COMP 5361, Assignment 1 programming

Ryan Mokarian (40080413)

Program Flow Summary:

(1) Program starts with calling the Main function, located at the end of the program.

(1.1) q1_in.txt and q2_in.txt files are opened and read.

(1.2) Q1_Output_Result and Q2_Output_Result functions are called and from them answers to the assignment's question 1 and 2 are obtained.

(1.3) Answers are written in 'q1_out.txt' and 'q2_out.txt' files.

(2) Q1_Output_Result and Q2_Output_Result functions are called.

(2.1) From them, Solve function is called.

(2.2) Input of the solve function are proposition statements.

(2.3) Output of the solve function are the statements' truth value (for Q1) and the statements' equivalence classification (for Q2).

(3) Solve function is called From it, Simply function is called to make a proposition statement decomposed to several simple logical statements.

(4) Simply function is called. From it, eval_logic function is called to evaluate truth value of each simplified logical statements.

(5) eval_logic function is called.

(5.1) Takes a simple logical statement (either an inner most parenthesis proposition from the "simplify" function or a proposition statement without parenthesis and with more than one operator).

(5.2) Call several logical operations functions to evaluate the simple logical statement.

(5.3) Returns a binary number associated to the simple logic statement.

Note: Calculated binary numbers are returned back from eval_logic function and are composed in simply and solve functions and then passed back to the previous functions all the way up to the main function. Other not mentioned functions, which are explained at the beginning of each function, are used for the facilitation of the evaluation process.

FUNCTIONS

def truthValuesCombination(propVariablesNum)

Takes the number of propositional variables and Returns all possible combinations of 1 or 0 (Ture/False) values.

">>> truthValuesCombination(1)

```
[{'P1': '0'}, {'P1': '1'}]
```

```
def truthValuesCombination(n):
    """
    Takes the number of propositional variables and Returns all possible combinations of 1 or 0 (Ture/False) values.

   # >>> truthValuesCombination(1)
    [{'P1': '0'}, {'P1': '1'}]
    """
    outerList =[]
    for i in range(n):
        outerList.append([])
    for i in range(n):
        c = 0
        while c< pow(2, n):
            for j in range(int(pow(2, n - i) / 2)):
                outerList[i].append(0)
                c = c + 1
            for m in range(int(pow(2, n - i) / 2)):
                outerList[i].append(1)
                c = c + 1
    truthValuesCombination = []
    propTruth = dict()
    temp = dict()
    for i in range(int(pow(2, n))):
        for j in range(n):
            temp['P' + str(j+1)] = str(outerList[j][i])
            propTruth = dict(temp)
        truthValuesCombination.append(propTruth)
```

```
    return truthValuesCombination
```

def parenthetic_contents(string):

Takes a proposition statement, identify its parenthesis and Return content of each inner and or outer parenthesis as pairs (parenthesis level, contents).

">>> (parenthetic_contents('(-(P1 ∨ P2) ↔ (-P1 ∧ -P2))')

```
[(1, 'P1 ∨ P2'), (1, '-P1 ∧ -P2'), (0, '-(P1 ∨ P2) ⇔ (-P1 ∧ -P2)')]
```

```
def parenthetic_contents(string):
    """
    Takes a proposition statement, identify its parenthesis and Return content of each inner and or outer parenthesis
     as pairs (parenthesis level, contents).

    #>>> (parenthetic_contents('(-(P1 ∨ P2) ⇔ (-P1 ∧ -P2))')
    [(1, 'P1 ∨ P2'), (1, '-P1 ∧ -P2'), (0, '-(P1 ∨ P2) ⇔ (-P1 ∧ -P2)')]
    """

    stack = []
    for i, char in enumerate(string):
        if char == '(':
            stack.append(i)
        elif char == ')' and stack:
            start = stack.pop()
            yield (len(stack), string[start + 1: i])
```

def parenthetic_contents(string):

Takes a proposition statement and truth values of proposition variables and Return it in a proper syntax to be used in an evaluation function.

