

**Modus Ponens (M.P.)**

$$\frac{p \supset q \quad p}{\therefore q}$$

**Modus Tollens (M.T.)**

$$\frac{p \supset q \quad \sim q}{\therefore \sim p}$$

**Hypothetical Syllogism (H.S.)**

$$\frac{p \supset q \quad q \supset r}{\therefore p \supset r}$$

**Simplification (Simp.)**

$$\frac{p \cdot q}{\therefore p} \quad \frac{p \cdot q}{\therefore q}$$

**Conjunction (Conj.)**

$$\frac{p \quad q}{\therefore p \cdot q}$$

**Dilemma (Dil.)**

$$\frac{p \supset q \quad r \supset s \quad p \vee r}{\therefore q \vee s}$$

**Disjunctive Syllogism (D.S.)**

$$\frac{p \vee q \quad \sim p}{\therefore q} \quad \frac{p \vee q \quad \sim q}{\therefore p}$$

**Addition (Add.)**

$$\frac{p}{\therefore p \vee q} \quad \frac{q}{\therefore p \vee q}$$

**Double Negation (D.N.)**

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

**Duplication (Dup.)**

$$p :: (p \vee p) \\ p :: (p \cdot p)$$

**Commutation (Comm.)**

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

**Association (Assoc.)**

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

**Contraposition (Contrap.)**

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

**DeMorgan's (DeM.)**

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

**Biconditional Exchange (B.E.)**

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

**Conditional Exchange (C.E.)**

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

**Distribution (Dist.)**

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

**Exportation (Exp.)**

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

C. Restrictions on the Use of C.P. and I.P.

1. Every assumption made in a proof must eventually be discharged.
2. Once an assumption has been discharged, neither it nor any step that falls

within its scope may be used in the proof again.

3. Assumptions inside the scope of other assumptions must be discharged in the reverse order in which they were made; that is, not two scope markers may cross.

#### D. General Instructions for Using C.P. and I.P

1. For both C.P. and I.P., an assumption may be introduced at any point in the proof, provided we label it as an assumption.
2. In using C.P., we assume the antecedent of the conditional to be proved and then derive the consequent. In using I.P., we assume the opposite of what we want to prove and then derive a contradiction. All the steps from the assumption to the consequent (for C.P.) or the contradiction (for I.P.) are said to be *within the scope* of the assumption.

#### Quantifier Negation (Q.N.) Equivalences

1.  $\sim (\exists x)\phi x \equiv (x) \sim \phi x$
2.  $\sim (x)\phi x \equiv (\exists x) \sim \phi x$
3.  $\sim (\exists x) \sim \phi x \equiv (x)\phi x$
4.  $\sim (x) \sim \phi x \equiv (\exists x)\phi x$

### STATEMENT OF THE QUANTIFIER RULES, WITH ALL NECESSARY RESTRICTIONS

#### A. Preliminary Definitions

1.  $\phi x$  is a propositional function on  $x$ , 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.  $\phi a$  is a formula just like  $\phi x$ , except that every occurrence of  $x$  in  $\phi x$  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 subproof that 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.)**

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

**Existential Generalization (E.G.)**

**Universal Generalization (U.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.

```

#!/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.setResultName("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
```

```
#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.__contral(form1, form2) or
            self.__contral(form2, form1))

def __contral(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.__bel(form1, form2) or
                self.__bel(form2, form1))
    except:
        return False

def __bel(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.__cel(form1, form2) or
                self.__cel(form2, form1))
    except:
        return False

def __cel(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.__distan(a,b,c,d,e)
        else:
            return self.__distor(a, b, c, d, e)

    except:
        return False

def __distan(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.__expl(form1, form2) or
                self.__expl(form2, form1))

    except:
        return False

def __expl(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

#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 rangel in lst1:  
            if (ref <= rangel[1] and  
                ref >= rangel[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

```
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) == form1:
            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 v
            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.__qnloneway(form1, form2) or
            self.__qnloneway(form2, form1))

def __qnloneway(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

```

#CQN Rules

#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')

if __name__ == '__main__':
    a = Prop()
    # print a.confirm_validity_string(file1)
    # a.mt("Za->(Ha*Wa)", "~(Ha*Wa)", "~Za")
    #
    # file1 = a.prompt_for_file()
    # print a.confirm_validity(file1)
    #
    # print a.split_form("(F::G) -> (A -> F)")

    file1 = open("proofs/proof14.txt", 'r')
    print a.confirm_validity(file1)

```

```
#!/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)"))
    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,
                                                (9,4),(10,8,9),(11,10),(12,2,11),
                                                (13,7,12),(14,13),(15,3,6),(16,14
                                                (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",
        "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()
```