Semantics in Python

Semantics 2 :: 2023

Group project work package

Note: For this project basic Python programming skills are required,
for instance the level attained in the simultaneous course Python for Linguists.

1 Introduction and motivation

An important part of the motivation for formal semantics is that one can *compute* the predictions of one's formal semantic theory. Instead of having to rely on intuitive-but-vague paraphrases to characterize what something means, a formal syntactic + semantic grammar generates precise meaning descriptions for us, along with formal tools for e.g. determining truth/falsity in a model, and meaning relations such as entailment. Nevertheless, as we have noticed, with more complex sentences manually exploring all potential logical forms can become a lot of work. Moreover, what if we want to look in a large corpus for sentences with a particular type of meaning? In this project we explore whether the formal semantic theory we have been developing can be implemented computationally, to let our computer do the dirty work for us.

Goal of this project: To solidify your understanding of formal semantics, to gain experience with implementing a rule-based formal system for natural language processing and the types of data structures this involves (grammar, trees, dictionaries), and to become familiar with possible ways of making semantics more computational.

2 Literature

NLTK Selected pages from `www.nltk.org`, the documentation of the Natural Language ToolKit which we will use, in particular:

- `https://www.nltk.org/howto/grammar.html` (just the first example CFG)
- `https://www.nltk.org/howto/tree.html` (until (but excluding) the 'Tree Parsing' section)
- `https://www.nltk.org/api/nltk.tree.tree.html`

J&M Jurafsky and Martin (2022). Speech and Language Processing (3rd ed. draft). In particular:

- Chapter 12 on context-free grammars, until (incl.) section 12.2 (`https://web.stanford.edu/~jurafsky/slp3/12.pdf`).

3 What you should hand in

- For this project you will collectively hand in only the resulting code, in a single .zip file. No written 'report' from the group is necessary, although some of the assignments below will ask you to include some information in a docstring in your code.
- However, you *are* asked (as in all the other projects) to separately hand in a short (half page) *individual* reflection, see the end of the file.

- I will be reviewing your effort and insight, more than exactly how many of the assignments in this document you will manage to solve (though of course I will take that into account, too). About 10% of your grade will be determined by the code style (clarity, suitable division into functions, transparent names, helpful docstrings), taking into account that this is likely your first substantial Python project.
- If your program does not run, you should consider removing some functionality so that it does run. Or better: make sure you have a version that runs, a few days before the deadline, and keep it as backup ('minimal viable product'), just in case any last-minute changes introduce errors. I will not review any commented-out code, or any functions that are not called by the main program.
- If your program spans multiple files, make sure it is clear to me which Python file I should run (and, if non-trivial, how) for properly evaluating your program.

4 Part 1: Formal grammar as the backbone for formal semantics

Read J&M, chapter 12, until (incl.) section 12.2, about context-free grammars.

Assignment 1. Reference the NLTK page about context-free grammars, and use the `nltk` library to implement a grammar that recognizes sentences like the following, generating a simple syntactic tree with, say, S = NP VP as the top node (the type of tree we had before applying quantifier raising, and before we added event phrase EP and thematic roles).

> John sees Mary.
> Some girl sees every boy.
> A student walks.
> Every eager student passed.

The following code can be used as a starting point:

```
import nltk

grammar = nltk.CFG.fromstring("""
    <define your grammar here>
""")

parser = nltk.parse.RecursiveDescentParser(grammar)

sentence = "some girl saw every boy"
tokens = sentence.split()

parse_trees = parser.parse(tokens)

if not parse_trees:
    print("No parse tree found.")

for parse_tree in parse_trees:
    print(parse_tree)
```

Some advice:

- Your grammar's production rules cannot contain quotation marks, except for defining leaf nodes (lexical entries; see the example on the NLTK webpage). For a node like V' (daughter of VP, for transitive verbs) you can avoid the quotation mark by calling it (e.g.) Vbar instead.

- It may be convenient to include unary-branching nodes for the lexical categories like 'Det' (determiner), 'N' (noun), 'PN' (proper name), 'IV' (intransitive verb), 'TV' (transitive verb). The resulting parse trees will have such unary-branching nodes at the bottom, but other than that they should look similar to the kinds of trees we used in class.

