Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science
6.863J/9.611J Natural Language Processing, Spring, 2004

Laboratory 4
Semantics: word semantics & sentence semantics

Handed Out: April 21 **Due: April 30**

1. Introduction: Goals of the Laboratory

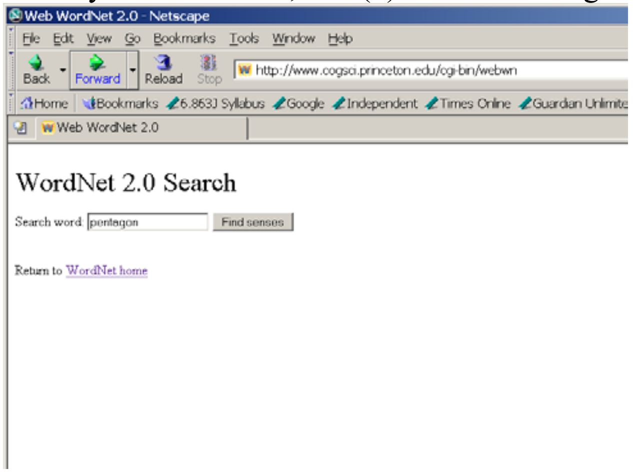
In this laboratory you will investigate two aspects of assigning meaning to sentences and how to use that information: (1) word meaning, as illustrated by the hand-constructed ontology, Wordnet; and (2) sentence meaning, as illustrated by a very simple lambda-calculus interpreter. For the first part, you will see how to use Wordnet to examine the relationships amongst word meanings, as might be used in a hypothetical word reasoning system. For the second part, you will be asked to extend. When you have completed the assignment, you should understand the principle of compositionality in semantics and how that relates to syntactic analysis. You will also know how to 'put it all together': you will have extended an English interface to a simple database.

2. Word semantics

2.1 The WordNet database (version 2.0) is available online via the following link. (You may also download it to a windows or linux machine for local use – just follow the instructions there, but this isn't necessary for the exercise.) The second link is the online browser, which is all that you need here.

`http://www.cogsci.princeton.edu/~wn/`
`http://www.cogsci.princeton.edu/cgi-bin/webwn`

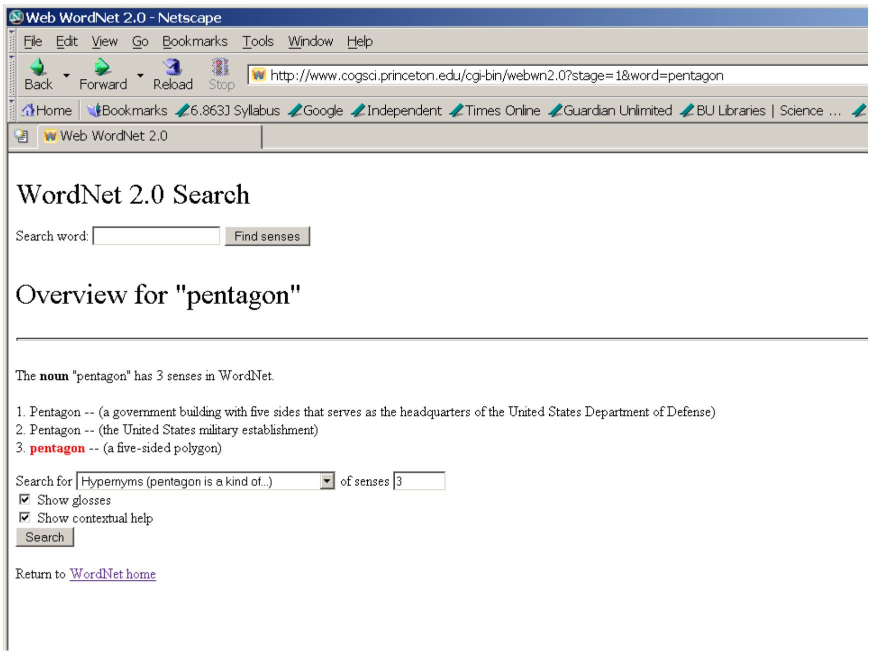
Wordnet is a hand-built directed graph of linked synonym sets or synsets. If you click on the second link, you'll pull up a web browser window that allows you to explore them. Typing 'pentagon' into the search window, Wordnet will display 3 meaning senses for 'pentagon': (1) a government building; (2) the US military establishment; and (3) a five-sided figure:


Web WordNet 2.0 - Netscape
File Edit View Go Bookmarks Tools Window Help
Back Forward Reload Stop
http://www.cogsci.princeton.edu/cgi-bin/webwn
Home Bookmarks 6.863J Syllabus Google Independent Times Online Guardian Unlimite
Web WordNet 2.0

WordNet 2.0 Search

Search word: pentagon [Find senses]

Return to WordNet home


Web WordNet 2.0 - Netscape
File Edit View Go Bookmarks Tools Window Help
Back Forward Reload Stop
Home Bookmarks 6.863J Syllabus Google Independent Times Online Guardian Unlimited BU Libraries | Science ...
Web WordNet 2.0

http://www.cogsci.princeton.edu/cgi-bin/webwn2.0?stage=1&word=pentagon

WordNet 2.0 Search

Search word: [ ] Find senses

Overview for "pentagon"

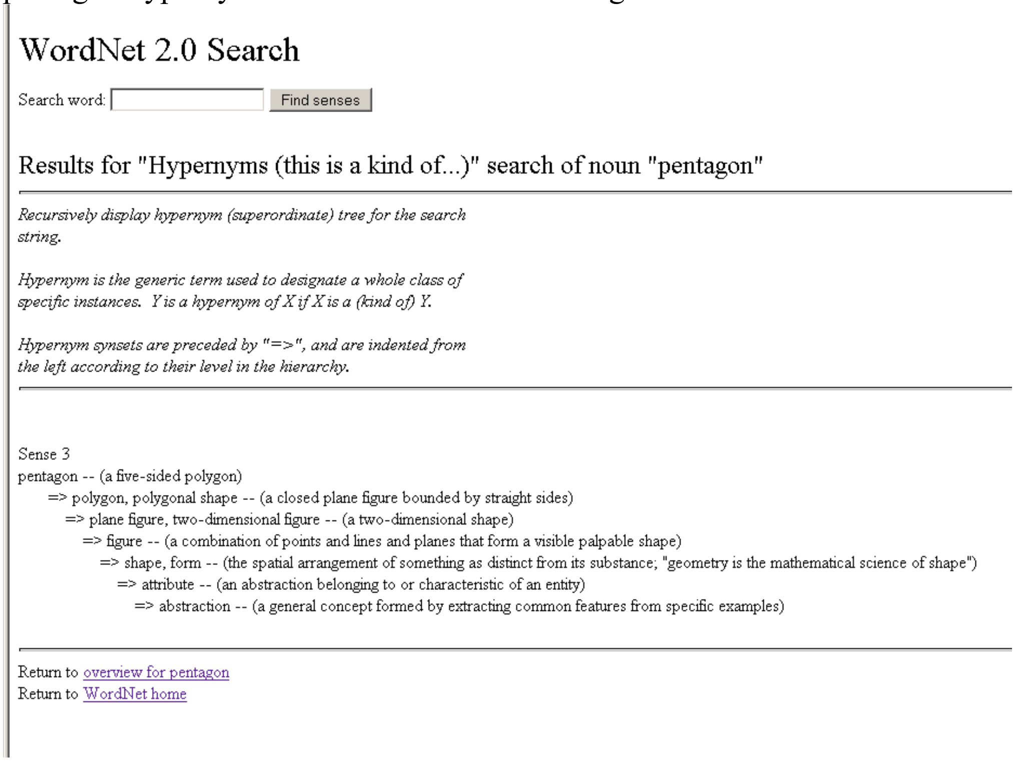
The **noun** "pentagon" has 3 senses in WordNet.

