CS187 Lab 4-3: Semantic ambiguity and quantifier scope

September 7, 2024

[ ]:
```
# Please do not change this cell because some hidden tests might depend on it.
import os

# Otter grader does not handle ! commands well, so we define and use our
# own function to execute shell commands.
def shell(commands, warn=True):
    """Executes the string `commands` as a sequence of shell commands.

       Prints the result to stdout and returns the exit status.
       Provides a printed warning on non-zero exit status unless `warn`
       flag is unset.
    """
    file = os.popen(commands)
    print (file.read().rstrip('\n'))
    exit_status = file.close()
    if warn and exit_status != None:
        print(f"Completed with errors. Exit status: {exit_status}\n")
    return exit_status

shell("""
ls requirements.txt >/dev/null 2>&1
if [ ! $? = 0 ]; then
 rm -rf .tmp
 git clone https://github.com/cs187-2021/lab4-3.git .tmp
 mv .tmp/tests ./
 mv .tmp/requirements.txt ./
 rm -rf .tmp
fi
pip install -q -r requirements.txt
""")
```

[ ]:
```
# Initialize Otter
import otter
grader = otter.Notebook()
```

When discussing syntactic representations, we encountered a central issue in natural language — ambiguity. In lab 3-3, we introduced PP attachment ambiguity to show how a single sentence ("Twain bought a book for Howells") can have multiple distinct syntactic structures, each bearing

a different meaning.

Semantic ambiguity can arise even for a *single* parse of a sentence giving rise to different meanings. In this lab, we'll introduce one example of this phenomenon, quantifier scope ambiguity. We'll take a look at a sentence that elicits this type of ambiguity and propose two possible FOL representations of the sentence, confirming that they produce different truth values in a model. Then, we will use a syntactic-semantic grammar similar to those of lab 4-1 to parse and interpret the sentence, pointing out a weakness of this method.

Preparation

[ ]:
```
import os
import sys
import wget

import nltk
```

[ ]:
```
# Download code for augmented grammars
remote_script_dir = "https://raw.githubusercontent.com/nlp-course/data/master/
 ↪scripts/"
local_script_dir = "./scripts/"

# Create and search the local script directory
os.makedirs(local_script_dir, exist_ok=True)
sys.path.insert(1, local_script_dir)

# Download files to script directory
wget.download(remote_script_dir + "trees/transform.py", out=local_script_dir)

# Import functions for transforming augmented grammars
import transform as xform
```

1 The flight world model

We'll be using the flight world as the domain for the lab as we did for lab 4-1, so we provide a simple model for it here.

[ ]:
```
# Constants

London = "London"
NewYork = "New York"
Paris = "Paris"
Boston = "Boston"
TelAviv = "Tel Aviv"

DL10 = "DL10"
```

```
DL11 = "DL11"
DL13 = "DL13"
LY01 = "LY01"
LY12 = "LY12"

# Properties

Object = {London, NewYork, Paris, Boston, TelAviv, DL10, DL11, DL13, LY01, LY12}
Flight = {DL10, DL11, DL13, LY01, LY12}
City = {London, Paris, NewYork, Boston, TelAviv}
Capital = {London, Paris}

# Relations

Origin = {
    (DL10, London),
    (DL11, London),
    (DL13, Paris),
    (LY01, Paris),
    (LY12, London)
}
Destination = {
    (DL10, NewYork),
    (DL11, TelAviv),
    (DL13, Boston),
    (LY01, NewYork),
    (LY12, TelAviv),
}
```

2 Quantifier meanings

Before discussing quantifier scope ambiguity, we need to establish an appropriate notion of quantifier meanings in natural language. In FOL, there are only two quantifiers, the universal quantifier $\forall$ ("for all") and the existential quantifier $\exists$ ("there exists"). In natural language, we find many more quantifiers. A few examples:

- Universal: "every", "each"
- Existential: "some", "a(n)"
- Exact counts: "one", "two", "three", "infinitely many"
- Approximate counts: "few", "many", "most", "several"
- Count bounds: "more than three", "fewer than 100"

Natural language quantifiers can be thought of as expressing a *relationship between two properties*.

> Recall that a *property* can be thought of as a set of objects (those bearing the property) or a function from objects to truth values (which takes those objects that bear the property to *true* and objects not bearing the property to *false*). In our flight world, *Flight* is a property, as is $\lambda x.Origin(x, Boston)$ ("objects originating in Boston").

