Computational Linguistics

LIN 567/467 – Spring 2018

Project

Due date: by 11.59pm, May 14th
Late submissions will not be accepted

1 Instructions

- This project can be completed by groups of two students, or individually. If a project is submitted by a group of two students, both students get **the same grade**.
- If there is any evidence that code was borrowed from other students, oral examinations will be used to determine if unethical behavior occurred.
- Upload one zip file containing:
  - **Code** that successfully compiles and implements the system discussed below. The file should contain information about the authors, and the code should be commented, so that students can explain in detail what is achieved. Please use intuitive variable and predicate names , whenever possible. Submitting a non-text file that Prolog cannot compile results in **point deduction**.
  - **PDF Report** that again identifies the authors, and illustrates how the modules of the system work, with execution examples. Submitting examples that do not match the output will incur in **point deduction**.

2 Project description

Suppose you are asked to develop a *natural language understanding* system that can interface with a 'smart' refrigerator and allow users to text queries or instructions to their fridge. If the message is a declarative sentence, the fridge agrees or disagrees according to its contents. If the message is a question, the fridge answers, according to its contents. Your goal as a language engineer, is to develop a First-Order Logic based natural language understanding system that can create exact semantic representations for complex linguistic input, and respond appropriately, as illustrated below:

```
?- chat.

|: Every blue container on the top shelf contains a sandwich that has no meat.
```

```
> That's right.

|: Every white container on the bottom shelf contains a banana.
> That is not correct.

|: What does the yellow bowl on the middle shelf contain?
> Two eggs.

|: Are there two watermelons in the fridge?
> Yes.

|: Is there milk?
> No.

|: Who drank the almond milk?
> Sam.

|: Which milk did Sam drink?
>  Skim.

|: Is there a sandwich that does not contain meat?
> Yes.

|: Is there an empty box of popsicles in the freezer?
> Yes.

|: Bye.
> Bye.

?-
```

There should not be any limit to the size or complexity of the input. You must construct a vocabulary with at least 5 lemmas of different parts of speech: verbs (some intransitive, some transitive, etc.), prepositions, adjectives, nouns, proper names, and determiners (including numerals from *one* to *ten*). The vocabulary should also contain the interrogative pronouns *who*, *which*, and *what*, and the semantically vacuous *there*. Your grammar should at least correctly deal with all the execution examples discussed above. The main code of the project is shown below. You may wish to modify some of it.

```
% Main cycle. Accept spaced sequences of tokens.
% Terminates when user types 'bye'.
chat:-
 repeat,
   readinput(Input),
   process(Input),
  (Input = [bye| _] ),!.
```

```
% Read input into tokenized list
readinput(TokenList):-
   read_line_to_codes(user_input,InputString),
   string_to_atom(InputString,CharList),
   string_lower(CharList,LoweredCharList),
   tokenize_atom(LoweredCharList,TokenList).

% Deal with 'bye...' input
process([bye| _]):-
   write('> bye!').

% Use grammar to process non-'bye...' input, and use model checker to respond
process(Input):-
  parse(Input,SemanticRepresentation),
  modelchecker(SemanticRepresentation,Evaluation),
  respond(Evaluation),!,
  nl,nl.

% ...  use either the left corner parser or a chart parser of your choosing

% Declarative true in the model
respond(Evaluation) :-
    Evaluation = [true_in_the_model],
    write('That is correct'),!.

% Declarative false in the model
respond(Evaluation) :-
    Evaluation = [not_true_in_the_model],
    write('That is not correct'),!.

% Yes-No interrogative true in the model
respond(Evaluation) :-
    Evaluation = [yes_to_question],
    write('yes').

% Yes-No interrogative false in the model
respond(Evaluation) :-
    Evaluation = [no_to_question],
    write('no').

% wh-interrogative true in the model
% ...

% wh-interrogative false in the model
% ...
```

As indicated above, one of your main tasks is to define the predicate `parse`/2. This predicate must access a vocabulary, a grammar, and a parser. The result of the parse is a semantic representation that is given to a model checker, via `modelchecker/2`, another predicate that you must define.

Finally, the evaluation produced by the model checker is returned to the user via `respond/1`, only partially defined above. More details about the parser are discussed immediately below.

2.1 Grammar

1. Each lemma must be encoded as `lemma`/2. Lemmas should **not be inflected** with regular inflection patterns. For example, *contain* is a lemma, but *contains, contained*, and *containing* are not lemmas. For the purpose of this project, irregularly inflected variants of the same verb **are** lemmas, e.g. *drink, drinks, drank, drunk*.

   ```
   lemma(box,n).
   lemma(contain,tv).
   ```

2. A morphological parser must convert an input token into a (possibly inflected) lexical item annotated with the corresponding syntactic and semantic representation. Some execution examples are given below. Each lexical item is of the form `lex`/2, where the first argument contains categorial and semantic information and where the second argument contains the word form in the user input.

   ```
   ?- lex(X,box).
   X = n(_G293^box(_G293))

   ?- lex(X,boxes).
   X = n(_G293^box(_G293))

   ?- lex(X,contained).
   X = tv(_G294^_G297^contain(_G294,_G297),[ ])

   ?- lex(X,contains).
   X = tv(_G294^_G297^contain(_G294,_G297),[ ])

   ?- lex(X,contain).
   X = tv(_G294^_G297^contain(_G294,_G297),[ ])

   ?- lex(X,containing).
   X = tv(_G294^_G297^contain(_G294,_G297),[ ])
   ```

3. As seen above, the morphological parser should be able to handle nominal and verbal inflection. Note that the relation or predicate element in the $\lambda-$term is always the corresponding lemma. For simplification purposes, the semantic correlates of verbal inflection should be ignored. Your implementation does not need to consider any form of derivation.

