\/~(A\\/D)", "~~(A\\/D)", "~(B\\/C)"))

    def test_add(self):
        self.assertTrue(self.prop.add("(A->B)","(A->B)\\/C"))
        self.assertTrue(self.prop.add("((((A\\/B)\\/C)\\/D)\\/E)\\/F","(((((A\\/B)\\/C)\\/D)\\/E)\\/F)\\/
        self.assertFalse(self.prop.add("(B)","(A)"))


#Tests for the Replacement Rules
    def test_dn(self):
        self.assertTrue(self.prop.dn("A","~~A"))
        self.assertTrue(self.prop.dn("~~A","A"))
        self.assertTrue(self.prop.dn("~~(A->(B*C))","A->(B*C)"))

    def test_dup(self):
        self.assertTrue(self.prop.dup("A","A*A"))
        self.assertTrue(self.prop.dup("A","A\\/A"))
        self.assertTrue(self.prop.dup("A*A","A"))
        self.assertTrue(self.prop.dup("A\\/A","A"))

    def test_comm(self):
        self.assertTrue(self.prop.comm("E*F","F*E"))
        self.assertTrue(self.prop.comm("E*(F->G)","(F->G)*E"))

    def test_assoc(self):
        self.assertTrue(self.prop.assoc("(A*B)*C","A*(B*C)"))
        self.assertTrue(self.prop.assoc("A*(B*C)","(A*B)*C"))
        self.assertTrue(self.prop.assoc("(A*B)*(C->D)","A*(B*(C->D))"))
        self.assertTrue(self.prop.assoc("(A*B)*(C*D)","A*(B*(C*D))"))
        self.assertTrue(self.prop.assoc("(A\\/B)\\/C","A\\/(B\\/C)"))
        self.assertFalse(self.prop.assoc("",""))

    def test_assocand(self):
        self.assertTrue(self.prop.assocand("(A*B)*C","A*(B*C)"))
        self.assertTrue(self.prop.assocand("A*(B*C)","(A*B)*C"))
        self.assertTrue(self.prop.assocand("(A*B)*(C->D)","A*(B*(C->D))"))

    def test_assocor(self):
        self.assertTrue(self.prop.assoc("(A\\/B)\\/C","A\\/(B\\/C)"))
        self.assertFalse(self.prop.assoc("(A\\/B)*C","A\\/(B\\/C)"))

    def test_contra(self):
        self.assertTrue(self.prop.contra("A->B","~B->~A"))
        self.assertTrue(self.prop.contra("~B->~A","A->B"))
        self.assertFalse(self.prop.contra("",""))

    def test_dem(self):
        self.assertTrue(self.prop.dem("~(A*B)","~A\\/~B"))
        self.assertTrue(self.prop.dem("~(A\\/B)","~A*~B"))
        self.assertTrue(self.prop.dem("~A\\/~B","~(A*B)"))
        self.assertTrue(self.prop.dem("~A*~B","~(A\\/B)"))
        self.assertFalse(self.prop.dem("",""))

    def test_be(self):
        self.assertTrue(self.prop.be("A::B","((A->B)*(B->A))"))
        self.assertTrue(self.prop.be("((A->B)*(B->A))","A::B"))
        self.assertFalse(self.prop.be("",""))

    def test_ce(self):
        self.assertTrue(self.prop.ce("A->B","~A\\/B"))
        self.assertTrue(self.prop.ce("~A\\/B","A->B"))
        self.assertFalse(self.prop.ce("",""))

    def test_dist(self):
        self.assertTrue(self.prop.dist("A*(B\\/C)","(A*B)\\/(A*C)"))
        self.assertTrue(self.prop.dist("Ax*(By\\/Cz)","(Ax*By)\\/(Ax*Cz)"))
        self.assertTrue(self.prop.dist("(A*B)\\/(A*C)","A*(B\\/C)"))
        self.assertTrue(self.prop.dist("A\\/(B*C)","(A\\/B)*(A\\/C)"))
        self.assertFalse(self.prop.dist("",""))

    def test_exp(self):
        self.assertTrue(self.prop.exp("(A*B)->C","A->(B->C)"))
        self.assertTrue(self.prop.exp("A->(B->C)","(A*B)->C"))
        self.assertFalse(self.prop.exp("",""))



#Predicate Logic Methods

    def test_ui(self):
        self.assertTrue(self.prop.ui("(x)(Cx->Mx)","Ca->Ma"))
        self.assertTrue(self.prop.ui("(x)(Mx->Vx)","Ma->Va"))
        self.assertTrue(self.prop.ui("( x )( Mx -> Vx )","Ma->Va"))
        self.assertFalse(self.prop.ui("",""))

    def test_eg(self):
        self.assertTrue(self.prop.eg("Oa*Ea*Na","(\exists x)(Ox*Ex*Nx)"))

    def test_ei(self):
        self.assertTrue(self.prop.ei("(\exists x)(Ox*Ex*Nx)","Oa*Ea*Na"))
        self.assertFalse(self.prop.ei("(\exists x)(Ox*Ex*Nx)","Oa*Ea*Na"))

    def test_ug(self):
        self.assertTrue(self.prop.ug("a","Ca->Ma","(x)(Cx->Mx)"))

    def test_fs(self):
        self.assertTrue(self.prop.fs("a"))
        self.assertTrue(self.prop.fs("b"))

