# CPSC 312 Project 1 Group Report

## General Usage Steps:

- Switch into working directory and load 312-pess.pl.
- Load our custom rules by calling load_rules('smite.kb').
- Call *main.* to start the main interpreter loop.
- Enter *goal* to set a new goal if so you desire.
- Enter *solve* to fire away.
- For other operations, use *help* for additional information and commands.

## Task 1

The domain we chose is a selection of characters in a game called Smite (a MOBA game like LoL/Dota). While writing rules for this domain, it is easy when the classification is obvious (race/origin/role). But for some complex feature, it is difficult to fit the object in simple categories. Another thing is, natural language tends to use synonyms to represent the similar meaning. But when it comes to building a knowledge base, we have to remove all the synonyms and use as little words variation as possible because available vocabulary is very limited. In all, there are quite a lot preprocessing required to convert a common knowledge to the knowledge base NLP can understand.

## Task 2

This section was a bit strange to implement. It involved trying to make something that follows an imperative flow fit into a declarative language, which as it turns out involves a lot of cuts to make it follow only one path in the code. The actually implementation of the loop wasn't too bad once repeat/0 was understood as well as how to assert an end/0 so that the loop exits. Unifying each of the commands was something that was made to be easy since the remaining tasks required adding commands by others that may not understand the loop completely, so a simple list of commands each taking a single value to select was chosen. Implementing the commands themselves was relatively easy and just involved grabbing statements from knowledge base processing and reading lines or sentences into them. Extending from that, listing predicates was just a matter of allowing rules to be unified then printing them out until it ended.

## Task 3

### Preprocessing: (312-pess)

To set a new goal, we can either specify it in the main interpreter loop, or we can set it as part of the knowledge base file.
In both cases, goal setting will first clear the database and then call into specific goal parsing routine.

## Goal Parsing: (312-pess-grammar under 'Goal Sentence parser')

Goal parsing was done by breaking down possible forms of question sentences, which were given as examples in the project outline, and then rearrange the sentence in a form that can be parsed using the existing parser.
In order to break down question sentences, there are several cases we must consider:

- Case 1: "is it a brown swan?"
  This case is broken down into a general form of:
  **[VIS = is][DT/PRP = it][NP = a brown swan]**
  (Parsed using: nlp.stanford.edu:8080/parser/index.jsp)
  Thus we see that this sentence can be rearranged into: **it is a brown swan**.
  Using this new form, we can properly parse it into the expected result given in the project outline and set it as the new goal.

  CAVEAT 1:
  Note, we can parse the sentence this way because we must specify a third person subject - it - in the question to make the goal. Otherwise, this type of parsing does not work.
  This caveat also carries over to other parsing cases, but this note will not be repeated.
  However, this case does not take care of sentence in the form of: "is it a swan?"
  Case 5 will explain further on this deficiency.

- Case 2: "Does it eat insect?"
  This case is broken down into a general form of:
  **[NNP = does][NP = it eat insect]**
  Again, we can see that, by stripping 'does', we obtain a parse-able sentence (NP) that may be parsed into a goal, albeit with bad grammar.
  Caveat 1 also applies here.

- Case 3: "What (the heck) does it have?"
  This case is broken down into a general form of:
  **[WDT = what][NP(optional) = the heck][VBZ = does][DT/PRP = it][V = have]**
  Looking at the breakdown, we see the question really is just: "it has what?"
  Which we can obtain by reassembling the original sentence breakdown:
  [DT/PRP = it][V = have][WDT = what]

  After obtaining this new sentence, we can then send it to the modified parser, which is covered in Case 5.

- Case 4: "What (the heck) is THAT?" BONUS
  This bonus case is broken down into a general form of:
  **[WDT = what][NP(optional) = the heck][VIS = is][DT = THAT]**

This construct breaks down into:
"what is it?"
Which has a trivial solution of:
"it is what"

After mapping the original sentence to the new one, we again send it to the modified parser.

- Case 5: "It is a small what."
  This is the general case that is the common code paths for all other cases.
  In this case, we modify the existing sentence parsing code to recognize sentence which contains NPT "what".
  With the original parser, any sentence with "what" may be converted into a "has_a" sentence attr.
  By seperating out sentences with "what", we can skip the converion step and make the parser spit out the parsed sentence straight away.

  This gives us a parsed sentence rule of the following form to work with:
  rule(attr(V, what, Attrs))

### Post Processing: (in 312-pess)

Given result rule(attr(V, Q, Attrs)), we can extract from the rule the attributes of the parsed sentence. After extraction, we obtain V,  Q and Attrs where we can then assert as the new goal. During the assertion, we check if Q is "what" to see if we need to replace Q with X, thus setting the goal key as an unknown variable. Otherwise, Q is already specific, and thus we set the goal using this Q as the key.

## Task 4

This section was implemented in parallel with 3 in order to provide a faster way to test how goals were parsed. A common procedure was defined, set_goal/1, which would take the sentence and process it as a goal. There’s not too much to this implementation other than that the work done in 2 to make adding additional commands easier paid off.

## Task 5

Adding new facts and rules are trivial as goal setting is already done and routines for setting facts and rules already exist.
Thus, we can simply add new commands in the interpreter loop to ask for user input and send the sentence directly to the set fact/rule routines.
These set fact/rule routines call into specific parsing routines and assert the result into the database, which is not in the scope of discussion for this question.

## Task 6

This section was easy to implement since wordnet provides an argument of the word type to unify with. Using that it becomes a task of having generic word queries that check the type of that query. A wrapper was created to do this, as well as break the word into its components using pronto-morph. Each of the results from pronto-morph is then checked against wordnet to see if it matches the specified type, if not it just continues. Finding which type to compare the query to was in the documentation for wordnet so for each of the word types adding the appropriate wrapper case added the necessary functionality.

## Task 8.a) BONUS

Commenting capabilities for the knowledge base is done by checking the first letter of a line.
In this case, we used '#' (sharp) as well as '%' for our comments. By filtering out all lines with sharp letter at the beginning, we can discard these sentences by doing nothing in the processing step.
Note, however, we must have period or some other delimiter to end the sentence, otherwise the line will go on to consume the next block of text before a line end symbol.

## Task 8.b) BONUS

Vowel checking is performed when determining noun phrase and when loading vocabulary. “a” and “an” are no longer considered determiners with no effect, but treated specially before other determiners are resolved. If the determiner is “a”, the next phrase will be checked whether the start letter is a consonant or “h”, “y”. If the determiner is “an”, similar check is performed against vowel.
One issue which came up is adj(a) returns true, and this causes attr(is_like, a, []) is discovered as APTerms and as a result, “a ash” is able to be parsed without error. The temporary solution is to hardcode “a” in adj/1 so that adj(a) will always return false. This is safe because “a” is indeed a special case to handle.
For the purpose of NL system, we do not think enforcing this is a good idea because it increases risks of having error in knowledge base, while not having benefits because a/an are just determiners and normally have no effect.

## Task 8.c) BONUS

Multiple words phrase in double quotes are now supported for user-defined vocabulary.
When parsing vocabulary sentence, consider the head can be either multiple words and single word, and use similar code from lit/2, it is easy to parse multiples words phrase in double quotes to atom. When parsing rules, we also try parse multiple words to atom and check for its corresponding types.
It seems to slow down the processing a bit.