4. Singular common nouns are very easy to handle, as illustrated below.

   ```
   lex(n(X^P),Word):- lemma(Word,n), P =.. [Word,X].
   ```

2.2 Semantic analysis

Use a parser of your choosing to parse the input and construct semantic representations like those seen below. These examples need not be exactly the same that your system obtains but **crucially** the FOL translations should impose the same conditions, so that the sentences are correctly deemed true/false according to any given model.

```
?- parse([a,blue,box,contains,some,ham],X).
X= s(exists(_G387,and(and(box(_G387),blue(_G387)),exists(_G400,and(ham(_G400),contain(_G387,
    _G400)))))),[])

?- parse([a,blue,box,contains,ham],X).
X= s(exists(_G387,and(and(box(_G387),blue(_G387)),exists(_G400,and(ham(_G400),contain(_G387,
    _G400)))))),[])

?- parse([the,white,box,that,the,freezer,contains,belongs,to,sue],X).
X= s(the(_G400,and(and(and(box(_G400),white(_G400)),the(_G387,and(freezer(_G387),
contain(_G387,_G400))))),belong(_G400,sue)),[])

?- parse([is,there,an,egg,inside,the,blue,box],X).
X = ynq(exists(_G406,and(egg(_G406),the(_G419,and(box(_G419),blue(_G419
),contain(_G419,_G406))))))

?- parse([are,there,two,eggs,inside,the,blue,box],X).
X = ynq(two(_G406,and(eggs(_G406),the(_G419,and(box(_G419),blue(_G419
),contain(_G419,_G406))))))

?- parse([what,does,the,green,box,contain],X).
X = q(_G386,and(thing(_G386),the(_G394,and(box(_G394),green(_G394)),contain(_G394,_G386))))

?- parse([who,put,every,yellow,box,on,the,white,bowl],X).
X = q(_G366,and(person(_G366),forall(_G374,imp(and(box(_G374),yellow(_G374)),
the(_G367,and(bowl(_G367),white(_G367)),put(_G366,_G374,_G367))))))
```

2.3 Model checker

1. Create a model based on the examples above, and expand it, so that the examples provided above are evaluated as shown (or sentences with the same structure but different words). Show in your report five other examples that illustrate the range of formulas that the parser can handle beyond those discussed here.

2. You must revise `sat/3` in order to deal with numerals, plurals and potentially other cases. When evaluating formulas with plurals, e.g. $Two_x(box(x) \wedge \phi(x))$ you can use `findall/3` to obtain a list L of all values of x that satisfy $box(x) \wedge \phi(x)$ and then make sure L contains at least two solutions.

3. The revised model checker `sat`/3 should be encapsulated by `modelchecker`/2, which manages the output of `sat`/3. The examples below illustrate how the system should work. Your system

must give appropriate answers given any model that is chosen, not just the one you provide.

```
? -modelchecker(s(forall(_G387,if(table(_G387),exists(_G395,and(and(box(_G395),black(_G395)),
    contain(_G395,_G387))))),[]), Result)
Result = [true_in_the_model]

?- modelchecker(q(_G234,and(thing(_G234),exists(_G235,and(and(box(_G235),green(_G235)),
    contain(_G235,_G234))))), Result)
Result = [[_G234,p],[_G235,d]]
```

4. You must allow for lexical inferences of the following kind. Suppose the freezer contains *ham*, and *chicken*. If the user asks *Is there meat in the freezer?* then the system should be able to answer *yes*, even though *meat* is not in the model, only *ham*, and *chicken*. This can be done by introducing a small ontology (like a hand-built WordNet fragment) that allows predicates to be satisfied directly in the model, or indirectly, via inference.

2.4 Question answering

Compute the appropriate response, depending on the value of `Result`:

1. If the input is a true declarative sentence, then the system must agree with the user, by printing something like `That is correct.`
2. If the input is a false declarative sentence, then the system must disagree with the user, by printing something like `That is not correct.`
3. If the input is a yes-no question, then the system must answer it according to the output of the model checker: `Tes.` or `No.`
4. If the input is a content question, then the system must answer it, according to the output of the model checker. The answer should consist of all the properties P that the object in the answer set has, according to the model. For example, assuming 'd' is a blue bowl and 'p' is a yellow box, and that both are contained by a green box on the top shelf, then the answer to the following question should be:

```
> What does the green box on the top shelf contain?
: a blue bowl, a yellow box
```

3 Grading

The 31 points awarded by the project are as follows:

- A **zip** file is submitted, containing (i) a **pdf** file with a complete and well-formatted report and (ii) a single well-formatted **text** file that runs without compilation errors or warnings.

[1 point]

- The lexicon contains enough elements for the system to be able to parse (at least) all the above examples.

[6 points]

- The morphological parser avoids listing all inflections for all words by identifying the stem of each token and tagging the word appropriately.

[6 points]

- Parsing module uses a CFG to obtain FOL representations of arbitrarily complex input (i.e. the grammar is recursive so that sentences with any number of words are processed correctly and yield well-formed semantic representations that can be dealt with appropriately by the model checker).

[6 points]

- Model checker module computes the truth value of the (arbitrarily complex) formulas in the input, and outputs the appropriate results, including for those formulas that involve plurals, numerals, and lexical inferences.

[6 points]

- Response module computes the appropriate reply to the user's input.

[6 points]

4 Programming

The recommended programming language is Prolog, but those students who which to use the NLTK's Python modules can do so. The latter is a particularly viable option for those students who understand the Prolog approach but wish to see how the exact same goals are achieved in Python, using off-the-shelf tools. However, be advised that the latter answer requires considerably more coding effort than the Prolog approach, and that no Python question or instruction will be provided by the instructor or TA.