#Tests for conditional and indirect proofs.
    def test_cp(self):
        self.assertTrue(self.prop.cp("A","F->~E","A->(F->~E)"))
        self.assertTrue(self.prop.cp("Ax","Fy->~Ez","Ax->(Fy->~Ez)"))

    def test_ip(self):
        self.assertTrue(self.prop.ip("E","I*~I","~E"))
        self.assertTrue(self.prop.ip("E","~I*~~I","~E"))
        self.assertTrue(self.prop.ip("E","(I)*~I","~E"))
        self.assertTrue(self.prop.ip("E","(A->B)*~(A->B)","~E"))
        self.assertTrue(self.prop.ip("E","~(A->B)*~~(A->B)","~E"))
        self.assertTrue(self.prop.ip("Ex","~(Ax->By)*~~(Ax->By)","~Ex"))
        self.assertFalse(self.prop.ip("E","~~(A->B)*~~(A->B)","~E"))


#Tests for QN

    def test_qn(self):
        self.assertTrue(self.prop.qn1("~(x)(Ax)","(\\exists x)(~Ax)"))
        self.assertTrue(self.prop.qn1("(\\exists x)(~Ax)","~(x)(Ax)"))
        self.assertTrue(self.prop.qn1("(\\exists x)~(Ax)","~(x)(Ax)"))
        self.assertTrue(self.prop.qn2("~(\exists x)(Ax)","(x)(~Ax)"))
        self.assertTrue(self.prop.qn3("~(x)~(Ax)","(\exists x)(Ax)"))
        self.assertTrue(self.prop.qn4("~(\exists x)~(Ax)","(x)(Ax)"))

#Tests for utilities
    def test_split_form(self):
        self.assertEqual(self.prop.split_form("(F::G)->(A->F)"), ("F::G","A->F","imp"))
        self.assertEqual(self.prop.split_form("(F::G)->(A ->F)"), ("F::G","A->F","imp"))
        self.assertEqual(self.prop.split_form("(F::G) -> (A -> F )"), ("F::G","A->F","imp"))
        self.assertEqual(self.prop.split_form("~(A*B) -> C"), ("~(A*B)","C","imp"))
        self.assertEqual(self.prop.split_form("~(A\\/B) \\/ C"), ("~(A\\/B)","C","or"))
        self.assertEqual(self.prop.split_form("~(B\\/C)\\/~(A*D)"), ("~(B\\/C)","~(A*D)","or"))
        self.assertEqual(self.prop.split_form("~(A*B)"), ("A*B",'neg'))
        self.assertTrue(self.prop.split_form("A") == None)

    def test_find_main_op(self):
        self.assertEqual(self.prop.find_main_op("~(A*B)->C"), (6,'imp'))
        self.assertEqual(self.prop.find_main_op("(A\\/B)->~C"),(6,'imp'))
        self.assertEqual(self.prop.find_main_op("((A*B)\\/(A*B))->(B\\/C"),(14,'imp'))
        self.assertEqual(self.prop.find_main_op("(A\\/B)*~C"),(6,'and'))
        self.assertEqual(self.prop.find_main_op("(A\\/B)\\/~C"),(6,'or'))
        self.assertEqual(self.prop.find_main_op("(A\\/B)::~C"),(6,'equiv'))
        self.assertEqual(self.prop.find_main_op("~(A\\/D)"),(0,'neg'))
        self.assertEqual(self.prop.find_main_op("~~(A\\/D)"),(0,'neg'))
        self.assertTrue(self.prop.find_main_op("A") == None)
        self.assertTrue(self.prop.find_main_op("") == None)

    def test_strip_form(self):
        self.assertEqual(self.prop.strip_form("( A \\/ B ) -> ~C"),"(A\\/B)->~C")
        self.assertEqual(self.prop.strip_form(" ( ( A \\/ B ) -> ˜C ) "),"(A\\/B)->˜C")
        self.assertEqual(self.prop.strip_form(" ( ( A \\/ B )) "),"A\\/B")
        self.assertEqual(self.prop.strip_form("(F::G) -> (A -> F )"),"(F::G)->(A->F)")
        self.assertEqual(self.prop.strip_form("( x )(Cx ->Mx)"),"(x)(Cx->Mx)")

    def test_confirm_structure(self):
        self.assertTrue(self.prop.confirm_structure([(7,16),(6,17),(5,18),(4,19)],
                                                     [(1,),(2,),(3,),(4,),(5,),(6,),(7,),(8,1,5),
                                                      (9,4),(10,8,9),(11,10),(12,2,11),
                                                      (13,7,12),(14,13),(15,3,6),(16,14,15),
                                                      (17,),(18,),(19,),(20,)]))



