3. Phrase Structure Rules and Semantic Interpretation

In this section we will deal with a small fragment of real English, “Toy English” This fragment will be extremely limited. Nevertheless it will illustrate a few important principles of semantic theory, in particular the way how syntax and semantics are related to each other and how the meaning of complex expressions can be computed from the meaning of more basic expressions.

3.1. The Syntax of Toy English

3.1.1. The Task of Syntax

We expect from the syntax of a natural language that it will specify exactly those expressions that native speakers will consider to be correct, or well-formed. This cannot be done by simply enumerating all the correct expressions of a language. The reason is that this set is infinite, for every natural language. This can be illustrated very easily by looking at the following sentences of English:

(1) Mary ran.
Mary ran and Mary ran.
Mary ran and Mary ran and Mary ran.
…

Whenever we have a correct English sentence, we can add to it *and Mary ran*, and we get another correct English sentence. (Of course these sentences are not very interesting or informative, but they are certainly grammatically correct -- and that’s the point!) It is obvious that the set of all sentences of English is infinite, and can never be fully specified by simply listing all the correct expressions.

Our example illustrates another point: We obviously followed some rule in constructing ever-new English sentences. This should remind you of the way a recursive definition works: We have basic cases, and we have induction steps that allow us to derive new cases. We could try to develop something similar for the description of correct expressions of a language:

· The basic expressions of a natural language are the words of the lexicon, like *Mary, came, car, red, and* etc. (or perhaps more precisely the morphems, as words can be complex, cf. *grandmother, great-grandmother, great-great-grandmother,* etc.).

· The induction steps are the syntactic rules of the language that tell us how expressions can be combined to form new expressions.

Notice that this way of defining the correct expressions of a language allows us to represent an infinite set with finite means, provided that

· the set of basic expressions is finite;

· the set of syntactic rules is finite.

This is of course an attractive view when we have the idea that syntax should model (parts of) the capacity of the speaker of a language. Speakers of a language certainly do not have an immediate grasp of infinite sets!

3.1.2. Phrase Structure Grammars

One popular way of specifying the set of correct expressions of a natural language is by using a phrase structure grammar. A phrase structure grammar, in a technical sense, is a mathematical object that consists of the following components:

· A finite set of labels for linguistic categories,
  like S for "sentence", NP for "noun phrase", VP for "verb phrase",
  N for "noun", DET for "article", etc.

· A finite set of basic expressions (the "lexicon").

· A finite set of phrase structure rules that have the following form:

  $X \rightarrow Y_1\ Y_2 \ldots Y_n$,

  where X is a label for a linguistic category,
  and $Y_1\ Y_2 \ldots Y_n$ are either labels for linguistic categories or basic expressions.

· One label for a linguistic category as a starting symbol, usually S.

The idea behind phrase structure rules is the following: A rule like X → Y Z allows us to replace or rewrite a label X by the sequence of symbols Y and Z, where Y and Z are either labels or words. By this we express that the sequence Y Z is considered to be of the syntactic category X.

Let us specify the syntax of Toy English, which is a very small, but yet infinite fragment of real English.

3.1.3. Phrase Structure Rules for Toy English

We specify the syntax of English as follows:

· Labels for categories: S, NP, VP, V, Coor, Mod

· Basic expressions: *Leopold, Stephen, Molly, sleeps, snores, loves, knows, and, or, it-is-not-the-case-that*

· Phrase-Structure rules:

(2) a. S → NP VP
  b. VP → V NP
  c. S → S Coor S
  d. S → Mod S
  e. NP → *Leopold*, NP → *Stephen*, NP → *Molly*
   (we abbreviate these three rules by NP → {*Leopold*, *Stephen*, *Molly*})
  f. VP → {*sleeps*, *snores*}
  g. V → {*loves*, *knows*}
  h. Coor → {*and*, *or*}
  i. Mod → *it-is-not-the-case-that*

· Starting symbol: S

We can derive well-formed expressions of toy English by starting with the symbol S and replacing it step by step, following the phrase-structure rules, until we arrive at a string of basic expressions. Here is an example:

(3)	S	(Starting symbol)
	S Coor S	(Rule c)
	NP VP Coor S	(Rule a)
	NP VP Coor NP VP	(Rule a)
	NP VP Coor NP V NP	(Rule b)
	Leopold VP Coor NP V NP	(Rule e)
	Leopold snores Coor NP V NP	(Rule f)
	Leopold snores and NP V NP	(Rule h)
	Leopold snores and Molly V NP	(Rule e)
	Leopold snores and Molly loves NP	(Rule g)
	Leopold snores and Molly loves Stephen	(Rule e)

The resulting string, *Leopold snores and Molly loves Stephen*, is a correct expression of toy English: It consists only of basic expressions, and it was generated from the starting symbol S by recursive applications of the phrase-structure rules.

