1 Semantic Types

It is convenient to classify the semantic objects in the model according to what kind of objects they are. To this end, we will label them with **semantic types**.

Semantic types are constructed out of two primitive types, e and t. Type e is the type of individuals/entities. Type t is the type of truth-values. Everything in the domain of the model, D, is of type e. We only have two things that are of type t, namely the truth-values, 0 and 1.

In our semantic universe, everything else is a function. The types of functions always look like $\langle\sigma,\tau\rangle$, where σ is the type of the input, and τ is the type of the output.

For example, any function that takes an individual and returns a truth-value is of type $\langle e,t\rangle$, because its input is of type e and its output is of type t. So $[\![\text{smokes}]\!]^M$ is a type-$\langle e,t\rangle$ function.

There are more complex types. We can think of a function that takes an individual and returns a function of type $\langle e,t\rangle$. Such a function is of type $\langle e,\langle e,t\rangle\rangle$, because its input is of type e and its output is of type $\langle e,t\rangle$. Notice that $\langle e,\langle e,t\rangle\rangle$ is an instance of the general format $\langle\sigma,\tau\rangle$ with $\sigma = e$ and $\tau = \langle e,t\rangle$.

Please keep in mind that things like $\langle e,e,t\rangle$ or $\langle e\rangle$ are NOT semantic types. Between each pair of angled brackets $\langle\ \rangle$, you should find exactly two semantic types separated by ','.

Just for the sake of explicitness, let us define all semantic types recursively as follows.

(1) a. e is a semantic type.
  b. t is a semantic type.
  c. Whenever σ and τ are semantic types, $\langle\sigma,\tau\rangle$ is also a semantic type.
  d. Nothing else is a semantic type.

This definition yields infinitely many semantic types, and at the same time excludes things like $\langle e,e,t\rangle$ or $\langle e\rangle$. In particular, besides e, t, $\langle e,t\rangle$, $\langle e,\langle e,t\rangle\rangle$ we can also have much more complex semantic types like $\langle\langle t,e\rangle,\langle e,\langle\langle t,t\rangle,\langle e,\langle t,e\rangle\rangle\rangle\rangle\rangle$. This is probably overkill, because such complex types seem to be unnecessary for analyzing natural language expressions. But you never know. And having unused semantic types does not do any harm.

It should also be noted that it is a common practice to write et for $\langle e,t\rangle$. We will not use this abbreviation in this lecture, but we will in future lectures. You will be reminded of it then.

Now, we have defined the semantic types. We should also state what they represent. We do so by defining the 'domain' of each semantic type. For each type τ, its domain is written as D_τ.

(2) a. D_e is the set of all individuals in the model i.e. D
  b. D_t is the set of all truth-values, i.e. $\{0,1\}$
  c. For any other type $\langle\sigma,\tau\rangle$, $D_{\langle\sigma,\tau\rangle}$ is the set of functions from D_σ to D_τ

The last case (2c) can also be written as $D_{\langle\sigma,\tau\rangle} = \{ f \mid f\colon D_\sigma \to D_\tau \}$ (remember '$f\colon D_\sigma \to D_\tau$' means f is a function whose domain is D_σ and whose range is D_τ).

This is just a formal way of stating what we said above. That is, e is the semantic type of individuals, t is the semantic type of truth-values and everything else is a type of functions. For instance, $[\![\text{smokes}]\!]^M$ is a function from individuals to truth-values, or equivalently a function from D_e to D_t, so its semantic type is $\langle e,t\rangle$, or equivalently, $[\![\text{smokes}]\!]^M \in D_{\langle e,t\rangle}$.

2 Transitive Verbs

So far the Lexicon of our grammar is limited to proper names and intransitive verbs.

(3) For any model M

a. $\llbracket \text{John} \rrbracket^M$ = some individual determined by M
b. $\llbracket \text{Mary} \rrbracket^M$ = some individual determined by M
c. $\llbracket \text{smokes} \rrbracket^M = \lambda x \in D_e.\ 1$ iff x smokes in M
d. $\llbracket \text{left} \rrbracket^M = \lambda x \in D_e.\ 1$ iff x left in M

Proper names denote type-e elements, i.e. individuals and intransitive verbs denote functions of type $\langle e, t \rangle$. We often say 'Proper names are of type e, intransitive verbs are of type $\langle e, t \rangle$', but what we really mean is that their denotations are of type e and type $\langle e, t \rangle$.