Assignment 2. Reference the NLTK pages about trees. Next, parse some example sentences and learn how to navigate the resulting parse trees. Concretely:

- Use the `pretty_print` method on the parse tree to print the tree in a very readable way. Make sure you use this method for inspecting your trees frequently in the assignments to come.
- Positions in trees are tuples like `(0, 1, 1)` (this example meaning, from the top node, enter the left branch (0), then take the right branch (1), then the right branch again (1)). Use this to select particular nodes in your parse trees, e.g., try to select the verb, the subject, a determiner.
- Practice iterating over the tree (and printing the items), in the following ways:
  - `for subtree in parse_tree.subtrees(): ...`
  - `for pos in parse_tree.treepositions(): ...`
  - `for pos in treepositions(order="leaves"): ...`
  - `for pos in treepositions(order="postorder"): ...`
- Define a function that takes a parse tree, and returns the position of the verb (i.e., a tuple). If your grammar follows the textbook closely, then the position of the verb will depend on whether it is a transitive or intransitive verb (i.e., depending on the presence/absence of a Vbar node).
- Define a function that takes a position in the parse tree (i.e., a tuple like `(0, 1, 1)`) and returns the position of the parent node (`(0, 1)` in this case), or `None` if it has no parent.
- Define a function that takes a parse tree, and returns the position of the subject DP.
- Define a function that takes a parse tree, and returns the position of the object DP if one exists, and `None` otherwise.
- Define a function that returns a list of positions in the tree, corresponding to all quantificational DPs (i.e., the constituents that might undergo quantifier raising, which will be implemented later).

Assignment 3. Think of a way to represent logical translations in Python. You are free to use any approach you see fit, but the easiest (given the level of Python attained in the PyLing course) might be to represent each formula as a string. For instance, `"all x [P(x) -> Q(x)]"` could be used to denote the formula $\forall x[P(x) \rightarrow Q(x)]$. Using your chosen representation, enter translations for the example sentences from Assignment 1.
Advice: If you represent formulas as strings, then it will likely be convenient to always delimit the scope of lambda operators with some kind of brackets/parentheses, in particular brackets that are not yet used for any other purpose. For instance, `"lambda x <SLEEP(x)>"` could be used to denote $\lambda x\text{SLEEP}(x)$. Using such brackets will make it easier to determine which variables to substitute when doing function application, in the next assignment.

Assignment 4. Think of a way to list lexical entries with their logical translations, and try to introduce/maintain a separation of grammar and lexicon: first define a lexicon, and then automatically add the various lexical entries to the right places in your grammar definition (string), *before* compiling the latter into a CFG/parser.

Assignment 5. Implement a function that takes two logical formulae and applies (extensional) function application and beta reduction (= simplification), returning the result.

- Your function should check for type-compatibility, i.e., determine whether one expression can combine with another, and then choose the right formula as the function and the other as the argument. This is easiest to check if you adopt some conventions, e.g., that lowercase x, y, z are always of type e and that uppercase words are of type $\langle e, t \rangle$. But you can also add type annotations to your formulae if you want (e.g., `lambda x_e`, `lambda P_et`), or add them as additional information in your lexicon (cf. the use of dictionaries as 'feature structures' in one of the Python coding quests).
- Your 'function application'-function should also apply 'beta reduction' as much as possible, that is, simplify the formula by feeding arguments into functions. Test your function on simple cases, like combining `SLEEP` with `susie`, before trying things with lambdas, like combining `lambda x <SLEEP(x)>` with `susie`, and combining `lambda Q <all x [STUDENT(x) then Q(x)]>` with `SLEEP`.

5 Part 2: Compositional semantics

Assignment 6. Implement a function that takes syntactic tree and returns a logical translation according to our *extensional semantics*, or `None` if no well-formed formula of SL can be derived. It should derive suitable translations for the simple sentences in Assignment 1, *except* for the one with a quantifier in object position (remeber why?).

