Discrete Mathematics Spring 2023

Coding Assignment 2: Propositional Logic

Due Date: February 20, 2024

Academic Honesty Policy:

Assignments are **to be completed individually**. No collaboration is permitted. Please do not include any of your code snippets or algorithms in public EdStem posts. You cannot 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.

Read Carefully:

- 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.
- You **must name your file** "coding2.py". Any submission that does not follow this naming will not be graded and **will receive a zero.**
- You **must make sure your file extension is .py.** Remember that online environments like Google Colab might export a **.ipynb** file; this is **not the correct filetype.** Colab (and other environments) have the option to export as **.py.** If you are using some sort of Python notebook, *make sure your code compiles in a single block before exporting.*
- A skeleton of each function has been provided to you. **You are expected to ONLY write code in the functions and blocks specified.** In any case, DO NOT modify the function signatures, return variables, or any code that is not specified to be modifiable (we will include comments in the code to distinguish these sections) for any reason. Also, though you may modify the `main` function to test other cases, **do not turn in an assignment with a modified main function, i.e. anything below the line shown below.** This will be run by our autograder, so any unexpected modifications that make it malfunction **will receive a zero.**

```
### DO NOT TURN IN AN ASSIGNMENT WITH ANYTHING BELOW HERE MODIFIED ###
if __name__ == "__main__":
```

- Test cases will be provided in the `main` function in the code. Several of the test cases for each part will be provided to you, and the others will be hidden test cases on our side (all graded via autograder).
- **Only Python 3.x versions will be graded.** To check your Python version locally, open a terminal window and enter:

```
python --version
```

  Google Colab should use Python 3 by default, but, to check, under "Runtime," navigate to "Change runtime type."

- To receive points, make sure your code runs. We recommend using Spyder, 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. Code that does not run will not receive any points.**

Part A: Format Prop

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 (up to two) will be the atomic propositions in the compound proposition.

For example:

- $\neg p_1$ will be represented with the list

```
["not", ["p1"]]
```

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

```
["if", ["p1"], ["p2"]]
```

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

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

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). In the code, they will be written, respectively, as: `or`, `and`, `if`, `iff`, `not`, `xor`.

You may safely assume that we consider only the unary operator as `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`, which may be a (possibly nested) list of lists (of strings), as above. 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 strings in parentheses even if they are not required.

Guidelines/Hints:

- You will likely have to use recursion for this. We have added comments and starting code in the skeleton code to guide you in structuring your recursion.
- To achieve recursion, you will have to call `format_prop` in your function. We have placed comments and starting code to guide you in this.
- However, you do not need to follow our starting code – if you have a solution that you think works better/you understand better, feel free to delete any hint code. Just make sure you return a single, correctly formatted string in the end.
- You will have three possible cases for `format_prop`: atomic proposition (list of length 1), unary operator (list of length 2), and binary operator (list of length 3). It is important to think about what your **base case** is, and what to do when you reach that case.

Part B: Evaluate Prop

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. The proposition will be given to you in the same form as Part A.

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 `eval_prop` should take 2 arguments `prop`, a proposition in the list representation, and `values`, values of atomic propositions. It should return 0 for `False` and 1 for `True` (*not* the Boolean value itself).

```
def eval_prop(prop, values):
    #your code
    return # Integer (0 or 1, for True/False)
```

Guidelines/Hints:

- Again, you will have to use recursion for this part. We have laid out the basic structure of the recursion in the skeleton code; it is up to you to fill in the details.
- As in Part A, if you have a solution that doesn't adhere to the structure we've placed down to help you start, you're free to delete any of that starting code. Just make sure you are returning either 0 or 1 in the end.
- The **base case** is similar to Part A, but you will now have to use standard logic rules in Python to evaluate truth values. These include: `and`, `or`, etc. Python has these binary operators already built-in exactly as they are in English. Refer to this documentation for the full list: `https://www.w3schools.com/python/python_operators.asp`
- You will likely continue running into `'list index out of range.'` Always make sure you are indexing into values correctly. You can do this by: (1) Checking that you are correctly taking the integer value from the proposition string (2) Making sure your `values` list is exactly the size you need to fit your proposition.
- You may index into any string like it is a list of characters. This will allow you to extract the position/index in the `values` list you want from each `prop` string.