**NAME:SAKCHI CHOUDHARY**

**REGISTRATION NO:RA1911042010035**

**BATCH:CSBS-R1**

**EXPERIMENT NO: 8**

# TOPIC: IMPLEMENTATION OF PREPOSITIONAL LOGIC FOR REAL WORLD PROBLEMS

# CODE:

#!/usr/bin/env python3

"""

MIT License

Copyright (c) 2017 Parham Pourdavood

Permission is hereby granted, free of charge, to any person obtaining a copy

of this software and associated documentation files (the "Software"), to deal

in the Software without restriction, including without limitation the rights

to use, copy, modify, merge, publish, distribute, sublicense, and/or sell

copies of the Software, and to permit persons to whom the Software is

furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all

copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR

IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,

FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE

AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER

LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,

OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE

SOFTWARE.

"""

import re

class Expression:

    """

    Expression class takes a logical expression in form of string and creates

    an object with useful methods to manipulate the expression or to get useful

    information from it.

    Attributes

    ----------

    expression: str

            A logical argument using OR, AND, and NOT operations

    keywords:   ["OR", "AND", "IF", "THEN"]

            List of keywords used in this program

    and\_regex:  str

            Regular expression used to identify conjunctive arguments

    or\_regex:   str

            Regular expression used to identify disjunctive arguments

    conditional\_regex:  str

            Regular expression used to identify conditional arguments

    Methods

    -------

    get()

    set(new\_expression)

    recognizer()

    valid\_parentheses\_checker()

    expression\_parser()

    is\_pure\_proposition()

    negative\_inverter()

    temp\_negative\_inverter()

    """

    def \_\_init\_\_(self, expression):

        self.expression = expression

        self.keywords = ["OR", "AND", "IF", "THEN"]

        self.and\_regex = r"(\(+.\*?\)+) AND (\(+.\*\)+)( AND \(+.\*\)+)\*$"

        self.or\_regex = r"(\(+.\*?\)+) OR (\(+.\*\)+)( OR \(+.\*\)+)\*$"

        self.conditional\_regex = r"IF (\(.\*\)) THEN (\(.\*\))$"

    def \_\_eq\_\_(self, other):

        return self.get() == other.get()

    def \_\_ne\_\_(self, other):

        return not self.\_\_eq\_\_(other)

    def \_\_hash\_\_(self):

        return hash(self.expression)

    def \_\_str\_\_(self):

        return self.expression

    def get(self):

        """

        Returns the expression of the object in string.

        :return: self.expression

        :rtype: str

        """

        return self.expression

    def set(self, new\_expression):

        """

        Sets the self.expression to a new string

        :param new\_expression

        :type new\_expression: str

        :return: -

        """

        self.expression = new\_expression

    def recognizer(self):

        """

        Recognizes the main type of a logical argument.

        :return:

        "AND", "OR", "Conditional", "Pure", "Broken":   str

        """

        if re.match(self.or\_regex, self.expression):

            p = re.compile(self.or\_regex)

            m = re.match(p, self.expression)

            for i in m.groups()[:-1]:

                if "))" in i and "((" not in i or "((" in i and "))" not in i:

                    return "AND"

            return "OR"

        elif re.match(self.and\_regex, self.expression):

            p = re.compile(self.and\_regex)

            m = re.match(p, self.expression)

            for i in m.groups()[:-1]:

                if "))" in i and "((" not in i or "((" in i and "))" not in i:

                    return "OR"

            return "AND"

        elif re.match(self.conditional\_regex, self.expression):

            return "Conditional"

        else:

            # Here we check whether the argument contains any keywords or it

            # is just stating a pure expression as truth.

            flag = True

            for i in self.keywords:

                if i in self.expression:

                    flag = False

                    break

            if flag:

                return "Pure"

            else:

                # If none was applied it means the expression is broken

                return "Broken"

    def valid\_parentheses\_checker(self):

        """

        Checks whether expression of an object is using valid from parentheses

        :return: True | False

        :rtype: bool

        """

        if "(" not in self.expression and ")" not in self.expression:

            return False

        if "(" not in self.expression or ")" not in self.expression:

            return False

        else:

            return True

    def expression\_parser(self):

        """

        Parses the expression of an Expression object to its main parts and

        creates new objects with the sub-parts and places them in a list.