""">>>changeToLogValSymb([P1, AND, (, P1, THEN, False,)],{P1:1})

[1, ∧, (, 1, →, 0,)]

```
def changeToLogValSymb(rawList,propTruth):
    """
    Takes a proposition statement and truth values of proposition variables and Return it usable for an
     evaluation function.
```

```
    # >>> changeToLogValSymb([P1, AND, (, P1, THEN, False, )],{P1:1})
     [1, ∧, (, 1, →, 0, )]
     """

     cookedList = []
     for i in range(len(rawList)):
         if rawList[i] == 'P1':
             cookedList.append(propTruth['P1'])
         elif rawList[i] == 'P2':
             cookedList.append(propTruth['P2'])
         elif rawList[i] == 'P3':
             cookedList.append(propTruth['P3'])
         elif rawList[i] == 'P4':
             cookedList.append(propTruth['P4'])
         elif rawList[i] == 'AND':
             cookedList.append('∧')
         elif rawList[i] == 'OR':
             cookedList.append('∨')
         elif rawList[i] == 'THEN':
             cookedList.append('→')
         elif rawList[i] == 'EQ':
             cookedList.append('↔')
         elif rawList[i] == 'TRUE':
             cookedList.append('1')
         elif rawList[i] == 'FALSE':
             cookedList.append('0')
         elif rawList[i] == ')\n':
             cookedList.append(')')
         elif rawList[i] == 'P1\n':
             cookedList.append(propTruth['P1'])
         elif rawList[i] == 'P2\n':
             cookedList.append(propTruth['P2'])
         elif rawList[i] == 'P3\n':
             cookedList.append(propTruth['P3'])
         elif rawList[i] == 'P4\n':
             cookedList.append(propTruth['P4'])
         else:
             cookedList.append(rawList[i])
     return cookedList
```

Logival Functions

In the following, functions for logical evaluations are presented.

```
def and_func(p, q):
    """ Evaluates truth of AND operator for boolean variables p and q."""
```

```
    return p and q

def or_func(p, q):
    """ Evaluates truth of OR operator for boolean variables p and q."""
    return p or q
def conditional(p, q):
    """Evaluates truth of conditional for boolean variables p and q."""
    return False if p and not q else True

def biconditional(p, q):
    """ Evaluates truth of biconditional for boolean variables p and q."""
    return (True if p and q
            else True if not p and not q
    else False)

def negate(p):
    """ Evaluates truth of NOT operator for boolean variables p and q."""
    return not p

def apply_negations(string):
    """
    Applies the '-' operator when it appears directly before a binary number.

   # >>> apply_negations('-1 ∧ 0')
    '0 ∧ 0'
    """

    new_string = string[:]
    for i, char in enumerate(string):
        if char == '-':
            try:
                next_char = string[i+1] # Character proceeding '-'
                num = int(next_char)
                negated = str(int(negate(num)))
                new_string = new_string.replace('-'+string[i+1], negated)
            except:
                # Character proceeding '-' is not a number
                pass
    return new_string
```

def solve(valued_statement):

Takes a proposition statement containing binary numbers and logical operators, evaluate and Returns answer as binary integer.

""">>> solve([(, -, (, 0, ∧, 0,), ↔, (, -, 0, ∨, -, 0,),)])

1

```
def solve(valued_statement):
    """
    Takes a proposition statement containing binary numbers and logical operators, evaluate and Returns answer
     as binary integer.

    #>>> solve([(, -, (, 0, ∧, 0, ), ⇔, (, -, 0, ∨, -, 0, ), )])
    1
    """

    while len(valued_statement) > 1:
        valued_statement = simplify(valued_statement)
    return int(valued_statement)
```

def simplify(valued_statement):

Take a proposition statement from "solve" function that contains binary numbers and logical operators and evaluate the statements contained in the innermost parentheses. For example, a proposition statement with n levels of parenthesis is taken, inner most parenthesis is evaluated and a string type statement with n-1 levels of parenthesis is returned.

""">>> simplify([(, -, (, 0, ∧, 0,), ↔, (, -, 0, ∨, -, 0,),)])

'(-0 ↔ 1)'

```
def simplify(valued_statement):
    """
    Take a proposition statement from "solve" function that contains binary numbers and logical operators and
    evaluate the statements contained in the innermost parentheses. For example, a proposition statement with n
    levels of parenthesis is taken, inner most parenthesis is evaluated and a string type statement with n-1
    levels of parenthesis is returned.

    #>>> simplify([(, -, (, 0, ∧, 0, ), ⇔, (, -, 0, ∨, -, 0, ), )])
    '(-0 ⇔ 1)'
    """

    valued_statement = ''.join(valued_statement)
    brackets_list = list(parenthetic_contents(valued_statement))
    if not brackets_list:
        # There are no brackets in the statement
        return str(eval_logic(valued_statement))
    deepest_level = max([i for (i,j) in brackets_list]) # Deepest level of nested brackets
    for level, string in brackets_list:
        if level == deepest_level:
            bool_string = str(eval_logic(string))
```

```
            valued_statement = valued_statement.replace('('+string+')', bool_string)
    return valued_statement
```

def eval_logic(string):

Takes a simple logical statement (either an inner most parenthesis proposition from the "simplify" function or a proposition statement without parenthesis and with more than one operator) and Returns a binary numbers.