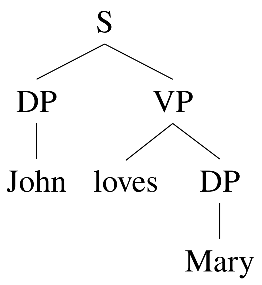
Because our Lexicon is limited, our semantic theory only captures a tiny corner of English, namely just four sentences. We will add more vocabulary items (and compositional rules, as necessary), so that ultimately we can account for everything in English. We of course cannot go that far in this course, and certainly semanticists haven't been there yet (and they haven't gotten made redundant!). But we will demonstrate how this step-by-step method works throughout the course. Today, we will add **transitive verbs** and **connectives** to our grammar.

2.1 Syntax

Transitive verbs are those verbs that take two DPs, a subject and object, e.g. 'love', 'saw', etc. It is easy to modify the syntax to include these verbs. Specifically, we just need to add two rules, which are underscored in (4).

(4) a. S → DP VP
b. DP → John | Mary
c. VP → smokes | left | <u>loves DP</u> | <u>saw DP</u>

With these new rules, we can now generate sentences like (5).

(5)


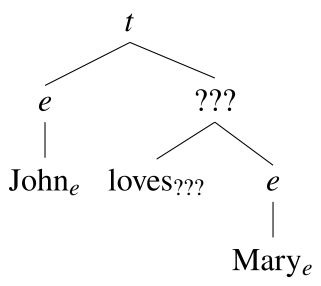
2.2 The Semantic Type of Transitive Verbs

What should be the denotations of the transitive verbs? Let's start with their semantic types. We can actually infer what their semantic types should be based on what we already know.

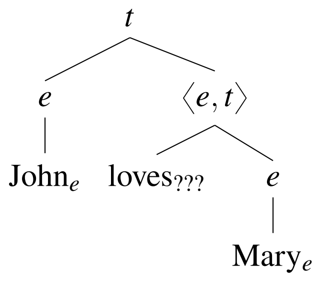
- We know that the entire sentence denotes a truth-value, i.e. it's of type t.
- We also know that $[_{\text{DP}}$ John$]$ denotes an individual, i.e. it's of type e.
- Given the Branching Node Rule, the VP $[_{\text{VP}}$ loves $[_{\text{DP}}$ Mary$]]$ should denote a function of type $\langle e, t \rangle$ (because $\llbracket [_{\text{VP}} \text{ loves } [_{\text{DP}} \text{ Mary}]] \rrbracket^M (\llbracket [_{\text{DP}} \text{ John}] \rrbracket^M)$ should be a truth-value).
- Furthermore, we know that $[_{\text{DP}}$ Mary$]$ denotes an individual, i.e. it's of type e.
- Given the Branching Node Rule, $\llbracket \text{loves} \rrbracket^M$ should be a function that $\llbracket [_{\text{DP}} \text{ Mary}] \rrbracket^M$ and returns $\llbracket [_{\text{VP}} \text{ loves } [_{\text{DP}} \text{ Mary}]] \rrbracket^M$. Then its input is of type e, and its output is of type $\langle e, t \rangle$. This means that $\llbracket \text{loves} \rrbracket^M$ itself is of type $\langle e, \langle e, t \rangle \rangle$.

Reasoning about semantic types, as demonstrated here, is a very important technique in formal semantics. We will make similar inferences about semantic types many times throughout this course.

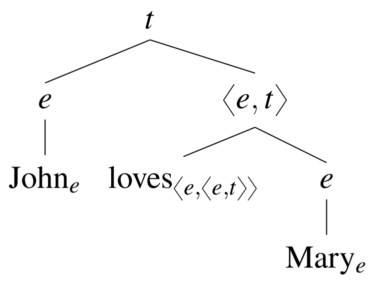
It is instructive to visualise this process by annotating each constituent in the tree diagram with its semantic type. Before the reasoning above, we knew the following:

(6)


Given the Branching Node Rule, the VP needs to denote a function of type $\langle e, t\rangle$, because its sister (= its input) is of type e and its mother (= its output) is of type t. So we have:

(7)


Similarly, $[\![\text{loves}]\!]^M$ needs to be of type $\langle e, \langle e, t\rangle\rangle$, because its sister (i.e. its input) is of type e and its mother (i.e. its output) is of type $\langle e, t\rangle$.