        :return: parsed\_expression\_list

        :rtype: list

        """

        if self.recognizer() == "AND":

            parsed\_expression = []

            match = re.match(self.and\_regex, self.expression)

            groups = match.groups()[:-1]

            for expression in groups:

                parsed\_expression.append(expression)

            for i in parsed\_expression:

                if "))" in i and "((" in i:

                    list\_index = parsed\_expression.index(i)

                    i = i[1:-1]

                    parsed\_expression[list\_index] = i

                elif "((" in i:

                    list\_index = parsed\_expression.index(i)

                    double\_paren\_index = i.index("((")

                    i = i[:double\_paren\_index] + i[double\_paren\_index + 1:]

                    parsed\_expression[list\_index] = i

                elif "))" in i:

                    list\_index = parsed\_expression.index(i)

                    double\_paren\_index = i.index("))")

                    i = i[:double\_paren\_index + 1] + i[double\_paren\_index + 2:]

                    parsed\_expression[list\_index] = i

            parsed\_expression\_list = []

            for i in parsed\_expression:

                new\_expression\_object = Definer(i)

                parsed\_expression\_list.append(new\_expression\_object)

            return parsed\_expression\_list

        elif self.recognizer() == "OR":

            parsed\_expression = []

            match = re.match(self.or\_regex, self.expression)

            groups = match.groups()[:-1]

            for expression in groups:

                parsed\_expression.append(expression)

            for i in parsed\_expression:

                if "))" in i and "((" in i:

                    list\_index = parsed\_expression.index(i)

                    i = i[1:-1]

                    parsed\_expression[list\_index] = i

                elif "((" in i:

                    list\_index = parsed\_expression.index(i)

                    i = i[1:]

                    parsed\_expression[list\_index] = i

                elif "))" in i:

                    list\_index = parsed\_expression.index(i)

                    i = i[:-1]

                    parsed\_expression[list\_index] = i

            parsed\_expression\_list = []

            for i in parsed\_expression:

                new\_expression\_object = Definer(i)

                parsed\_expression\_list.append(new\_expression\_object)

            return parsed\_expression\_list

        elif self.recognizer() == "Conditional":

            condt\_matched = re.match(self.conditional\_regex, self.expression)

            condt\_object1 = Definer(condt\_matched.group(1))

            condt\_object2 = Definer(condt\_matched.group(2))

            parsed\_expression = {"IF": condt\_object1,

                                 "THEN": condt\_object2}

            return parsed\_expression

    def is\_pure\_proposition(self):

        """

        Checks whether a proposition is pure, meaning it is just stating the

        truth of an expression without any keywords.

        :return: True | False

         :rtype: bool

        """

        if self.recognizer() == "Conditional":

            parsed\_expression = self.expression\_parser()

            for expression in parsed\_expression:

                if parsed\_expression[expression].recognizer() != "Pure":

                    return False

            return True

        else:

            for i in self.expression\_parser():

                if i.recognizer() != "Pure":

                    return False

            return True

    def negative\_inverter(self):

        """

        Removes NOT from the expression of an Expression object and sets it.

        :return: -

        """

        expression = self.expression

        expression = expression[0] + expression[5:]

        self.expression = expression

    def temp\_negative\_inverter(self):

        """

        Returns a string that has the NOT in the expression of an Expression

        object removed.

        :return: temp\_inverted\_object

        :rtype: str

        """

        expression = self.expression

        expression = expression[0] + expression[5:]

        temp\_inverted\_object = Expression(expression)

        return temp\_inverted\_object

# ------------------------------------------------------------------------------

class Definer(Expression):

    """

    Definer class is a subclass of Expression that provides methods useful for

    defining logical expressions into our knowledge dictionary.

    Attributes

    ----------

    expression: str

            A logical argument using OR, AND, and NOT operations

    Methods

    -------

    and\_definer()

    or\_definer()

    conditional\_definer()

    definer()

    and\_in\_or\_checker(or\_expression)

    and\_temp\_transformer()

    """

    def \_\_init\_\_(self, expression):

        Expression.\_\_init\_\_(self, expression)

    def and\_definer(self):

        """

        Defines a conjunctive expression into the knowledge dictionary and set

        the expression as True.