Natural language quantifiers like "every" express a relationship between two properties, the *restriction* of the quantifier and the *scope* of the quantifier. For instance, in the sentence "every flight leaves from Boston", the restriction property is expressed by "flight" and the scope property is expressed by "leaves from Boston". The relationship that "every" expresses is simply that *everything bearing the first property also bears the second property.* Similarly, "some" expresses the relationship that *there is something bearing the first property that also bears the second property.* And "most" expresses that *most (at least half, say) of the objects bearing the first property bear the second property.*

This way of thinking about quantifiers as expressing relationships between two properties is referred to as *generalized quantifiers*, because it generalizes the FOL notion of quantifier by adding an explicit and separate restriction.

> In FOL, quantifiers $\forall$ and $\exists$ have only a scope and not a restriction. When we write $\forall x. ...$, the ... is the scope of the quantifier; we can think of this as expressing that $\forall$ holds of the single property $\lambda x. ...$. How then can restriction properties be expressed in FOL? By folding them into the scope. * "every flight leaves from Boston": $\forall x. Flight(x) \implies Origin(x, Boston)$ * "some flight leaves from Boston": $\exists x. Flight(x) \wedge Origin(x, Boston)$
>
> Notice though that in incorporating the restriction ($Flight(x)$) into the scope of the FOL quantifier, we use different ways of combining them depending on the quantifier. That's somewhat inelegant. More importantly, for most NL quantifiers, *there is no way of encoding the relationship between restriction and scope with just a single property*, as we happen to be able to for the universal and existential quantifiers. Hence, semanticists have moved to the generalized quantifier approach to quantifier meanings, as we do here.

To allow for meanings for the first order quantifiers, we'll define some Python functions that capture this generalized quantifier approach. Each quantifier function takes the two properties, restriction and scope, and returns a truth value.

[ ]:
```
def for_all(R_property, S_property):
    """Returns `true` just in case all objects bearing `R_property`
    also have `S_property`."""
    restriction_objects = [x for x in Object if R_property(x)]
    scope_objects = [x for x in Object if S_property(x)]
    return all([x in scope_objects for x in restriction_objects])

def there_exists(R_property, S_property):
    """Returns `true` just in case at least one object bearing `R_property`
    also has `S_property`."""
    restriction_objects = [x for x in Object if R_property(x)]
    scope_objects = [x for x in Object if S_property(x)]
    return any([x in scope_objects for x in restriction_objects])
```

We can see how these work by verifying that it's not the case that all flights originate in London, but some flight does.

[ ]:
```
## expressing "every flight leaves from London" -- should be false
for_all(lambda x: x in Flight, lambda y: (y, London) in Origin)
```

```
[ ]: ## expressing "some flight leaves from London" -- should be true
there_exists(lambda x: x in Flight, lambda y: (y, London) in Origin)
```

Now it's your turn. Implement a generalized quantifier for the English determiner "two", which holds if exactly two of the objects in the restriction are in the scope as well.

Hint: we used `Object` to store all objects.

```
[ ]: #TODO - Implement the `two` function
def two(R_property, S_property):
    """Returns `true` just in case exactly two objects bearing `R_property`
    also have `S_property`."""
    ...
```

```
[ ]: grader.check("two_quantifier")
```

We can use this implementation of the quantifier to demonstrate that two flights have New York as destination:

```
[ ]: ## expressing "two flights go to New York" -- should be true
two(lambda x: x in Flight, lambda y: (y, NewYork) in Destination)
```

Express and test the proposition that two flights originating in London have a capital as destination. You'll end up using two quantifiers.

```
[ ]: #TODO - express "there are two flights that leave from London and go to a␣
  ↪capital" -- should be false
two_london_capital = ...
two_london_capital
```

```
[ ]: grader.check("two_london_capital")
```

Express and test the proposition that two flights originating in a capital have New York as a destination.

```
[ ]: #TODO - express "there are two flights that leave from a capital and go to New␣
  ↪York"
two_capital_newyork = ...
two_capital_newyork
```

```
[ ]: grader.check("two_capital_newyork")
```

3 Quantifier scope ambiguity

Consider the sentence

- every flight leaves from a capital

Under one reading of this sentence, it is true just in case each flight leaves from a (possibly different) city that is a capital. This interpretation can be represented with the following FOL formula:

$$\forall x.(Flight(x) \implies \exists y.(Capital(y) \wedge Origin(x, y))) \tag{1}$$

However, there is another possible reading of this sentence. A less intuitive yet possible interpretation could be that we are claiming that there is a single capital from which all flights leave. This interpretation gives rise to a different FOL representation:

$$\exists y.(Capital(y) \wedge \forall x.(Flight(x) \implies Origin(x, y))) \tag{2}$$

This type of ambiguity is referred to as *quantifier scope ambiguity*, and appears almost inevitably when a sentence has multiple quantified noun phrases.

