CPSC 312 Project 1
November 17th, 2015

Name: Sam Sung
student number: 13714126
ugrad id: u2u8

Name: Rob Wu
student number: 42764118
ugrad id: y4d8

Name: Velina Ivanova
student number: 22644116
ugrad id: b6i8

2

The implementation involves adding a interpretation loop. The code is included under the heading “Intrepretation loop” in the pess.pl file. I start the implementation by referring the “Amezi’s discussion of interpreter shell. The go predicate in the document provides me some insight and given me a template for the implementation. The rest of the task is writing the interactive message and show another prompt to ask for the file name. The interactive message is created using the examples in the question as a template. This is to ensure that I fulfil the requirement of the question. **Note: the interpretation loop would result in error if the terminal is closed before the loop is quit (most likely due to the nature of interpretation loop).**

* During load, if the end of the file has any extra empty lines, it will trigger the goal to reset to default. If the user wish to define his/her own goal, he would have to remove any empty lines at the end of the kb file. <- this is resolved by removing the empty line check and moving the goal reset code to earlier stages. The default goal is retracted when process detects a defined goal.

Example:
1 ?- main.
This is the CPSC 312 Prolog Expert System Shell.
Based on Amzi's "native Prolog shell".
Type help. load. solve. list. goal. assert. or quit.
at the prompt. Notice the period after each command!
> load.
Enter filename in single quotes, followed by a period
(eg. 'bird.kb'.: 'bird.kb'.)
|: 'bird.kb'.
(...debug messages about knowledge base entries…)
No Goal Found, reset to default
rules loaded
> |: solve.
Would you say that it has external tubular nostrils ?> |: yes.
Would you say that it lives at sea ?> |: 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 ?> |: yes.
Would you say that it is white ?> |: yes.

```
Would you say that it ponderously flies ?> |: no.
Would you say that it is plump ?> |: yes.
Would you say that it powerfully flies ?> |: yes.
Would you say that it winters in the united states ?> |: no.
Would you say that it summers in canada ?> |: yes.
Would you say that it has black head ?> |: no.
The answer is snow goose
> goal.
Enter the new goal followed by a period.
what does it eats.
Understood goal to prove:
[it,eats]
> assert.
if it is a green goose then it has a flat tail.
Parsed rule is: [rule(attr(has_a,tail,[attr(is_like,flat,[])]),[attr(is_a,goose,[attr(is_like,green,[])])])]
Understood: if it is green goose then it has flat tail
> |: help.
Type help. load. solve. goal. assert. or quit.
at the prompt. Notice the period after each command!
> |: quit.

true .
```

3

Overview:

This section aims to expand the functionality of the program to parse questions in addition to sentences, and allow a user to set a top goal in their kb file.

We can divide the types of questions to be supported into a few templates:

- *Is it NOUN_PHRASE?*
- *Does it VERB_PHRASE?*
- *What does it VERB_PHRASE?*

In addition we also have the questions

- *What is it?*
- *What does it have?*

which don't nicely fit into any of the above categories.

To implement support for parsing questions we can then use the existing grammar for parsing noun and verb phrases. The simple case is for the first of the above two templates, which are yes/no questions. In this case the top goal to prove is "yes", and the body of the rule is just the parsed structure of the phrase.

The other three types of questions need to find an unknown answer, rather than just prove one, so the top goal defined needs to leave a variable for the part of speech that is of interest. However, if for example the knowledge base includes a rule for "it is a brown swan", and we ask the question "what is it", then if the rule holds, we would expect to receive the full answer "brown swan", and not just "swan". That means we need to also include associated attributes in the answer. In the case of "what is it" and "what does it have", we look for is_a and has_a relationships respectively. For the template "what does it VERB_PHRASE" we need to match the user inputted verb to one in a does relationship.

Examples:

Given a knowledge base containing:
goal: is it a white goose.
rule:
if its family is goose and
it is white
then it is a white goose.

solve output is:
?- solve.
Would you say that it has family that is goose ?> yes.
Would you say that it is white ?> yes.
The answer is yes
true .

Given a knowledge base containing:
goal: what does it eats.
rule:
if it is a swan and
it has a tail
then it eats green insects.

?- solve.
Would you say that it is swan ?> yes.
Would you say that it has tail ?> yes.
The answer is it eats green insects
true .

