Project1: Prolog Expert System Shell
19 October, 2015

In this project, you will design a generic, natural language (NL) based expert system for answering questions based on a database of expert knowledge. This system may never replace *human* experts in, e.g., determining what kind of little, black bird is eating your vegetable garden or telling you what inscrutable and expensive problem is plaguing your car. Nonetheless, we do hope that you'll see how Prolog's built-in grammar support and the good fit between logic programming and many reasoning tasks make it relatively easy to create powerful NL reasoning systems. You will also encounter some of Prolog's limitations as a programming language.
This project statement is long, but please read the whole thing before diving in.

Authors: This project is developed by Steve Wolfman who taught this course a decade ago, who based it on a code developed by David Lowe, who taught the course some time before Steve did.

Learning Goals

Besides the high-level goals described above (call them "Prolog appreciation"), the following are also learning goals of the project:

- Write Prolog grammars using the built-in DCG notation.
- Use and modify large Prolog packages written by someone else (and get a sense for what makes a package good and bad).
- Appreciate the need for and exercise the mechanisms for efficiency in Prolog (e.g., goal/rule ordering and cuts).
- Encounter several common idioms of Prolog programming (like the "repeat" looping predicate).

Project Resources

You will need the following resources during this project. Some of the files are included in the zip file you downloaded from Piazza. The rest are available for download or use at the provided links:

- The starter files: 312-pess.pl, 312-pess-grammar.pl, and bird.kb (included).
- Amzi's online text on expert systems in Prolog (particularly, chapters 1 and 2 and section 3.6).

- WordNet and the online interface (for exploration). Wordnet database is available in Prolog and can be downloaded from the same website. For ease of access this package is included in your project package.
- ProNTo set of tools: At the bottom of the page, under "Accessing WordNet from Prolog", a package can be downloaded which contains a technical article describing the use of wordnet from prolog and some example codes. You may find some of these documents helpful in trying to make sense of using wordnet.
- On the same page, look for "ProNTo Morph" and download the package. You will use this database for stemming words in your program. An accompanying article explains the usage.

(Side Note: Don't feel overwhelmed! remember that this is your toolset and there are only very specific parts of it that you will need to understand, master, and use in order to accomplish this project. Try to keep a broad perspective in your exploration so as not to get lost in details. The first part of your project is dedicated to this exploration, so don't feel rushed to take everything in all at once.)

Collaboration

The project must be completed in teams of 3. Teams of 2 or 4 may also be accepted if you can't form a group of 3. While the workload will be adjusted accordingly, you should keep in mind that the load may still be a bit too high for a team of 2 and collaboration can be difficult in a team of 4.
Teams may divide work as they like bearing in mind that: 1) the final may include questions referring to the project and 2) all team members need to submit peer evaluations and some written questions individually, 3) the instructor/TAs reserve the right to have a one-on-one questioning session with any of the individual members necessary as part of the grading procedure.
Collaboration or consultation outside your team is not allowed. (with an exception that's explained under 'The First Week Tasks')

The Deliverables

Overview of the project deliverables (extensively explained below):

- *As a team*: your team description (names, student numbers, ugrad ids, and optional team name), should be included with each one of your submissions. One interim checkpoint submission. One final submission with one report.
- *As an individual*: your *private* team member ratings and your individual answers to a list of questions.

0. The First task (team make-ups):
ASAP OR BY THE MIDNIGHT OF Friday, October 23rd. Have one team member submit the information about your team make-up: names, student numbers, and ugrad ids.
Use handin for this submission: course name is **cs312** and submission name is **project1teams**
Anyone who does not have a team on Monday morning (Oct 26th), will receive an email from the instructor or TAs that randomly assigns them to a team. You can use Piazza to find teammates as well.

