Task

The task in this programming is to implement, a knowledge base and an inference engine for the wumpus world. First of all, you have to create a knowledge base (stored as a text file) storing the rules of the wumpus world, i.e., what we know about pits, monsters, breeze, and stench. Second, you have to create an inference engine, that given a knowledge base and a statement determines if, based on the knowledge base, the statement is definitely true, definitely false, or of unknown truth value.

Command-line Arguments

The program should be invoked from the commandline as follows:

```
check_true_false wumpus_rules.txt
[additional_knowledge_file] [statement_file]
```

For example:

```
check_true_false wumpus_rules.txt kb1.txt
statement1.txt
```

- Argument wumpus_rules.txt specifies the location of a text file containing the wumpus rules, i.e., the rules that are true in any possible wumpus world, as specified above (once again, note that the specifications above are not identical to the ones in the book).
- Argument [additional_knowledge_file] specifies an input file that contains additional information, presumably collected by the agent as it moves from square to square. For example, see kb3.txt.
- Argument [statement_file] specifies an input file that contains a single logical statement. The program should check if, given the information in wumpus_rules.txt and [additional_knowledge_file], the statement in [statement_file] is definitely true, definitely false, or none of the above.

Output

Your program should create a text file called "result.txt". Depending on what your inference algorithm determined about the statement being true or false, the output file should contain one of the following four outputs:

- **definitely true**. This should be the output if the knowledge base entails the statement, and the knowledge base does not entail the negation of the statement.
- **definitely false**. This should be the output if the knowledge base entails the negation of the statement, and the knowledge base does not entail the statement.
- **possibly true, possibly false**. This should be the output if the knowledge base entails neither the statement nor the negation of the statement.
- **both true and false**. This should be the output if the knowledge base entails both the statement and the the negation of the statement. This happens when the knowledge base is always false (i.e., when the knowledge base is false for every single row of the truth table).

Note that by "knowledge base" we are referring to the conjunction of all statements contained in wumpus_rules.txt AND in the additional knowledge file.
Also note that the sample code provided below stores the words "result unknown" to the result.txt file. Also, the "both true and false" output should be given when the knowledge base (i.e., the info stored in wumpus_rules.txt AND in the additional knowledge file) entails both the statement from statement_file AND the negation of that statement.

Syntax

The wumpus rules file and the additional knowledge file contain multiple lines. Each line contains a logical statement. The knowledge base constructed by the program should be a conjunction of all the statements contained in the two files. The sample code (as described later) already does that. The statement file contains a single line, with a single logical statement.
Statements are given in prefix notation. Some examples of prefix notation are:

```
(or M_1_1 B_1_2)
(and M_1_2 S_1_1 (not (or M_1_3 M_1_4)))
(if M_1_1 (and S_1_2 S_1_3))
(iff M_1_2 (and S_1_1 S_1_3 S_2_2))
(xor B_2_2 P_1_2)
P_1_1
B_3_4
(not P_1_1)
```

Statements can be nested, as shown in the above examples.

Note that:

- Any open parenthesis that is not the first character of a text line must be preceded by white space.
- Any open parenthesis must be immediately followed by a connective (without any white space in between).
- Any close parenthesis that is not the last character of a text line must be followed by white space.
- If the logical expression contains just a symbol (and no connectives), the symbol should NOT be enclosed in parentheses. For example, (P_1_1) is not legal, whereas (not P_1_1) is legal. See also the example statements given above.
- Each logical expression should be contained in a single line.
- The wumpus rules file and the additional knowledge file contain a set of logical expressions. The statement file should contain a single logical expression. If it contains more than one logical expression, only the first one is read.
- Lines starting with # are treated as comment lines, and ignored.
- You can have empty lines, but they must be totally empty. If a line has a single space on it (and nothing more) the program will complain and not read the file successfully.

