```
# -*- coding: UTF-8 -*-

from Util.Constants   import OPERATORS, RESERVEDCHARS
from Logic.Expressoes  import Variavel, NOT, AND, OR, XOR, IMPLIES, BIIMPLIES,\
                       Binaria
from Util.Excessoes   import InferenceRuleException
from Logic.Inferencias import ModusPonens, ModusTollens, SilogismoHipotetico,\
                        SilogismoDisjuntivo, Simplificacao, Adicao, Conjuncao,\
                        Contraposicao, Resolucao, DilemaConstrutivo,\
                        DilemaDestrutivo, AnyInference

class ExpressionInterpreter:
    ''' INTERPRETADOR DE EXPRESSOES
        Classe que contem um metodo de avaliacao de string e constituicao de
        um objeto Expression a partir das informacoes contidas nela. '''

    def __init__(self):
        self.processesLog = []

    def eval(self, string, vars = None):
        ''' string :: String -> Expression
            Analisa uma string e retorna uma expressao. '''
        if string == '':
            return None

        members       = []
        operador      = ''

        # Variavels auxiliares
        subExpression = ''
        variable      = ''
        negacao       = False
        parentesis    = 0

        # Pequeno artificio técnico vital para o bem-estar deste algoritmo
        string += ' '

        for char in string:

            # Leitura de subexpressoes limitadas por parenteses
            if char == '(' and parentesis == 0:
                parentesis = 1
                continue
            elif parentesis > 0:
                if char == '(':
                    parentesis += 1
                elif char == ')':
                    parentesis -= 1

                if parentesis > 0:
                    subExpression += char
                else:
                    sub = self.eval(subExpression, vars)
```

```
                if negacao:
                    members.append( NOT(sub) )
                    negacao = False
                else:
                    members.append(sub)
                subExpression = ''
            continue

        # Flag usada para indicar uma negacao
        if char == '!':
            negacao = True
            continue

        # Leitura de VARIAVEIS
        elif char not in RESERVEDCHARS:
            variable += char
            continue
        elif variable != '':
            if vars == None:
                v = Variavel(variable, True)
            else:
                v = vars[variable]

            if negacao:
                members.append( NOT(v) )
                negacao = False
            else:
                members.append(v)
            variable = ''
            continue

        # Leitura de OPERADORES
        elif char in OPERATORS:
            operador += char
            continue

    expression = None

    # Se houver um operador, deduz-se que a expressao seja binaria
    if operador != '':
        if   operador == '&':
            expression = AND(members[0], members[1])
        elif operador == '|':
            expression = OR (members[0], members[1])
        elif operador == '*':
            expression = XOR(members[0], members[1])
        elif operador == '->':
            expression = IMPLIES(members[0], members[1])
        elif operador == '<->':
            expression = BIIMPLIES(members[0], members[1])
        else:
            return None
```

```
            result = "\n P. "  + str(expression)              + \
                     "\n 1. "  + str(expression.exp1)         + \
                     " = ["    + str(expression.exp1.eval() ) + "]" + \
                     "\n 2. "  + str(expression.exp2)         + \
                     " = ["    + str(expression.exp2.eval() ) + "]" + \
                     "\n :: [" + str(expression.eval() )      + "]"

        # Senao, ela deve ser Unaria, ou uma variavel
        else:
            expression = members[0]

            # Se for uma negacao
            if isinstance(expression, NOT):
                result = "\n P. "  + str(expression)             + \
                         "\n 1. "  + str(expression.exp)         + \
                         " = ["    + str(expression.exp.eval() ) + "]" + \
                         "\n :: [" + str(expression.eval() )     + "]"

            # Senao, eh uma variavel
            else:
                result = "\n P. "  + str(expression)         + \
                         "\n :: [" + str(expression.eval() ) + "]"

        self.processesLog.append(result)

        return expression


class InferenceInterpreter(object):
    ''' INTERPRETADOR DE REGRAS DE INFERENCIAS
        Classe que contem metodos de Prova e Identificacao de Regras
        de Inferencia. '''

    def proof(self, premissas, obj):
        ''' premissas :: [Expression], obj :: Expression -> ([String],
[Exp.])
