CSCE 420
Programing Assignment #4
due: Tues, Dec 4, 2018, 11:59pm (submit on eCampus)

Objective

The objective of this project is to implement a Knowledge-Based System using **Back-chaining**, and then use it to solve some reasoning problems. The input will be conjunctive rules in First-Order Logic (in a file format to be defined). Thus, to handle variables in the inference procedure, you will have to implement **Unification**. You may use C++, Java, or Python. Here is an example:

```
> cat animals.kb
# facts
((mammal dog))
((mammal cat))
((bird penguin))
((car ford))
# rules
((mammal ?m) (animal ?m)) # forall m mammal(m) -> animal(m)
((bird ?m) (animal ?m))

> inferencer animals.kb "((animal ?x))"
((animal dog))
((animal cat))
((animal penguin))
```

I passed in the query on the command line, and it printed out all 3 solutions it found (with variables from the resulting unifiers substituted into the query).

File Format for Knowledge Bases

The input files contain facts and conjunctive rules represented as Expressions. Expressions consist of either symbols or a list of sub-expressions enclosed in parentheses. Predicates are represented as list where the first element in the list is the predicate name, and the rest of the elements are the arguments. For example the predicate *capital(texas,austin)* would be expressed as `(capital texas austin)`. Variables are symbols prefixed with a '?'. Rules are a list of predicates, consisting of the list of antecedents (implicitly connected by "^") followed by the consequent. Hence the consequent is the LAST literal in a rule. Every line in the file consists of a list of literals (since facts are like rules with 0 antecedents). Hence even facts have to be enclosed in double parentheses. Blank lines are allowed, lines beginning with '#' are treated as comments and ignored.

```
line ::= blank | comment | Fact | Rule
Fact ::= ( Literal )
Rule ::= ( Literal+ Literal ) // antecedents followed by the consequent
Literal ::= ( predicateName arg* )
arg ::= symbol | ?symbol // variables are symbols prefixed with a '?'
```

A more abstract way to think about the syntax is that each line contains an Expression, which is just a bunch of symbols nested in parentheses:

```
line ::= blank | comment | Expr
Expr ::= Sym | List
List ::= ( Expr* )
```

You can write your own parser if you choose. However, to make things easier, some C++ code is being provided that defines an Expr class and gives a parser you can call for reading input files and converting them to a list of Expr instances.
(download: `http://orca2.tamu.edu/tom/cs420-fall18/Expr.hpp` and `Expr.cpp`)

Unification

An important aspect of doing inference in FOL is unification. You should implement the pseudo-code in the textbook (Figure 9.1) for unifying a pair of Expressions (and returning a substitution list; or NULL if they don't unify). You will probably want to define a 'Unifier' class, which contains a mapping from variables to Expressions (i.e. a substitution list). In addition, you will probably want to implement a 'subst(Expr,Unifier)' function for applying a unifier to an Expression.

For the Unification algorithm, I recommend the following simplifications. First, you can skip the "OccursCheck", which won't come up in our examples. Second, since our Expression representation already treats predicates as lists, you can skip the line in the algorithm referring to "compound" expressions. Here is a test example:

```
void test_unifier() {
  Expr* a=parse(tokenize(string("(author huckleberry_finn ?b)")));
  Expr* b=parse(tokenize(string("(author ?a mark_twain)")));
  Unifier* u=unify(a,b);
  u->print();
  cout << subst(a,u) << EOL; }
```

prints...

```
?b=mark_twain
?a=huckleberry_finn
(author huckleberry_finn mark_twain)
```

Back-chaining

Once you have implemented Expressions and Unification as described above, it should be relatively straightforward to implement Back-chaining as an inference method. You should implement the algorithm as shown in Figure 9.6 in the textbook. Thus, given a knowledge base (i.e. a set of facts and rules read-in from the input file) and a query, you can set up a goal-stack and try to reduce it to empty (using facts or rules), showing the query is entailed. Don't forget that if some branches of the search fail, then you will have to back-track to the most recent choice-point and try unifying the top subgoal with other rules or facts. If the query has variables in it, then your code should return the unifier it has built-up so you can see the variable bindings that satisfy it. Furthermore, you should keep track of all the unifiers that lead to successful proofs (empty stack), so you can print out all solutions the the query (with variables substituted, as in the animals example above).

Note that Back-chaining is fundamentally recursive. So don't forget to make copies of data structures like Expressions, Unifiers, and the goal stack before you modify them, so you avoid changing the instances of these variables in the caller.