There are six connectives: and, or, xor, not, if, iff. No other connectives are allowed to be used in the input files. Here is some additional information:

- A statement can consist of either a single symbol, or a connective connecting multiple (sub)statements. Notice that this is a recursive definition. In other words, statements are symbols or more complicated statements that we can make by connecting simpler statements with one of the six connectives.
- Connectives "and", "or", and "xor" can connect any number of statements, including 0 statements. It is legal for a statement consisting of an "and", "or", or "xor" connective to have no substatements, e.g., (and). An "and" statement with zero substatements is true. An "or" or "xor" statement with zero substatements is false. An "xor" statement is true if exactly 1 substatement is true (no more, no fewer).
- Connectives "if" and "iff" require exactly two substatements.
- Connective "not" requires exactly one substatement.

The only symbols that are allowed to be used are:

- M_i_j (standing for "there is a monster at square (i, j)).
- S_i_j (standing for "there is a stench at square (i, j)).
- P_i_j (standing for "there is a pit at square (i, j)).
- B_i_j (standing for "there is a breeze at square (i, j)).

NO OTHER SYMBOLS ARE ALLOWED. Also, note that i and j can take values 1, 2, 3, and 4. In other words, there will be 16 unique symbols of the form M_i_j, 16 unique symbols of the form S_i_j, 16 unique symbols of the form P_i_j, and 16 unique symbols of the form B_i_j, for a total of 64 unique symbols.

The Wumpus Rules

Here is what we know to be true in any wumpus world, for the purposes of this:

- If there is a monster at square (i,j), there is stench at all adjacent squares.

- If there is stench at square (i,j), there is a monster at one of the adjacent squares.
- If there is a pit at square (i,j), there is breeze at all adjacent squares.
- If there is breeze at square (i,j), there is a pit at one or more of the adjacent squares.
- There is one and only one monster (no more, no fewer).
- Squares (1,1), (1,2), (2,1), (2,2) have no monsters and no pits.
- The number of pits can be between 1 and 11.
- We don't care about gold, glitter, and arrows, there will be no information about them in the knowledge base, and no reference to them in the statement.

Efficiency

Brute-force enumeration of the 2^{64} possible to the 64 Boolean variables will be too inefficient to produce answers in a reasonable amount of time. Because of that, we will only be testing your solutions with cases where the additional knowledge file contains specific information about at least 48 of the symbols.

For example, suppose that the agent has already been at square (2,3). Then, the agent knows for that square that:

- There is no monster (otherwise the agent would have died).
- There is no pit (otherwise the agent would have died).

Furthermore, the agent knows whether or not there is stench and/or breeze at that square. Suppose that, in our example, there is breeze and no stench.

Then, the additional knowledge file would contain these lines for square 2,3:

```
(not M_2_3)
(not P_2_3)
B_2_3
(not S_2_3)
```

You can assume that, in all our test cases, there will be at least 48 lines like these four lines shown above, specifying for at least 48 symbols whether they are true or false. Assuming that you implement the TT-

Entails algorithm, your program should identify those symbols and their values right at the beginning. You can identify those symbols using these guidelines:

- Note that the sample code stores the knowledge base as a LogicalExpression object, whose connective at the root is an AND. Let's call this LogicalExpression object knowledge_base.
- Suppose that you have a line such as "B_2_3" in the additional knowledge file. Such a line generates a child of knowledge_base that is a leaf, and has its "symbol" variable set to "B_2_3". You can write code that explicitly looks for such children of knowledge_base.
- Suppose that you have a line such as "(not M_2_3") in the additional knowledge file. Such a line generates a child of knowledge_base whose connective is NOT, and whose only child is a leaf with its "symbol" variable set to "M_2_3". You can write code that explicitly looks for such children of knowledge_base.

This way, your program will be able to initialize the model that TT-Entails passes to TT-Check-All with boolean for at least 48 symbols, as opposed to passing an empty model. The list of symbols passed from TT-Entails to TT-Check-All should obviously NOT include the symbols that have been assigned values in the initial model. This way, at most 16 symbols will have unspecified values, and TT-Check-All will need to check at most 2^{16} rows in the truth table, which is quite doable in a reasonable amount of time (a few seconds).