# CS187 Lab 4-1: First-order logic, lambda calculus, semantic parsing

November 12, 2023

[ ]:
```
# 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/lab4-1.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()
```

In this lab, you'll use first-order logic (FOL), augmented by the lambda calculus of functions, as the formal representation of meaning for a language in the by-now-familiar air travel (ATIS) domain.

## Preparation

```
[ ]: import os
     import pprint
     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 First-order logic

Recall that FOL formulas are composed of constants, variables, predicates over them, logical operators over the predicates, and quantifiers.

A first-order model describes what the constants denote, and what values the predicates are true of. For example, a model of the flight world, might consist of the following:

- **Constants**:
  - (places) Boston, New York, Tel Aviv
  - (times) Morning, Evening
  - (flights) DL10, DL11, DL13, LY01, LY12
- **Predicates**:
  - *Flight*: DL10, DL11, DL13, LY01, LY12
  - *Origin*: (DL10, Boston), (DL11, Boston), (DL13, New York), (LY01, Tel Aviv), (LY12, New York)
  - *Destination*: (DL10, New York), (DL11, Tel Aviv), (DL13, Boston), (LY01, New York), (LY12, Tel Aviv)
  - *DepartureTime*: (DL10, Morning), (DL11, Evening), (DL13, Evening), (LY01, Evening), (LY12, Morning)
  - *ArrivalTime*: (DL10, Morning), (DL11, Morning), (DL13, Evening), (LY12, Evening)

Using this model, we can express propositions such as "DL13 is a flight" and "DL13 departs in the evening" with formulas of FOL: * Flight(DL13) * DepartureTime(DL13, Evening)

We can also combine expressions using Boolean operators to express statements like "DL13 departs from Boston in the evening": * DepartureTime(DL13, Evening) $\wedge$ Origin(DL13, Boston)

We use variables to refer to objects that are not specified. Variables are quantified to either express the existence of an object or to refer to all objects:

1. $\exists x.Flight(x) \wedge Origin(x, Boston)$
2. $\forall x.Flight(x) \implies Origin(x, Boston)$

Write a plain English description of the meaning (a "gloss") for each of the two expressions above:

[ ]:
```
# TODO -- Provide an English gloss in the form of a string for each formula
gloss1 = ...
gloss2 = ...
```

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

Determine the truth values of the above propositions under the model of the flight world given above:

[ ]:
```
# TODO -- Fill in the truth values for the two propositions as Python booleans
statement1 = ...
statement2 = ...
```

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

### 1.1 Implementing a model of the flight world

We will need a Python implementation of the flight world. Our constants will be string objects:

[ ]:
```
# Constants

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

DL10 = "DL10"
DL13 = "DL13"
LY01 = "LY01"
LY12 = "LY12"
DL11 = "DL11"

Morning = "Morning"
Evening = "Evening"
```

Our predicates will be sets of objects (for unary predicates, which express *properties*) or tuples (for binary predicates, which express *relations*). We have defined some of the predicates below.

[ ]:
```
# Predicates