Suggested approach:

- Translation composes from the leaf nodes upwards in the tree. Therefore, you could iterate over the nodes in your tree bottom-up, by using the method `treepositions(order='postorder')`.
- When you have computed the translation for a node, you can store it in a dictionary that maps a unique identifier of each node (e.g., a string representation, or its position tuple) to its translation. That way, branching nodes higher up can easily retrieve the translation of their daughters lower in the tree by looking up their translations in this dictionary.
- If you followed the advice in Part 1, you will have some unary-branching nodes above the lexical leaves, unlike the textbook. These should simply adopt the intepretation of their daughters unchanged, merely 'passing them on' to nodes higher up in the tree.

Assignment 7. Implement a function corresponding to quantifier raising, that takes a basic syntactic tree, and returns a *list* of (one or multiple) new structures, the 'logical forms', obtained by raising the quantifiers in all the various ways permitted: we have seen that quantificational DPs can raise in any order, one above the other, giving rise to multiple translations.

Some advice:

- Use `parse_tree.copy(deep=True)` to copy your base structure before modifying it. That way you can save the original base structure while computing the various possible logical forms.
- Earlier you should have made a function to find all quantificational DPs, that you can use here.
- Recall that quantifier raising leaves in place a 'trace', translated as the same variable bound by the quantifier; meaning composition from the trace upwards can proceed as usual, until you reach the (now raised) quantifier, at which point you should lambda-abstract over the trace before combining it with the quantifier.
- It can be helpful to first define an auxiliary function that takes a base structure plus the position of a particular quantifier, and a position of the target node to (above) which it is raised, and applies quantifier raising for just those parameters. Separately, define a function that does this for each quantifier and possible target position (and repeatedly, on the obtained structure, for each (hitherto not-yet-raised) quantifier remaining).

- You may have to keep track of which quantifiers have already been raised, which you can do either by labeling them as such (e.g., appending the marker '¡ to their label), or by checking if their trace variable exists lower down in the tree, or by checking if they appear above what used to be the highest possible node prior to raising, e.g., S or VP.

Assignment 8. Define a single 'wrapper' function that handles both quantifier raising and translation. Then test your system thoroughly. Does it yield the right, well-formed translations for sentences containing (single or multiple) quantifiers? (Note: No need to handle scope islands yet.)

6 Part 3: Extensions

For the following assignments, make sure you always keep a 'minimal viable product' as backup, i.e., a program that handles a few simple examples. Try to approach each assignment by extending this minimal case, rather than combining everything (the latter is encouraged, but optional).

Assignment 9. Make your program work for *intensional* semantics. This will involve new lexical entries and a new composition rule: intensional function application. Add a propositional attitude verb like *believe* and show that your system derives a de-re and de-dicto interpretation (at least without island constraints. . .).

Assignment 10. Make your program work with *events*, and make sure to add an adjective to illustrate. This will involve:

- Modified the lexicon: change verbs to be predicates over events, and add thematic role labels like AGENT and PATIENT to the lexicon.
- Modifying the LFs: LFs should now contain EP (with specEP translating as an existential over events) and ThetaP, which you can achieve either by adding placeholders like AGENT and EXISTSEVENT to the raw input string *before* parsing by the CFG (and add rules for EP and thetaP to the grammar specification), or by parsing the sentence as usual and then, *after* parsing, modifying the structure to insert EP and thetaP nodes.
- Modifying the composition: besides function application, the rule 'predicate conjunction' should now be available too, which is invoked instead of function application whenever both sister nodes are of type $\langle e, t \rangle$.

Assignment 11. Implement island constraints on quantifier raising.

Assignment 12. Clean up your code before submitting :)

7 Also hand in...

Besides handing in, collectively as a group, your code (in a single .zip file, with a single program containing your attempted solutions to the assignments), also (separately) hand in a short *individual* reflection (max. half a page) on the project and your role in it. Things I would love to hear about include:

- (How) has the project increased your understanding of the kind of phenomena semanticists are interested in, and the way we use formal tools to learn more about how language works?
- Did you feel the 'group work' format was useful? Was it too much work, too little, or just right?
- Did you feel that the ground we covered in Semantics 1 and 2 (and Python for Linguists) prepared you enough for this more complex and independent type of work?
- Were you happy with your group mates and the division of labour, and is there any reason why I shouldn't grade your group collectively?