(8)


2.3 The Denotation of Transitive Verbs

Now we know the semantic type of $[\![\text{loves}]\!]^M$. Since it's of type $\langle e, \langle e, t\rangle\rangle$, it will look like

$$[\lambda x \in D_e.\ [\lambda y \in D_e.\ 1 \text{ iff } ???]]$$

Let's remind ourselves what x and y are. The first DP that 'loves' combines with is the object DP. In the above example, it is Mary. This is going to replace all occurrences of x in the body of the function, when λ-conversion takes place. Then the resulting function of type $\langle e, t\rangle$ will combine with the subject DP, John. So John is going to replace y in the body of the function.

If this function applies to Mary and then to John, we will get a truth-value. What kind of truth-value should we get? Well, we already know the truth-conditions of the entire sentence, so it's easy to figure this out. Specifically, if John loves Mary in the model, we should get 1, and if not, we should get 0. So what we want is:

(9) $[\lambda x \in D_e.\ [\lambda y \in D_e.\ 1 \text{ iff } y \text{ loves } x \text{ in } M]]$

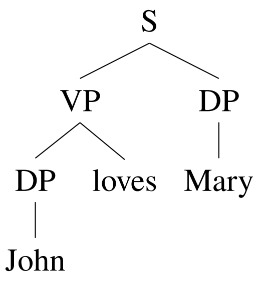
Importantly, it is '*y* loves *x*' rather than '*x* loves *y*', because *x* is going to be the denotation of the object and *y* is going to be the denotation of the subject, so *x* should be the person who is loved and *y* should be the person who loves *x*. So, (10) is not the right meaning of 'loves'.

(10) $[\lambda x \in D_e.\ [\lambda y \in D_e.\ 1 \text{ iff } x \text{ loves } y \text{ in } M]]$

This function of also of type $\langle e, \langle e, t \rangle \rangle$, so we could actually compute the meaning of (5) with it, but the predicted truth-conditions would be wrong. That is, it predicts the sentence to denote 1 iff Mary loves John (in *M*), because it says the first DP it combines with will be the person who loves the other person, and the second DP will be the person who is loved.

This demonstrates a very important point: How we construct our semantics depends crucially on the syntax. That is, if the syntactic structure of the sentence were not as represented as in (5), but something like (11) instead, then (10) would actually be the correct analysis.

(11)



However, by assumption, this is not the right syntax. It is not a purpose of this course to explain why this is not the right structure, but in order to do formal semantics, you need to be familiar with syntax. And because of this dependency, our semantic theory actually has a lot to say about what the syntax should look like.

So we will add, two transitive verbs, 'loves' and 'saw', to our Lexicon:

(12) For any model *M*,

a. $[\![\text{loves}]\!]^M = [\lambda x \in D_e.\ [\lambda y \in D_e.\ 1 \text{ iff } y \text{ loves } x \text{ in } M]]$
b. $[\![\text{saw}]\!]^M = [\lambda x \in D_e.\ [\lambda y \in D_e.\ 1 \text{ iff } y \text{ saw } x \text{ in } M]]$

As in the case of intransitive verbs, it is quite easy to generalize this to other transitive verbs.

(13) For any model *M*,

a. $[\![\text{hit}]\!]^M = [\lambda x \in D_e.\ [\lambda y \in D_e.\ 1 \text{ iff } y \text{ hits } x \text{ in } M]]$
b. $[\![\text{kick}]\!]^M = [\lambda x \in D_e.\ [\lambda y \in D_e.\ 1 \text{ iff } y \text{ kicks } x \text{ in } M]]$

To sum up, our grammar now looks like the following.

(14) *Syntax*

a. S $\rightarrow$ DP VP
b. DP $\rightarrow$ John | Mary
c. VP $\rightarrow$ smokes | left | loves DP | saw DP

(15) *Lexicon*: For any model *M*

a. $[\![\text{John}]\!]^M$ = some individual determined by *M*
b. $[\![\text{Mary}]\!]^M$ = some individual determined by *M*
c. $[\![\text{smokes}]\!]^M = \lambda x \in D_e.\ 1 \text{ iff } x \text{ smokes in } M$

d. $[\![\text{left}]\!]^M = \lambda x \in D_e.\ 1 \text{ iff } x \text{ left in } M$
e. $[\![\text{loves}]\!]^M = [\lambda x \in D_e.\ [\lambda y \in D_e.\ 1 \text{ iff } y \text{ loves } x \text{ in } M]]$
f. $[\![\text{saw}]\!]^M = [\lambda x \in D_e.\ [\lambda y \in D_e.\ 1 \text{ iff } y \text{ saw } x \text{ in } M]]$