# Properties
Object = {Boston, NewYork, TelAviv, DL10, DL11, DL13, LY01, LY12, Morning,␣
 ↪Evening}
Flight = {DL10, DL11, DL13, LY01, LY12}
# Relations
Origin = {
    (DL10, Boston),
    (DL11, Boston),
    (DL13, NewYork),
    (LY01, TelAviv),
    (LY12, NewYork),
}
Destination = {
    (DL10, NewYork),
    (DL11, TelAviv),
    (DL13, Boston),
    (LY01, NewYork),
    (LY12, TelAviv),
}
```

Complete the definition of the predicates by defining `DepartureTime` and `ArrivalTime` according to the flight world given above. Assume that the first element of the tuple is the flight object and the second element is the time.

[ ]:
```
# TODO
DepartureTime = ...
ArrivalTime = ...
```

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

Given this implementation, we can determine truth values of simple FOL propositions. To check whether DL10 has an origin of Boston, that is, that the proposition $Origin(DL10, Boston)$ is true, we use set membership:

[ ]:
```
(DL10, Boston) in Origin
```

Next, construct a Python expression that expresses the proposition that DL10 has an origin of Boston and a destination of New York. Name it `prop1`. We can then use it to test whether the proposition is true.

[ ]:
```
# TODO -- Construct a Python expression that tests the proposition
prop1 = ...
print(prop1)
```

[ ]:
```
grader.check("dl10_endpoints")
```

## 2 Lambda calculus

The lambda calculus is a simple notation for expressing functions. By embedding FOL in the lambda calculus, it allows us to build functions over propositions of various sorts, and build up propositions from component parts by applying those functions.

For example, $\lambda x.\, Origin(x, NewYork)$ is a function from entities to truth values, true just in case the entity is a thing (presumably a flight, though that isn't checked) whose origin is New York. This expression thus defines a *property*, the property of originating in New York.

These lambda functions can be applied to arguments. Applying the expression above to $DL10$, gives us the expression $(\lambda x.\, Origin(x, NewYork))(DL10)$. This can be simplified through the normal lambda calculus rules ($\beta$-reduction) to $Origin(DL10, NewYork)$, expressing the proposition that flight DL10 has New York as its origin. (As it turns out, that proposition is false. Nothing stops us from expressing falsehoods.)

You may recognize the similarity between this $\lambda$ notation, due to Alonzo Church, inventor of the lambda calculus, and Python's syntax for defining anonymous functions, such as `lambda x: (x, NewYork) in Origin`. (This is not a coincidence, even though Guido van Rossum is not a fan.) The happy fact that Python already embeds a proper implementation of the lambda calculus means that we do not have to implement the lambda calculus ourselves. We'll just use Python's `lambda`.

Given our implementation of the flight world, define Python `lambda` functions corresponding to the following glosses:

1. Things that have a destination of New York
2. Flights from Tel Aviv arriving in the evening

[ ]:
```
# TODO
expr1 = ...
expr2 = ...
```

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

You can apply your functions on objects from the world and verify that you get the expected result. Use the expressions you just defined to check whether

1. flight DL10 has a destination of New York; and
2. whether flight LY01 is an evening-arriving flight from Tel Aviv.

[ ]:
```
# TODO - check (1) and (2) using `expr1` and `expr2`
result1 = ...
result2 = ...
print(result1)
print(result2)
```

[ ]:
```
grader.check("simple_lambda_2")
```

**Question:** Notice that our flight world is underspecified, since flight LY01 does not have an arrival time. Is the result you got in `result2` desired? Would you suggest a different result?

*Type your answer here, replacing this text.*

# 3 Semantic parsing

Semantic parsing (or semantic interpretation or analysis) is the task of converting a natural language sentence to a semantic representation, such as first-order logic. We will use a syntax-driven approach to semantic parsing, where context-free grammar rules are augmented with semantic augmentations, providing meanings for constituents derived by each rule. The meaning of each expression will be a function of the meaning of the expression's subconstituents. The meanings and the ways to combine them are defined in a syntactic-semantic grammar. For example, given a syntactic rule $A \rightarrow B\,C$, the meaning of $A$ is a function of the meaning of $B$ and the meaning of $C$.

We want to be able to convert syntactic trees of natural language queries to FOL representations of the meanings of the associated queries. For example, given a sentence "flights from Boston to New York", we want to obtain the FOL expression $Flight(x) \wedge Origin(x, Boston) \wedge Destination(x, NewYork)$.

## 3.1 An example grammar of flights

Let us define a simple syntactic grammar for natural language queries about flights in our flight world. The grammar should cover NP expressions such as the following:

- "flights from Boston to New York"
- "flights from Tel Aviv departing in the morning"

  In working with this grammar here and below, do not change the order of the productions in the grammar, as our unit tests depend on the order.

[ ]:
```
grammar, _ = xform.parse_augmented_grammar("""
    NP -> 'flights'
    NP -> NP PP

    PP -> PP_PLACE
    PP -> PP_TIME

    PP_PLACE -> 'from' LOC
              | 'leaving' LOC
              | 'to' LOC
              | 'arriving' 'at' LOC

    PP_TIME -> 'arriving' TIME
             | 'departing' TIME
             | 'leaving' TIME

    LOC -> 'Boston'
    LOC -> 'New' 'York'
    LOC -> 'Tel' 'Aviv'