1. The Interim Checkpoint Submission (See "First Week Tasks"):
DUE ON THE MIDNIGHT OF Friday, October 30th. *(otherwise you'll turn into pumpkins!)*
For this checkpoint, the team should make a single submission responding to both the coding and written questions in the "First Week Tasks" section. To receive credit for this part, your submission must be clear, easy to read, and include all pertinent information (team members' names, student numbers, and ugrad ids). The codes must be in a single executable file (not doc or pdf), and each predicate must be accompanied by comments that explain its intended meaning and functionality. As before, code with no documentation cannot be accepted.

The Final *Team* Submission (See "Main Project"):
DUE ON THE MIDNIGHT of Tuesday, November 17th.
The team should make a single submission that responds to all the requirements in the "Main Project". Once again all the team information should be included. In this part, you're extending the codes you are given as the starting point of your project. In your submission, include all relevant project files that are needed for running your project, even those that you haven't modified. Your additions to the original code should be accompanied by comments that explain the intended meanings of the predicates, their functionalities, as well as the *instantiation expectation of their arguments* (e.g. "this predicate will work only when the first argument is ground" Or "it will work inefficiently if the second argument is a variable", i.e. non-ground).
Your report is a separate doc or pdf file which should include a section for each of the questions under the "Main Project". In each section, give a brief overview (one or two paragraphs) of how you implemented the requirements of that section. This should include which part in the original code you modified, and an overview of the implementation, including how the various predicates you've written interact with each other to achieve the results. You can use examples to clarify your explanation. You can document your program's behaviour by explaining a few common use cases or scenarios, or explain it in

abstract terms. Anyway, explain it however you understand it and explain it as clear as you can and as brief as possible. Keep it under 10 pages.
If your code has any shortcomings, such as, known scenarios that don't work, known inefficiencies, etc. include them as well as why you think that's happening or what you've done to fix it or any other relevant information that can prove your understanding of the issue.

The Final *Individual* Submission (See "Peer Evaluation" and "Written Questions):
DUE AT THE SAME TIME AS THE FINAL TEAM SUBMISSION
For this part your submission has two parts (can be in one file or two files). One part should include your answers to the "Written Questions", which is made up of some reflection questions you will answer after finishing your project and a second file where you give a rating, on a scale of 0 to 5, to *yours* and to *each of your team members'* participation in the project. The individual submission is absolutely necessary for you to receive your individual grade. Missing individual submission will be interpreted as lack of participation in the project and will cause you to receive a zero for your first project.

LATE SUBMISSIONS ARE SUBJECT TO UP TO 25% GRADE REDUCTION. If both your team submission and your individual submission is late, your grade will only be reduced once for your team submission. If your team submission is on time but your individual submission is late, you will still face a penalty if you're *significantly* late. (Use your best judgement to determine what *significantly* means in this context. Your instructor reserves the right to do the same.)

Note on Bonus Points: If you wish to complete bonus work, (which you should) include a separate section on your final report for the bonus point(s) you've attempted. As with other sections, document the functionality and use of your code for the bonus questions, and detail any shortcomings in your solution. Bonus points will be assigned at the discretion of instructional staff based on functionality, clarity, and understanding demonstrated in the report.

Note on including the shortcomings of your program: You are about to embark on a complex programming task in which multiple approaches and solutions are possible and is likely that no solution you find will be completely perfect. Your solution is not expected to be perfect, but to be reasonably functional, efficient, and responsive. Including your observed shortcomings of your code will not necessarily reduce your grade. If shortcomings are with respect to basic operations of the code, the markers will catch it nonetheless. So you won't save any grades by not including them in the report. On the other hand, a reasonable explanation of a deficiency, its possible causes, as well as attempted

fixes (even if unsuccessful), proves an honest effort and a higher level of understanding and can elevate your mark.