1. Pentagon -- (a government building with five sides that serves as the headquarters of the United States Department of Defense)
2. Pentagon -- (the United States military establishment)
3. **pentagon** -- (a five-sided polygon)

Search for [Hypernyms (pentagon is a kind of...)] of senses [3]
☑ Show glosses
☑ Show contextual help
Search

Return to WordNet home

Below these three word senses is a new search box with pull down menu items. We have selected a 'hypernym' search. This is a class hierarchy description based on a "this is a kind of…" relation. For example, a pentagon in the third sense is "a kind of polygon". We have also typed in '3' into the 'senses' box, meaning that we'll look for hypernyms only for the <u>third</u> sense of pentagon. We have also checked 'contextual help' which will explain the format and definitions of the terms used in the display. The pentagon hypernym search returns the following screen:



WordNet 2.0 Search

Search word: [ ] Find senses

Results for "Hypernyms (this is a kind of...)" search of noun "pentagon"

Recursively display hypernym (superordinate) tree for the search string.

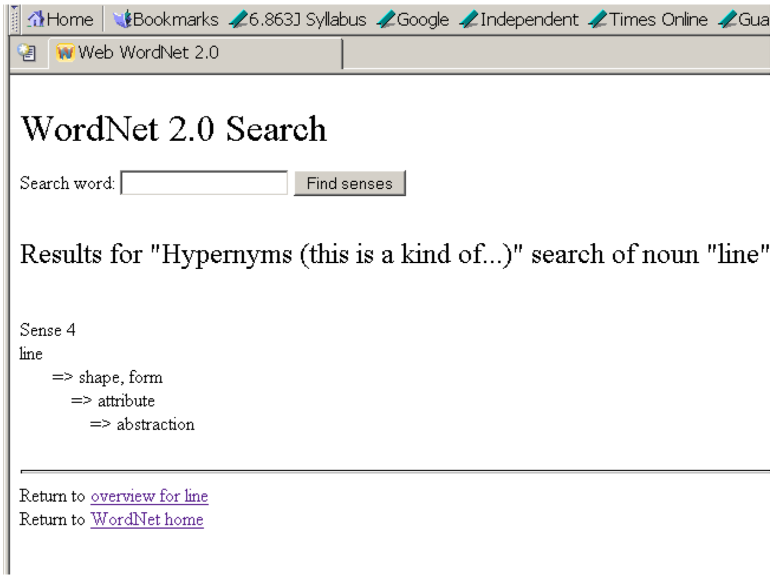
Hypernym is the generic term used to designate a whole class of specific instances. Y is a hypernym of X if X is a (kind of) Y.

Hypernym synsets are preceded by "=>", and are indented from the left according to their level in the hierarchy.

```
Sense 3
pentagon -- (a five-sided polygon)
    => polygon, polygonal shape -- (a closed plane figure bounded by straight sides)
       => plane figure, two-dimensional figure -- (a two-dimensional shape)
          => figure -- (a combination of points and lines and planes that form a visible palpable shape)
             => shape, form -- (the spatial arrangement of something as distinct from its substance; "geometry is the mathematical science of shape")
                => attribute -- (an abstraction belonging to or characteristic of an entity)
                   => abstraction -- (a general concept formed by extracting common features from specific examples)
```

Return to overview for pentagon
Return to WordNet home

The word “line” has 29 senses (!!) as a noun, and 6 as a verb. In its sense as a moving point in geometry, its hypernyms are:



WordNet 2.0 Search

Results for "Hypernyms (this is a kind of...)" search of noun "line"

Sense 4
line
    => shape, form
        => attribute
            => abstraction

Return to overview for line
Return to WordNet home

The lowest common ancestor for these two senses is the hypernym **shape, form.** A hypernym path goes up the hypernym hierarchy from the first word to a common ancestor and then down to the second word. Note that a hypernym path from a node other than the lowest common ancestor will always be equal to or longer than the hypernym path provided by the lowest common ancestor. For the example above, the hypernym path is `pentagon` to `polygon, polygonal shape,` to `plane figure, two-dimensional figure,` to `figure` to `shape` to `line`.