    TIME -> 'in' 'the' 'morning'
    TIME -> 'in' 'the' 'evening'
""")
```

[ ]:
```
print(grammar)
```

There are several things to notice about this grammar.

1. We use the `parse_augmented_grammar` function provided by `scripts.transform` (which we downloaded during the setup), to provide a more pleasant format for specifying grammars and augmented grammars. This format allow for blank lines and comment lines, separating alternatives on separate lines, and (although we haven't used it yet) adding semantic augmentations to the syntactic rules.
2. The grammar mixes familiar nonterminals like parts of speech and phrases (NP for noun phrases, PP for prepositional phrase) with nonterminals of a more semantic flavor, like LOC for location or TIME for time. Syntactic grammars with semantics-based nonterminals are often referred to (perhaps confusingly) as "semantic grammars". But, the grammar is still syntactic in the sense that it operates on expressions to provide their structure.
3. The function returns two values, a grammar in standard NLTK format, and a dictionary storing the augmentations, about which more later.

### 3.1.1 Augmenting the grammar with semantic composition functions

Next, we will augment this grammar with rule meanings to construct logical expressions. For each production in the grammar, we'll provide a function that takes as arguments the meanings of the subconstituents (one for each nonterminal on the right-hand side of the production) and returns the meaning of the full constituent. For those rules that have no right-hand side nonterminals, their meaning will thus be a function that takes no arguments (in Python, written as `lambda: ...`).

Before going further, make sure you fully understand this idea. The idea behind compositional semantics systems like this is that **the meaning of a constituent is determined by (is a *function* of) the meanings of its subconstituents**. We'll call these functions *composition functions*. In our implementation, we are taking this *literally*, by having the function of the subconstituent meaning be an *actual Python function*. Some perhaps surprising things follow:

1. Since the meanings might themselves be functions, these composition functions may themselves return functions.
2. Since the subconstituent meanings might themselves be functions, these composition functions may take functions as arguments.
3. Since there may be zero subconstituents of a constituent, these composition functions may take no arguments at all.

For instance, trees admitted by the syntactic production `NP -> 'flights'` have no (nontrivial) subconstituents. The production might have a composition function `lambda: lambda x: x in Flight`. Similarly, the production `TIME -> 'in' 'the' 'morning'` might have a semantic composition function `lambda: Morning`.

We add augmentations to the grammar by placing them on the same line as the syntactic rule they augment, after a colon (`:`). Here we've added a few augmentations.

[ ]:
```
grammar_spec_1 = """
    NP -> 'flights'                 : lambda: lambda x: x in Flight
    NP -> NP PP
```

```
    PP -> PP_PLACE
    PP -> PP_TIME

    PP_PLACE -> 'from' LOC
              | 'leaving' LOC
              | 'to' LOC
              | 'arriving' 'at' LOC

    PP_TIME -> 'arriving' TIME
             | 'departing' TIME
             | 'leaving' TIME

    LOC -> 'Boston'
    LOC -> 'New' 'York'
    LOC -> 'Tel' 'Aviv'

    TIME -> 'in' 'the' 'morning'    : lambda: Morning
    TIME -> 'in' 'the' 'evening'    : lambda: Evening
"""

grammar_1, augmentations_1 = xform.parse_augmented_grammar(grammar_spec_1)
```

Now it's your turn to add augmentations for the other productions that have no nonterminals on the right-hand side (just those for now).

[ ]:
```
## TODO - copy grammar_spec_1 from above and add augmentations
#         for productions with no nonterminals on the right-hand side
# Note: do not change the order of productions!
grammar_spec_2 = ...

grammar_2, augmentations_2 = xform.parse_augmented_grammar(grammar_spec_2)
```

[ ]:
```
grader.check("q_grammar_spec_2")
```

What about the rule `PP_TIME -> 'arriving' TIME`? A phrase like "arriving in the morning" ought to be associated with a Python expression that is true of things that arrive in the morning, that is, `lambda x: (x, Morning) in ArrivalTime`. We can work backwards from there.

We know that the composition function for the rule will be a function of one argument, the meaning of the `TIME` subconstituent "in the morning", which we've already determined to be `Morning`.

(Ask yourself, why isn't it `lambda: Morning`? Make sure you understand why before moving on.)

Thus, the augmentation will be of the form `lambda Time: ...`, which will end up being applied to `Morning`, so that `Time` in this particular case will end up being `Morning`. What should you fill in for the `...` so that the result of the application will be `lambda x: (x, Morning) in ArrivalTime`?

Add an augmentation for the `PP_TIME -> 'arriving' TIME` rule to the grammar based on your solution.