Grading Scheme
TBD

The Basic System

Your whole team should do this part ASAP!
Start with the files 312-pess.pl and 312-pess-grammar.pl which together constitute a working but limited NL expert system. This system allows the user to make assertions about the world (enter facts into the database) and to pose questions about the state of the world.
1. Your first task should be to study and understand the three starter files at a high level, although you need not understand the details of all the files.
You can use load_rules/1 to read facts from a NL knowledge base and solve/0 to solve the top goal for that database. Currently, the top goal is to figure out what the object/creature under identification is. For the sample bird knowledge base, this means identifying the bird. Here is an example of use:

```
1 ?- load_rules('bird.kb').
[Much debugging output flies by, describing how 312PESS understood]
[the rules in the knowledge base.]
rules loaded
true.

2 ?- solve.
Would you say that it has external tubular nostrils ?> why.
    it has external tubular nostrils
    it has order that is tubenose
    it has family that is albatross
    it is laysan albatross
    top_goal(laysan albatross)
> |: no.
Would you say that it has webbed feet ?> |: yes.
Would you say that it has flat bill ?> |: yes.
```

```
Would you say that it has long neck ?> |: no.
Would you say that it is plump ?> |: no.
Would you say that it feeds on the water's surface ?> |: yes.
Would you say that it agilely flies ?> |: yes.
Would you say that it quacks ?> |: yes.
Would you say that it has green head ?> |: yes.
The answer is mallard
true.

3 ?- solve.
Would you say that it has external tubular nostrils ?> no.
Would you say that it has webbed feet ?> |: no.
Would you say that it eats meat ?> |: no.
Would you say that it has one long backward toe ?> |: no.
No answer found.
true.
```

(The yes., no., and why. responses after '> |:' were typed by the user.)

2. Now, try adding some rules to bird.kb or parsing sentences or parts of sentences to see how the grammar produces the parsed rules. You can try parsing individual rules using the try_parse predicate:

```
?- try_parse.
it is an insect.
Parsed structure is: [rule(attr(is_a, insect, []), [])]

Understood: it is insect
true .
```

("it is an insect" on the second line was written by the user)
That parses the input sentence "it is an insect" into the parse structure shown and "glosses" (i.e., transforms into English-like text for output) the parse structure into the text understood. In this case, there is one rule, which has a head but an empty body (i.e., a fact). The head is attr(is_a, insect, []), indicating that object/creature under consideration "is a" insect. The third argument indicates that we know no more about this "is a" attribute. (For comparison, try parsing "it is a large insect" or "its talons are sharp".)
The comments in the pess and grammar-pess files can help you better understand each of these sections.

First Week Tasks

Check under "Deliverables" for the due date of this task.

In the first week, your task will be to become an expert on different parts of the problem. You should assign group members to learn about each of the subjects below. Note that WordNet and ProNTo_Morph will play a relatively small role in the final project. On the other hand, *everyone* in the class is expected to learn about many of the general Prolog use items (e.g, DCGs, good Prolog style, the cut, and difference lists).

1. WordNet's Prolog interface: what its relations mean, especially s and g. (links under "Project's Resources". Library files included with the project package)
2. ProNTo_Morph and how to use its top_level predicates in your program. (links under "Project's Resources")
3. Consult the AMZI's chapters on expert system to understand how the interactive shell in 312-pess.pl works. (links under "Project's Resources", in particular section 2.3)
4. Explore 312-pess.pl and 312-pess-grammar and/or SWI-Prolog manual for the uses of:
    a. Built-in input/output predicates used in the code (e.g., write, get_char, and atom_chars), and other built-in predicates, namely, fail, repeat, var, functor, arg. Notice the role some play in implementing the shell.
    b. DCGs (including {} and ;) and how the grammar rules are expressed.
    c. assert/retract, cuts and difference lists (List1\List2 is a difference list. Section 15.1 of your textbook discusses difference lists).

Note on Collaboration in this part of the project: When exploring the above listed aspects of the project, if your team gets to a point where you can't figure out how something works on your own, you *can* post your questions on Piazza. These questions however, will be strictly for your fellow students to answer. Students who actively respond to others' questions in this part gain 1 or 2 bonus points. The instructors will not get involved, unless there is a question about the assignment description. Do try to avoid *lazy* questions: how do I use wordnet? what is a difference lists? Show that you have made some effort on your own to understand something, so that others will be encouraged to help you. Answering questions as well should be prioritized on the basis of how much effort the person has made to resolve the issue on their own.