Question 1: Find the lowest common ancestor across all senses for each of the following word pairs. Provide the hypernym path through the lowest common ancestor in each case.

1. *English* and *Tagalog*
2. *United States* and *Cambridge*

2.2 Consider the following text:
The pilots screamed as the airplane skidded along the runway. The surface was covered with ice and the vehicle was slowly getting out of control. Finally the crew managed to stop.

We use the term coreference (verb, corefer) to specify a pair of items that denote the same object or event in a text or dialog. If coreference does not hold between an item pair, we say they are non-coreferent Noun phrases and pronouns most often corefer, e.g., *Vincent drank. He coughed on the water,* but this is not always the case. For example, in the above text, *runway* and *surface* may be said to corefer.

Question 2: Use WordNet’s hypernym function to find evidence for the following coreference or lack of coreference, and briefly explain your reasoning:

1. coreference between *runway* and *surface*
2. coreference between *airplane* and *vehicle*
3. non-coreference between *vehicle* and *runway*
4. *crew* and *pilot* (you decide from WordNet whether coreference or non-coreference should hold)

3. Sentence semantics

In this section, you will extend a very simple lambda-calculus based semantic interpreter that uses the notion of paired syntactic and semantic rules. Let's first see what it can do. To run the semantic interpreter, `add 6.863` as usual. Then, invoking **`semantic -h`** provides a list of command line options:

```
athena% cd /mit/6.863/python-semantics
athena% semantic -h
usage: semantic.py [options] [rule_file]

6.863 Semantic Framework
version 1.1 (April 14, 2004)
by Rob Speer
Distributed under the GPL. See LICENSE.TXT for information.

options:
-h, --help            show this help message and exit
-bFILE, --batch=FILE  Batch test: interpret all the lines in a file
-v, --verbose         output lots of lambda applications [default]
-q, --quiet           output only the responses [default in batch mode]
```

To enter a command loop that reads a sentence in turn, you specify a file of paired syntactic, semantic rules, and the program will respond with a prompt:

```
athena% semantic lab_rules.py
Loading semantics...
Hello.
>
```

We can now type in sentences for the interpreter to process. The system will first attempt to parse the sentence, using the Lab 3 Earley parser (with a given grammar). It then applies the semantic rules associated with each context-free rule that applied to build a node of the parse tree, to yield a representation that can either add to a simple database of facts or query the database, and construct a response. You can then type in another sentence, and so forth. A carriage return terminates the session and returns to the shell prompt.

A declarative sentence adds a statement to the database; the system responds with "OK". A question can be one of three types: a yes-no question ("Did John see Mary"), in which case the system responds Yes or No; a wh-question ("Who did John see"), in which case the system responds with an answer retrieved from its database, possibly "I don't know" ; or a wh-locative question ("Where did John see Mary"), in which case the system responds with a prepositional phrase or "I don't know." (Note that if the parser cannot succeed in parsing a sentence, the system simply returns "I don't understand"). Thus, in effect the system has a very primitive notion of a 'speech act' – the top-level 'action' that is implied by what you type in. Thus, the system knows about only three types of speech acts: a request to add something to a database; a yes-no request; and an information request. As we shall see, each corresponds to a different procedure that is executed after semantically interpreting a sentence. Here is an example of an interaction with the system:

```
> John saw Mary
OK
> Who did John see
Mary
> Did John see Mary
Yes.
> Did Mary see John
No.
> Where did John see Mary
I don't know.
> John saw Mary in the park
OK.
> Where did John see Mary
in the park
> John quizzed Mary
I don't understand.
>  <cr>
athena%
```

How the system works. To see how this all fits together, we next illustrate the intermediate semantic representations produced by the interpreter. After this, we shall describe the paired syntactic and semantic rules in detail, and follow with a step-by-step explanation of the interpreter processing the simple sentence "John saw Mary".

The intermediate representation.