We applied the rules to derive our example in a particular order. But it is obvious that nothing hinges on the particular order that we followed. If you have time to spare, you can try to figure out how many different derivations of the sentence there are!

One interesting fact about the syntax of toy English is that it will generate infinitely many sentences. This is due to rule (c),

S → S Coor S,

which allows us to rewrite the symbol "S" by the sequence "S Coor S", in which S occurs again. This allows derivations that start as follows

S
S Coor S
S Coor S Coor S
S Coor S Coor S Coor S
…

and may lead to sentences like

Leopold snores and Molly sleeps or Leopold loves Molly and Stephen sleeps or Leopold snores and it-is-not-the-case-that Molly snores.

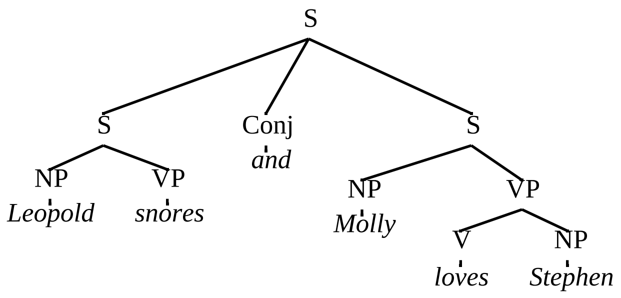
Obviously, there is no maximal length for correct sentences of toy English. So our little grammar of toy English captures one fact about real English, namely that it generates infinitely many sentences.

3.1.4. Phrase Structure Trees

The above derivations of correct expressions of toy English are rather clumsy and cover aspects that we are not really interested in, namely the order in which the phrase structure rules are applied.

A more perspicuous way to specify the derivation of a string with a phrase structure grammar is by a tree, which gives us just the essential information.

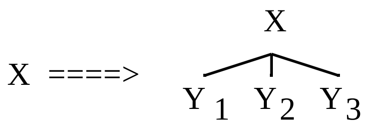
(4)



It is obvious how to construct a tree: We start by writing down the starting symbol, S, and proceed in the following way:

Whenever we have in a derivation a rule X → Y_1 Y_2 … Y_n, we add to the node that represents X the branches that lead to the nodes Y_1, Y_2,… Y_n below X (in that order).

(5) A rule X → Y_1 Y_2 Y_3 allows us to perform the following replacement:



Ideally, the phrase structure grammar of a language should lead to phrase structure trees that show syntactic groupings or **constituents** that are independently motivated by syntactic tests. For example, we analyze a sentence with a transitive verb like *Molly loves Stephen* in a way where *loves* and *Stephen* form a phrase, a VP, and not in a way where *Molly* and *loves* form a phrase. There are syntactic reasons for that; for example, there are pronominal forms like *so* for a VP, but not for a purported constituent that consists of a subject NP and a transitive verb:

(6) *Molly loves Stephen, and so does Leopold*.
(i.e., Leopold loves Stephen as well; it cannot mean that Molly loves Leopold as well!)

The reasons for favouring particular phrase structure analyses over other ones are particular constituent tests; they will be dealt with in the syntax course.

Frequently, trees will be specified in a linear way, with the help of brackets and indices. The above tree could have been given in the following way as well:

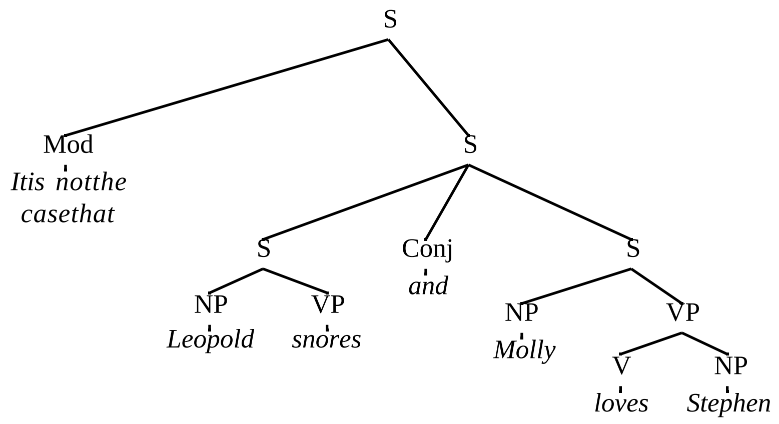
(7) $[_S[_S[_{NP}$*Leopold*$][_{VP}$*snores*$]][_{Conj}$*and*$][_S[_{NP}$*Molly*$][_{VP}[_V$*loves*$][_{NP}$*Stephen*$]]]]$

Often we represent only those parts in brackets and with labels that we are currently interested in.

