**LAB 8: Implementation of Prepositional logic for real world problems**

**Name: Priyanshu Rishi**

**Reg No: RA1911042010041**

**Batch: CSBS-(R1)**

**Code:-**

import re

class Expression:

    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):

        return self.expression

    def set(self, new\_expression):

        self.expression = new\_expression

    def recognizer(self):

        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:

            flag = True

            for i in self.keywords:

                if i in self.expression:

                    flag = False

                    break

            if flag:

                return "Pure"

            else:

                return "Broken"

    def valid\_parentheses\_checker(self):

        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):

        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):

        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):

        expression = self.expression

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

        self.expression = expression

    def temp\_negative\_inverter(self):

        expression = self.expression

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

        temp\_inverted\_object = Expression(expression)

        return temp\_inverted\_object

class Definer(Expression):

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

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

    def and\_definer(self):

        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):

        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):

        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):

        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):

        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):

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

            return True

        else:

            return None

    def and\_temp\_transformer(self):

        self.expression = self.expression + "@"

class Resolver(Expression):

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

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

    def and\_resolver(self):

        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):

        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):

        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):

        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):

    flag = False

    if expression[-1] == "@":

        flag = True

        expression = expression[0:-1]

    expression\_object = Definer(expression)

    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

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

        if flag:

            knowledge\_dict[

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

        else:

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

    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()

        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()

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

                    interpreter(expression.get())

        else:

            for expression in parsed\_expression:

                expression\_type = expression.recognizer()

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

                    expression.and\_temp\_transformer()

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

                    interpreter(expression.get())

def validator(expression):

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

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

    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]

    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()

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

                    validator(expression.get())

        else:

            for expression in parsed\_expression:

                expression\_type = expression.recognizer()

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

                    validator(expression.get())

if \_\_name\_\_ == '\_\_main\_\_':

    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):

        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 = 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):

                    try:

                        validator(user\_input)

                    except Exception:

                        "There is a problem with this argument."