Note that posts that are directly about any of the questions below are strictly excluded from this arrangement. As before, the discussion board will be monitored by the instructors to ascertain a fair and honest interaction among students according to these terms. □

Now, to prove your familiarity with these, you will need to answer the following questions in your checkpoint submission:

1. Write a predicate `definition(Word, Meaning)` that is true if Meaning is a definition (i.e. gloss) for `Word`. For example:

```
?- definition('hello', G).
G = 'an expression of greeting; "every morning they exchanged
polite hellos"'.

?- definition('discipula', G).
false

?- definition('hypernym', G).
G = 'a word that is more generic than a given word'
```

2. Write a predicate word_line_morphs/0, a utility that reads a line of text from the keyboard, converts the text to an atom (just assume the text is a single word; no need to check), and finally writes out ProNTo_Morph's complete list of morphological parsings for that atom. (Be careful about checking for the end of line! You might want to use peek_char to help. Understanding the "read_sentence" predicate in 312-pess-grammar.pl should also help.) For example:

```
?- word_line_morphs.
says.
[[[says]], [[say, -s]]]

?- word_line_morphs.
triples.
[[[triples]], [[tripl, -pl]], [[triplis, -pl]], [[triple, -s]]]
```

We're not concerned that your output exactly match ours as long as you find all the stems (e.g., find triples, tripl, triplis, and triple).

It is up to you to decide which built-in predicates or starter file predicates you use. It's better to use what you already have in starter files, because ultimately the codes you're writing in this first part of your project are going to be merged with existing code in starter files toward the goals under the "Main Project". Your choice of builtin vs starter file predicate use will not affect your grade as long as your program returns correct information for a given input. If your program's behaviour or interface is different from what's shown in the above examples, or if it doesn't work with certain input formats, you should make that clear in your comments or documentation.

3. Add a clause to `process/1` (in 312-pess.pl) and appropriate rules to the grammar to handle vocabulary. With your new rules, knowledge base files will be able to define their own vocabulary. For example, if a knowledge base file contained the lines:

```
words: thing is a noun.

words: lift is verb.

words:
  late adjective and
  project noun and
  silly adverb and
  tired adjective and
  instructor is a noun.

words:
  last is an adjective
  word is a noun.
```

Your process rule would handle each of these vocabulary statements and assert the following facts:

```
n(thing).
v(lift).
adj(late).
n(project).
adv(silly).
adj(tired).
n(instructor).
adj(last).
```

```
n(word).
```

It's up to you to devise a grammar that will allow you to read vocabulary "sentences" of these types. Do note, however, that your sentence will already be a list of terms by the time your grammar needs to deal with it (read_sentence takes care of that), and process itself should probably take care of the "words:" part.

4. **[For four-person teams only - bonus question for smaller teams]** Create a simplify_attr/2 predicate. simplify_attr will make attr/3 structures easier to understand. simplify_attr(A, SimpleA) is true if SimpleA represents the same goal as A except with the following (recursive) transformations:

   attr(X, Y, Z) → attr(X(Y), Z)
   attr(X(Y), []) → X(Y)
   attr(X(Y), [Z]) → attr(X(Y), Z)
   rule(H, []) → fact(H)
   rule(H, [B]) → rule(H, B)

   For example,

   ```
   rule(attr(does, eats, [attr(is_how, slowly, []), attr(is_a,
   insects, [attr(is_like, large, [])])]), [])
   ```

   would become

   ```
   fact(attr(does(eats), [is_how(slowly), attr(is_a(insects),
   is_like(large))])).
   ```

   (It really is easier for humans to read, trust me!)
   You'll probably find the functor/3 and arg/3 predicates helpful for the first step above (use help(functor). and help(arg). in Prolog to find out more).
   Your predicate need only work if the first argument is ground; however, it would be good practice to make it work both ways, and for a bonus point you can try doing just that (perhaps using the var predicate and cuts to "cheat" and use different predicates depending on which argument is non-ground).

