CSE 463/563, Spring 2005

Programming Project 1
AUTOMATED THEOREM PROVING

In this project, you will write programs that could have passed the CS department's old graduate-level AI Qualifying Exam questions on logic :-)

As described in the syllabus, your final report should be a conference-style paper containing your **annotated** code and **commented** sample runs either as appendices or incorporated into the body of the report.

1. **(WARNING: THIS PART IS RELATIVELY EASY.)**

   (a) In lecture, you will be given an algorithm for **converting a sentence of first-order logic into "clause form"** (this algorithm will be on the Web at:

   http://www.cse.buffalo.edu/∼rapaport/563S05/clause-form.pdf
   and
   http://www.cse.buffalo.edu/∼rapaport/563S05/clause-form.html).

   Using your favorite programming language, write a fully-commented algorithm (e.g., a Lisp function) that takes a well-formed formula of first-order logic as input and that returns an equivalent sentence in clause form.

   Suggestion 1: You should choose an input notation for wffs that is convenient for the programming language that you are using. E.g., for Lisp,

   ```
   (^ P Q)
   ```

   might be more convenient than:

   $(P \wedge Q)$

   You do not need to write a parser (i.e., a compiler) that converts our text's FOL notation for wffs into yours. All you need to do is write a program that converts your notation into clause form.

   Suggestion 2: When you rename variables so that variables bound by different quantifiers have unique names, you can use rewrite rules of the following form:

   $$(Q_1 v_1 F(v_1^*) \# Q_2 v_1 G(v_1^*)) \rightarrow (Q_1 v_1 F(v_1^*) \# Q_2 v_2 G(v_2^*))$$

   where the Q_i are quantifiers (either the same or different), # is either $\vee$ or $\wedge$, the v_i are variables such that 'v_1' $\neq$ 'v_2', and '$F(v_1^*)$' represents a sentence containing 0 or more occurrences of 'v_1'. An example would be:

   $$(\forall x P(a,x) \wedge \exists x R(a)) \rightarrow (\forall x P(a,x) \wedge \exists y R(a))$$

   (b) Apply your algorithm to the following sentence:

   $$\forall x[Animal(x) \supset (Predator(x) \equiv \exists y[Animal(y) \wedge Eats(x,y)])]$$

2. **(WARNING: THIS PART IS RELATIVELY HARD.)**

(a) Preferably using the same programming language as for the previous part of this project (since the two parts will eventually need to communicate), implement a fully-commented **unification algorithm** (either the one in Brachman & Levesque, the version of that algorithm (to be) given in lecture, or one that you find documented elsewhere; please be sure to give a full citation to whichever version you choose).

More precisely, your algorithm should take a pair of sentences as input and either return a most general unifier (MGU) for them, if they are unifiable, or else return a message such as "NOT UNIFIABLE". (Again, you may assume that the notation $f(x,g(x))$ can be understood as: `(f x (g x))`, if you prefer using Lispish notation.)

(b) Use your algorithm to answer the following question: For each of the following pairs of terms, if they unify, show an MGU; if they don't, say so, and state why. (Note: You only need to state *in your report* why a unification failed. Although a "trace" of your unification algorithm might be useful, your *program* does not have to indicate exactly where unification failed). Assume that *u*, *v*, *x*, *y*, and *z* are variables, and that *a*, *b*, and *c* are individual constants:

i. $P(a,x,c)$ and $P(y,b,z)$

ii. $P(a,x,c)$ and $P(y,b,y)$

iii. $P(x,x,c)$ and $P(u,v,u)$

iv. $P(x, f(x), f(y))$ and $P(f(a), f(z), z)$

v. $P(x, f(x), f(a))$ and $P(f(z), f(z), z)$

3. (a) **(WARNING: THE COMPUTATIONAL IMPLEMENTATION OF THIS PART IS RELATIVELY HARD.)**
Write a **resolution + unification + refutation theorem prover** for *first-order predicate* logic.

To do this, you will need to incorporate the two previous parts of the project into a fully-commented theorem prover that uses a resolution algorithm with a refutation strategy. You may either write such an algorithm from scratch (recommended, but difficult) or adapt one that you find elsewhere. If you choose the latter option, please keep the following in mind:

- You must give a full citation for the algorithm that you use.
- You must fully comment the algorithm so that the reader of your report can understand it (this is especially important if the reader is not fluent in the programming language that the algorithm is written in) and so that *we* know that *you* understand it!
- Although this option has (as the philosopher Bertrand Russell once commented) all the advantages of theft over honest toil, it is not necessarily going to be easier than writing your own, since you will have to fully understand the one you will be using and you will probably have to either adapt it to work with the previous components of this project or else adapt those previous components to work with *it*.

There are several strategies for selecting two target clauses to resolve. Instead of implementing one or more of these strategies, your program can ask the user to input two clauses for resolution (i.e., this aspect of resolution need not be automated). This interaction can take place with each pass through the algorithm.

Since this part of the project relies on a working implementation of the clause-form converter and unification algorithm (from parts 1 and 2, you should test those implementations thoroughly before you implement this part. If you find that you are getting stuck on one of the first two parts (and *only if* you are getting stuck), you should "hand code" the clause form inputs or unified results for this part. In general, you should "hand code" anything in your automated theorem prover that you are unable to automate.

(b) **DO PART 3b (AT LEAST BY HAND) WHETHER OR NOT YOU ARE ABLE TO DO PART 3a.**

This part is inspired by an example in Moore 1982: 429; in your report, explain the relationship between Moore's example and this problem: Using resolution, show that the following set of clauses is inconsistent. Assume that *a*, *b*, and *c* are individual constants, and that *x* and *y* are variables:

i. $[Next\text{-}to(a,b)]$

ii. $[Next\text{-}to(b,c)]$

iii. $[Green(a)]$

iv. $[\neg Green(c)]$

v. $[\neg Green(x), Green(y), \neg Next\text{-}to(x,y)]$

NOTE: Please do *all* exercises at least *by hand* (in addition to, or instead of, implementing them in a programming language) as part of your *report.* I will hand out a tentative grading scheme to make it easier for you to organize your final report.

********** NEW **********

DUE AT START OF LECTURE, *FRIDAY, APRIL 15*.

********** NEW **********