Discrete Mathematics Fall 2019

Coding Assignment 1: Propositional Logic

Due date October 7, 2019

Academic Honesty Policy:

You are expected to submit your own work. Assignments are **to be completed individually**, NOT in groups. No collaboration is permitted, unless otherwise specified. Please do not include any of your code snippets or algorithms in public Piazza posts. You cannot not use solutions from any external source. You are not permitted to publish code or solutions online, nor post the course questions on forums including stack overflow. We run plagiarism detection software. If you have any questions at all about this policy please ask the course staff before submitting your assignment.

Please note:

- Feel free to import any standard Python libraries you need as far as output and input of functions meet the requirement. Use the provided template to implement the functions.
- Please name your file as "UNI_coding1.py".
- A skeleton of each function has been provided to you. Please expect further instructions as we finalize auto-grading scripts. In any case, DO NOT modify the function signatures or return variables for any reason. Your job is to complete the body of each function. We have included test cases so you can check your solutions. These test cases are under the line:

  ```
  if __name__ == "__main__":
  ```

- We provide more propositions at the end of this document to try your code.
- Make sure to use Python 3.x. To check your Python version, open a terminal window and enter:

  ```
  python --version
  ```

- Although you will not be graded on your coding style, it is good to familiarize yourself with PEP8 style guide for Python. Learn more about PEP8 here:
  `https://www.python.org/dev/peps/pep-0008/`
- To receive full points, make sure your code compile. We recommend using Spider, Pycharm or Google colab. They all allow you to download .py files. Be aware that if you write your code in some platforms like Codio and copy and paste it in a text file, there may be spurious characters in your code, and your code will not compile. Always ensure that your .py compiles.

Specification:

One way to represent a compound proposition in Python is to use lists as follows: the first element is a logical connective, and the remaining elements will be the atomic propositions in the compound proposition.
For example:

- $p_1 \wedge \neg p_2 \implies p_3$ will be represented with the list

```
["if", ["and", "p1", ["not", "p2"] ] , "p3"]
```

- $p_1 \iff p_2 \vee \neg p_3$ will be represented with the list

```
["iff", "p1", ["or", "p2", ["not", "p3"]]]
```

(a) Write a python function `format_prop` that allows to print a proposition in Propositional Logic using the indicated list representation. Consider the connectives: $\vee$, $\wedge$, $\implies$, $\iff$, $\neg$. and $\oplus$ (XOR).

Remark. To facilitate the problem assume that we consider only unary operator `not` and binary operations: functions that take only two inputs.

For example, such list will be a valid input:

```
["or", "p1", "p2"]
```

List with more variables is an invalid input:

```
["or", "p1", "p2", "p3"]
```

Your python function `format_prop` should take one argument `prop`, list of strings and other lists. It should return a formatted string corresponding to the desired compound proposition. Assume all inputs are valid.

```
def format_prop(prop):
    #your code
    return #your string
```

For example, inputs from our sample example

```
lst1 = ["if", ["and", "p1", ["not", "p2"]], "p3"]
lst2 = ["iff", "p1", ["or", "p2", ["not", "p3"]]]
print(format_prop(lst1))
print(format_prop(lst2))
```

produce the following strings:

```
((p1 and (not p2)) -> p3)
(p1 <-> (p2 or (not p3)))
```

You may assume that atomic propositions are in the format of `p[1-9]`, meaning that they have a prefix `p` and one digit from 1 to 9, or a string *true* or *false*. The list of valid operators is

```
["or", "and", "if", "iff", "not", "xor"]
```

You may wrap all operations into parentheses even if they are not required.
Hint: You may find recursion to be very helpful.

(b) Given a proposition p over atomic propositions p_1, p_2, $\cdots$, p_n, $n \leq 9$, and a truth value assigned to each atomic proposition, evaluate whether p is true or false under this assignment.

Assume that the values of atomic propositions are given as a Python list of length n where each element is 0 or 1. For example if we have $p_1 = 0, p_2 = 1, p_3 = 0$ the corresponding list is:

```
values = [0, 1, 0]
```

Your python function `evaluate` should take 2 arguments `prop`, a proposition in the list representation, and `values`, values of atomic propositions. It should return a Boolean value or a corresponding integer, 0 or 1. Assume all inputs are valid.

```
def eval_prop(prop, values):
    #your code
    return #your boolean or integer (0 or 1)
```

Hint: Again you may find recursion to be very helpful.

(c) Print the truth table for p. Feel free to use any formatting as far as the table your code outputs is readable.

Your python function `print_table` should take two arguments: `prop` in the list representation and an integer defining the number of atomic propositions `n_var`. It should print a corresponding truth table. Assume all inputs are valid.

```
def print_table(prop, n_var):
    #your code
```

Hint: Use functions `format_prop` and `eval_prop`

Example of propositions to try:

1. $p_1 \rightarrow p_2$
2. $true \vee (p_1 \wedge p_2)$
3. $p_1 \wedge p_2$
4. $p_1 \vee p_2$
5. $p_1 \leftrightarrow p_2$
6. $\neg p_1 \rightarrow p_2$
7. $(p_1 \vee p_2) \vee (p_1 \vee \neg p_2)$
8. $(\neg(\neg p_1)) \leftrightarrow p_1$
9. $((p_1 \rightarrow p_2) \wedge (p_2 \rightarrow p_3)) \rightarrow (p_1 \rightarrow p_3)$
10. $(p_1 \vee p_2) \wedge (\neg p_1 \wedge \neg p_2)$
11. $\neg((p_1 \rightarrow false) \rightarrow \neg p_1)$
12. $\neg(p_2 \rightarrow p_1) \wedge ((p_1 \wedge p_2) \rightarrow p_3) \wedge p_4$
13. Try more!