CS 1571: Homework 3 (programming)

FOL Inference: Forward Chaining (100 pts)

Assigned: October 25, 2017

Due: November 13, 2017

I. Implement basic (not necessarily efficient) FOL Forward-Chaining in Python, Java, or C#.

For this homework project, we will use the term *rule* to refer to an implication which has a left-hand side (LHS - a conjunction of non-negated atoms) and a right-hand side (RHS - one non-negated atom). Also, we will use *fact* to refer to a single atom. We will say *applying or firing a rule* to refer to a legal application of Generalized Modus Ponens to that rule, thus deriving a new fact.

For the purposes of this assignment, there are several simplifying assumptions to make your job easier:

- Facts must be fully instantiated atoms -- no variables. Also, they cannot be negated.
- All variables appearing in a rule's RHS must also appear on the LHS. This ensures that when a rule fires, it results in a rule that has no variables. (Your code can assume this without checking for it.)
- No use of functions; only predicates, variables and constants.

In general, there is a set of rules, and a separate set of facts. Each time a new fact comes in, the Forward-Chain algorithm is invoked to set off all possible rule firing, like the "domino effect", adding new facts to the knowledge base. In addition to implementing Forward-Chain, you must write code to invoke it for each new incoming fact.

Unification algorithm. Due to the above simplifying assumptions, the unification algorithm can be simpler than that given in the text. Here is the unification algorithm you should implement:

Unify(f1, f2) takes two atoms as input, the second of which has no variables (a fact). If they can be matched, it returns a list of variable bindings. Otherwise, it returns "fail".

```
1. Initialize substitutions (a local variable) to the empty list.
2. If f1 and f2 have different predicate names or different numbers of arguments, Then return "fail".
3. For each argument, a1, in f1, with corresponding argument a2 in f2:
       If a1 is a variable
       Then If a1 is bound to something other than a2 in substitutions
           Then return "fail"
           Else add the binding a1/a2 to substitutions
       Else If (constant) a1 doesn't equal (constant) a2
           Then return "fail"
4. return substitutions
```

Make sure that when you invoke Unify you give the two arguments in the correct order (as shown above, the second argument must have no variables).

Text-based I/O. Your system should be able to handle (piped as standard input) input such as that shown below in Part II. As in the text, variables are lower-case, and constants are capitalized. You need not check for errors in the input -- you can assume the input is syntactically correct. You may expect the input to first list all the rules, and then all the facts. You can assume no spaces within each predicate-argument structure, but spaces everywhere else.

II. Test Your System.

Example 1: You must show your system working on the following example similar to the book, typed here for your convenience so you can cut-and-paste:

```
American(x)  ^  Weapon(y)  ^  Nation(z)  ^  Hostile(z)  ^  Sells(x,z,y)  ->  Criminal(x)
Owns(Nono,x)  ^  Missile(x)  ->  Sells(West,Nono,x)
Missile(x)  ->  Weapon(x)
Enemy(x,America)  ->  Hostile(x)
American(West)
Nation(Nono)
Enemy(Nono,America)
Owns(Nono,M1)
Missile(M1)
Nation(America)
PROVE   Criminal(West)
```

From this, it should derive that West is a criminal.

Note that test cases will include a final line at the end stating the required thing to PROVE.

Example 2: You must also show your system working on the following example:

```
Instrument(y)  ^  Musician(x)  ->  Plays(x,y)
Instrument(y)  ^  Plays(x,y)  ->  NotToneDeaf(x)
Musician(Grace)
Instrument(I1)
PROVE   NotToneDeaf(Grace)
```

From this, it should derive that Grace is not tonedeaf.

Other Examples: We will also test your system on new examples which will be announced on November 7. Note that some examples may involve cases where the proof fails. You should modify Example 1 or 2 to make sure you can handle this case.

III. Incremental Forward Chaining

Modify your code to do incremental forward chaining and retest your system. Your trace should give enough detail for you to include a writeup explaining how it shows that incremental forward-chaining is more efficient.

IV. What to Submit and Grading Guidelines

See Ahmed's Instructions.

What to submit

Implement your **forward chaining** program in one file (FOL_FC.java, FOL_FC.cs, FOL_FC.py). Implement the **incremental forward chaining in another file** (FOL_IFC.java, FOL_IFC.cs, FOL_IFC.py)

Submit a .zip package with the name **"YourLastName_YourFirstName_HW3.zip"**, the package contains the following:

- (FOL_FC.jar, FOL_FC.exe or FOL_FC.py) and (FOL_ IFC.jar, FOL_ IFC.exe or FOL_ IFC.py) files for your implementation, these files should be runnable from command line. The files take only one argument which is the test case file. In case of java or C#, submit also the code (FOL_FC.java or FOL_FC.cs) and (FOL_IFC.java or FOL_IFC.cs) alongside the executable files. These executable files should print your output on the screen.

- Submit transcript files (two files for each test case one for running the FC and the other for the IFC) that contains your inferred arguments and if you found the requested argument or not. The transcript files should be named (TestCaseName_out_FC.txt) and (TestCaseName_out_IFC.txt)

- A Readme.txt file, in which mention:
    - The version of Python you used (if python used), framework used in case of C# or Java.
    - Any additional libraries needed to run your code.
    - Any additional resources, references, or web pages you've consulted.
    - List any person with whom you've discussed the assignment and describe the nature of your discussions.

- A Report.pdf or Report.doc file, in which you summarize your implementation decision and then explain how your implementation of incremental forward-chaining is more efficient.

Grading Criteria

5 (100%): The program works as expected for both implementations; has a clear README for the grader; includes a complete set of transcripts; complete report.

4 (93%): The program works as expected for both implementations, but possibly with a minor flaw; has a clear README for the grader; includes a complete set of transcripts; complete report.

3 (80%): One of the two implementations only works, OR, the program works for both implementations but one of these three requirements is not complete or missing (Clear README, Complete set of transcripts, complete report).

2 (60%): One of the two implementations only works, and one of these three requirements is not complete or missing (Clear README, Complete set of transcripts, complete report), OR, the program works for both implementations but two of these three requirements are not complete or missing (Clear README, Complete set of transcripts, complete report).

1 (40%): None of the implementations is working but a serious attempt has been made on the assignment, or the three requirements is not complete or missing (Clear README, Complete set of transcripts, complete report).