Assignment 4: Resolution Theorem Prover

UMaine COS 470/570

Fall, 2021

Assigned: 11/1/2021 Due: 11/22/2021

Fair warning: Historically, students have considered this one hard, folks, so start early. (Did I mention, start early?)

Theorem proving is a very useful AI technique that is widely used in other kinds of programs, such as planners. An simple, but powerful, theorem proving technique is *resolution theorem proving* (RTP). It operates on a pre-compiled, homogeneous knowledge base of axioms and has the same power as any other deductive theorem proving method.

In class, you have seen how the various pieces of a resolution theorem prover are designed and work. In this assignment, you'll write one.

You have to have your program prove theorems in two domains: one chosen from among the domains given in the section "Axioms" below and a second domain of your own devising.

You do *not* need to have your program convert your axioms to clause form—you can enter them that way. However, if you have a conversion mechanism in your program, I'll give you some extra points. (Don't spend a great deal of time on it, though, until you get RTP working.)

What you will need are the following:

1. a pattern matcher;
2. two different knowledge bases of axioms; and
3. the resolution control structure.

Pattern matcher (unify)

The pattern matcher you will use has been described in class—the Unify function. In fact, since I'm feeling magnanimous, I'll even *give* you a Unify implementation. (It will be posted on BrightSpace.)

You should consider the code provided as a starting point or guide rather than as something you can just use "out of the box"—you'll almost certainly need to change it for your purposes. For example, you may have to change it and its supporting functions to conform to your representation of predicate calculus. (Some suggestions for how to represent predicate calculus were given in lecture and some are in the comments in Lisp code.) You may also have to modify the Unify function to handle OR and NOT, if you represent them explicitly or have multiple negation in an axiom.

Note that there are variable-handling functions in the code you may find useful, even if you don't use the rest of the code. For example,

```
(variable? '?foo)
```

returns `t`,

```
(varname '?foo)
```

will return `FOO`, and

```
(make-var 'bar)
```

will return `?BAR`. There are also functions to help generate new symbols based on old symbols, as we had for the second assignment. For example,

```
(newSymbol '?foo))
```

or

```
(newSymbol '?foo3)
```

will both return a unique variable whose name is based on `FOO`; if `?foo3` is the last variable created in the "foo" series, then either of these calls will return `?FOO4`.

The file also contains a function, `instantiate`, that will replace all variables in an expression (e.g., a literal or clause) with their bindings. See the file for more information.

Knowledge bases

The knowledge bases for your program are sets of *clauses*. Each clause is a list-structured version of a predicate calculus clause. You do *not* have to write code to translate from First-Order Predicate Calculus to Conjunctive Normal Form—you can start with CNF.

One way to represent CNF clauses in Lisp is to just use a prefix form of the logical connective OR; example, the predicate calculus clause:

$$\neg Roman(x_2) \vee loyalto(x_2, Caesar) \vee hate(x_2, Caesar)$$

which could come from the axiom:

$$\forall\ x_2\ Roman(x_2) \wedge \neg loyalto(x_2, Caesar) \Longrightarrow hates(x_2, Caesar)$$

could be represented in your program as:

```
            (or (not (Roman ?x2))
                (loyalto ?x2 Caesar)
                (hate ?x2 Caesar))
```

This representation implies several things. First of all, you will need to make your program smart enough to recognize OR (unless you figure out the trick to get rid of or, that is) and NOT—it needs to handle such things as "`(not (Roman Marcus))` is the negation of `(Roman Marcus)`". It also needs to know how to access each term of an ORed clause.

Finally, it has to have a way of knowing what things are variables. I suggest using the question mark prefix discussed above. An alternative is to use *property lists* of symbols to determine if they are variables. For example, you could use any symbol as a variable by defining:

```
(defun make-var (symbol)
  (setf (getf symbol 'var) t)
  symbol)

(defun is-var (symbol)
  (getf symbol 'var))
```

To make a variable, you could then do:

```
(make-var 'this-is-a-variable-now)
```

Resolution

To resolve two clauses, your program will need to recognize the term in each that allows resolution. In other words, you need to find two terms that unify except for their "signs"—not negated (positive) or negated (negative)—which must be opposite one another. For example, if the two clauses are:

1. `(or (not (bird ?x) (flies ?x)`
2. `(or (not (flies Tweety)) (color Tweety yellow))`

the two terms that unify are `(flies ?x)` and `(not (flies Tweety))`; resolving 1 & 2 using those terms gives us:

```
(or (not (bird Tweety)) (color Tweety yellow))
```

This implies that you'll need a function to create the negation of a term:

- `(bird ?x)` ⇒ `(not (bird ?x))`, or
- `(not (bird ?x))` ⇒ `(bird ?x)`

Note that here may be many such corresponding pairs of terms, which represents a choice point in your search tree.

The Resolution control structure

This is the portion of the program that actually does resolution, as well as decides on which clauses to use. General algorithms for this were given in class and in your text.