        :return: True

        :rtype: bool

        """

        for expression in self.expression\_parser():

            if "NOT" not in expression.expression:

                knowledge\_dict[expression] = True

            else:

                # If NOT is in the expression we invert it and set it to False.

                expression.negative\_inverter()

                knowledge\_dict[expression] = False

        return True

    def or\_definer(self):

        """

        Defines a disjunctive expression into the knowledge dictionary and set

        the expression to True or False based on evaluation of previously

        entered expressions.

        :return: True | False | None

        :rtype: bool

        """

        expression\_in\_dict = False

        for expression in self.expression\_parser():

            if "NOT" in expression.get():

                reversed\_temp = expression.temp\_negative\_inverter()

                if reversed\_temp in knowledge\_dict:

                    expression\_in\_dict = True

                    break

                continue

            if expression in knowledge\_dict:

                expression\_in\_dict = True

                break

        if expression\_in\_dict is True:

            for expression in self.expression\_parser():

                if "NOT" in expression.get():

                    reversed\_temp = expression.temp\_negative\_inverter()

                    if reversed\_temp not in knowledge\_dict:

                        knowledge\_dict[reversed\_temp] = None

                else:

                    if expression not in knowledge\_dict:

                        knowledge\_dict[expression] = None

            # count to check if all elements are false

            count = 0

            for expression in self.expression\_parser():

                if "NOT" in expression.get():

                    expression.negative\_inverter()

                    if knowledge\_dict[expression] is False:

                        return True

                    elif knowledge\_dict[expression] is True:

                        count += 1

                else:

                    if knowledge\_dict[expression] is True:

                        return True

                    elif knowledge\_dict[expression] is False:

                        count += 1

            if count == len(self.expression\_parser()):

                return False

            else:

                return None

        else:

            for expression in self.expression\_parser():

                if "NOT" in expression.get():

                    reversed\_temp = expression.temp\_negative\_inverter()

                    if reversed\_temp not in knowledge\_dict:

                        knowledge\_dict[reversed\_temp] = None

                else:

                    if expression not in knowledge\_dict:

                        knowledge\_dict[expression] = None

            return True

    def conditional\_definer(self):

        """

        Defines a conditional expression into the knowledge dictionary and set

        the expression to True or False based on evaluation of previously

        entered expressions.

        :return: True | False | None

        :rtype: bool

        """

        for expression in self.expression\_parser():

            if "NOT" in self.expression\_parser()[expression].expression:

                continue

            if self.expression\_parser()[expression] not in knowledge\_dict:

                knowledge\_dict[self.expression\_parser()[expression]] = None

        if "NOT" in self.expression\_parser()["IF"].expression:

            if\_proposition = self.expression\_parser()["IF"]

            if\_proposition.negative\_inverter()

            if if\_proposition not in knowledge\_dict:

                return None

            if knowledge\_dict[if\_proposition] is False:

                if "NOT" in self.expression\_parser()["THEN"].expression:

                    then\_proposition = self.expression\_parser()["THEN"]

                    then\_proposition.negative\_inverter()

                    knowledge\_dict[then\_proposition] = False

                else:

                    knowledge\_dict[self.expression\_parser()["THEN"]] = True

                return True

            elif knowledge\_dict[if\_proposition] is True:

                return True

            else:

                return None

        else:

            if knowledge\_dict[self.expression\_parser()["IF"]] is True:

                if "NOT" in self.expression\_parser()["THEN"].expression:

                    then\_proposition = self.expression\_parser()["THEN"]

                    then\_proposition.negative\_inverter()

                    knowledge\_dict[then\_proposition] = False

                else:

                    knowledge\_dict[self.expression\_parser()["THEN"]] = True

                return True

            elif knowledge\_dict[self.expression\_parser()["IF"]] is False:

                return True

            else:

                return None

    def definer(self):

        """

        Used as a general definer to be used in interpreter function. For the

        sake of simplicity, definer checks the type of the expression in its

        body and uses the right definer accordingly.

        :return: True | False | None

        :rtype: bool

        """

        if self.recognizer() == "AND":

            return self.and\_definer()

        elif self.recognizer() == "OR":

            return self.or\_definer()

        elif self.recognizer() == "Conditional":

            return self.conditional\_definer()

    def special\_definer(self):

        """

        Special definer is used for defining the AND expressions that have been

        previously been in an OR expression. It is special since we don't want

        to set them to True.

