Project Description

Today, your dad is opening his dream restaurant after working at a desk job for the last few decades. He has always been passionate about food, but there is one other thing he loves more: Money. Trying to cut some expenses, he came up with an amazing idea and convinced you to design an automated system to manage the restaurant for him. This automated system will take over most of the dining room duties: It will decide whether there is a table to seat the incoming customers, take their orders according to restaurant policies and current stock, and bring them their check once they are done eating. Using this system, your dad can instead spend most of his budget on an amazing chef and fresh ingredients. You are hoping the customers would love this concept and make the restaurant very popular!

You sit down with your dad to develop a beta version of the system. Having just taken CSCI561 last semester, you decide to implement it using ***first-order logic resolution***. Current restaurant status, policies, ingredient stock and customer status will all be encoded as first order logic sentences in the knowledge base. The knowledge given to you contains sentences with the following defined operators:

```
NOT X            ~X
X OR Y           X | Y
X AND Y          X & Y
X IMPLIES Y      X => Y
```

The program takes a query and provides a logical conclusion to whether it is true or not.

Format for input.txt:

```
<QUERY>
<K = NUMBER OF GIVEN SENTENCES IN THE KNOWLEDGE BASE>
<SENTENCE 1>
…
<SENTENCE K>
```

The first line contains a query as one logic sentence (further detailed below). The line after contains an integer *K* specifying the number of sentences given for the knowledge base. The remaining *K* lines contain the sentences for the knowledge base, one sentence per line.

Query format: The query will be a single literal of the form `Predicate(Constant_Arguments)` or `~Predicate(Constant_Arguments)` and will not contain any variables. Each predicate will have between 1 and 25 constant arguments. Two or more arguments will be separated by commas.

KB input format: Each sentence to be inserted into the knowledge base is written in FOL using operators &, |, =>, and ~, with the following conventions:

1. `&` denotes the *conjunction* operator.
2. `|` denotes the *disjunction* operator.
3. `=>` denotes the *implication* operator.
4. `~` denotes the *negation* operator.
5. No other operators besides `&`, `|`, `=>`, and `~` are used in the input to the knowledge base.
6. There will be NO parentheses in the input to the KB except to mark predicate arguments. For example: `Pred(x,y)` is allowed, but `A & (B | C)` is not.
7. Variables are denoted by a single lowercase letter.
8. All predicates (such as `Order(x,y)` which means person x orders food item y) and constants (such as `Broccoli`) are case sensitive alphanumeric strings that begin with an uppercase letter.
9. Thus, when parsing words in the input to the KB, use the following conventions:
    - 9.1. Single lowercase letter: variable. E.g.: `x, y, z`
    - 9.2. First letter is uppercase and opening parenthesis follows the current word: predicate. E.g.: `Order(x,y), Pred52(z)`
    - 9.3. Otherwise: constant. E.g.: `Harry, Pizza123`
10. Each predicate takes at least one argument (so, all predicate names are always followed by an opening parenthesis). Predicates will take at most 25 arguments. A given predicate name will not appear with different number of arguments.
11. Predicate arguments will only be variables or constants (no nested predicates).
12. There will be at most 100 sentences in the knowledge base.
13. See the sample input below for spacing patterns.
14. You can assume that the input format is exactly as it is described.
15. There will be no syntax errors in the given input.
16. The KB will be true (i.e., will not contain contradictions).
17. **Note that the format we just specified is broader than both Horn form and CNF. Thus, you should first convert the given input sentences into CNF and then insert the converted sentences into your CNF KB for resolution.**

Format for output.txt:

Your program should determine whether the query can be inferred from the knowledge base or not, and write a single line to output.txt:

`<ANSWER>`

Each answer should be either TRUE if you can prove that the corresponding query sentence is true given the knowledge base, or FALSE if you cannot. This is a so-called "closed-world assumption" (things that cannot be proven from the KB are considered false).

Notes and hints:

- Please name your program "**homework.xxx**" where 'xxx' is the extension for the programming language you choose. ("**py**" for python3, "**cpp**" for C++11, and "**java**" for Java).
- If you decide that the given statement can be inferred from the knowledge base, every variable in each sentence used in the proving process should be unified with a Constant (i.e., unify variables to constants before you trigger a step of resolution).
- All variables are assumed to be universally quantified. There is no existential quantifier in this homework. There is no need for Skolem functions or Skolem constants.
- Operator priorities apply (e.g., negation has higher priority than conjunction).
- The knowledge base is consistent.
- If you run into a loop and there is no alternative path you can try, report FALSE. For example, if you have two rules ***(1)*** `~A(x) | B(x)` and ***(2)*** `~B(x) | A(x)` and wanting to prove A(Teddy). In this case your program should report FALSE.
- Note that the input to the KB is not in Horn form. So you indeed must use resolution and cannot use generalized Modus Ponens.

