CSC 8520 Introduction to Artificial Intelligence
Spring 2002
Programming Assignment #3
DUE DATE: Sunday, April 7, 2002

For this assignment you will complete partial code to build a resolution theorem prover for predicate logic. In particular, you will be given code in the file called:

http://www.csc.vill.edu/~klassner/csc8520/handouts/resolution.lisp,

and complete the definitions of the functions prove-by-refutation and resolve-clauses. Their stubs are defined at the end of the file. Resolve-clauses will need to use the unify function to do resolution on two clauses, which have already been converted into CNF form for you. Prove-by-refutation repeatedly apply the resolution function to a knowledgebase until a contradiction is found. You will implement prove-by-refutation twice. First, in whatever straightforward way you can develop, and second, in a way that incorporates some heuristic you developed to speed up the proof process.

I will first describe the 2-clause resolution code, and then describe the theorem-proving code.

A. RESOLUTION OF TWO CLAUSES. The UNIFY functions permit us to unify (or attempt to unify) two clauses, which is a basic step in the resolution algorithm. Using UNIFY, we will implement the RESOLVE function, which is prototyped below:

```
(DEFUN resolve (a b)
  "This function takes as input two disjunctive clauses in the minimal-clause
  form ("minimal-CNF") defined for input to UNIFY. If A and B are clauses that
  can be resolved, then this function will return the result of resolving A with B.
  If A and B cannot be resolved, this function returns T. If the result of the
  resolution is a contradiction, NIL is returned."
)
```

From our in-class discussions, we know that two predicate logic disjuncts X and Y can be resolved if and only if

1. one of the clauses contains a predicate that is a negated form of a predicate in the other clause [i.e. X <-- '( (P $x $y) (Q $z $y) )
   and Y <-- '( (M $r $s) (not (Q $d $e)) (H Bob) ) ].

We will call this pair the positive-negative pair for clarity.

In the resolution inference rule:

```
(A OR (NOT B)), (B OR G)
--------------------------------------------------------
A OR G
```
the B and (not B) would be the positive-negative pair. (Note: All the A’s, B’s and G’s have arguments, but for clarity I dropped them in the rule).
and 2. these two predicates (obviously ignoring the negation) can be UNIFYed, and the resulting binding list can be used to substitute into the new resolution result (formed by ORing together all the predicates in X and Y that weren’t in the positive-negative pair).

In our example, the result of resolution is

`'( (P $x $e)  (M $r $s)  (H Bob) ).`

**B. RESOLUTION THEOREM PROVING.** Once we have a RESOLVE-CLAUSES function, we can build a *very* simple resolution theorem prover that is first given a knowledgebase containing CNF clauses, to which a single statement (the negation of which is what we're trying to prove by refutation) is added. The theorem prover then exhaustively uses RESOLVE to resolve pairs of disjuncts in the knowledgebase, adding the results of each successful resolution to the knowledgebase until we get a contradiction (RESOLVE-CLAUSES returns NIL).

You will therefore also be required to hand in a function (with any helper functions you need to define) called PROVE-BY-REFUTATION:

```
(defun PROVE-BY-REFUTATION (kb statement)
  "This function takes a knowledgebase data structure in KB, which has been initialized with a consistent set of facts. It also takes the argument STATEMENT, which we assume is the negation of what we are trying to prove. The function then adds STATEMENT temporarily to the KB, and runs the simple resolution procedure on the facts in the augmented KB, continuing until either no resolutions are left (return NIL), or a contradiction is found (return T)."
)
```

**C. RESOLUTION PRUNING STRATEGIES.** Clearly, for even small knowledgebases the exhaustive resolution approach is rather time-consuming. Accordingly, once you have written your first version of PROVE-BY-REFUTATION, you will write a second version that uses one of the resolution strategies described on pages 285 -- 286 of the text, or one of your own. You will have to describe the heuristic pruning strategy in a 2- or 3- paragraph writeup, explaining what the heuristic is, and how (approximately, no long detailed proofs are necessary) it cuts down on the number of extraneous resolutions attempted.
IMPLEMENTATION DETAILS: Be sure to note the following implementation details:

I. Knowledgebases are implemented as a record by the “defstruct” command you see listed before the definitions for ASK, TELL, and RETRACT, and POSSIBLE-RESOLVERS. Basically, you will use TELL to have the system convert a logic statement into CNF form, then insert the statement into the knowledgebase. The KB is implemented as a pair of hash tables. Each clause in a disjunct is classified as either negative (has a NOT) or positive (does not have a NOT), and is inserted into the appropriate hash table. Then, inside prove-by-refutation, when you call POSSIBLE-RESOLVERS with a predicate (“literal”) and a KB data structure, you can “quickly” retrieve all clauses that might resolve with it. Be sure to read all comments in the function definitions carefully.

II. Through the LOGIC function, you can take a string with INFIX notation like “P(x,y)=>Q(z,y)” and convert it into the prefix version we use in LISP:

```
(=> (P $x $y) (Q $z $y)).
```

You can use the string-infix form or the list-prefix form, as you prefer, for input to ASK, and TELL. You should assume that you can only use list-prefix form everywhere else as input. NOTE: The string-to-list conversion process assumes that any “word” that starts with a lowercase letter is a VARIABLE! So, make sure that you always capitalize predicate names and “constants” like “BOB.”

The infix-to-prefix conversions and their meanings appear below:

<table>
<thead>
<tr>
<th>INFIX</th>
<th>PREFIX</th>
<th>MEANING</th>
</tr>
</thead>
<tbody>
<tr>
<td>~(P)</td>
<td>(not P)</td>
<td>negation</td>
</tr>
<tr>
<td>P ^ Q</td>
<td>(and P Q)</td>
<td>conjunction</td>
</tr>
<tr>
<td>P</td>
<td>Q</td>
<td>(or P Q)</td>
</tr>
<tr>
<td>P =&gt; Q</td>
<td>(=&gt; P Q)</td>
<td>implication</td>
</tr>
<tr>
<td>P &lt;=&gt; Q</td>
<td>(&lt;= P Q)</td>
<td>logical equivalence</td>
</tr>
<tr>
<td>P(x,y)</td>
<td>(P $x $y)</td>
<td>predicate with 2 args</td>
</tr>
<tr>
<td>forall(x,y,P(x,y))</td>
<td>(forall ($x $y) (P $x $y))</td>
<td>universal quantification</td>
</tr>
<tr>
<td>exists(x,y,P(x,y))</td>
<td>(exists ($x $y) (P $x $y))</td>
<td>existential quantification</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>the TRUE logical constant</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
<td>the FALSE logical constant</td>
</tr>
</tbody>
</table>

III. It is PROVE-BY-REFUTATION’s job to print out a trace of the resolutions it is trying (and the result), as it performs the resolution theorem-proving algorithm. You should use the FORMAT function to create “pretty” traces on standard output.

Your trace should appear in your emacs buffer as you run the function ASK (which calls PROVE-BY-REFUTATION). To hand in your trace, simply save that buffer when the function finishes executing.

WHAT YOU NEED TO HAND IN:

1) a hard copy of your code, which should show everything you added to the RESOLUTION.LISP file (in a separate file).
2) the trace of your system when run on the “Did Curiosity kill the cat?” example in the textbook on pages 282 -- 283.
3) the trace of your system when run on the knowledgebase of your choice.
4) a short description of how you designed the “first pass” of the theorem prover, followed by your writeup of what heuristic(s) you added for the “second pass” to get better run times in 2 & 3. (this should take no more than 2 pages).