    def test_confirm_validity(self):
        self.assertTrue(self.prop.confirm_validity(open("./proofs/proof.txt",’r’)))
        self.assertTrue(self.prop.confirm_validity(open("./proofs/proof2.txt",’r’)))
        self.assertTrue(self.prop.confirm_validity(open("./proofs/proof3.txt",’r’)))
        self.assertFalse(self.prop.confirm_validity(open("./proofs/proof4.txt",’r’)))
        self.assertFalse(self.prop.confirm_validity(open("./proofs/proof5.txt",’r’)))
        self.assertFalse(self.prop.confirm_validity(open("./proofs/proof6.txt",’r’)))
        self.assertTrue(self.prop.confirm_validity(open("./proofs/proof7.txt",’r’)))
        self.assertTrue(self.prop.confirm_validity(open("./proofs/proof8.txt",’r’)))
        self.assertTrue(self.prop.confirm_validity(open("./proofs/proof9.txt",’r’)))

    def test_confirm_validity_string(self):
        self.assertEqual(self.prop.confirm_validity_string(open("./proofs/proof6.txt",’r’)),
                         "There is a problem with the following lines: 5, 6")
#        print self.prop.confirm_validity(open("./proofs/proof16.txt",’r’))





    def test_confirm_wff(self):
        self.assertTrue(self.prop.confirm_wff("A\\/B"))
        self.assertTrue(self.prop.confirm_wff("(A\\/B)"))
        self.assertTrue(self.prop.confirm_wff("(A\\/B) -> C"))
        self.assertFalse(self.prop.confirm_wff("A\\/B)"))

if __name__ == ’__main__’:
    unittest.main()
```

9 Unfinished Version

This version of the main program was not finished in time to meet the end of the semester deadline, but it is here for anyone who is interested at looking at it. The main difference in this version is the use of the confirm method, which provides a general method for the rules of inference and the replacement rules.

```
#!/usr/bin/python
"""
```

```
Version 3 of prop.py
"""
import re
from itertools import permutations
#Has the rules of propositional calculus.
class Prop:
    def __init__(self):
        pass

    def confirm(self,forms, mold):
        """
        This generalized confirm method is used by all of
        the rules of inference methods.
        """

        if len(forms) != len(mold):
            return False

        variable_dict = {}

        for index in range(len(forms)):
            form1 = Wff(forms[index])
            form2 = Wff(mold[index])
            variable_dict = form1.similar(form2, variable_dict)
            if not variable_dict:
                return False

        return True

#Rules of inference.
    def mp(self, premise1, premise2, premise3):
        """
        Checks for the correct use of Modus Ponens.
        Both A->B,A,B and A,A->B,B are valid.
        """
        mp_rule = ('p->q','p','q')

        return (self.confirm((premise1, premise2, premise3), mp_rule) or
                self.confirm((premise2, premise1, premise3), mp_rule))




    def mt(self, premise1, premise2, conclusion):
        return (self.__mt_one_way(premise1, premise2, conclusion) or
                self.__mt_one_way(premise2, premise1, conclusion))

    def __mt_one_way(self, premise1,premise2,conclusion): # Modus Tollens

        premise1 = Wff(premise1)
        premise2 = Wff(premise2)
        conclusion = Wff(conclusion)
        try:
            return (premise1.main_op == 'imp' and
                    premise2.main_op == 'neg' and
                    conclusion.main_op == 'neg' and
                    premise1.left == conclusion.left and
                    premise1.right == premise2.left)
        except:
            return False

    def hs(self, premise1, premise2, conclusion):
        return (self.__hs_one_way(premise1, premise2, conclusion) or
                self.__hs_one_way(premise2, premise1, conclusion))

    def __hs_one_way(self, premise1,premise2,conclusion): #Hypothetical Syllogism
        """
        All three formulas are split and compared to one another.
        """

        premise1 = Wff(premise1)
        premise2 = Wff(premise2)
        conclusion = Wff(conclusion)

        try:
            return (premise1.main_op == 'imp' and
                    premise2.main_op == 'imp' and
                    conclusion.main_op == 'imp' and
                    premise1.left == conclusion.left and
                    premise1.right == premise2.left and
                    premise2.right == conclusion.right )

        except:
            return False




    def simp(self, premise,conclusion): #Simplification
        """
        The first formula is split and compared to the
        second.
        """

        premise = Wff(premise)
        conclusion = Wff(conclusion)

        try:
            return (premise.main_op == 'and' and
                    (premise.left == conclusion.string or
                     premise.right == conclusion.string)
                    )

        except:
            return False
```

```
def conj(self, premise1, premise2, conclusion): #Conjunction
    """
    Conjunction uses the simplification method to
    validate that premise1 is tuple_of_form simplification of conclusion
    and premise2 is tuple_of_form simplification of conclusion.
    """
    return self.simp(conclusion, premise1) and self.simp(conclusion, premise2)