(16) *Compositional Rules*

a. **Branching Node Rule**

For any model M, $\left[\!\!\left[\begin{array}{c} \text{A} \\ \wedge \\ \text{B}\ \ \text{C} \end{array} \right]\!\!\right]^M = [\![\text{B}]\!]^M([\![\text{C}]\!]^M)$ or $\left[\!\!\left[\begin{array}{c} \text{A} \\ \wedge \\ \text{B}\ \ \text{C} \end{array} \right]\!\!\right]^M = [\![\text{C}]\!]^M([\![\text{B}]\!]^M)$

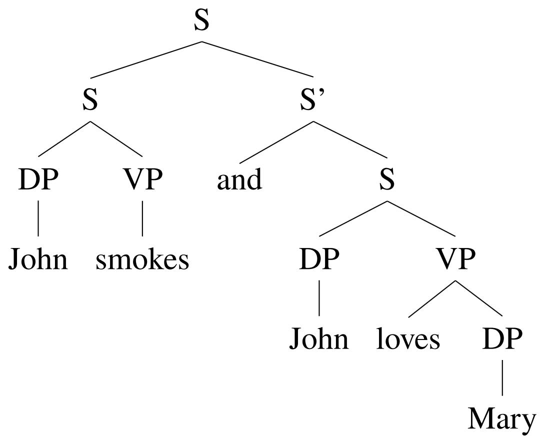
b. **Non-Branching Node Rule**

For any model M, $\left[\!\!\left[\begin{array}{c} \text{A} \\ | \\ \text{B} \end{array} \right]\!\!\right]^M = [\![\text{B}]\!]^M$

3 Connectives

Now, let us add some more items, specifically, two connectives, 'and' and 'or'. As for the syntax, we assume a structure like (17), where branching is always binary.

(17)



In order to generate such a sentence, we will add the underlined rules to the syntax.

(18) a. S → DP VP | <u>S S'</u>
b. <u>S' → and S | or S</u>
c. DP → John | Mary
d. VP → smokes | left | loves DP | saw DP

The new rule in (18a) creates a branching to a sentence S and S'. Then (18b) says S' is made up of a connective and another sentence S.

Notice that the syntax now generates infinitely many sentences, because the new rules create a loop. That is, S can branch into S and S', and this new S can itself branch off to S and S'. This process can apply to each of the newly created S's and repeat itself indefinitely. Such a loop is called **recursion**, which is exactly the mechanism that gives natural language syntax to generate infinitely many sentences, while being a finite device.

Now, assuming the above syntactic structure, let's us analyze the denotation of 'and' and 'or'. First, we need to figure out the semantic types of these connectives. As in the case of transitive verbs, we can reason about them, based on the semantics types of things we know.

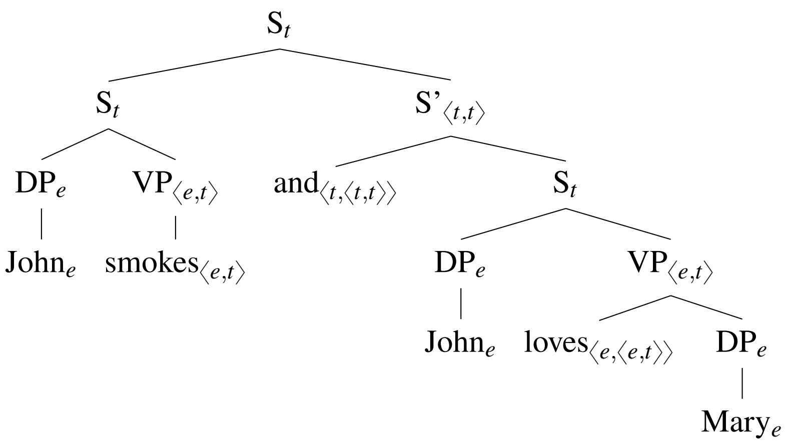
- We know that all sentences denote truth-values.
- Because truth-values are not functions, in order to combine S and S', S' needs to be a

function. In particular, it has to be of type $\langle t, t\rangle$, because it takes a truth-value denoted by its sister constituent, and returns a truth-value.

- Then, the denotation of 'and' should be a function of type $\langle t, \langle t, t\rangle\rangle$, because it takes a truth-value denoted by its sister, and returns the denotation of S', which is of type $\langle t, t\rangle$.