As a first step, the system parses an input sentence, using the Earley parser from Lab 3. We assume here that only zero or one parse will be returned (what should we do if there is more than one parse? That's a good term project.) This parse tree is then passed to the semantic interpreter, which traverses the tree according to the lambda-calculus method we have described in class, and outputs an 'event structure': a template with the main verb at the front, and various attribute-value pairs filling in the remainder, corresponding to a kind of meaning snapshot of the sentence. For example, in the case of "John saw Mary" the system produces the following event structure, where the 'agent' is supposed to denote the initiator of the even, and the 'patient' denotes the 'affected object':

```
Event[action see, agent Object[name John], patient Object[name Mary], tense
                                                                       past]
```

The labels 'agent' and 'patient', sometimes called the thematic roles for the verb are arbitrary. The number and kind of thematic roles needed to adequately describe the semantics of sentences is a topic of much current research, called lexical semantics.
Here are the corresponding event structures for the sentences, "John gave Fido to Mary" and "Did John see Mary". The 'give' sentence illustrates a new thematic role lable, the `beneficiary`, roughly, the person/thing that "receives" the object in an event:

John gave Fido to Mary:

```
Event[action give, agent Object[name John], beneficiary Object[name Mary],
                                  patient Object[name Fido], tense past]
```

A yes-no question has the same event structure as its corresponding declarative:

Did John see Mary

```
Event[action see, agent Object[name John], patient Object[name Mary],
                                                      tense past]
```

However, event structures are not the end of the line for semantic interpretation. As mentioned above, according to what kind of 'speech act' the sentence corresponds to, the event structure will be processed in one of three different ways, a dispatch case analysis that will hand off the event structure to one of three different top-level routines: **`processSentence, ynQuestion,`** or **`whQuestion:`**

- If it's a declarative sentence, the interpreter adds a 'semantic fact' to its database:
```
def processSentence(data):
        sem.learned.add(data)
        return "Okay."
```

- If it's a yes-no question, the interpreter tries to find a match for this event structure in its current database:
```
def ynQuestion(data):
        if sem.learned.match(data): return "Yes."
        else: return "No."
```

- If it's a wh-question, the interpreter tries to find a match between the variable denoting the wh-element (e.g., 'who', 'which book', etc.) in this event structure and a previously stored event structure (the 'translate' routine is just pretty-printing for either a single name or something more complex):
```
def whQuestion(data):
vars = sem.learned.matchVar(data)
if vars is None: return "I don't know."
else:
        if len(vars.keys()) == 1:
                return translate(vars.values()[0])
        else:
                return translate(sem.learned.match(data))
```

Building the event structure: lambda-calculus semantic interpretation. To actually build the event structure, we use the rule-to-rule syntactic, semantic method and lambda calculus compositionality: with each context-free rule we associate a corresponding lambda calculus procedure. As a parse proceeds, each syntactic node is decorated with a corresponding semantic lambda form or constant. The resulting parse tree has a lambda-calculus formula attached to each node, including lexical leaves. The event structure is then constructed by 'tree walking' through the parse structure and evaluating the lambda forms, bottom-up. Let's first take a look at the paired syntactic, lambda form rules for a simple example, to illustrate their construction, and then walk through the simple example, "John saw Mary".

The syntactic, semantic rules. Below we give the paired rules from the file lab_rules.py, simplified just a bit from their actual python representation. The context-free rules should be familiar. We will assume the following context-free rules to handle a simple sentence like "John saw Mary":
```
1. Start -> S
2. S -> NP[wh -] VP
3. VP -> V+args
4. V+args -> V2[tense +] NP
5. V2 -> saw
6. NP[pro -, wh -] -> Name
7. Name -> {John, Mary}
```

We next illustrate the (syntactic rule, semantic rule) pairs for this simple grammar. The semantic rules are all procedures written in a lambda-calculus notation. The keyword 'lambda' comes first, followed by the variables that are to be bound in the lambda form separated by commas, and then a colon. After this follows the actual functional form for the lambda expression – the function that will be applied to the arguments bound by the lambda variables. (Note that as usual this form could be 'higher order' in the sense that lambda variables can also bind functions. This is the case for verbs, as shown in the 4th rule below.) For instance, the first semantic rule, associated with the context-free rule **`Start -> S`**, specifies that the function **`processSentence(s)`** will be executed to compute the 'meaning' of the entire 'root' sentence. By this point, the variable 's' will have been bound to the lambda form 'passed up' by interpreting the rest of the parse tree – namely, the whole sentence. Thus the effect of the rule is to execute **`processSentence`** with an argument whose value is the event structure frame constructed from the rest of the sentence, thus adding it to the database as desired.