Note for two person teams: I recommend one of you work on the first two questions and the other one on question 3.
Note on submission: It's up to you whether you submit your solutions to the above tasks in a separate file you start from scratch (and import what you need) or add your code to existing starter files. Just keep in mind that in the next part of your project ("Main Project") you're going to add the codes you've written to the starter files and use them as part of the PESS program. If you do submit them as part of the starter files, make sure you submit sufficient documentation that makes it clear where your additions to those files are to be found.

Main Project

Check under "Deliverables" for the due date of this task.

This section lists a set of requirements your final submission must meet. You need not work on these tasks in the order listed. Many of the tasks do not directly depend on each other, and even where they interact to an extent, much progress can be made independently:

1. Create a new knowledge base describing some simple domain. (Note that, if you want the goal PESS solves to be something other than "what is it?", you will have to manually change the rule for top_goal in the clear_db predicate in 312-pess.pl.) Include the file in your submission. In your report, briefly explain your domain and some discussion about what was especially easy, difficult, or frustrating about writing the rules.
2. Write a predicate main/0 that starts an interpreter loop. The interpreter loop should accept commands like "load", "solve", "help", and "quit" from the user, allowing them to load new knowledge bases, solve the current goal, get help on available commands, and quit the interpreter. Amzi's discussion of interpreter shells may be helpful here. Note that, as you complete later parts, you will need to add new commands. So, you should ensure that adding new commands is easy!

   Here's an example session with the interpreter:

```
?- main.
This is the CPSC312 Prolog Expert System Shell.
Based on Amzi's "native Prolog shell".
Type help. load. solve. or quit.
at the prompt.  Notice the period after each command!
> load.
Enter file name in single quotes, followed by a period
(e.g 'bird.kb'.): 'bird.kb'.
Understood: if it has external tubular nostrils and it
lives at sea and it has hooked bill then it has order
that is tubenose
[And so forth..]
```

```
rules loaded
> solve.
Would you say that it has external tubular nostrils ?> no.
Would you say that it has webbed feet ?> yes.
[And so forth..]
Would you say that it quacks ?> yes.
Would you say that it has green head ?> yes.
The answer is mallard
> help.
Type help. load. solve. or quit.
at the prompt.  Notice the period after each command!
> quit.

Yes
```

Bonus Question: For 1 bonus point to each member, implement the command 'list' in the interpreter that prints out a list of all the loaded rules. The output will be similar to the debugging output when the rules are being loaded. If no rules are loaded, the command should not fail, but print a message saying no rules are loaded.

3. Allow the user to specify the goal (i.e., top_goal, currently specified in clear_db/0) in the knowledge base. So, rather than trying to figure out what "it" is when you type solve., your program will now try to solve whatever goal the user specifies. (However, if the user specifies no goal, you should still default to figuring out what "it" is.)

   This will require parsing questions as opposed to just sentences (as the grammar currently does). For example, users might be able to specify the top goal as "what is it" (which would translate to attr(is_a, X, [])), "what does it have" (which would translate to attr(has_a, X, [])), "is it a brown swan" (which would translate to attr(is_a, swan, [attr(is_like, brown, [])])), or "does it eat insects" (which would translate to attr(does, eat, [attr(is_a, insects, [])])).

   The user would specify these goals with a new type of knowledge base statement starting with "goal:" (just as vocabulary can now be added with the "words:" statement). So, for example, the file could contain: "goal: does it eat insects", which would set the top_goal to the one described above.

You may, if you wish, instead specify goals as statements, but you must allow the statements to be missing pieces. For example, you could add the word "what" as a noun that leaves the value of the "is_a" attribute unbound. "goal: it is a small what." would become attr(is_a, X, [attr(is_like, small, [])]). (You should support all the types of questions described above, but they can be in statement form instead.)

Try to use as much of the existing grammar rules as possible to solve this problem. It will make the problem *much* easier! To get you started, a good rule to put in for top_goal for the "does it eat insects" example might be: rule(top_goal(yes), [attr(does, eat, [attr(is_a, insects, [])])]).