        :return: True | False | None

        :rtype: bool

        """

        for expression in self.expression\_parser():

            if expression not in knowledge\_dict:

                knowledge\_dict[expression] = None

        true\_count = 0

        for expression in self.expression\_parser():

            if knowledge\_dict[expression] is None:

                return None

            if knowledge\_dict[expression] is False:

                return False

            if knowledge\_dict[expression] is True:

                true\_count += 1

        if true\_count == len(self.expression\_parser()):

            return True

    def and\_in\_or\_checker(self, main\_expression):

        """

        Checks whether the expression we are looking is an AND expression that

        is part of an OR expression.

        :param main\_expression:

        :type main\_expression: Expression

        :return: True | False

        :rtype: bool

        """

        if self.recognizer() == "AND" and main\_expression.recognizer() == "OR":

            return True

        else:

            return None

    def and\_temp\_transformer(self):

        """

        Sets a mark in the expression so we can recognize it's different later.

        :return: -

        """

        self.expression = self.expression + "@"

# ------------------------------------------------------------------------------

class Resolver(Expression):

    """

    Resolver class is a subclass of Expression that provides methods useful for

    resolving logical expressions in our knowledge dictionary into proof\_dict.

    Attributes

    ----------

    expression: str

            A logical argument using OR, AND, and NOT operations

    Methods

    -------

    and\_resolver()

    or\_resolver()

    conditional\_resolver()

    general\_resolver()

    """

    def \_\_init\_\_(self, expression):

        Expression.\_\_init\_\_(self, expression)

    def and\_resolver(self):

        """

        Resolves an AND expression based on other expressions stored in

        proof\_dict.

        :return: True | False | None

        :rtype: bool

        """

        for expression in self.expression\_parser():

            if "NOT" in expression.get():

                continue

            if expression not in proof\_dict or expression is None:

                proof\_dict[expression] = None

                return None  # Can't be determined

        true\_count = 0

        for expression in self.expression\_parser():

            if "NOT" in expression.get():

                expression.negative\_inverter()

                if proof\_dict[expression] is True:

                    return False

                if proof\_dict[expression] is False:

                    true\_count += 1

            else:

                if proof\_dict[expression] is False:

                    return False

                if proof\_dict[expression] is True:

                    true\_count += 1

        if true\_count == len(self.expression\_parser()):

            return True

    def or\_resolver(self):

        """

        Resolves an OR expression based on other expressions stored in

        proof\_dict.

        :return: True | False | None

        :rtype: bool

        """

        for expression in self.expression\_parser():

            if "NOT" in expression.get():

                continue

            if expression not in proof\_dict or proof\_dict[expression] is None:

                proof\_dict[expression] = None

        # count to check if all elements are false

        count = 0

        for expression in self.expression\_parser():

            if "NOT" in expression.get():

                expression.negative\_inverter()

                if expression not in proof\_dict:

                    proof\_dict[expression] = None

                if proof\_dict[expression] is False:

                    return True

                if proof\_dict[expression] is True:

                    count += 1

            else:

                if proof\_dict[expression] is True:

                    return True

                if proof\_dict[expression] is False:

                    count += 1

        if count == len(self.expression\_parser()):

            return False

        else:

            return None

    def conditional\_resolver(self):

        """

        Resolves a conditional expression based on other expressions stored in

        proof\_dict.

        :return: True | False | None

        :rtype: bool

        """

        if\_statement = self.expression\_parser()["IF"]

        then\_statement = self.expression\_parser()["THEN"]

        if then\_statement not in proof\_dict:

            proof\_dict[then\_statement] = None

        if "NOT" in if\_statement.get() and "NOT" not in then\_statement.get():

            if\_statement.negative\_inverter()

            if if\_statement not in proof\_dict:

                proof\_dict[if\_statement] = None

                return None

            if proof\_dict[if\_statement] is False and proof\_dict[

                then\_statement] is False:

                return False

            elif proof\_dict[if\_statement] is True:

                return True

            elif proof\_dict[if\_statement] is False and proof\_dict[

                then\_statement] is True:

                return True

            else:

                return None

        elif "NOT" in if\_statement.get() and "NOT" in then\_statement.get():

            if\_statement.negative\_inverter()

            then\_statement.negative\_inverter()

            if if\_statement not in proof\_dict:

                proof\_dict[if\_statement] = None

                return None

            if proof\_dict[if\_statement] is False and proof\_dict[

                then\_statement] is True:

                return False

            elif proof\_dict[if\_statement] is True:

                return True

            elif proof\_dict[if\_statement] is False and proof\_dict[

                then\_statement] is False:

                return True

            else:

                return None

        elif "NOT" not in if\_statement.get() and "NOT" in then\_statement.get():

            then\_statement.negative\_inverter()

            if if\_statement not in proof\_dict:

                proof\_dict[if\_statement] = None

                return None

            if proof\_dict[if\_statement] is True and proof\_dict[

                then\_statement] is True:

                return False

            elif proof\_dict[if\_statement] is False:

                return True

            elif proof\_dict[if\_statement] is True and proof\_dict[

                then\_statement] is False:

                return True

            else:

                return None

        else:

            if if\_statement not in proof\_dict:

                proof\_dict[if\_statement] = None

                return None

            if proof\_dict[if\_statement] is True and proof\_dict[

                then\_statement] is False:

                return False

            elif proof\_dict[if\_statement] is False:

                return True

            elif proof\_dict[if\_statement] is True and proof\_dict[

                then\_statement] is True:

                return True

            else:

                return None

    def general\_resolver(self):

        """

        Used as a general resolver to be used in validator function. For the

        sake of simplicity, general\_resolver checks the type of the expression

        in its body and uses the right resolver accordingly.

        :return: True | False | None

        :rtype: bool

        """

        if self.recognizer() == "AND":

            return self.and\_resolver()

        elif self.recognizer() == "OR":

            return self.or\_resolver()

        elif self.recognizer() == "Conditional":

            return self.conditional\_resolver()

# ------------------------------------------------------------------------------

def interpreter(expression):

    """

    interpreter function is a recursive function that uses divide and conquer

    to go through nested arguments of the input expression and defines them

    accordingly.

    :param expression:

    :type expression: str

    :return: -

    """

    # Let's check to see if we have an AND operator that was part of an AND

    flag = False

    if expression[-1] == "@":

        # When it was delete the mark but set the flag to True so we will treat

        # it differently in the future.

        flag = True

        # Flag has become True and we can normalize the expression again

        expression = expression[0:-1]

    # Create an expression class with the expression string.

    expression\_object = Definer(expression)

    # First base case

    if expression\_object.recognizer() == "Pure":

        if "NOT" not in expression\_object.get():

            knowledge\_dict[expression\_object] = True

        else:

            expression\_object.negative\_inverter()

            knowledge\_dict[expression\_object] = False

    # Second base case

    elif expression\_object.is\_pure\_proposition():

        if flag:

            knowledge\_dict[

                expression\_object] = expression\_object.special\_definer()

        else:

            knowledge\_dict[expression\_object] = expression\_object.definer()

    # Third base case

    else:

        parsed\_expression = expression\_object.expression\_parser()

        if flag:

            knowledge\_dict[

                expression\_object] = expression\_object.special\_definer()

        else:

            knowledge\_dict[expression\_object] = expression\_object.definer()

        # Check to see if it is conditional so we can mark its IF proposition

        if expression\_object.recognizer() == "Conditional":

            for expression in parsed\_expression.values():

                expression\_type = expression.recognizer()

                if expression\_type == "AND":

                    parsed\_expression["IF"].and\_temp\_transformer()

                # Recursive step

                if expression\_type != "Pure" and expression\_type != "Broken":

                    interpreter(expression.get())

        else:

            for expression in parsed\_expression:

                expression\_type = expression.recognizer()

                # Check to see if any AND  was part of an OR proposition

                if expression.and\_in\_or\_checker(expression\_object) is True:

                    expression.and\_temp\_transformer()

                # Recursive step

                if expression\_type != "Pure" and expression\_type != "Broken":

                    interpreter(expression.get())

# ------------------------------------------------------------------------------

def validator(expression):

    """

    validator function is a recursive function that uses divide and conquer

    to go through nested arguments of the input expression and validates them

    accordingly. (Uses arguments previously defined in knowledge dictionary that

    was later on copied into proof\_dict. validator stores the result in

    proof\_dict.