""">>> eval_logic('1 ∧ 0')

0

```
def eval_logic(string):
    """
    Takes a simple logical statement (either an inner most parenthesis proposition from the "simplify" function
    or a proposition statement without parenthesis and with more than one operator) and Returns a binary numbers.

   # >>> eval_logic('1 ∧ 0')
    0
    """
    string = apply_negations(string) # Switch -0 to 1, -1 to 0
    # Convert string to list
    statement = list(string)
    operators = { '∧': and_func, '∨': or_func, '→': conditional, '↔': biconditional }
    # # recursive base condition (only one operator)
    if len(statement) == 1:
        return string
    elif len(statement) == 3:
        for i in range(3):
            if statement[i] in operators:
                boolean = operators[statement[i]](int(statement[i - 1]), int(statement[i + 1]))
        return str(int(boolean))  # Return boolean as 0 or 1
    # recursive other conditions (more than one operator)
    elif len(statement) > 3:
        temp = []
        if statement[1] in operators:
            boolean = operators[statement[1]](int(statement[0]), int(statement[2]))
            statement.pop(0)
            statement.pop(0)
            statement.pop(0)
            statement.insert(0, str(int(boolean)))
            for j in range(len(statement)):
                temp.append(statement[j])
        result = eval_logic(temp)
        return result
```

```
    else:
        try:
            len(statement) < 2
        except:
            # None of the logical operators were found in the statement
            return print('propositional statement is not correct')
```

def Q1_Output_Result(file_handler):

Takes the opened file handler related to q1_in.txt file from the main function and for each line of the input file call the solve function to evaluate the proposition statement and Returns a list of the evaluations to the main function as answer to Question 1.

""">>> Q1_Output_Result(file_handler1)

[[proposition statement], [valued statement], [statement's truth value]]

```
def Q1_Output_Result(file_handler):
    """
    Takes the opened file handler related to q1_in.txt file from the main function and for each line of the input file
    call the solve function to evaluate the proposition statement and Returns a list of the evaluations to the main
    function as answer to Question 1.

   # >>> Q1_Output_Result(file_handler1)
    [ [proposition statement], [valued statement], [statement's truth value ] ]
    """

    # Create  list for output results
    q1_output_result = []
    # Convert the file text to the list of lines
    line = file_handler.readlines()
    for lineNumber in range(len(line)):
        # Sperate each line to a list of (1) propositional
        # variables; (2)  truth assignment; (3) propositional sentence
        listLine = line[lineNumber].split('\t')
        # Make a list from the propositional variables
        propVal = listLine[0].split(',')
        # Make a list from the truth assignments
        truthBool = listLine[1].split(',')
        # Change T and F to 1 and 0
        truthVal0 = []
        for i in range(len(truthBool)):
            if truthBool[i] == 'T':
                truthVal0.append('1')
            elif truthBool[i] == 'F':
                truthVal0.append('0')
```

```
        # Use a dictionary structure to pair propositions with their truth values
        propTruth = dict()
        for j in range(len(propVal)):
            propTruth[propVal[j]] = (truthVal0[j])
        # Make a list from the propositional statement
        statement0 = listLine[2].split(' ')
        # Convert propositional statement to a list of logical values and symbols
        valued_statement = (changeToLogValSymb(statement0, propTruth))
        q1_output_result.append([statement0, valued_statement, solve(valued_statement)])
    return q1_output_result
```

def Q2_Output_Result(file_handler):

Takes the opened file handler related to q2_in.txt file from the main function and for each line of the input file call the solve function to evaluate the proposition statement for all combinations of truth value assignments to classify the statement to 'Tautology', 'Contingency' or 'Contradiction'. Returns a list of the statements' classifications to the main function as answer to Question 2.