Bonus question: For 1 bonus point to each team member, also allow the goal "what the heck is THAT", which should translate to the same thing as the default goal ("what is it").

4. **[For three or four person teams only - bonus question for pairs]** Allow the user to set the top-level goal from the interpreter loop. This will involve a new command goal that works as follows:

```
> goal.
Enter the new goal, followed by a period: does it contain gluten.
Understood goal: it has gluten
> solve.
[Many questions asked and answered..]
[Eventually reports]
The answer is yes
>
```

5. **[For three or four person teams only - bonus question for pairs]** Allow the user to assert new facts and rules at the interpreter prompt. As above, this will involve a new command like assert. (This should be very easy after completing the previous problem!)

6. Use WordNet and ProNTo_Morph to dramatically expand your program's vocabulary. You should use ProNTo_Morph to stem words and then find the stems in WordNet. WordNet's s/6 predicate will tell you what part of speech the word is (although note that a word may belong to multiple parts of speech!). You should add rules to the grammar that incorporate these new words. Ensure that backtracking across the different parts of speech in WordNet and the different possible stemmings in ProNTo_Morph works correctly!

Your code should now be able to handle sentences like "it is an amusing hyena" without explicitly adding those terms.

Notes: You should consider WordNet's "satellite" adjectives as normal adjectives. Also, you may want to check whether the *unstemmed* word is in WordNet before trying stemming to avoid, e.g., turning adverbs into adjectives. Finally, it's up to you what to do with plural forms (e.g., should "eats" match "eat"), but decide what you're going to do and justify your decision!

7. **[For four person teams only - 2 bonus points for others]**
   Properly reason about implication in PESS. Right now, if the user adds the facts: "its nose is external" and "its nose is tubular" to bird.kb, the "tubenose" rule will still ask whether it has an external, tubular nose. This is because only information in the known/1 predicate is checked in the implication check. PESS doesn't try to prove each of the implied goals. If it did, unfortunately, it would end up asking the user separately about many of the goals (which would be worse than one larger question!). So, create a "prove" mode in which PESS makes the closed world assumption and use this mode to check implication in the implied predicate. Once you're done, adding the nose facts above should avoid PESS ever asking about the implied, combined external/tubular fact.
   Additionally, if a combined question needs to be asked of the user, simplify it as much as you can, if possible. For example, if it's already known that "its nose is external", and prove needs to know that "its nose is tubular and external", pare the question down to just whether "its nose is tubular". (This is somewhat related to the previous problem, but my sense is that it's harder.)
8. **[For four person teams only - each item a bonus question for others]**
   Add these extra features to your parser:

a. Add commenting capability to the knowledge base files. Everything from a '%' to the end of the line should be ignored by the parser.
b. Disallow the determiner "a" before words starting with a vowel and "an" before words starting with a consonant. Allow either before "h" and "y". Discuss (in your report) whether you think this is a good idea given the purpose of the NL system (to allow users to engineer knowledge bases).
c. Allow user-defined vocabulary (from the "words:" statements in the knowledge base) to be multiple words long. So, the user should be able to say that "head cheese" is a noun and have sentences written like: "it contains head cheese" parse correctly. In other words, the multiple words do not have to be in quotes inside the rule. What this looks like in the "words:" statement is up to you, however. (That is, there may need to be quotes present around the words in the "words:" statement.)

Written Questions

Check under "Deliverables" for the due date of this task.

Briefly answer the following reflection questions:

1. What impressed you most about Prolog in the course of this project? Least?
2. What was the most frustrating thing about the toolkits and other code you used (WordNet, ProNTo_Morph, and 312PESS), and (roughly) what would the authors have to do to fix the problem?
3. What application (besides this one and adventure games) could you apply your natural language understanding code to?
4. What could we have done better on this project?
5. About how many hours did you spend on the project?

Peer Evaluation

To be submitted with the previous part:

Each team member must individually and privately rate their teammates' contributions to the project and their own, on a scale of 0 to 5.