def dil(self, premise1,premise2,premise3,conclusion):
    boolean = False
    for permutation in permutations((premise1,premise2,premise3),3):
        boolean = self.__dil_one_way(permutation[0],permutation[1],permutation[2], conclusion)
        if boolean:
            break
    return boolean

def __dil_one_way(self, premise1, premise2, premise3, conclusion): #Dilemma

    premise1 = Wff(premise1)
    premise2 = Wff(premise2)
    premise3 = Wff(premise3)
    conclusion = Wff(conclusion)


    return (premise1.main_op == ’imp’ and
            premise2.main_op == ’imp’ and
            premise3.main_op == ’or’ and
            conclusion.main_op == ’or’ and
            {premise3.left, premise3.right} == {premise1.left, premise2.left} and
            {conclusion.left, conclusion.right} == {premise1.right, premise2.right})

def ds(self, form1, form2, form3):
    return (self.__ds_one_way(form1, form2, form3) or
            self.__ds_one_way(form2, form1, form3))


def __ds_one_way(self, a,b,c): #Disjunctive Syllogism

    a = Wff(a)
    b = Wff(b)
    c = Wff(c)

    try:
        return (a.main_op == ’or’ and
                b.main_op == ’neg’ and
                (a.left == b.left and
                a.right == c.string)
                or
                (a.right == b.left and
                 a.left == c.string)
                )
        except:
            return False

    def add(self, a,b): #Addition

        a = Wff(a)
        b = Wff(b)

        try:
            return (b.main_op == ’or’ and
                    (a.string in b.form))

        except:
            return False


#Replacement Rules
    def dn(self, form1,form2): #Double Negation

        form1a = Wff(form1)
        form2a = Wff(form2)
        form1b = Wff(form1a.left)
        form2b = Wff(form2a.left)

        #The second wff is the double negation.
        if (form1a.string == form2b.left and form2a.main_op == ’neg’
            and form2b.main_op == ’neg’):
            return True

        #The first wff is the double negation.
        else:
            return (form2a.string == form1b.left and form1a.main_op == ’neg’
            and form1b.main_op == ’neg’)


    def dup(self, form1, form2):

        form1 = Wff(form1)
        form2 = Wff(form2)

        if (form1.main_op == ’or’ and form1.left == form2.string):
            return form1.left == form1.right

        if (form1.main_op == ’and’ and form1.left == form2.string):
            return form1.left == form1.right

        if (form2.main_op == ’or’ and form2.left == form1.string):
            return form2.left == form2.right

        if (form2.main_op == ’and’ and form2.left == form1.string):
            return form2.left == form2.right
    return False

def comm(self, form1, form2): #Commutation

    form1 = Wff(form1)
    form2 = Wff(form2)

    if ((form1.main_op == 'or' and form2.main_op == 'or') or
        (form1.main_op == 'and' and form2.main_op == 'and')):
        return (form1.left == form2.right and
                form1.right == form2.left)

    return False

def assoc(self,form1, form2): #Association
    """
    First we will decide which way the association rule is applied.
    Then we will apply it and finally we will decide if
    it is valid.
    """

    mold = ('((p\/q)\/r)','(p\/(q\/r))')

    if (self.confirm((form1,form2), mold) or self.confirm((form2,form1),mold)):
        return True

    mold = ('(p*q)*r','p*(q*r)')

    return self.confirm((form1,form2), mold) or self.confirm((form2,form1),mold)

def contra(self, form1, form2):

    mold = ('p->q','~q->~p')

    return (self.confirm((form1, form2), mold) or
            self.confirm((form2,form1), mold))

def dem(self, form1, form2):

    mold = ('~(p\/q)','~p*~q')

    if (self.confirm((form1, form2), mold) or
        self.confirm((form1, form2), mold)):
        return True

    mold = ('~(p*q)','~p\/~q')

    return (self.confirm((form1, form2), mold) or
        self.confirm((form1, form2), mold))


def be(self, form1, form2):
    mold = ('p::q','(p->q)*(q->p)')
```

```
        return (self.confirm((form1,form2), mold) or
                self.confirm((form1,form2), mold))

    def ce(self, form1, form2):
        mold = (’p->q’, ’˜p\/q’)

        return (self.confirm((form1,form2), mold) or
                self.confirm((form1,form2), mold))


    def dist(self, form1, form2):
        mold = (’p⋆(q\/r)’, ’(p⋆q)\/(p⋆r)’)

        if (self.confirm((form1,form2), mold) or
                self.confirm((form1,form2), mold)):
            return True

        mold = (’p\/(q⋆r)’, ’(p\/q)⋆(p\/r)’)

        return (self.confirm((form1,form2), mold) or
                self.confirm((form1,form2), mold))


    def exp(self, form1, form2): #Exportation

        mold = (’(p⋆q)->r’,’p->(q->r)’)

        return (self.confirm((form1,form2), mold) or
                self.confirm((form1,form2), mold))