> Other scope-taking elements can also give rise to scope ambiguities. For instance, modals (like "may", "must", "can"), and negation ("not") also have scope and can engender ambiguities. For instance, the sentence "you may not go" arguably has two readings, paraphrasable as "you have permission to not go" (that is, you may stay) and "you do not have permission to go" (that is, you must stay). Examining these further kinds of ambiguity is beyond the scope (so to speak) of this lab.

It is easy to see that both representations have the same components, just structured differently. The different ordering possibilities of the quantifiers is what generates the ambiguity. In representation (1), $\forall x$ has the outer scope, while in representation (2), $\exists y$ has the outer scope. Reflecting this ordering, the two readings are sometimes referred to as the *AE reading* and the *EA reading*, respectively.

> As another example of the phenomenon, consider the old joke: In my town, a person is mugged every 15 minutes…and boy is he getting tired of it.

Express the two readings of the sentence

- every flight leaves from a capital

using the Python implementation of generalized quantifiers. Start with the AE reading.

[ ]:
```
AE_reading = ...
AE_reading
```

[ ]:
```
grader.check("ambiguity_AE")
```

Now do the same for the EA reading.

[ ]:
```
EA_reading = ...
EA_reading
```

[ ]:
```
grader.check("ambiguity_EA")
```

If you've implemented the two readings correctly, you have confirmed that the two readings of the ambiguous sentence are indeed different; they generate different answers.

4 A compositional semantics for quantifiers

Instead of manually constructing a meaning representation as we did, let's use a syntactic-semantic grammar to perform this interpretation process. We will use the following augmented grammar:

[ ]:
```
grammar_spec = """
  S -> NP VP                    :  lambda NP, VP: NP(VP)

  VP -> V NP                    :  lambda V, NP: lambda x: NP(V(x))

  V -> 'leaves' 'from'          :  lambda: lambda Subj: lambda Obj: (Subj, Obj)␣
 ↪in Origin
     | 'leave' 'from'
     | 'goes' 'to'              :  lambda: lambda Subj: lambda Obj: (Subj, Obj)␣
 ↪in Destination
     | 'go' 'to'

  S_COMP -> 'that' VP           :  lambda X: exec('raise NotImplementedError')

  NP -> DET NOM                 :  lambda DET, NOM: DET(NOM)
  NOM -> NOM S_COMP             :  lambda N, S: exec('raise NotImplementedError')
  NOM -> N                      :  lambda X: X

  NP -> 'New' 'York'            :  lambda: lambda P: P(NewYork)
      | 'Paris'                 :  lambda: lambda P: P(Paris)
      | 'Tel' 'Aviv'            :  lambda: lambda P: P(TelAviv)
      | 'London'                :  lambda: lambda P: P(London)
      | 'Boston'                :  lambda: lambda P: P(Boston)

  N -> 'flight' | 'flights'     :  lambda: lambda x: x in Flight
     | 'capital' | 'capitals'   :  lambda: lambda x: x in Capital
     | 'city' | 'cities'        :  lambda: lambda x: x in City

  DET -> 'every'                :  lambda: lambda R: lambda S: for_all(R, S)
       | 'a'                    :  lambda: lambda R: lambda S: there_exists(R, S)
       | 'two'                  :  lambda: exec('raise NotImplementedError')
"""
```

Some things to note about the grammar:

- Some of the nonterminals may be unfamiliar. In particular, the nonterminal `S_COMP` is for complementized relative clauses, phrases like "that leaves from Boston" in the sentence "every flight *that leaves from Boston* goes to New York".
- Some of the productions have no augmentation. You may recall that in this format, productions with no explicit augmentation just use the one for the preceding production. Thus the "goes to" and "go to" rules have the same augmentation.
- Some of the augmentations (those involving `S_COMP` and the determiner `two`) raise

`NotImplementedError`. We'll have you fill those in in a bit. Hold off for now.