    :param expression:

    :type expression: str

    :return: -

    """

    # Create an object with the expression string.

    expression\_object = Resolver(expression)

    expression\_object\_type = expression\_object.recognizer()

    parsed\_expression = expression\_object.expression\_parser()

    # First base case

    if expression\_object\_type == "Pure":

        if "NOT" in expression\_object.get():

            temp\_inverted = expression\_object.temp\_negative\_inverter()

            if temp\_inverted not in knowledge\_dict:

                proof\_dict[expression\_object] = None

                return None

            elif knowledge\_dict[temp\_inverted] is True:

                proof\_dict[expression\_object] = False

                return False

            else:

                proof\_dict[expression\_object] = True

                return True

        else:

            if expression\_object not in knowledge\_dict:

                proof\_dict[expression\_object] = None

                return None

            else:

                proof\_dict[expression\_object] = knowledge\_dict[

                    expression\_object]

                return proof\_dict[expression\_object]

    # Second base case

    elif expression\_object.is\_pure\_proposition() is True:

        proof\_dict[expression\_object] = expression\_object.general\_resolver()

    else:

        proof\_dict[expression\_object] = expression\_object.general\_resolver()

        if expression\_object.recognizer() == "Conditional":

            for expression in parsed\_expression.values():

                expression\_type = expression.recognizer()

                # Recursive step

                if expression\_type != "Pure" and expression\_type != "Broken":

                    validator(expression.get())

        else:

            for expression in parsed\_expression:

                expression\_type = expression.recognizer()

                # Recursive step

                if expression\_type != "Pure" and expression\_type != "Broken":

                    validator(expression.get())

# ------------------------------------------------------------------------------

if \_\_name\_\_ == '\_\_main\_\_':

    # Knowledge dict is where we store all the definitions defined by definer

    # function.

    knowledge\_dict = dict()

    user\_input = ""

    print("Please keep entering the logical arguments you would like to" +

          " define.\nTo see the results and further validate new arguments" +

          " based on your arguments enter -1.")

    input\_list = list()

    while user\_input != "-1":

        user\_input = input("\nNew argument:\t")

        if user\_input != "-1":

            expression\_object = Expression(user\_input)

            if expression\_object.recognizer() == "Broken":

                print("Incorrect Syntax. Please try again: ")

                continue

            elif expression\_object.recognizer() == "Pure":

                if expression\_object.valid\_parentheses\_checker() is False:

                    print("Parentheses do not exist or aren't in a valid form.")

                    continue

            input\_list.append(user\_input)

    failed\_expression = set()

    for count in range(2):

        # We repeat the procedure of defining to make sure all the elements get

        # updated.

        for expression in input\_list:

            try:

                interpreter(expression)

            except Exception:

                failed\_expression.add(expression)

    for expression in failed\_expression:

        print("\nThis expression could not be submitted due to a problem: ",

              expression)

    print(40 \* "-" + "\nExpressions and arguments you defined: ")

    for expression in knowledge\_dict:

        print(expression.get(), "--->", knowledge\_dict[expression])

    print(40 \* "-" + "\nEnter the new argument you would like to validate: " +

          "\nEnter 'view' at any time to see the full list of arguments and" +

          " their results\nEnter 'exit' to quit.")

    # proof\_dict is where we store all the validation results we resolved.

    # At the beelining we copy all the elements of knowledge\_dict so resolver

    # can use previously defined arguments.

    proof\_dict = knowledge\_dict.copy()

    while user\_input != "exit":

        user\_input = input("\nValidate:\t")

        if user\_input != "exit":

            if user\_input == "view":

                for expression in proof\_dict:

                    print(expression.get(), "--->", proof\_dict[expression])

            else:

                for i in range(2):

                    # We repeat the procedure of defining to make sure all the

                    # elements get validated.

                    try:

                        validator(user\_input)

                    except Exception:

                        "There is a problem with this argument."

                        continue

                for expression in proof\_dict:

                    if expression.get() == user\_input:

                        print(user\_input, "----->", proof\_dict[expression])

**OUTPUT:**

Graphical user interface, text, application, email

Description automatically generated