#Conditional proof methods and structural checks.
    def cp(self, form1, form2, form3):
        tuple_of_form = self.split_form(form3)
        form1 = self.strip_form(form1)
        form2 = self.strip_form(form2)

        return (form1 == tuple_of_form[0] and
                form2 == tuple_of_form[1] and
                tuple_of_form[2] == ’imp’)

    def ip(self, form1, form2, form3):
        form1 = self.strip_form(form1)
        form3 = self.strip_form(form3)
        return (self.__is_contradiction(form2) and
                (form1 == ’˜’ + form3 or
                     form3 == ’˜’ + form1 or
                     form1 == ’˜(’ + form3 + ’)’ or
                     form3 == ’˜(’ + form1 + ’)’))


    def __is_contradiction(self,form1):
        tuple_of_form = self.split_form(form1)
        try:
            return (tuple_of_form[2] == ’and’ and
                    (tuple_of_form[0] == ’˜’ + tuple_of_form[1] or
                     tuple_of_form[1] == ’˜’ + tuple_of_form[0] or
                     tuple_of_form[0] == ’˜(’ + tuple_of_form[1] + ’)’ or
                     tuple_of_form[1] == ’˜(’ + tuple_of_form[0] + ’)’))

        except:
            return False

    def confirm_structure(self, ip, refs):
        for tuple1 in refs:
            if not len(tuple1) == 1:
                lst1 = []

                for tuple2 in ip:
                    # if the line number is outside the scope of
                    # an assumption we must be cautious
                    if tuple1[0] > tuple2[1]:
                        lst1.append(tuple2)
                for ref in tuple1[1:]:
                    if self.__is_between(ref,lst1):
                        return False
        return True


    def __is_between(self,ref,lst1):
        if lst1:
            for range1 in lst1:
                if (ref <= range1[1] and
                    ref >= range1[0]):
                    return True
        return False

    def ip_do_not_cross(self,lst1):
        lst2 = []
        for element in lst1:
            if (element[1] == ’assp’ or
                element[1] == ’fs’):
                lst2.append(element[0])
            if element[1] in (’ip’,’cp’, ’ug’):
                x = lst2.pop()
                if not element[2] == x:
                    return False
        return not bool(lst2)



#Predicate Logic Methods

class Pred(Prop):
    def __init__(self):
        super(Pred, self).__init__()
        self.flagset = set()

    def ui(self, form1, form2):
        try:
            dict1 = {}
            form1 = self.strip_form(form1)
            form2 = self.strip_form(form2)
            var = form1[1]
            form1 = form1[4:-1]
            for i in range(len(form1)):
                if form1[i] == var:
                    if not dict1.has_key(var):
                        dict1[var] = form2[i]
                    else:
                        return re.sub(var,dict1[var],form1) == form2

        except:
            return False


    def eg(self, form1, form2):

        try:
            dict1 = {}
            form1 = self.strip_form(form1)
            form2 = self.strip_form(form2)
            var = form2[8]
            form2 = form2[11:-1]
            for i in range(len(form2)):
                if form2[i] == var:
                    if not dict1.has_key(var):
                        dict1[var] = form1[i]
                    else:
                        return re.sub(var,dict1[var],form2) == form1

        except:
            return False

    def ei(self, form1, form2):
        try:
            dict1 = {}
            form1 = self.strip_form(form1)
            form2 = self.strip_form(form2)
            var = form1[8]
            form1 = form1[11:-1]
            for i in range(len(form1)):
                if form1[i] == var:
                    if not dict1.has_key(var):
                        dict1[var] = form2[i]
                        break
            if dict1[var] not in self.flagset and re.sub(var,dict1[var],form1) == form2:
                self.flagset.add(dict1[var])
                return True
            except:
                return False

        def ug(self, flag, form1, form2): #Universal Generalization
            """
            This method compares the flagged variable, the first
            formula in the Universal Generalization subproof and
            the conclusion to the subproof. The flagged variable
            is discarded if the proof is valid.
            """
            try: #Anything that goes wrong here means that something is incorrect.
                form1 = self.strip_form(form1) #Get rid of access space
                form2 = self.strip_form(form2)
                var = form2[1]
                form2 = form2[4:-1]
                bool1 = bool(re.sub(var,flag,form2) == form1)
                if bool1:
                    self.flagset.discard(flag) #Once the ug subproof is complete flagged variable can be usec
                    return True
                else:
                    return False

            except:
                return False

        def fs(self, flag):

            try:

                bool1 = bool(flag not in self.flagset)
                if bool1:
                    self.flagset.add(flag)
                    return True
                else:
                    return False

            except:
                return False


    #QN Rules

        def qn1(self, form1, form2):
            return (self.__qn1oneway(form1, form2) or
                    self.__qn1oneway(form2, form1))


        def __qn1oneway(self, form1, form2):