Context-free Syntactic Rule	Corresponding Semantic Rule
`1. Start -> S`	**`lambda s: processSentence(s)`**
`2. S -> NP[wh -] VP`	**`lambda np, vp: vp(np)`**
`3. VP -> V+args`	**`lambda v: lambda subj: v(subj)`**
`4. V+args -> V2[tense +] NP`	**`lambda v2, np: lambda subj:   v2(subj, np)`**
`5. V2 -> saw`	**`lambda root, tense: lambda word: lambda agent, patient: C("Event", action=root, agent=agent, patient=patient,tense=tense))`**
`6. NP[pro -, wh -] -> Name`	**`lambda x: x  (e.g., identity function)`**
`7. Name -> {John, Mary}`	**`lambda name: C("Object", name=name)`**

The actual format for the rules in the python file **`lab_rules.py`** is a simple syntactic sugaring for these pairs provides as argument to the function **`sem.add`**. Since your job in this part of the lab will be to modify these rules, you will have to know about this format, so we provide the actual python examples here. The format for rules aside from lexical items is:

```
sem.add("context-free rule in quotes", lambda rule) e.g.

sem.add("Start -> S", lambda s: processSentence(s))
sem.add("S -> NP[wh -] VP", lambda np, vp: vp(np))
sem.add("VP -> V+args", lambda v: lambda subj: v(subj))
sem.add("V+args -> V2[tense +] NP", lambda v2, np: lambda
                                               subj: v2(subj, np))
sem.add("NP[pro -, wh -] -> Name", identity)
```

For lexical items (preterminals), e.g., **`V2 -> saw`**, or **`Name -> John`**, the semantic form must be a bit different. The trick is to remember what we want the procedure to return as its result. For instance, for a name, the lambda procedure builds a piece of the event structure, a category, with the keyword "object" and the value following set to whatever the name happens to be (e.g., "John"). The verb semantic lexical entries are the most complex of all, since they generally must specify a double lambda expression – a higher order function – taking first the object lambda form to construct the meaning of the VP, and then taking that resulting lambda form, and applying it to the subject NP meaning. Again, the best way to think of the entry is as a template whose values will be filled in from the result of the rest of the semantic interpretation. Consider the entry for a V2 verb like "saw". The key is the template following the "C" function name:

```
          lambda root, tense:
               lambda word: lambda agent, patient:
                  C("Event", action=root, agent=agent, patient=patient,tense=tense)
```

The template says that we want to build an event structure, with an action value set equal to whatever is filled in as the 'root' action (passed up from the verb); and the agent, patient values set to the values filled in from the semantics of the subject and object NPs.

It's probably best to see how this works by studying the trace of the interpreter in action on the sentence "John saw Mary". Unless the 'quiet' switch is set, interpreter will display which node it is evaluating, along with its associated lambda form. Let's follow the interpreter through the parse tree for "John saw Mary". The first four 'evaluating' forms below show how the interpreter moves down from the Start node of the parse tree, then S, then NP, then Name, all the way down to the bottom lexical leaf "John". At each step, you can see the lambda expression associated with that part of the parse tree:

```
>> Evaluating: Start -> S
         Text: John saw Mary
   Expression: (lambda s: processSentence(s))

>> Evaluating: S -> NP[pro -, wh -] VP
         Text: John saw Mary
   Expression: (lambda np, vp: vp(np))

>> Evaluating: NP[pro -, wh -] -> Name
         Text: John
   Expression: (lambda x: x)

>> Evaluating: Name -> 'John'
         Text: John
   Expression: (lambda name: C('Object', name=name))
```