Shortcomings:
In respose to a question of the form,
What does it VERB_PHRASE?
Our implementation will respond with all attributes associated with the verb specified. That means that if we have a rule in our knowledge base that concludes “it slowly eats big insects”, then the answer to the goal “what does it eat” will include the entirety of “it slowly eats big insects”. Ideally, we would have supported two types of questions for this instead,
What does it eat?
How does it eat?
which in this case would have resulted in ‘big insects’, and ‘slowly’ respectively.

4.
Overview:
This section aims to expand the functionality of the program by allowing the user to set the top level goal from the interpreter loop with a simple command, goal. This command will prompt the user to enter a new goal to be parsed. This section takes the implementation of section 3 into a more user friendly approach by integrating it as a command.

Before defining a new top goal, we clear any goal that may have been set previously.

Example:
?- define_goal.

Enter the new goal followed by a period.
is it a green goose.
[attr(is_a,goose,[attr(is_like,green,[])])]
Understood goal to prove:
[it,is,green,goose]
true .

Shortcomings:

5.
Overview:
This section aims to expand the functionality of the program by allowing the user to add new facts and rules. This is done by implementing a new command in the interpreter loop. Once the assert command is entered, the user is directed to input their custom fact or rules to be parsed then asserted into the database. This section is relatively simple as it is very similar to section 4 in term of concept.

Our implementation makes use of the already existing try_parse/0.

Example:
?- process_rule.
if it is a green goose then it has a flat tail.
Parsed rule is: [rule(attr(has_a,tail,[attr(is_like,flat,[])]),[attr(is_a,goose,[attr(is_like,green,[])])])]
Understood: if it is green goose then it has flat tail
true .

Shortcomings:

6.
Overview:
This section aims to expand the program's vocabulary by integrating WordNet and ProNto_Morph. It uses ProNto_Morph's morph_atoms_bag to stem the words then pass it to WordNet's s predicate to determine the part of speech. We implemented wordType/2 that creates a 'bag' of the word in morph forms, then attempts to match the word with the word type in WordNet's database. In WordNet's s/6, each word is defined as s(Synset_ID, Word Number, Actual Word, synset type, sense number, Tag). Looking at WordNet's synset type (pos), we also addressed the concern over multiple type of adjectives in the WN database by adding both types of adjective generic identifiers. We had to manually retract the letter 'a', since it was causing problems.

Example:
1 ?- try_parse.
it is an amusing hyena.
Parsed structure is: [rule(attr(is_a,hyena,[attr(is_like,amusing,[])]),[])]

Understood: it is amusing hyena
true .

2 ?- try_parse.
it is an amazing hyena.
Parsed structure is: [rule(attr(is_a,hyena,[attr(is_like,amazing,[])]),[])]

Understood: it is amazing hyena
true .

3 ?- try_parse.
it is a computer.
Parsed structure is: [rule(attr(is_a,computer,[]),[])]

Understood: it is computer
true .

Bonus

8 a.
Overview:
This section aims to widen the flexibility of the knowledge base file by allowing commenting capability. We thought of 2 different approach to this problem. We could either perform char check before it passes through process/1 or after. The former modification is done on read_sentence/1. It would've require a separate check for the comment sign and another one for the ends at an 'end_of_line' char. We would have to make sure that end of line check does not get called unless there is a comment sign present. The latter proposed solution is simpler from a development standpoint. However, it requires minimum effort the user to input a single period('.') at the end of each comment in order to safely exit out of ignore mode. After some evaluation, we selected the latter as our solution. Following standard prolog commenting, the new process/1 is actively looking for the percentage sign as the start of the comment. It will only stop ignoring the input at a period or 'end_of_file' char.

Example:

% water based sport are fun
% water based sport are cool**.** <- only require a single period if the comments are continuous.
rule….
…..
% now on to land based sport**.** <- separated by rules, requires another period to end it

1 ?- load_rules('sport.kb').
comment ignored. <- an indicator text printed acknowledging successful comment identification
Understood: if it is played in water then it has waterbased label
Understood: if it requires teammates then it has type that is team
Understood: if it has waterbased label and it has type that is team and it requires sailboats then it is sailing
Understood: if it has waterbased label and it has type that is team and it requires balls then it is water polo
Understood: if it has waterbased label and it has individual type then it is swimming
Understood: if it is played on land then it has landbased label

Shortcoming:
Although this solution requires almost no effort from the user, it still is a constraint that could cause inconveniences. The former solution, although tricky, should allow it to perform comment ignoring without any user effort, similar to most IDEs.