CS4710: Artificial Intelligence
Homework 1: Theorem Prover

Introduction:

For this assignment, you will be building a rule-based logic system similar to CLIPS (which we saw in class). You will only be implementing a limited amount of functionality.

System Commands:

Your system will be a command line-based program that supports the following commands:

Teach <ARG> <VAR > = <STRING> Example: Teach -R S = "Sam likes Ice Cream"

This command teaches the system a new variable of the form variable = string. The string will always be surrounded in double quotations. There will always be one space on each side of the equals sign. The string may contain spaces. This command simply adds a variable definition, but does not set that variable to be a true fact. Variables have a default value of false (see next command). A variable name cannot be used twice. Variable names can be any case-sensitive string containing the characters A-Z, a-z, and underscores. No other characters will appear in a variable.

The <ARG> is an argument (must be either –R or –L). The –R stands for a 'Root' variable, while –L stands for a learned variable. A root variable is a direct observation that can only be set to true directly by the user (using the teach command below this one). A learned variable cannot be set true directly, but must be inferred via inference rules.

Teach <ROOT VAR> = <BOOL> Example: Teach S = true

Teaches the system that a defined root variable has been observed to be true (or false). The first teach command above simply defines the variable. This command tells the system that the variable is true or false. 'true' and 'false' will always be all lowercase with no quotations. There will be one space on each side of the equals sign as shown above. This command can ONLY be used with root variables (see above command). The system should return an error if the user attempts to set a learned variable directly.

**Note that if I change a root variable's value, it may alter the truth value(s) of learned variables (see Learn command below). You may assume that this command also resets all learned variables to the value of false (this simplifies the assignment for you a bit).*

Teach <EXP> -> <VAR> **Example:** **Teach S -> V**

Teaches the system a new rule. The antecedent may be an entire expression, but the expression will only contain AND, OR, and NOT operators. The expression may contain parentheses. All variable defined here must have already been defined via the previous 'Teach' command, otherwise you can ignore this command. There will be one space on each side of the '->' symbol. However, the expression defined as <EXP> will contain no spaces. This <EXP> will also not be fully parenthesized (it might be) and will contain only ! (not), & (and), and | (or) symbols. If no parentheses, use the following order of operations: Not, And, Or. The <EXP> may contain a mix of root and learned variables, however the consequence <VAR> must be a learned variable.

List **Example:** **List**

Lists out all of the fact and rules currently known by the system. Use the format shown at the bottom of this document. List the word 'Variables:' on one line by itself, then on each successive line (and one tab over), list every variable with its string as it was given using the 'Teach' command (e.g., with spaces around the equals sing, quotations around the string, etc. The system should maintain the order in which variables, rules, and facts were learned.

Learn **Example:** **Learn**

Uses forward chaining to apply all of the rules of the system to the facts of the system to create newly formed facts. Continues until no new knowledge can be acquired from the given rules. If I type this command, and then type 'List', I should see new learned rules listed out. This command does not accept any parameters.

Query <EXP> **Example:** **Query (S&V)**

Returns true if an only if the given expression is true given the rules and facts within the system. Some queries may be simple enough to look up in the system's "list of facts". Others will need to apply backward-chaining. The list of facts should not be altered by this process (i.e., the 'List' command should produce the same output before and after any query).

Why <EXP> **Example:** **Why (S&V)**

This command's input format is identical to the Query command above. However, the 'Why' command explains the reasoning behind the true or false claim to the user. This command should output, on the first line, either 'true' or 'false'. Then, the system should

explain, using one inference rule per line, the reasoning that leads to this value. The system should explain the full (forward) reasoning regardless of the state of the knowledge base.

Example Output:

>Why (S&V)

true

I KNOW THAT Sam Likes Ice Cream //Use this syntax if fact is true but no rule leads to it

BECAUSE Sam Likes Ice Cream I KNOW THAT Sam Eats Ice Cream //Use this when rule applied

I THUS KNOW THAT (Sam Likes Ice Cream AND Sam Eats Ice Cream) //When concluding AND, OR, NOT Statements

If the expression given is false, then the logic should break down somewhere. Output the correct logic until you hit the step that breaks that logic and afterwards output the correct (but false) reasoning that you would want to apply to prove your statement.

Example Output:

>Why (S&V)

false

I KNOW IT IS NOT TRUE THAT Sam Likes Ice Cream

BECAUSE IT IS NOT TRUE THAT Sam Likes Ice Cream I CANNOT PROVE Sam Eats Ice Cream //when concluding a rule cannot be proven

THUS I CANNOT PROVE (Sam Likes Ice Cream AND Sam Eats Ice Cream) //when concluding an expression or sub-expression is false.

*Notice that the parentheses in the output here. You should output parentheses in your text versions of expressions if and only if the original expression has parentheses in those locations.

More Output Details for the Why Command:

This section lists other implementation and special case situations that might arise as you work on why. Make sure your implementation follows these rules exactly. If you think anything is underspecified, let us know so we can address it.