At this point, the interpreter can finally <u>return</u> a value, since it can supply the constant "John" at the leaf node as the binding for the lambda variable 'name': **`(lambda name: C('Object', name=name)). John`** Note that the result builds (returns) a piece of the final event structure, **`Object[name John].`**

```
<< Evaluated: Name -> 'John'
        Text: John
       Value: Object[name John]

<< Evaluated: NP[pro -, wh -] -> Name
        Text: John
       Value: Object[name John]
```

This value is then also returned as the value associated with the NP, since the lambda form here is just the identity function. This is the final 'semantic value' of the NP.

Next the interpreter turns to the VP syntactic node and evaluates it. Here is the associated lambda form for the VP, the functional application VP(NP):

```
>> Evaluating: VP -> V+args
         Text: saw Mary
   Expression: (lambda v: (lambda subj: v(subj)))
```

Proceeding, we go all the way down to the V2 and then the lambda form associated with V2 verbs, in this case, “saw”:

```
>> Evaluating: V+args -> V2[tense +] NP[pro -, wh -]
         Text: saw Mary
   Expression: (lambda v2, np: (lambda subj: v2(subj, np)))
```

When the interpreter reaches the actual leaf node ‘saw’ it can finally bind this as the value of ‘action’, and then return the resulting partially-filled in event structure as the result:

```
>> Evaluating: V2[tense +] -> 'saw'
         Text: saw
   Expression: (lambda word: (lambda agent, patient: C('Event', action='see',
                                          agent=agent, patient=patient, tense='past')))

<< Evaluated: V2[tense +] -> 'saw'
        Text: saw
       Value: (lambda agent, patient: C('Event', action='see', agent=agent,
                                                 patient=patient, tense='past'))
```

Next the NP object semantics must be constructed (just like the NP subject):

```
>> Evaluating: NP[pro -, wh -] -> Name
         Text: Mary
   Expression: (lambda x: x)

>> Evaluating: Name -> 'Mary'
         Text: Mary
   Expression: (lambda name: C('Object', name=name))

<< Evaluated: Name -> 'Mary'
        Text: Mary
       Value: Object[name Mary]

<< Evaluated: NP[pro -, wh -] -> Name
        Text: Mary
       Value: Object[name Mary]
```

With the NP object evaluated, now the lambda form at the V+args node can be applied to the NP object value to obtain a new lambda expression that is the semantic value for ‘saw Mary’. In the trace, the ‘@’ sign is used to indicate the (delayed) function application of two arguments: the subject (whose value still has to be filled in when we get back to the S), and the object NP (whose value is now the piece of event structure just built). This form is in turn passed up one more level, to the VP, which wraps another lambda around it:

```
<< Evaluated: V+args -> V2[tense +] NP[pro -, wh -]
        Text: saw Mary
       Value: (lambda subj: (lambda agent, patient: C('Event', action='see',
                agent=agent, patient=patient, tense='past'))@(subj, Object[name Mary]))

<< Evaluated: VP -> V+args
        Text: saw Mary
       Value: (lambda subj: (lambda subj: (lambda agent, patient: C('Event',
                                             action='see', agent=agent,
                                                patient=patient, tense='past'))
                                                   @(subj, Object[name Mary]))@(subj))
```

In its next to last step, the interpreter returns to the S node, where it can apply this function to the subject semantic value it has already built. If we carry out the lambda applications/substitutions, we arrive at the semantic interpretation for the S node, which is an event structure:

```
<< Evaluated: S -> NP[pro -, wh -] VP
        Text: John saw Mary
       Value: Event[action see, agent Object[name John], patient Object[name Mary], tense past]
```

Finally the interpreter reaches the Start node, which carries out the execution of the top-level procedure processSentence, with the value given by the event structure. In this case, the database is updated with the event structure information and the system returns OK:

```
<< Evaluated: Start -> S
        Text: John saw Mary
       Value: 'Okay.'
```