Note that your program must have the ability to know when it reaches a dead end. If there was more than one way to do resolution at some point (multiple clauses could be resolved, multiple literals within a clause could resolve with another clause, etc.), then if one doesn't work, it should try another until no more are possible; then it should backtrack. If this reminds you of depth-first search, it should.[1] Your program must also be able to backtrack to try different variable bindings, when more than one is found. You should think carefully about which control strategy(ies) you want to use to select clauses (e.g., set of support, etc.) and stick to it (them).

[1] You could also implement this without backtracking by using a breadth-first search. DON'T DO THIS! Recall that the space complexity of BFS is exponential—your programs will run out of memory rather quickly.

Your program **must** handle variables—you should show their use in the script you turn in.

Don't forget to deal with the fact that resolution theorem proving is *semi-decidable*! (What does that mean for how you code it?)

Output of the program

Your program's output should be sufficiently detailed to show off your program. If some capability it has isn't shown, I will assume that it can't do it. It should also be clear to the reader what is being output by the program. This means that you should have the program nicely format its output and clearly label things that you want the reader to understand. If we can't understand your output, you won't get credit for it.

You should turn in several examples of your program proving things, and one example of it not being able to prove something.

You should also output the knowledge base used prior to each set of sample runs of the program.

Write-Up

Your program must be **well**-documented, especially with respect to the axioms you provide the program with. For each clause, describe what it means—**both in predicate calculus and in English**.

It is important that you provide good instructions in your write-up on how to run your program, since I *may* run them. If your program does not run, tell me why—the quality of your analysis will determine how much partial credit you get. In any case, tell which control strategy or strategies you selected and explain why you chose it/them.

Be sure to point out any interesting or neat features of your programs, and discuss any problems you had, ways you overcame them, etc.

Your write-up **must be in PDF**.

Turn in:

You will turn in a zipfile or gzip'd tarfile containing all of your code and your write-up PDF file. You do *not* need to turn in a copy of `unify.lisp` unless you modified it—I already have a copy! ☺

Good luck! This is really *not* that hard, or at least it shouldn't be. However, **start early**!!! In the past, students have found this to be more time-consuming than they expected. Don't wait till the very last minute, or I can guarantee you won't get it done on time.

Some Common Lisp Mistakes

By this time of the semester, I usually notice some problems in one or more students' Lisp coding, so let me mention them here, so you can avoid them in this assignment. They mostly fall under the heading of "Let Lisp Help You, and Don't Fight It". If you fight it, it will usually win.

Unintentional global variables.

All variables inside a function or method that do not reference a true global variable (one you've defined at the top level with `defvar` or `defparameter`) need to have local scope. With very few exceptions, this means that they need to be one of the parameters or defined in a `let` form.[2] The `let` form is not just for variable assignment: its primary purpose is to open a new *scope* in which its variables reside.

[2] Some exceptions: variables appearing in a closure within whose lexical scope the function is defined, variables defined by the `loop` macro, variables in the parameter list of a `lambda`, and variables defined within a `multiple-value-bind` statement.

It's not just good programming practice, by the way, nor is it just that I'll count off (though I will) if you have global variables in your code where they don't belong. If the function is recursive, then local variables (i.e., those scoped locally by the parameter list, `let`, etc.) are stored on the stack and fresh copies are created with each invocation. Global variables are not created afresh with each invocation, but shared between invocations that are active. This is almost certainly what you do not want, and it will bite you.[3]

[3] Worse, global variables are technically "special" variables, which in Lisp means that they are *dynamically* scoped. Dynamically-scoped variables do an excellent job of emulating Satan. For example, if you define "foo" using `defvar` or by accident, then later use "foo" as a variable in a `let` in function A, inside of which you call another function B, which calls another function C, which tries to use the global variable "foo", it will instead use A's version. Confusing, to say the least.

It's a symbol processing language—ergo, use symbols!

I've in the past noticed students doing things like either this:

```
        (defvar forward "forward")
```

then later:

```
(if (string-equal dir forward)...)
```

or even, heaven forbid,

```
        (defvar forward 1)
```

then later:

```
(if (= dir forward)...)
```

If you are doing this, *stop it*. Lisp has symbols for a reason.

What you are doing is using a string or an integer to stand for (in this case) “go forward”. In other words, the string or number is a *symbol* for “go forward”. Why not just use the symbol `forward` or `:forward`? A lot easier and clearer for the reader and for you. I.e., no “defvars”, just:

```
(if (eql dir :forward)...)
```

or

```
(if (eql dir 'forward)...)
```

Oh, and this:

```
        (defvar forward 'forward)
```

is only marginally better. You're going to use it somewhere like:

```
       ...(if (eql direction forward) ...)
```

Although this is clear-ish, it is again using a variable to contain a symbol that has the same name as the variable... why? Just, why?

Booleans, not numbers.