- As described in the introductory lecture, this grammar makes use of Richard Montague's idea that noun phrase meanings should apply to verb phrase meanings, rather than the other way around. By so doing, we allow for quantifiers within noun phrases to scope over their verb phrases. But this approach necessitates changing the types of even simple noun phrases like "Boston". Instead of denoting an object, it too must denote a function from properties to truth values, for example, $\lambda P.P(Boston)$.

We parse the grammar specification to extract an NLTK grammar and the augmentation dictionary, and construct a parser for the grammar.

[ ]:
```
grammar, augmentations = xform.parse_augmented_grammar(grammar_spec,␣
 ↪globals=globals())
parser = nltk.parse.BottomUpChartParser(grammar)
```

Let's create a syntactic parse tree of the sentence we've been working with:

[ ]:
```
sentence = "every flight leaves from a capital".split()
parses = [p for p in parser.parse(sentence)]
for tree in parses:
    tree.pretty_print()
```

To carry out the semantic interpretation process, we will use the same notation as in lab 4-1, where the variable `A__some_words` will be given the meaning for the constituent with nonterminal `A` with the span "some words". We start building the semantic representation by applying the semantic composition rules from the syntactic-semantic grammar to the meanings of the subconstituents bottom up. Here are the first few steps:

[ ]:
```
DET__every = (lambda: lambda R: lambda S: for_all(R, S)) () # lambda R: lambda␣
 ↪S: for_all(R, S)
N__flight = (lambda: lambda x: x in Flight) () # lambda x: x in Flight
NOM__flight = (lambda X: X) (N__flight) # lambda x: x in Flight
NP__every_flight = (lambda DET, NN: DET(NN)) (DET__every, NOM__flight) # lambda␣
 ↪S: for_all(lambda x: x in Flight, S)
```

Now you complete the derivation, computing the meanings for each of the remaining constituents.

[ ]:
```
DET__a = ...
N__capital = ...
NOM__capital = ...
NP__a_capital = ...
V__leaves_from = ...
VP__leaves_from_a_capital = ...
S__every_flight_leaves_from_a_capital = ...
```

[ ]:
```
grader.check("complete_derivation")
```

Now that you've completed the semantic derivation, let's see what truth value the model provides for this sentence.

[ ]:
```
S__every_flight_leaves_from_a_capital
```

Question: According to the result, which of the two readings for the sentence (AE or EA) did the augmented grammar produce? Can we get the other reading with the same augmented grammar? What problems do you see resulting from this behavior?

Type your answer here, replacing this text.

5 Extending the grammar

To make the language more interesting, we'll extend the grammar to allow for relative clauses like "that leaves from Boston" or "that goes to a capital", as well as allowing for the determiner "two". Fill in the augmentations that raise `NotImplementedError` in the grammar above to allow for these.

[ ]:
```
sentence = "every flight that goes to Tel Aviv goes to a capital"
parse = list(parser.parse(sentence.split()))[0]
parse.pretty_print()
```

To test the extended grammar, we'll make use of the `interpret` function you implemented in lab 4-2. The skeleton is provided here for your convenience. Copy in your solution from that lab.

[ ]:
```
def interpret(tree, augmentations):
  ...
```

[ ]:
```
grader.check("grammar_extension")
```

Now we can demonstrate the full range of this grammar fragment by testing out a bunch of different sentences to determine which are true and which are false.

[ ]:
```
def test(sentence):
  print(sentence)
  parses = parser.parse(sentence.split())
  for parse in parses:
    print(parse, "\n==>", interpret(parse, augmentations))
```

[ ]:
```
test("every flight goes to a capital")
```

[ ]:
```
test("two flights leave from Paris")
```

[ ]:
```
test("two flights leave from London")
```

[ ]:
```
test("every flight that goes to Tel Aviv goes to a capital")
```

[ ]:
```
test("every flight that leaves from a capital goes to a capital")
```

[ ]:
```
test("every flight that goes to a capital leaves from a capital")
```

[ ]:
```
test("every flight that goes to a city leaves from a capital")
```

6 Lab debrief

Question: We're interested in any thoughts your group has about this lab so that we can improve this lab for later years, and to inform later labs for this year. Please list any issues that arose or comments you have to improve the lab. Useful things to comment on might include the following, but you're not restricted to these:

- Was the lab too long or too short?
- Were the readings appropriate for the lab?
- Was it clear (at least after you completed the lab) what the points of the exercises were?
- Are there additions or changes you think would make the lab better?

Type your answer here, replacing this text.

End of Lab 4-3

To double-check your work, the cell below will rerun all of the autograder tests.

[ ]:
```
grader.check_all()
```