            try:

                form1 = self.strip_form(form1)
            form2 = self.strip_form(form2)
            var = form1[2]

            return (form1[0:4] == ’˜(’+var+’)’ and
                    form2[0:10] == "(\\exists" + var + ")" and
                    self.split_form(’˜’ + form1[4:]) == self.split_form(form2[10:]))

        except:
            return False


    def qn2(self, form1, form2):
        try:
            return (self.__qn2oneway(form1, form2) or
                    self.__qn2oneway(form2, form1))
        except:
            return False


    def __qn2oneway(self, form1, form2):

        try:
            form1 = self.strip_form(form1)
            form2 = self.strip_form(form2)
            var = form2[1]


            return (form1[0:11] == ’˜(\\exists’+var+’)’ and
                    self.split_form(’˜(’+form1[11:]+’)’) == self.split_form(form2[3:])
                    and
                    form2[0:3] == ’(’+var+’)’)
        except:
            return False


    def qn3(self, form1, form2):
        return (self.__qn3oneway(form1, form2) or
                self.__qn3oneway(form2, form1))


    def __qn3oneway(self, form1, form2):

        try:
            form1 = self.strip_form(form1)
            form2 = self.strip_form(form2)
            var = form1[2]

            return (form1[0:4] == ’˜(’+var+’)’ and
                    form2[0:10] == ’(\\exists’+var+’)’ and
                    self.split_form(form1[4:]) == self.split_form(’˜(’+form2[10:]+’)’))

        except:
            return False
        def qn4(self, form1, form2):
            return (self.__qn4oneway(form1, form2) or
                    self.__qn4oneway(form2, form1))


        def __qn4oneway(self, form1, form2):

            try:
                form1 = self.strip_form(form1)
                form2 = self.strip_form(form2)
                var = form2[1]

                return (form1[0:11] == ’˜(\\exists’+var+’)’ and
                        self.split_form(form1[11:]) == self.split_form(’˜(’+form2[3:]+’)’)
                        and
                        form2[0:3] == ’(’+var+’)’)
            except:
                return False


    #Class that represents a well-formed formula.
    class Wff:
        """
        Takes the string representation of the tuple. ("A","B","imp")
        Has left for left of main op, right for right of main op, and main_op for main_op
        The tuple form also contains these.
        """

        def __init__(self, form):

            #Change the the string "A -> B" to "A->B".
            self.string = self.strip_form(form)
            self.form = self.string

            #Find the name and index of main operator.
            tuple_of_form = self.find_main_op(self.form)

            #If any of the following if statements are true, the method is
            #broken out of with a return.

            #There is no main operator.
            if not tuple_of_form:
                self.form = (self.form,’’)
                self.left = self.form[0]
                self.right = None
                self.main_op = None
                return

            #Main operator is a ’negation’. Length is 1.
            if tuple_of_form[1] == ’neg’:
                self.form = (self.form[1:], ’neg’)
                self.left = self.form[0]
                self.right = None
                self.main_op = 'neg'
                return

            #Main operator is 'or', 'imp', or 'quiv'. All are length 2.
            if tuple_of_form[1] in ['or','imp','equiv']:
                self.form = (self.form[:tuple_of_form[0]], self.form[tuple_of_form[0]+2:],
                           tuple_of_form[1])
                self.left = self.form[0]
                self.right = self.form[1]
                self.main_op = self.form[2]
                return

            #Main operator is an 'and'. Length is 1.
            else:
                self.form = (self.form[:tuple_of_form[0]], self.form[tuple_of_form[0]+1:],
                           tuple_of_form[1])
                self.left = self.form[0]
                self.right = self.form[1]
                self.main_op = self.form[2]

        def strip_form(self, form):
            form = re.sub(' ','',form)
            depth = 0
            for i,char in enumerate(form):
                if char == '(':
                    depth += 1
                if char == ')':
                    depth -= 1
                if depth == 0 and i == len(form) -1 and len(form) > 1:
                    return self.strip_form(form[1:-1])
                elif depth == 0:
                    break
            return form

        def find_main_op(self, form):
            """
            Takes tuple_of_form wff and converts it to tuple_of_form tuple.
            ie ("name of operator", index_of_operator)
            """

            if form[0] == '~':
                return (0, 'neg')

            #This could be better implemented as tuple_of_form stack.