Your turn.
Now we can turn to the actual questions for this section. You will be asked to modify or add to the rules in **`lab_rules.py`,** so please first make a copy of this file in your directory and remember to load your rules, e.g**., `mylab_rules.py`**, and not the lab rules into the semantic interpreter.
Consider the rules that allow the parser to handle sentences and questions such as “John gave Fido to Mary”, “Did John give Fido to Mary”, “Who did John give Fido to”, and “What did John give to Mary”. Verbs of type V3, like ‘give’ are required to take two arguments after the verb, and a subject, as indicated by the rule below. Note the arguments to the V3 function:

```
sem.add("V+args -> V3[tense +] NP[wh -] PP.dat",
     lambda v3, np, pp: lambda subj: v3(subj, pp, np))
```

To handle a yes-no question like “Did John give Fido to Mary” we need new syntactic rules and a semantic rule to handle the syntactic form “did” (called a “modal do”) followed by the subject NP, and then a VBAR nonterminal – the ‘do’ is an added function to be called which in this case is just the identity function.

```
sem.add("Q[wh -] -> Do_Modal NP[wh -] VBAR[fin +]",
     lambda dm, np, vbar: dm(vbar(np)))

sem.add("Do_Modal -> Modal", identity)
```

The following VBAR expansion will capture the syntactic form Verb-NP-PP (‘did give Fido to Mary’):

```
sem.add("VBAR[fin +] -> V3[tense -] NP[wh -] PP.dat",
     lambda v3, np, pp: lambda subj: v3(subj, pp, np))
```

Note that these new rules are actually a bit repetitive since it’s almost exactly the same as the rules to handle the corresponding declarative forms – a better grammar would somehow accommodate this.
But we’ll pass on this.

For our first actual exercise, note that for a Wh-question, we also need rules that introduce “gaps” that correspond to either the object Fido (the ‘patient’ thematic role) or the object of the PP, Mary (the ‘beneficiary’ thematic role), so that these may be ‘displaced’ to the front of the sentence.

So, we ask you to pick out which other two VBAR rules in the **`lab_rules`** file that do this – which two VBAR rules introduce the gaps that are needed to handle the sentences "What did John give to Mary" and "Who did John give Fido to."?

Question 3. Examine the lab rule file and please find and then write down these two VBAR rule from `lab_rules.py` that lets the system handle a Wh-questions such as these.

Question 4. Adding new syntactic/semantic rules. We'd also like the system to be able to process and answer related sentences and questions like the following:

> John gave Mary Fido
> John did give Mary Fido
> Did John give Mary Fido
> What did John give Mary

4.1 To do this, you'll have to add new syntactic/semantic rules to the existing lab rules. The first sentence above clearly means the same thing as "John gave Fido to Mary", so the event structure that is output should be the same. Currently though, the parser cannot even parse a sentence like this. So you will have to add both a new syntactic rule and a corresponding semantic rule to handle a 'double object' sentence. In fact, this should require exactly <u>one</u> new rule. It requires only a minor tweak to the **`sem.add`** rule for 'give' in declarative sentences that we have already provided. Please add this new **`sem.add`** rule, and then, via interaction with the system, demonstrate that your addition works correctly.

4.2 Next, you will need to add a syntactic/semantic rule to handle "John did give Mary Fido". You can do this by examining the rule for processing the similar sentence, "John did give Fido to Mary." Once you have this working, try the yes-no question sentence, "Did John give Mary Fido". The system should answer properly. Please explain why this works without having to add <u>any</u> more new syntactic or semantic rules.

4.3 Now consider "What did John give Mary". You should be able to add just <u>one</u> new **`sem.add`** rule to get the job done. The rule is similar to the first VBAR expansion rules for Wh-questions that you found for your answer in Question 3. Please add this single new **`sem.add`** rule, and then, via interaction with the system, demonstrate that your addition works correctly: turn in output showing that your system works (you can use the batch file option). Why don't you have to add <u>two</u> different VBAR rules, as required for Question 3?

This concludes Laboratory 4