""">>> Q2_Output_Result(file_handler2)

[[proposition statement], [truth table values], [classification], [number of proposition variables]]

```
def Q2_Output_Result(file_handler):
    """
    Takes the opened file handler related to q2_in.txt file from the main function and for each line of the input
    file call the solve function to evaluate the proposition statement for all combinations of truth value
    assignments to classify the statement to 'Tautology', 'Contingency' or 'Contradiction'. Returns a list of
    the statements' classifications to the main function as answer to Question 2.

   # >>> Q2_Output_Result(file_handler2)
    [[proposition statement], [truth table values], [classification], [number of proposition variables]]
    """

    # Create  list for output results
    q2_output_result = []
    # Convert the file text to the list of lines
    line = file_handler.readlines()
    for lineNumber in range(len(line)):
        # Separate each line to a list of (1) propositional
        # variables; (2)  truth assignment; (3) propositional sentence
        listLine = line[lineNumber].split('\t')
        # Make a list from the propositional variables
        propVal = listLine[0].split(',')
        # Make a list from the truth assignments
        statement0 = listLine[1].split(' ')
```

```
        # Make a combination list of truth assignments
        propVariablesNum = len(propVal)
        combo_list = truthValuesCombination(propVariablesNum)
        truth_table_values = []
        for propTruth in combo_list:
            # Convert propositional statement to a list of logical values and symbols
            # print('before func', statement0)
            valued_statement = (changeToLogValSymb(statement0, propTruth))
            answer = solve(valued_statement)
            truth_table_values.append(answer)
        if all(truth_table_values):
            propClassifier = 'Tautology'
        elif any(truth_table_values):
            propClassifier = 'Contingency'
        else:
            propClassifier = 'Contradiction'
        q2_output_result.append([statement0, truth_table_values, propClassifier, propVariablesNum])
    return q2_output_result
```

def Q1_Detailed_Result_AsAReference(Q1_Prop_Result):

Note this function is an additional to the requirement of the assignment as a complemental reference. Takes solution to question 1 from function Q1_Prop_Result and Returns the result in a more detail format in

'q1_DetailedOut_asReference.txt' file.

```
def Q1_Detailed_Result_AsAReference(Q1_Prop_Result):
    """
    Note this function is an additional to the requirement of the assignment as a complemental reference.
    Takes solution to question 1 from function Q1_Prop_Result and Returns the result in a more detail format in
    q1_DetailedOut_asReference.txt' file.
    """

    with open('q1_DetailedOut_asReference.txt','w') as file_handler6:
        for i in range(len(Q1_Prop_Result)):
            print('\nProposition Sentence(' + str(i+1)+ '): ' + ''.join(Q1_Prop_Result[i][0]) + 'i.e. ' + ''.join(Q1_Prop_Result[i][1]))
            print('Proposition Sentence Truth Value is "' + str(bool(Q1_Prop_Result[i][2]))+ '"\n')
            file_handler6.write('\nProposition Sentence(' + str(i+1)+ '): ' + ''.join(Q1_Prop_Result[i][0]))
            file_handler6.write('Proposition Sentence Truth Value is "' + str(bool(Q1_Prop_Result[i][2]))+ '"\n')
```

def Q2_Detailed_Result_AsAReference(Q2_Prop_Result):

Note this function is an additional to the requirement of the assignment as a complemental reference. Takes solution to question 2 from function Q2_Prop_Result and Returns the result in a more detail format in

```
    >'q2_DetailedOut_asReference.txt' file.
```

```
def Q2_Detailed_Result_AsAReference(Q2_Prop_Result):
    """
    Note this function is an additional to the requirement of the assignment as a complemental reference.
    Takes solution to question 2 from function Q2_Prop_Result and Returns the result in a more detail format in
    q2_DetailedOut_asReference.txt' file.
    """

    with open('q2_DetailedOut_asReference.txt','w') as file_handler5:
        for i in range(len(Q2_Prop_Result)):
            print('\n\nProposition Sentence: ' + ''.join(Q2_Prop_Result[i][0]))
            file_handler5.write('\nProposition Sentence(' + str(i+1)+ '): ' + ''.join(Q2_Prop_Result[i][0]))
            print('Truth Table Values:')
            file_handler5.write('Truth Table Values:')
            n = (Q2_Prop_Result[i][3])
            combo_list = truthValuesCombination(n)
            print(list(combo_list[0].keys()), '        Truth Value')
            file_handler5.write('\n' + str(list(combo_list[0].keys())) + '        Truth Value')
            for j in range(len(Q2_Prop_Result[i][1])):
                print(list(combo_list[j].values()), '             ', Q2_Prop_Result[i][1][j])
                file_handler5.write('\n' + str(list(combo_list[j].values())) + '             ' + str(Q2_Prop_Result[i][1][j]))
            print('Proposition Sentence is equivalent to "', Q2_Prop_Result[i][2], '"')
            file_handler5.write('\n' + 'Proposition Sentence is equivalent to "' + Q2_Prop_Result[i][2] + '"\n')
```

def main()

In the Main function, 'q1_in.txt' and 'q2_in.txt' are opened and read.