```
[ ]: ## TODO - copy grammar_spec_2 from above and add augmentations
     #         for production PP_TIME -> 'arriving' TIME
     # Note: do not change the order of productions!
     grammar_spec_3 = ...
     grammar_3, augmentations_3 = xform.parse_augmented_grammar(grammar_spec_3)
```

```
[ ]: grader.check("q_grammar_spec_3")
```

Once you get that augmentation in place, many others should be straightforward. Fill in augmentations for all of the `PP_PLACE -> *` and `PP_TIME -> *` productions.

And what about the productions `PP -> PP_PLACE` and `PP -> PP_TIME`? Their composition functions are simple, since the meaning of the `PP` is just the same as the meaning of its right-hand side element. What composition function can achieve that? Fill in the augmentations for those rules too.

Finally, the trickiest case is the composition function for the `NP -> NP PP` rule. Since it has two nonterminals on the right-hand side, it should be a function of two arguments, that is, something like `lambda NP, PP: ...`. This function will be applied to the meanings of the two subconstituents. The meanings for its two subconstituents, the NP and the PP, are each themselves going to be a function specifying a kind of property (like "being a flight" (`lambda x: x in Flight`) or "originating in New York" (`lambda x: (x, NewYork) in Origin`)). For the full constituent, its meaning should also be a property, namely the conjunction of the NP- and PP-provided properties ("being a flight and originating in New York" (`lambda z: z in Flight and (z, NewYork) in Origin`)).

> We used the variable `z` here just to emphasize that the variable names are arbitrary so long as they are sued consistently. Your semantic augmentations shouldn't assume that particular variables were used in the semantics for the subconstituents.

Define an appropriate augmentation that is appropriate for the `NP -> NP PP`, and add it to the grammar to complete the semantic augmentations.

> **Hint:** Consider what should the composition function for `NP -> NP PP` return. Should it be a function? If so, how many arguments should it take?

```
[ ]: ## TODO - copy grammar_spec_3 from above and add augmentations
     #         for productions PP_PLACE -> *, PP_TIME -> *,
     #         PP -> PP_PLACE, PP -> PP_TIME, and NP -> NP PP
     # Note: Do not change the order of productions!
     grammar_spec_4 = ...
     grammar_4, augmentations_4 = xform.parse_augmented_grammar(grammar_spec_4)
```

```
[ ]: grader.check("q_grammar_spec_4")
```

The `parse_augmented_grammar` function we've provided returns two values, an NLTK grammar based on the syntactic productions, and a dictionary that maps those productions onto the corresponding semantic augmentations. We can examine them individually.

```
[ ]: print(grammar_4)
```

```
[ ]: pprint.pprint(augmentations_4)
```

The semantic functions aren't much to look at. Python doesn't print out very useful information about them. But you can test them to see if they do the right thing. For instance, the semantic function for the `PP -> PP_PLACE` production ought to just be the identity function (something like `lambda PP_PLACE: PP_PLACE`).

> A typical alternative that comes up is to make explicit the function status of the `PP_PLACE` meaning by applying the `PP_PLACE` meaning to a variable, say, `x`. To generate a function of the right type for the `PP` meaning, we'd need to reabstract over `x`, resulting in the `PP` meaning `lambda x: PP_PLACE(x)`. If you took this approach, providing a composition function for the rule of the form `lambda PP_PLACE: lambda x: PP_PLACE(x)`, know that these two `PP` meaning options – `PP_PLACE` and `lambda x: PP_PLACE(x)` – are, in an appropriate sense, equivalent. In the lambda calculus, their equivalence is codified in the $\eta$ rule. You'll want to change to the simpler form for the tests below.

Let's check.

```
[ ]: pp_place_production = list(augmentations_4.keys())[2]
     pp_place_production
```

```
[ ]: pp_place_augmentation = augmentations_4[pp_place_production]
     pp_place_augmentation
```

```
[ ]: for x in [42, 'hello', True]:
       print(pp_place_augmentation(x))
```

So it sure looks like the augmentation for the `PP -> Place` rule is doing what we asked. (If not, check over your solution to the augmented grammar.)

### 3.2 Applying the grammar

With augmented grammar in hand, we can use it to get a FOL meaning representation for sentences. The procedure has two steps:

1. Run a syntactic parsing algorithm to get a syntactic tree.
2. Follow the tree derivation to obtain a meaning representation.

The first step you're familiar with from the last segment of the course; this is the role of parsing algorithms such as CKY.

The second step works by walking the tree, recursively constructing meanings for the subconstituents of a node, and then combining those subconstituent meanings by applying the production augmentation for the node to the subconstituent meanings to construct the meaning of the tree itself. This recursive method bottoms out when we come to a tree that has no nonterminal subconstituents; we just apply its production augmentation to the empty sequence of arguments.

You could implement such a function – in fact, you will in project segment 4 – but for purpose of the lab today, you'll just carry out this process by hand.

Let us walk through a semantic parsing of the expression "flights from Boston". First, we construct a syntactic parse tree for this sentence. The following function makes an `nltk` syntactic grammar from the syntactic-semantic grammar `grammar_4`:

We can use `nltk`'s `BottomUpChartParser` to parse the sentence:

[ ]:
```
parser = nltk.parse.BottomUpChartParser(grammar_4)