        '''
        logs     = []
        con      = None

        while con != obj:

            # Bi implicacoes
            for k, premissa in enumerate(premissas):
                if isinstance(premissa, BIIMPLIES):
                    con  = premissa.exp1
                    con2 = premissa.exp2

                    if con not in premissas:
                        premissas.append(con)
                        logs.append("BI-IMPLICACAO (" + str(k) + ")")

                    if con2 not in premissas:
```

```
                premissas.append(con2)
                logs.append("BI-IMPLICACAO (" + str(k) + ")")

            if con == obj:
                return logs, premissas
            elif con2 == obj:
                return logs, premissas


    for i, premissa in enumerate(premissas):
        # Simplificacoes
        if isinstance(premissa, AND):
            con  = Simplificacao(premissa).eval().conclusao
            con2 = Simplificacao(premissa).eval2().conclusao

            if con not in premissas:
                premissas.append(con)
                logs.append("SIMPLIFICACAO (" + str(i) + ")")

            if con2 not in premissas:
                premissas.append(con2)
                logs.append("SIMPLIFICACAO (" + str(i) + ")")

            if con == obj:
                return logs, premissas
            elif con2 == obj:
                return logs, premissas

        # DeMorgan
        resultado = None
        if isinstance(premissa, NOT) and \
             isinstance(premissa.exp, AND):

            con = OR( NOT(premissa.exp.exp1),
                      NOT(premissa.exp.exp2) )

            resultado = "DeMORGAN (" + str(i) + ")"

        # DeMorgan 2: O Retorno de Jaffar
        elif isinstance(premissa, NOT) and \
             isinstance(premissa.exp, OR):

            con = AND( NOT(premissa.exp.exp1),
                       NOT(premissa.exp.exp2) )

            resultado = "DeMORGAN (" + str(i) + ")"

        if con not in premissas and resultado != None:
            premissas.append(con)
            logs.append( resultado )

        if con == obj:
            return logs, premissas
```

```
        # Regras de Inferencia com 2 premissas
        for k in range( len(premissas) ):
            stat = [premissa, premissas[k] ]
            anyInference = AnyInference(stat)
            resultado = None

            if ModusPonens() == anyInference:
                con = ModusPonens(stat).eval().conclusao
                resultado = "MODUS PONENS (" + str(i) + "," + \
                                               str(k) + ")"

            elif ModusTollens() == anyInference:
                con = ModusTollens(stat).eval().conclusao
                resultado = "MODUS TOLLENS (" + str(i) + "," + \
                                                str(k) + ")"

            elif SilogismoHipotetico() == anyInference:
                con = SilogismoHipotetico(stat).eval().conclusao
                resultado = "SILOGISMO HIPOTETICO (" + str(i) + "," + \
                                                       str(k)+ ")"

            elif SilogismoDisjuntivo() == anyInference:
                con = SilogismoDisjuntivo(stat).eval().conclusao
                resultado = "SILOGISMO DISJUNTIVO (" + str(i) + "," + \
                                                       str(k)+ ")"

            elif Resolucao() == anyInference:
                con = Resolucao(stat).eval().conclusao
                resultado = "RESOLUCAO (" + str(i) + "," + str(k)+ ")"

            if con not in premissas and resultado != None:
                premissas.append(con)
                logs.append( resultado )

            if con == obj:
                return logs, premissas

        # Regras de Inferencia com 3 premissas
        for k in range( len(premissas) - 1 ):
            stat = [premissa, premissas[k], premissas[k + 1] ]
            anyInference = AnyInference(stat)
            resultado = None

            if DilemaConstrutivo() == anyInference:
                con = DilemaConstrutivo(stat).eval().conclusao
                resultado = "DILEMA CONSTRUTIVO (" + str(i) + "," + \
                                                     str(k)+ ")"

            elif DilemaDestrutivo() == anyInference:
                con = DilemaDestrutivo(stat).eval().conclusao
                resultado = "DILEMA DESTRUTIVO (" + str(i) + "," + \
                                                    str(k)+ ")"
```

```
            if con not in premissas and resultado != None:
                premissas.append(con)
                logs.append( resultado )

            if con == obj:
                return logs, premissas

    ''' Pela estratégia do algoritmo, para que CONJUNCOES sejam aplica
        das, deve-se saber pelo menos o que se busca conjugar. Entao,
        aqui é criado um array de expressoes do tipo AND, que deverão
        ser encontradas dentro das premissas, e só as conjuncoes que
        estiverem previstas dentro deste array serao adicionadas na
        lista de premissas. '''
    ANDs = []
    for premissa in premissas:
        if isinstance(premissa, Binaria):

            if isinstance(premissa.exp1, AND):
                ANDs.append(premissa.exp1)

            if isinstance(premissa.exp2, AND):
                ANDs.append(premissa.exp2)

    for i, premissa in enumerate(premissas):
        for k in range( len(premissas) ):
            stat = [premissa, premissas[k]]
            anyInference = AnyInference(stat)

            if Conjuncao() == anyInference:
                con = Conjuncao(stat).eval().conclusao

                if con == obj or con in ANDs:
                    if con not in premissas:
                        premissas.append(con)
                        logs.append("CONJUNCAO (" + str(i) + ","
                                                  + str(k) + ")")
                    if con == obj:
                        return logs, premissas

    ''' A mesma estrategia do algoritmo acima, sendo que agora usando
        ADICOES e procurando por expressoes OR'''
    ORs = []
    for premissa in premissas:
        if isinstance(premissa, Binaria):

            if isinstance(premissa.exp1, OR):
                ORs.append(premissa.exp1)

            if isinstance(premissa.exp2, OR):
                ORs.append(premissa.exp2)

    for i, premissa in enumerate(premissas):
```

```
                for k in range( len(premissas) ):
                    stat = [premissa, premissas[k]]
                    anyInference = AnyInference(stat)

                    if Adicao() == anyInference:
                        con = Adicao(stat).eval().conclusao

                        if con == obj or con in ORs:
                            if con not in premissas:
                                premissas.append(con)
                                logs.append("ADICAO (" + str(i) + ","
                                                       + str(k) + ")")
                            if con == obj:
                                return logs, premissas
        else:
            raise InferenceRuleException(" [ERRO] Resultado nao encontrado!")