            subdepth = 0

            for i, char in enumerate(form):
                if char == '(':
                    subdepth += 1
                if char == ')':
                    subdepth -= 1
                if char == '*' and subdepth == 0:
                    return (i, 'and')
                if char == '\\' and subdepth == 0:
                    if form[i+1] == '/':
                        return (i, 'or')
                if char == ':' and subdepth == 0:
                    if form[i+1]  == ':':
                        return (i, 'equiv')
                if char == '-' and subdepth == 0:
                    if form[i+1] == '>':
                        return (i, 'imp')


        def similar(self, form2, dictionary):
            """
            Confirms that the self fits the mold form2 and the dictionary is consitant.
            Any variables not in the dictionary are added and returned.
            If there is some conflict, false is returned else the dictionary.
            """

            if self.main_op != form2.main_op:
                return False

            if self.left in dictionary:
                if dictionary[self.left] != form2.left:
                    return False
            else:
                dictionary[self.left] = form2.left

            if self.right in dictionary:
                if dictionary[self.right] != form2.right:
                    return False
            else:
                dictionary[self.right] = form2.right

            return dictionary


    class Proof:
        """
        This class contains the attribute valid, which is true if
        there are no problems with the proof.
        Problems with the proof are stored in the following.
        rules_of_inference: stores a list of tuples (line_number, specific_rule)
        replacement_rules: list of (line_number, replacement_rule)
        flag: list of lines with incorrect flags
        """

        def __init__(self, file1):
            """
            Initialize Proof using a file. This automatically stores validity
            in valid and all of the problems of the proof.
            """
            self.logic = Pred()
            self.invalid_line = []
            self.invalid_number_of_arguments = []
            self.invalid_reasoning = []

            #Temporary list for proof. Elements are in the form
            #[premise1,premise2,...,conclusion,rule]
            proof_lst = self._file_to_list(file1)


            for index in range(len(proof_lst)):
                if proof_lst[index] == None:
                    #Add one to the index for line number.
                    self.invalid_line.append(index+1)
                    #Do nothing else if problem with format.
                    return

            #Only reached if format of proof is correct

            #Proof list changed to final format.
            proof_lst = self._replace_and_rearrange(proof_lst)

            #Check the individual parts of the proof and not the structure.
            for index, element in enumerate(proof_lst):
                if element[0] == 'pr':
                    pass
                else:
                    #If not true add to list of errors.
                    try:
                        if not eval('self.logic.'+element[1]+'(*element[1:])'):
                            self.invalid_reasoning.append((element[1], index+1))
                    except TypeError:
                        self.invalid_number_of_arguments.append(index+1)

            #Check that the flags are valid.

            #Check that the structure is valid.


        def _file_to_list(self, file1):
            """
            Takes proof file and returns proof as list.
            """

            #Proof as a list
            proof_lst = []

            for line in file1:
                #Remove newline from string
                line = line.rstrip()

                #Remove excess tabs
                line = re.sub(r"\t+","\t",line)
                line = re.sub(r"\.\t+","\t",line)

                #Split by tabs
                lst = line.split("\t")

                #The list should equal three except for blank lines
                if len(lst) == 3:
                    proof_lst.append(lst)

                #If line is not equal to three and not blank
                #append None
                elif re.sub(r"\s+","",lst2[0]):
                    lst1.append(None)

            return proof_lst

        def _replace_and_rearrange(proof_lst):
            """
            Takes a list with elements in the form [line_num, conclusion, reason, prem1ref,...]
            and returns a list with elements [reason, prem1,...,conclusion].
            Ready to be used with rules of inference and replacement rules.
            """
            #Holds result of proof_lst after the change.
            result = []

            for index, element in enumerate(proof_lst):
                #reason holds rule of inference, replacement, or pr
                if element[-1].lower() == ’pr’:
                    reason = ’pr’

                else:
                    #Temp lst
                    lst = element[-1].split(’ ’)
                    reason = lst[0]

                    #Will hold a list of line elements
                    line_lst = [reason]
                    for ref in lst[1:]:
                        line_lst.append(proof_lst[ref][1])

                    #Append the conclusion
                    line_lst.append(element[1])

                result.append(line_lst)

            return result

    #Confirms the validity of the proof.
    class Confirm:
        def __init__(self):
            self.logic = Pred()
```

```
def confirm_validity(self, file1):
    lst1,ip,refs = self.proof_to_list(file1)
    lst2 = []

    for element in lst1:
        lst2.append(self.test(element))

    return (all(lst2) and
            self.confirm_structure(ip, refs)
            and
            self.ip_do_not_cross(lst1))


def confirm_validity_string(self, file1):
    str1 = ("There is tuple_of_form problem with the " +
            "following lines: ")
    lst1,ip,refs = self.proof_to_list(file1)
    lst2 = []
    for element in lst1:
        lst2.append(self.test(element))
    if all(lst2):
        return "Proof is valid."
    else:
        for i,elem in enumerate(lst2):
            if elem == False:
                str1 += str(i+1) + ", "
        return str1[:-2]

def test(self, lst1):
    lst1[1]
    lst2 = []

    if lst1[0] == 'return False':
        return False

    if not (lst1[1] == 'pr' or lst1[1] == 'assp'
            or lst1[1] == 'fs'):
        str1 = "self." + lst1[1] + "(*lst2)"
        for x in lst1[2:]:
            lst2.append(x)
        lst2.append(lst1[0])
        try:
            return eval(str1)
        except:
            return False