There is a line in the Back-chaining algorithm the refers to a "standardize()" function, which is applied to rules each time you use them. Recall that standardizing an expression means replacing all the original variable names with completely unique new names, so that there will be no clash/mixing/reuse of variables between rules (or if a rule is used multiple times). For example, if "?x" is used in several rules, these are intended to be distinct instances. Code for a *standardize(Expr)* function that renames the variables is also provided for your convenience.

Problems to Solve

Once you have implemented your inference engine, you can use it to do some reasoning with conjunctive FOL knowledge bases (written the file format defined above). Here are several problems to try.

1. Write a knowledge base the Colonel West example in the text book and use your backchainer to prove that West is a criminal.

2. Write a knowledge base for the Wumpus World and use it to show that room (2,2) can be inferred to be safe, after the agent has visited rooms (1,1), (1,2), and (2,1).

3. Your inference engine can be used to emulate Datalog, which is like a database, augmented with Prolog-like rules for making queries. Here is some information on people occupations and where they live. Encode these as facts in a KB (using *binary predicates* like 'occupation' and 'lives_in'), and add general rules to define 'surgeons', 'lawyers', and residents of Texas. Then do a query to *find all surgeons living in Texas*. (hint: this will probably require a conjunctive query with multiple predicates and shared variables, effectively like a 'join' among tuples.)

name	*occupation*	*lives_in*
joe	hotel_manager	pittsburgh
sam	brain_surgeon	houston
bill	trial_lawyer	los_angeles
cindy	oral_surgeon	omaha
joan	civil_lawyer	chicago
len	graphic_artist	austin
lance	heart_surgeon	dallas
frank	patent_lawyer	dallas
charlie	plastic_surgeon	houston
lisa	investment_banker	albuquerque

```
> inferencer people.kb "((surgeon ?a)(resident_of_TX ?a))"
((surgeon sam) (resident_of_TX sam))
((surgeon lance) (resident_of_TX lance))
((surgeon charlie) (resident_of_TX charlie))
```

4. Write a knowledge base for playing tic-tac-toe. A board can be described using the 'occ' predicate, with symbols 'x', 'o', such as this: ((occ x 2 2)). Blank positions can be indicated this way: ((blank 1 3)). Given the state of a particular board, you should be able to query your knowledge base to determine the optimal move for each player. For example, if x has two in a row, and it is x's turn to move, then the best move is go for the win. In other circumstances, x might be forced to place a piece to block a potential win for o, if o has two in a row. If there are no intelligent moves, then a simple default move is to place a piece in any blank position. You might want to encode your own strategy, especially for making decisions on sparser boards. Note that if there are multiple solutions for "((move x ?p ?q))", they should be printed out in order of priority, such that the preferred move is listed first (which can be achieved by adjusting the order of your rules). Here is an example:

```
x  .  .

.  .  x

o  .  o
```

Facts: ((occ x 1 1)) ((blank 1 2)) ((blank 1 3)) ((blank 2 1)) ((blank 2 2)) ((occ x 2 3)) ((occ o 3 1)) ((blank 3 2)) ((occ o 3 3))

query: ((move o ?p ?q)) should produce ((move o 3 2)) because it is a winning move (possibly followed by several other less-optimal candidate moves).

```
o  .  o

.  .  .

x  .  .
```

In this case, the query ((move x ?a ?b)) would produce ((move x 1 2)) as a blocking move.

Finally here is a case where nobody can immediately win, but o can make a move to set up two in a row. Querying ((move o ?a ?b) should return ((move ?o 3 3)) as the top solution, which would force x to make a blocking move in the next turn, and allow o to win the game in the next move after that.

```
o  x  o

.  .  .

x  .  .  ← best move for o is here
```

Here is a hint in writing your rules. I found it helpful to be able to use antecedents like $x \neq y$. You can simulate this in a limited way for the numbers 1-3 by simply including the following facts: ((neq 1 2)) ((neq 2 1)) ((neq 1 3)) ((neq 3 1)) ((neq 2 3)) ((neq 3 2)). You can do something similar for equality or less-than, if you want.

You should write a short Word document describing the strategy implemented by your rules, and give some examples (of tic-tac-toe boards) where it makes good decisions.

What to Turn in

- You will submit your code for testing using **eCampus** (https://ecampus.tamu.edu/)
- You should include a Word document with **instructions on how to compile and run** your program.
- Include your **knowledge bases**.
- Include a **transcript** showing your solution traces**.**
- **Word document** describing your **strategy** for tic-tac-toe implemented by your rules, along with some **example board states and optimal moves** that it infers.