sentence = "flights from Boston".split()
for tree in [p for p in parser.parse(sentence)]:
    tree.pretty_print()
```

For mnemonic purposes below, we use Python variables like `A__some_words` to store the meaning for the constituent with nonterminal "A" spanning the phrase "some words".

For example, for the subtree `(NP 'flights')` at the bottom left of the tree, we'll store its meaning in the variable `NP__flights`. That meaning is constructed by applying the augmentation for the production `NP -> 'flights'` to the empty set of arguments. Hopefully, in the grammars starting with `grammar_spec_1` above, you've had the augmentation for that rule as `lambda: lambda x: x in Flight`. Applying this to the empty set of arguments, we get `(lambda: lambda x: x in Flight)()`, that is, `lambda x: x in Flight`. We record this for purposes of the lab as:

[ ]:
```
NP__flights = (lambda: lambda x: x in Flight)()
```

Now do the same for the subtree rooted in `LOC`.

[ ]:
```
#TODO
LOC__Boston = ...
```

[ ]:
```
grader.check("derivation_1")
```

Working bottom up, we consider the subtree rooted in `PP_PLACE`. What is the augmentation for the production used to form that subtree? Apply the augmentation to the meaning you've just computed for its one nonterminal-rooted subconstituent, which you've already stored as `LOC__Boston`, and call the result `PP_PLACE__from_Boston`.

[ ]:
```
#TODO
PP_PLACE__from_boston = ...
```

[ ]:
```
grader.check("derivation_2")
```

Again working bottom up, define the meaning for PP, by applying the meaning of the rule `PP -> PP_PLACE` to the meaning of the `PP_PLACE` just computed.

[ ]:
```
#TODO
PP__from_boston = ...
```

[ ]:
```
grader.check("derivation_3")
```

Finally, derive the meaning of the entire expression by applying the meaning of the rule `NP -> NP PP` to the meanings of its parts.

[ ]:
```
#TODO
NP__flights_from_boston = ...
```

[ ]:
```
grader.check("derivation_4")
```

To check that you've got the correct expression, we run through all objects in the flight world and apply the expression to them, printing out the ones that evaluate to `True`.

[ ]:
```
print([obj for obj in Object if NP__flights_from_boston(obj)])
```

If your grammar augmentations are correct and you've carried out the derivation correctly, you should get a list of just those objects that are flights from Boston, namely, DL10 and DL11.

Let's move to a more complex example. Now that you know the drill, construct the meaning for "flights from Boston to New York" using the same augmented grammar. Here's the parse tree:

[ ]:
```
sentence = "flights from Boston to New York".split()
parses = [p for p in parser.parse(sentence)]
for tree in parses:
    tree.pretty_print()
```

Create the meaning representation by walking through the tree bottom up. You can use the result you got for "flights from Boston" and compose it with the rest of the expression. The result will be stored as `NP__flights_from_boston_to_new_york`.

[ ]:
```
#TODO
...
NP__flights_from_boston_to_new_york = ...
```

[ ]:
```
grader.check("derivation_5")
```

Now run the expression against the flight world to verify you got the correct result:

[ ]:
```
print([obj for obj in Object if NP__flights_from_boston_to_new_york(obj)])
```

## 4 Scaling up

Constructing the meaning representation manually by traversing a tree is a tedious process. In practice, we would like an automated process, which, given any syntactic-semantic grammar and a tree consistent with the syntactic grammar returns a semantic representation. So far, we've seen how to use the syntactic parse tree to guide the composition of semantic representations. In project segment 4, you will automate this process by implementing such a generic semantic parser, which takes any tree with associated semantic rules, and constructs the meaning representation, doing just the work you've been doing by hand here.

## 5 Lab debrief

**Question:** We're interested in any thoughts you have 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 4-1

---

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

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