CSCI-561 - Spring 2023 - Foundations of Artificial Intelligence
Homework 3

Due April 18, 2023 23:59:59

Image from negativespace.co

Guidelines

This is a programming assignment. You will be provided with sample inputs and outputs (see below). Please understand that the goal of the samples is to check that you can correctly parse the problem definition and generate a correctly formatted output. The samples are very simple and you should not assume that if your program works on the samples it will work on all test cases. There will be more complex test cases and it is your task to make sure that your program will work correctly on any valid input. You are encouraged to try your own test cases to check how your program would behave in some complex special case that you might think of. Since **each homework is checked via an automated A.I. script**, your output should match the example format ***exactly***. Failure to do so will most certainly cost some points. The output format is simple and examples are provided. You should upload and test your code on vocareum.com, and you will submit it there. You may use any of the programming languages and versions thereof provided by vocareum.com.

Grading

Your code will be tested as follows: Your program should take no command-line arguments. It should read a text file called “input.txt” in the current directory that contains a problem definition. It should write a file “output.txt” with your solution. Format for files input.txt and output.txt is specified below. End-of-line convention is Unix (since vocareum is a Unix system).

The grading A.I. script will, 50 times:

- Create an input.txt file, delete any old output.txt file.
- Run your code.
- Compare output.txt created by your program with the correct one.
- If your outputs for all 50 test cases are correct, you get 100 points.
- If one or more test case fails, you **lose 2 points for each failed test case**. (Note that one test case involves only one query in this HW).

Note that if your code does not compile, or somehow fails to load and parse input.txt, or writes an incorrectly formatted output.txt, or no output.txt at all, or OuTpUt.TxT, **you will get zero points**. Please test your program with the provided sample files to avoid this. You can submit code as many times as you wish on vocareum, and the last submitted version will be used for grading.

Academic Honesty and Integrity

All homework material is checked vigorously for dishonesty using several methods. All detected violations of academic honesty are forwarded to the Office of Student Judicial Affairs. To be safe you are urged to err on the side of caution. Do not copy work from another student or off the web. Sanctions for dishonesty are reflected in *your permanent record* and can negatively impact your future success. As a general guide:

Do not copy code or written material from another student. Even single lines of code should not be copied.
Do not collaborate on this assignment. The assignment is to be solved individually.
Do not copy code off the web. This is easier to detect than you may think.
Do not share any custom test cases you may create to check your program's behavior in more complex scenarios than the simplistic ones considered below.
Do not copy code from past students. We keep copies of past work to check for this. Even though this problem differs from those of previous years, do not try to copy from homework submissions of previous years.
Do not ask on piazza how to implement some function for this homework, or how to calculate something needed for this homework.
Do not post code on piazza asking whether or not it is correct. This is a violation of academic integrity because it biases other students who may read your post.
Do not post test cases on piazza asking for what the correct solution should be.

Do ask the professor or TAs if you are unsure about whether certain actions constitute dishonesty. It is better to be safe than sorry.

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)
Open(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)
Open(Bestia)
Open(Kitchen,Bestia)
Open(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
```