As before, we can annotate the tree diagram with these types to make sure it works.

(19)


S_t
S_t
$S'_{\langle t,t\rangle}$
DP_e
$VP_{\langle e,t\rangle}$
$\text{and}_{\langle t,\langle t,t\rangle\rangle}$
S_t
John_e
$\text{smokes}_{\langle e,t\rangle}$
DP_e
$VP_{\langle e,t\rangle}$
John_e
$\text{loves}_{\langle e,\langle e,t\rangle\rangle}$
DP_e
Mary_e

At all the branching nodes, one of the daughters is a function that takes the other daughter as its argument, and the resulting semantic type is the semantic type of the branching node.

Now, which type-$\langle t, \langle t, t\rangle\rangle$ function should $[\![\text{and}]\!]^M$ be? Since $[\![\text{and}]\!]^M$ is of type $\langle t, \langle t, t\rangle\rangle$ it should look like:

$$[\lambda u \in D_t.\ [\lambda v \in D_t.\ 1 \text{ iff } ???]]$$

What should ??? say? Here we base our analysis on the logical connective $\wedge$ from Propositional Logic. Recall that $p \wedge q$ is true iff both p and q are true. Then, $[\![\text{and}]\!]^M$ should simply say both truth-values it takes are 1. Thus, we can write $[\![\text{and}]\!]^M$ as follows.

(20) For any model M,
$[\![\text{and}]\!]^M = [\lambda u \in D_t.\ [\lambda v \in D_t.\ 1 \text{ iff } u = 1 \text{ and } v = 1]]$

'Or' can be analyzed in the same way. Again, we model its meaning based on the logical connective $\vee$. Recall that $p \vee q$ iff $p = 1$ or $q = 1$. Thus:

(21) For any model M,
$[\![\text{or}]\!]^M = [\lambda u \in D_t.\ [\lambda v \in D_t.\ 1 \text{ iff } u = 1 \text{ or } v = 1]]$

4 Summary

To sum up, we have added two transitive verbs and two connectives to our grammar.

(22) *Syntax*

a. S → DP VP | S S'
b. S' → and S | or S
c. DP → John | Mary
d. VP → smokes | left | loves DP | saw DP

(23) *Lexicon*: For any model M,

a. $[\![\text{John}]\!]^M$ = some individual determined by M
b. $[\![\text{Mary}]\!]^M$ = some individual determined by M

c. $[\![\text{smokes}]\!]^M = \lambda x \in D_e.\ 1 \text{ iff } x \text{ smokes in } M$

d. $[\![\text{left}]\!]^M = \lambda x \in D_e.\ 1 \text{ iff } x \text{ left in } M$

e. $[\![\text{loves}]\!]^M = [\lambda x \in D_e.\ [\lambda y \in D_e.\ 1 \text{ iff } y \text{ loves } x \text{ in } M]]$

f. $[\![\text{saw}]\!]^M = [\lambda x \in D_e.\ [\lambda y \in D_e.\ 1 \text{ iff } y \text{ saw } x \text{ in } M]]$

g. $[\![\text{and}]\!]^M = [\lambda u \in D_t.\ [\lambda v \in D_t.\ 1 \text{ iff } u = 1 \text{ and } v = 1]]$

h. $[\![\text{or}]\!]^M = [\lambda u \in D_t.\ [\lambda v \in D_t.\ 1 \text{ iff } u = 1 \text{ or } v = 1]]$

(24) *Compositional Rules*

a. **Branching Node Rule**

For any model M, $\left[\!\!\left[\begin{array}{c} \text{A} \\ \diagup\diagdown \\ \text{B} \quad \text{C} \end{array} \right]\!\!\right]^M = [\![\text{B}]\!]^M([\![\text{C}]\!]^M)$ or $\left[\!\!\left[\begin{array}{c} \text{A} \\ \diagup\diagdown \\ \text{B} \quad \text{C} \end{array} \right]\!\!\right]^M = [\![\text{C}]\!]^M([\![\text{B}]\!]^M)$

b. **Non-Branching Node Rule**

For any model M, $\left[\!\!\left[\begin{array}{c} \text{A} \\ | \\ \text{B} \end{array} \right]\!\!\right]^M = [\![\text{B}]\!]^M$

We did not add any new compositional rules this time. Next time, we will see a case that might require a new compositional rule.