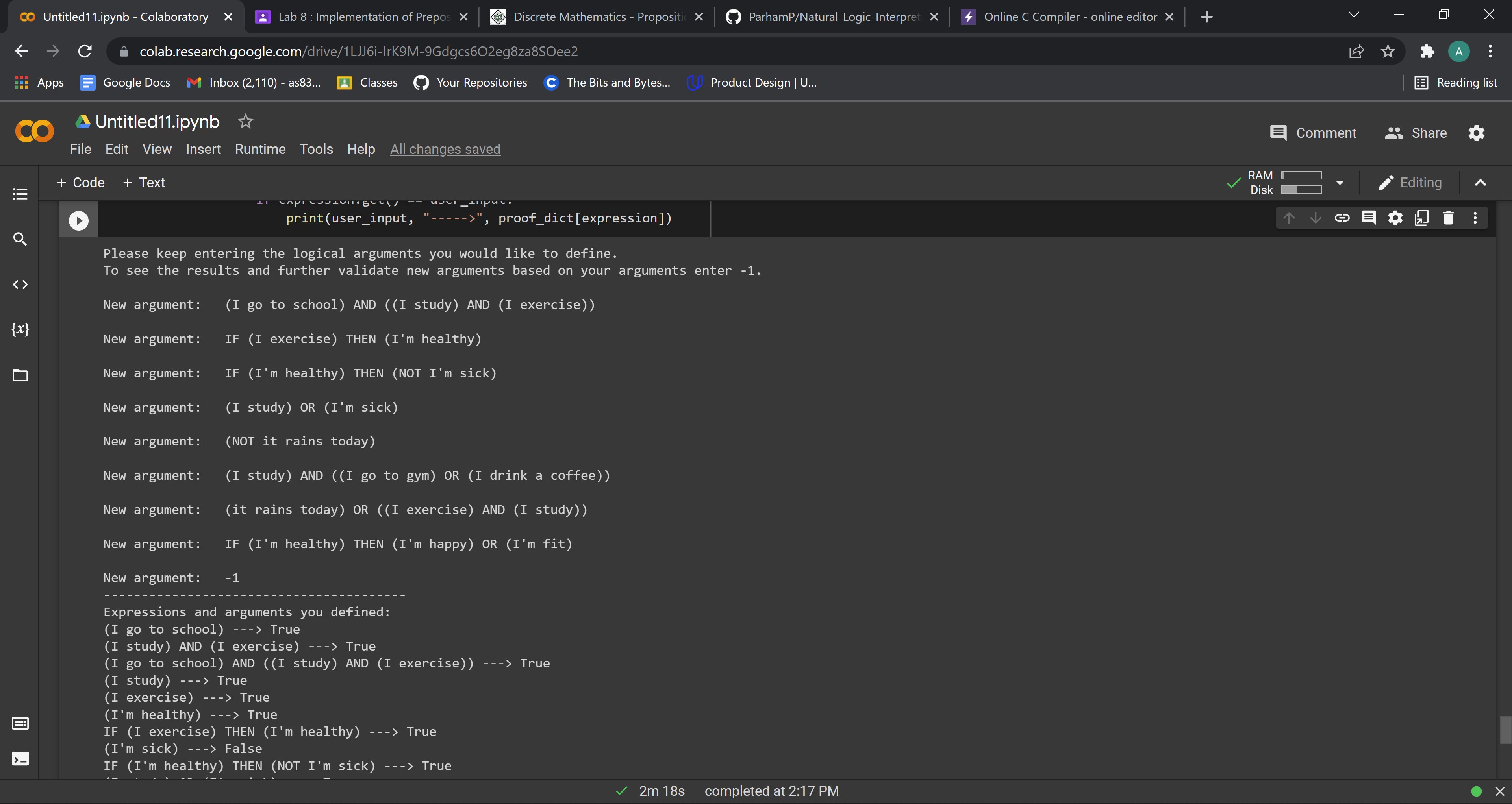
                        continue

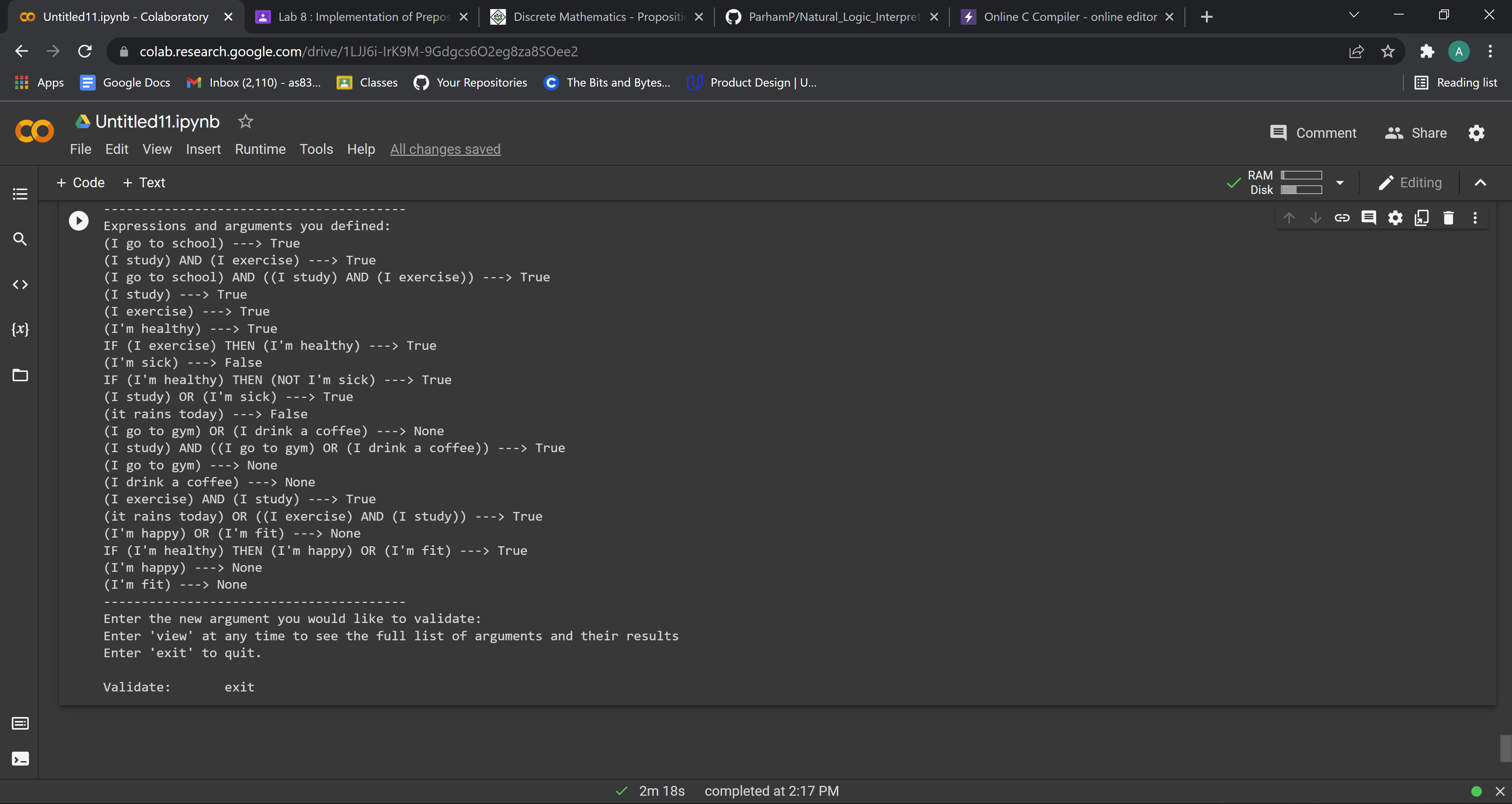
                for expression in proof\_dict:

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

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

**Output:-**

****

****