Example 1:

For this input.txt:

```
Order(Jenny,Pizza)
7
Order(x,y) => Seated(x) & Stocked(y)
Ate(x) => GetCheck(x)
GetCheck(x) & Paid(x) => Leave(x)
Seated(x) => Open(Restaurant) & Open(Kitchen)
Stocked(Hamburger)
Open(Restaurant)
Open(Kitchen)
```

your output.txt should be:

```
FALSE
```

Note that, equivalently, the following input.txt could be given, where the => symbols have been replaced using the definition of implication (P=>Q is the same as ~P|Q):

```
Order(Jenny,Pizza)
9
~Order(x,y) | Seated(x)
~Order(x,y) | Stocked(y)
~Ate(x) | GetCheck(x)
~GetCheck(x) | ~Paid(x) | Leave(x)
~Seated(x) | Open(Restaurant)
~Seated(x) | Open(Kitchen)
Stocked(Hamburger)
Open(Restaurant)
Open(Kitchen)
```

and your output.txt should again be:

```
FALSE
```

Hint: you will need some pre-processing, like we have done here to convert from the first version of this example to the second version (we eliminated the implications), to ensure that your resulting KB is in CNF and can be used for resolution.

Example 2:

For this input.txt:

```
Leave(Helena)
11
Seated(x) & Stocked(y) => Order(x,y)
Order(x,y) => Ate(x)
GetCheck(x) & HaveMoney(x) => Paid(x)
Ate(x) => GetCheck(x)
GetCheck(x) & Paid(x) => Leave(x)
Open(Restaurant) & Open(Kitchen) => Seated(x)
Stocked(Portabello) | Stocked(Tofu) => Stocked(VeganHamburger)
Stocked(Portabello)
Open(Restaurant)
Open(Kitchen)
HaveMoney(Helena)
```

your output.txt should be:

```
TRUE
```

Example 3:

For this input.txt:

```
Order(Tim,Italian)
15
Seated(x) & Stocked(y) => Order(x,y)
Order(x,y) => Ate(x)
GetCheck(x) & HaveMoney(x) => Paid(x)
Ate(x) => GetCheck(x)
GetCheck(x) & Paid(x) => Leave(x)
Open(Restaurant) & Open(Kitchen) => Seated(x)
Stocked(Pasta) | Stocked(Pizza) => Stocked(Italian)
Stocked(Flour) & Stocked(Cheese) => Stocked(Pizza)
Stocked(Penne) & Stocked(Pesto) => Stocked(Pasta)
Open(Restaurant)
HaveMoney(Tim)
HaveMoney(Lauren)
Stocked(Penne)
Stocked(Flour)
Stocked(Cheese)
```

your output.txt should be:

```
FALSE
```

Example 4:

For this input.txt:

```
Hangout(Leia,Teddy)
45
Likes(x,y) & Likes(y,x) | Meet(x,y,z) => Hangout(x,y)
Leave(x,z) & Leave(y,z) => Meet(x,y,z)
GetCheck(x,z) & Paid(x,z) => Leave(x,z)
GetCheck(x,z) & HaveMoney(x) => Paid(x,z)
Ate(x,y) => GetCheck(x,z)
Order(x,y) & Good(y) => Ate(x,y)
Seated(x,z) & Stocked(y,z) => Order(x,y)
OpenRestaurant(z) & Open(Kitchen,z) & HasTable(z) => Seated(x,z)
TableOpen(x,z) | TableOpen(y,z) => HasTable(z)
HasIngredients(y,z) & Open(Kitchen,z) => Stocked(y,z)
~Bad(x) => Good(x)
Has(Dough,z) & Has(Cheese,z) => HasIngredients(CheesePizza,z)
Has(Pasta,z) & Has(Pesto,z) => HasIngredients(PestoPasta,z)
Has(Falafel,z) & Has(Hummus,z) => HasIngredients(FalafelPlate,z)
Has(Rice,z) & Has(Lamb,z) => HasIngredients(LambPlate,z)
```

```
Has(LadyFingers,z) & Has(Mascarpone,z) => HasIngredients(Tiramisu,z)
Old(Cheese) | Burnt(CheesePizza) => Bad(CheesePizza)
Moldy(Pesto) => Bad(PestoPasta)
Bad(Lamb) | Soggy(Rice) => Bad(LambPlate)
Has(Dough,Bestia)
Has(Cheese,Bestia)
Has(Cheese,Dune)
Has(Pasta,Bestia)
Has(Pesto,Bestia)
Has(Falafel,Dune)
Has(Hummus,Dune)
Has(Rice,Dune)
Has(Lamb,Dune)
Has(LadyFingers,Bestia)
Has(Mascarpone,Bestia)
Burnt(CheesePizza)
Soggy(Rice)
~Bad(Tiramisu)
Bad(Lamb)
OpenRestaurant(Bestia)
Open(Kitchen,Bestia)
OpenRestaurant(Dune)
Open(Kitchen,Dune)
HaveMoney(Leia)
HaveMoney(Teddy)
Likes(Leia,Teddy)
Likes(Leia,Mary)
Likes(Teddy,Harry)
Likes(Harry,Teddy)
TableOpen(Patio,Bestia)
```

your output.txt should be:

```
TRUE
```