Conjunctions

When given a conjunction (e.g., A&B) that you are trying to prove. You should print out the reasoning for A first, then B, then have a line concluding A&B as below. NOTE that A and B might be expressions themselves requiring multiple lines of reasoning:

I KNOW THAT A

I KNOW THAT B

I THUS KNOW THAT A AND B

If you CANNOT prove A&B, then you ONLY need to explain the one line of reasoning that causes this to fail giving the left expression precedence. In other words, if A fails, explain it only. For example:

I KNOW IT IS NOT TRUE THAT A

THUS I CANNOT PROVE A AND B

Disjunctions

Similar rules exist for disjunctions (A|B). If you CAN prove the statement, you only need to print one line of reasoning, giving precedence to the left expression. If A and B are both true, only explain A. For example:

I KNOW THAT A

I THUS KNOW THAT A|B //no need to prove B one way or the other.

If A|B cannot be proven, then your system should fully explore both possible lines of reasoning, showing the they both fail. A simple example:

I KNOW IT IS NOT TRUE THAT A //explain the left expression FIRST

I KNOW IT IS NOT TRUE THAT B

THUS I CANNOT PROVE A OR B

The NOT Operator

The NOT operator can lead to some strange output. If handling something of the form !<EXP>, explain the reasoning of the expression first, then conclude how the NOT operator is applied. This can lead to output that appears to be redundant. Consider !A where A is currently false:

I KNOW IT IS NOT TRUE THAT A //just stating the A is not a true variable

I THUS KNOW THAT NOT A //seems redundant, but we are saying thus !A is true

Similar output can be seen when the expression ends up being true:

I KNOW THAT A //looked up the variable and it is true

THUS I CANNOT PROVE NOT A //concluding that !A is indeed false

Multiple Rules for Same Variable

If multiple rules can be applied to learn the same learned variable, then you should attempt to apply them in the same order they were initially taught to the system. Sometimes, multiple rules can be applied to

successfully prove a single learned variable (e.g., A->B and C->B where A and C are both true). In this case, you should ONLY explain one line of reasoning to prove the learned variable (B in this case) and give precedence to rules that were taught to the system first. In the example here, the A->B rule would be explained only to prove B.

If many rules exist for the same learned variable but none of them can successfully be used to prove it, you should attempt to explain each one and show that every possible rule breaks down and doesn't apply. This can produce a lot of output (and that is ok). Each of these failed explanations will conclude with a "BECAUSE IT IS NOT TRUE THAT X I CANNOT PROVE Y statement.

If No Rules Exist for a Learned Variable

While applying your logical rules, you may come to a situation where you want to try to prove a learned variable (let's say, A) and there doesn't exist any rules in the system that can prove A (i.e., there are no rules with A on the right side of the conditional). In this case, you can simply state that that the variable is false. For example, if A is learned and the production rules are X -> B and Y -> C, then:

```
Why A
I KNOW IT IS NOT TRUE THAT A          //no rule exists for it, so just state it is false.
```

Note About Chaining Conjunctions

Quick note about chaining conjunctions (or creating any other larger logical expressions). Your system will be pretty verbose. For example, if you try to explain why A&B&C&D, you would get the following output:

```
I KNOW THAT A          //left half of A & <everything else>
I KNOW THAT B          //left half of B & <everything right of it>
I KNOW THAT C          //left half of C&D
I KNOW THAT D          //right half of C&D
I THUS KNOW THAT C AND D              //concluding C&D
I THUS KNOW THAT B AND C AND D        //concluding B with C&D
I THUS KNOW THAT A AND B AND C AND D  //concluding A with B&C&D
```

This seems verbose (and it is) but it is the behavior we are looking for. I know many of you are capable of making the output more concise here, but the autograder won't like it.

Getting Started:

Here are some recommendations for getting started.

1. Choose your data structures. I recommend having one for storing the variable definitions, another for storing the facts in the system (maybe a list of variables that are true), and a third for the list of rules that can be applied. Don't try to shove all of this information into a single data structure, although this is up to you.

2. Write code that parses the commands correctly, but doesn't do anything with them. For example, write skeleton code that accepts the string "Teach S&P -> V" and correctly invokes your 'createNewRule(condition, consequence)' method. For now, leave those methods empty (or just print out the parameters to confirm they are being invoked correctly).

3. Implement the List command first. It should be the easiest and is how you will confirm your other methods are working correctly.

4. Implement the Teach methods one at a time. Use 'List' to confirm that they work.

5. Implement the Learn command. Use 'List' to confirm that new information is being added to the system's facts.

6. Implement the Query command. Test and make sure the true / false output is correct on several different test cases.

Input / Output:

Input should be accepted via standard input. We will pipe in our test input to your program when running your code. Input files will consist of individual commands to the system, one per line. Input will end with a single line containing a '0'. All commands will be well-formed.

Only the 'List' and 'Query' commands should produce output. The 'Query' command should produce either true or false (all lowercase) on its own line.

The 'List' command should produce output of the following form exactly. There should be no empty lines in between output lines (though it appears that way on this document):

Root Variables:

S = "Sam likes Ice Cream"

V = "Today is Sunday"

Learned Variables:

EAT = “Same will eat ice cream”

Facts:

S

V

Rules:

S^V -> EAT

Programming Language:

You may use Java, C++, or Python 3. All input and output should be directed through standard input/output.