    return True

def proof_to_list(self, file1):
    lst1 = []
    lst3 = []
    for line in file1:
        line = line.rstrip()
        line = re.sub(r"\t+","\t",line)
        line = re.sub(r"\.\t+","\t",line)
        lst2 = line.split("\t")
        if len(lst2) == 3:
            lst2 = lst2[1:]
            lst2 = self.convert1(lst2)
            lst1.append(lst2)
        elif re.sub(r"\s+","",lst2[0]):
            lst1.append(['return False','return False'])

    ip   = self.__ip(lst1)
    refs = self.__refs(lst1)


    for element in lst1:
        lst2 = self.convert2(element, lst1)
        lst2[1] = lst2[1].lower()
        lst3.append(lst2)

    return (lst3,ip,refs)


def __refs(self,lst1):
    lst2 = []
    for i,element in enumerate(lst1):
        if (not isinstance(element[-1],int)
            or
            element[-3].lower() in ('cp','ip','ug')):

            lst2.append((i+1,))

        elif not isinstance(element[-2],int):
            lst2.append((i+1,element[-1]))
        else:
            lst2.append((i+1,element[-2],element[-1]))
    return lst2

def __ip(self,lst1):
    lst2 = []
    for element in lst1:
        try:
            if element[1].lower() in ('cp','ip','ug'):
                lst2.append((element[-2],element[-1]))
        except:
            pass
    return lst2


def flatten(self, x):
    result = []
    for el in x:
        if hasattr(el, "__iter__") and not isinstance(el, basestring):
            result.extend(self.flatten(el))
            else:
                result.append(el)
        return result


    def convert1(self, lst1):
        lst1[1] = lst1[1].split(’ ’)
        try:
            lst1[1][1] = lst1[1][1].split(’,’)
        except:
            pass

        lst1 = self.flatten(lst1)

        if len(lst1) > 2:
            for i, x in enumerate(lst1[2:]):
                lst1[i + 2] = int(x)

        return lst1

    def convert2(self, lst1, lst2):
        if not len(lst1) == 2: #Not tuple_of_form premise or assumption.
            for i, x in enumerate(lst1[2:]):
                lst1[i+2] = lst2[x - 1][0]

        return lst1


if __name__ == ’__main__’:
    #Tests for Version 3
    a = Wff("A->B")
    b = Wff("˜A")
    c = Wff("A∗B")
    print a.form
    print b.form
    print c.form
    a = "A->B"
    b = "A"
    c = "B"
    prop = Prop()
    print prop.mp(a,b,c)
    print prop.mp(b,a,c)
    print prop.mp(c,a,c)
    print prop.mp(b,a,"B->C")
    b = "˜B"
    c = "˜A"
    print prop.mt(a,b,c)
    b = "B->C"
    c = "A->C"
    print prop.hs(a,b,c)
    print prop.hs(b,a,c)
    print prop.hs(b,c,c)
    a = "A∗B"
```

```
b = "A"
c = "B"
print prop.simp(a, b)
print prop.simp(a, c)
a = "A->B"
b = "C->D"
c = "A\\/C"
d = "B\\/D"
print prop.dil(a, b, c, c)
print prop.dil(a, b, c, d)
print prop.dil(b,a, c, d)
print prop.dil(a, b, c, c)
d = "˜C"
e = "˜A"
print prop.ds(c,d,"A")
print prop.ds(c,d,e)
a = "A"
b = "B"
c = "A\\/B"
print prop.add(a, c)
print prop.add(b,c)
print prop.dn("A","˜˜A")
print prop.dn("˜˜A","˜˜˜˜A")
print prop.dn("˜˜A","A")
print prop.dn("˜A","A")

a = ’A’
b = ’A\\/A’
c = ’A⋆A’

print prop.dup(a,b)
print prop.dup(a,c)
print prop.dup(b,a)
print prop.dup(c,"B")

a = ’A\\/B’
b = ’B\\/A’
c = ’A⋆B’
d = ’B⋆A’
print prop.comm(a,b)
print prop.comm(c,d)
print prop.comm("A","B")

a = ’((A\/B)\/C)’
b = ’(A\/(B\/C))’
c = ’((A⋆B)⋆C)’
d = ’(A⋆(B⋆C))’
print prop.assoc(a,b)
print prop.assoc(b,a)
print prop.assoc(c,d)
print prop.assoc(d,c)

a = ’A->B’
```

```
b = ’˜A->˜B’

print prop.contra(a,b)
print prop.contra(b,a)

a = ’˜(A\/B)’
b = ’˜A⋆˜B’

print prop.dem(a,b)

a = ’A::B’
b = ’(A->B)⋆(B->A)’

print prop.be(a,b)

a = ’A->B’
b = ’˜A\/B’

print prop.ce(a,b)

a = ’A⋆(B\/C)’
b = ’(A⋆B)\/(A⋆C)’

print prop.dist(a,b)

a = ’(A⋆B)->C’
b = ’A->(B->C)’

print prop.exp(a,b)
```