    def identify(self, premissas, conclusao):
        ''' premissas :: [Expression], conclusao :: Expression -> InferenceRule
            Atraves de premissas e uma conclusao passadas pelo usuario, este
            metodo avalia de qual regra de inferencia se trata. '''
        anyInference = AnyInference(premissas, conclusao)

        if len(premissas) == 1:
            if Adicao() == anyInference:
                return Adicao(premissas, conclusao)

            elif Simplificacao() == anyInference:
                return Simplificacao(premissas, conclusao)

            elif Contraposicao() == anyInference:
                return Contraposicao(premissas, conclusao)

        elif len(premissas) == 2:
            if ModusPonens() == anyInference:
                return ModusPonens(premissas, conclusao)

            elif ModusTollens() == anyInference:
                    return ModusTollens(premissas, conclusao)

            elif SilogismoHipotetico() == anyInference:
                return SilogismoHipotetico(premissas, conclusao)

            elif SilogismoDisjuntivo() == anyInference:
                return SilogismoDisjuntivo(premissas, conclusao)

            elif Conjuncao() == anyInference:
                return Conjuncao(premissas, conclusao)

            elif Resolucao() == anyInference:
                return Resolucao(premissas, conclusao)
```

```
        elif len(premissas) == 3:
            if DilemaConstrutivo() == anyInference:
                return DilemaConstrutivo(premissas, conclusao)

            elif DilemaDestrutivo() == anyInference:
                return DilemaDestrutivo(premissas, conclusao)

        raise InferenceRuleException("[ERRO] R.I. Inexistente!")


    def complete(self, premissas, conclusao = None):
        ''' premissas :: [Expression], conclusao :: Expression -> InferenceRule
            Identifica a Regra de inferencia, mesmo faltando uma premissao ou a
            conclusao. '''

        if premissas[0] == None:
            return self._primeirapremissa(premissas, conclusao)

        elif len(premissas) > 1 and premissas[1] == None:
            return self._segundapremissa(premissas, conclusao)

        elif len(premissas) > 2 and premissas[2] == None:
            return self._terceirapremissa(premissas, conclusao)

        elif conclusao == None:
            return self._conclusao(premissas)

        else: raise InferenceRuleException("[ERRO] Entrada invalida!")