Lisp understands Booleans very well indeed: `nil` is false, and non-nil values (including `t`) are true. Please don't use C's $0 =$ false and all else is true convention. To Lisp, both 0 and 1 (or anything else non-nil) are true. This makes extra work for you, will bite you when you least expect it, and makes your code look like you can't tell the difference between C and Lisp.

Oh, and you don't need usually to explicitly check to see if something is `t`, e.g.:

```
    (if (not (null thing))...)
```

or worse,

```
    (if (eql thing t) ...)
```

which might not even work (since things other than `t` are considered true). You can just do this instead:

```
    (if thing ...)
```

to mean "if thing is not nil" (i.e., it is true) and

```
    (if (not thing) ...)
```

to mean "if thing is false".

Ugly ifs

I'll be the first to admit that Lisp's `if` function is not the prettiest thing in the world, requiring as it does `progn`, etc., if you have more than one expression in the "then" or "else" clause. For that particular problem, I'd suggest using `cond`:

```
;;; This:
(if (eql direction :forward)
  (progn
    ...
    )
  (progn
    ...
    ))
;; looks better as:
(cond
 ((eql direction :forward)
  ...)
 (t
  ...))
```

Also, remember that there are other conditionals that are cleaner when you have only a single optional path for control or more than two paths:

```
;;; This
(if (not (eql direction :forward))
  (progn
    (setq direction :backward)
    (print "Retreat!")))
;; would be cleaner as:
(when (not (eql direction :forward))
  (setq direction :backward)
  (print "Retreat!")))
;; or even better:
(unless (eql direction :forward)
  (setq direction :backward)
  (print "Retreat!")))

;;; And:
(if (eql direction :forward)
  (print "forward")
  (if (eql direciton :backward)
    (print "backward")
    (if (eql direction :left)...)))
;;; would be better as one of these:
(case direction
  (:forward (print "forward"))
  (:backward (print "backward"))
  (:left...)
  ...
  (otherwise (print "unknown command")))
;; or
(print (case direction
 (:forward "forward")
 (:backward "backward")
 ...
 (otherwise "unknown")))
;; or, if you were going to do this a lot or needed to change stuff often:
(defvar mapping '((:forward "forward")
  (:backward "backward")
  ...))
...
(print (or (cadr (assoc direction mapping)) "unknown"))
;; Of course, for this particularly trivial use, you could have just done:
(princ direction)
;; if you didn't care about printing "unknown" for an unknown value.
```

Line length.

Please keep your lines of code to a reasonable length, preferably $\leqslant 80$ characters. This will keep me from getting annoyed as I grade them and the lines wrap around or disappear because I'm not using the editor or settings you happened to use.

In Emacs, you can set the "fill column" variable with:

```
C-u 80 C-x f
```

then tell Emacs to use that intelligently to wrap your lines (text &

code):

```
M-x auto-fill-mode
```

Comments and indentation.

Take another look at the Lisp packet do what it says for comments and indentation. Your IDE will help you. (If it doesn't, then use a different IDE, such as Emacs.) For example, Emacs understands how to indent code and it understands common commenting conventions.

Axiom sets

Use one of these axiom sets for the first part of the assignment, and then come up with one of your own for the second part. Remember: You can convert to CNF *before* you give the axioms to the program if you like.

Marcus axiom

1. Human(Marcus)
2. Pompeian(Marcus)
3. Born(Marcus,40)
4. Ruler(Caesar)
5. TryAssassinate(Marcus, Caesar)
6. Erupted(Vesuvius,79)
7. now = 2019
8. $\forall$ x Pompeian(x) $\Rightarrow$ Roman(x)
9. $\forall$ x $\forall$ y Roman(x) $\wedge$ Ruler(y) $\Rightarrow$ LoyalTo(x,y) $\vee$ Hate(x,y)
10. $\forall$ x $\forall$ y Human(x) $\wedge$ Ruler(y) $\wedge$ TryAssassinate(x,y) $\Rightarrow$ $\neg$LoyalTo(x,y)
11. $\forall$ x Human(x) $\Rightarrow$ Mortal(x)
12. $\forall$ x Pompeian(x) $\Rightarrow$ Died(x,79)
13. $\forall$ x, $\forall$ t_1 $\forall$ t_2 Mortal(x) $\wedge$ Born(x,t_1) $\wedge$ gt(t_2 - t_1, 150) $\Rightarrow$ Dead(x,t_2)
14. $\forall$ x $\forall$ t Alive(x,t) $\Leftrightarrow$ $\neg$Dead(x,t)
15. $\forall$ x, $\forall$ t_1 $\forall$ t_2 Died(x,t_1) $\wedge$ gt(t_2, t_1) $\Rightarrow$ Dead(x,t_2)

Garden world axiom set

1. John likes carrots.
2. Mary likes carrots.
3. John grows vegetables that he likes.
4. Carrots are vegetables.
5. When you like a vegetable, you grow it.
6. To eat something, you have to own it.

7. When you grow something, you own it.
8. In order to grow something, you must own a garden.

HAVE FUN