From Main function, Q1_Output_Result and Q2_Output_Result functions are called and from those functions answers to the assignment question 1 and 2 are obtained and written in 'q1_out.txt' and 'q2_out.txt' files.

```
def main():
    """
    Note this is the starting point of the program. 'q1_in.txt' and 'q2_in.txt' are opened and read, After
    calling Q1_Output_Result and Q2_Output_Result functions, answers to the assignment question 1 and 2 are
    obtained and written in 'q1_out.txt' and 'q2_out.txt' files.
    """
```

```
    # Open and read input file for question 1 and obtain its result
    with open('q1_in.txt') as file_handler1:
        Q1_Prop_Result = Q1_Output_Result(file_handler1)
        Q1_Detailed_Result_AsAReference(Q1_Prop_Result)
    # Write output result for question 1 in q1_out.txt
    with open('q1_out.txt','w') as file_handler2:
        for i in range (len(Q1_Prop_Result)):
            file_handler2.write(str(bool(Q1_Prop_Result[i][2])))
            file_handler2.write('\n')
    # Open and read input file for question 2 and obtain its result
    with open('q2_in.txt') as file_handler3:
        Q2_Prop_Result = Q2_Output_Result(file_handler3)
        Q2_Detailed_Result_AsAReference(Q2_Prop_Result)
    # Write output result for question 2 in q1_out.txt
    with open('q2_out.txt','w') as file_handler4:
        for i in range (len(Q2_Prop_Result)):
            file_handler4.write(Q2_Prop_Result[i][2])
            file_handler4.write('\n')
```

Starting point of the program

```
if __name__ == '__main__':
    main()
```

```
Proposition Sentence(1): ((P1ANDP2)OR(P3ANDTRUE))OR((-P1AND-P3)ANDP2)
i.e. ((1∧1)∨(1∧1))∨((-1∧-1)∧1)
Proposition Sentence Truth Value is "True"


Proposition Sentence(2): ((P1ANDP2)OR(P3ANDTRUE))OR((-P1AND-P3)ANDP2)
i.e. ((1∧0)∨(0∧1))∨((-1∧-0)∧0)
Proposition Sentence Truth Value is "False"


Proposition Sentence(3): (P1THENP2)AND(P2THENP1)
i.e. (1→1)∧(1→1)
Proposition Sentence Truth Value is "True"


Proposition Sentence(4): (P1THENP2)AND(P2THENP1)
i.e. (0→1)∧(1→0)
Proposition Sentence Truth Value is "False"


Proposition Sentence(5): (P1THENP2)AND(P2THENP1)EQ(P1EQP2)
i.e. (0→1)∧(1→0)⇔(0⇔1)
Proposition Sentence Truth Value is "True"


Proposition Sentence(6): P1ANDP2ANDP3ORP4
i.e. 1∧0∧1∨0
Proposition Sentence Truth Value is "False"


Proposition Sentence(7): -P1AND(P1THENP2)
i.e. -1∧(1→1)
Proposition Sentence Truth Value is "False"


Proposition Sentence(8): ((P1THENP2)OR(P2THENP3))EQ(P1THENP3)THENP2
i.e. ((0→0)∨(0→1))⇔(0→1)→0
Proposition Sentence Truth Value is "False"


Proposition Sentence(9): ((P1THENP2)OR(P2THENP3))THEN((P1THENP3)THENP2)OR-P1
```

```
i.e. ((0→1)∨(1→1))→((0→1)→1)∨-0
Proposition Sentence Truth Value is "True"


Proposition Sentence(10): (P1ORFALSE)AND(P2ORTRUE)i.e. (0∨0)∧(1∨1)
Proposition Sentence Truth Value is "False"



Proposition Sentence: ((P1ANDP2)OR(P3ANDTRUE))OR((-P1AND-P3)ANDP2)

Truth Table Values:
['P1', 'P2', 'P3']         Truth Value
['0', '0', '0']               0
['0', '0', '1']               1
['0', '1', '0']               1
```