    def _primeirapremissa(self, premissas, con):
        ''' Metodo acessor chamado quando a 1a premissa estiver faltando. '''
        anyInference = AnyInference(premissas, con)

        if len(premissas) == 1:
            if Adicao() == anyInference:
                return Adicao(con.exp1, con)

            elif Contraposicao() == anyInference:
                return Contraposicao([IMPLIES(NOT(con.exp2),NOT(con.exp1))],con)
            else:
                return Simplificacao([AND(con, 'q')], con)

        elif len(premissas) == 2:
            p2 = premissas[1]

            if SilogismoHipotetico() == anyInference:
                return SilogismoHipotetico([IMPLIES(con.exp1, p2.exp1), p2],con)

            elif Conjuncao() == anyInference:
                return Conjuncao([con.exp1, p2], con)

            elif Resolucao() == anyInference:
```

```
                return Resolucao([OR(NOT(p2.exp2), con.exp1), p2], con)

            elif SilogismoDisjuntivo() == anyInference:
                return SilogismoDisjuntivo([OR(p2.exp, con), p2], con)

            elif ModusTollens () == anyInference:
                return ModusTollens([IMPLIES(NOT(con), NOT(p2)), p2], con)

            elif ModusPonens() == anyInference:
                return ModusPonens([IMPLIES(p2, con), p2], con)

        elif len(premissas) == 3:
            p2 = premissas[1]
            p3 = premissas[2]

            if DilemaConstrutivo() == anyInference:
                return DilemaConstrutivo([IMPLIES(p3.exp1,con.exp1), p2, p3],
                                         con)

            elif DilemaDestrutivo() == anyInference:
                return DilemaDestrutivo([IMPLIES(NOT(con.exp1),
                                                 NOT(p3.exp1)),p2, p3], con)

        raise InferenceRuleException("[ERRO] 1a premissa nao encontrada")


    def _segundapremissa(self, premissas, con):
        ''' Metodo acessor chamado quando a 2a premissa estiver faltando. '''

        anyInference = AnyInference(premissas, con)

        if len(premissas) == 2:
            p1 = premissas[0]

            if SilogismoHipotetico() == anyInference:
                return SilogismoHipotetico([p1,IMPLIES(p1.exp2,con.exp2)], con)

            elif SilogismoDisjuntivo() == anyInference:
                if p1.exp1 == con:
                    return SilogismoDisjuntivo([p1, NOT(p1.exp2)], con)
                else:
                    return SilogismoDisjuntivo([p1, NOT(p1.exp1)], con)

            elif Conjuncao() == anyInference:
                return Conjuncao([p1, con.exp2], con)

            elif Resolucao() == anyInference:
                return Resolucao([p1, OR(con.exp2, NOT(p1.exp1))], con)

            elif ModusPonens() == anyInference:
                return ModusPonens([p1, p1.exp1], con)

            elif ModusTollens() == anyInference:
```

```
                return ModusTollens([p1, NOT(p1.exp2)], con)

        elif len(premissas) == 3:
            p1 = premissas[0]
            p3 = premissas[2]

            if DilemaConstrutivo() == anyInference:
                return DilemaConstrutivo([p1,IMPLIES(p3.exp2,con.exp2),p3], con)

            elif DilemaDestrutivo() == anyInference:
                return DilemaDestrutivo([p1, IMPLIES(NOT(con.exp2),
                                                     NOT(p3.exp2)),p3], con)


        raise InferenceRuleException("[ERRO] 2a premissa nao encontrada")


    def _terceirapremissa(self, premissas, con):
        ''' Metodo acessor chamado quando a 3a premissa estiver faltando. '''
        anyInference = AnyInference(premissas, con)
        p1 = premissas[0]
        p2 = premissas[1]

        if DilemaConstrutivo() == anyInference:
            return DilemaConstrutivo([p1, p2, OR(p1.exp1, p2.exp1)], con)

        elif DilemaDestrutivo() == anyInference:
            return DilemaDestrutivo([p1,p2,OR(NOT(p1.exp2),NOT(p2.exp2))], con)


        raise InferenceRuleException("[ERRO] 3a premissa nao encontrada")


    def _conclusao(self, premissas):
        ''' Metodo acessor chamado quando a conclusao estiver faltando. '''
        anyInference = AnyInference(premissas)

        if len(premissas) == 1:
            if Simplificacao() == anyInference:
                return Simplificacao(premissas).eval()

            elif Contraposicao() == anyInference:
                return Contraposicao(premissas).eval()

            else:
                return Adicao(premissas).eval()

        elif len(premissas) == 2:
            if SilogismoHipotetico() == anyInference:
                return SilogismoHipotetico(premissas).eval()

            elif SilogismoDisjuntivo() == anyInference:
                return SilogismoDisjuntivo(premissas).eval()
```

```
        elif Resolucao() == anyInference:
            return Resolucao(premissas).eval()

        elif ModusPonens() == anyInference:
            return ModusPonens(premissas).eval()

        elif ModusTollens() == anyInference:
            return ModusTollens(premissas).eval()

        elif Conjuncao() == anyInference:
            return Conjuncao(premissas).eval()

    elif len(premissas) == 3:
        if DilemaConstrutivo() == anyInference:
            return DilemaConstrutivo(premissas).eval()

        elif DilemaDestrutivo() == anyInference:
            return DilemaDestrutivo(premissas).eval()

    raise InferenceRuleException("[ERRO] Conclusao nao encontrada!")
```