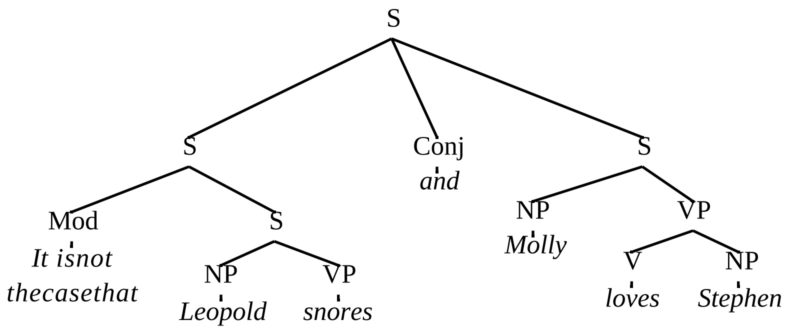
3.1.5. Syntactic Ambiguity

As we have seen, trees abstract away from spurious differences in which we derive a particular expression of our language. However, we still may derive one and the same string in a number of different ways that shows up in different trees. Example:

(8) a.



b.



In the first case, we have first replaced “S” by “Mod S”, following rule (d). In the second case, we have first replaced “S” by “S Coor S”, following rule (c). The result is the same expression (i.e., the same sequence of words).

There are semantic reasons to distinguish between the two derivations. We can argue that the first denies both that Leopold snores and that Molly loves Stephen, whereas the second asserts that Leopold does not snore, and that Molly loves Stephen. This is clearly something different; for example, if Leopold snores, the second reading is certainly false, whereas the first one still could be true.

The two derivations of our sentence are instances of a **structural ambiguity**. In the case of structural ambiguity we have that one and the same expression can have more than one syntactic structure, and typically it will have different meanings under these syntactic structures.

3.2. Semantics of Toy English: Names, Intransitive Verbs and Sentence Connectors

Let us come back to our main goal, the compositional semantics for a small fragment of English. So far we have described the correct expressions of toy English, using phrase-structure rules. Now let us turn to semantics. The purpose of this section is not to give a final account of the semantics of English constructions (or even a serious candidate for that), but to illustrate how a semantic theory for complex expressions of a language looks like.

The underlying principle that we follow is the compositionality principle: It must be possible to compute the meaning of a complex expression by knowing the meanings of the parts and the syntactic rules by which the parts are combined. This leads to the following procedure:

- We will assign meanings to the basic expressions of Toy English.
- We will specify, for each syntactic rule, a corresponding semantic rule.

One consequence of the second rule is that we cannot simply assign meanings to the expressions of Toy English. This is because one and the same expression can have different syntactic structures, and have different meanings under those syntactic structures.

3.2.1. NPs, Sentences, and Intransitive Verbs

The only NPs we have in Toy English are names. (We will discuss NPs like *a woman* or *every woman* later). Names have **saturated** meanings, they denote individuals (in this case, the heroes of James Joyce's "Ulysses"). Hence we have:

(9) a. ⟦*Leopold*⟧ = Leopold Bloom (abbreviated: LB)
b. ⟦*Molly*⟧ = Molly Bloom (MB)
c. ⟦*Stephen*⟧ = Stephen Dedalus (SD)

The set of all persons and things we care to talk about is called the **universe of discourse**. I will use the D_e for that, where D stands for "domain" and e for "entity". D_e contains at least LB, MB and SD as distinct entities, and perhaps more.

(10) D_e = the domain of **e**ntities = the universe of discourse.

How should we model the meanings of sentences? At the beginning of this course we have argued for a **truth-conditional** approach to sentence meanings, in which we take the truth conditions of a sentence to be crucial for the determination of its meanings. This led us to the idea that the meaning of a sentence is the set of all possible circumstances, or possible worlds, in which this sentence is true. For example,

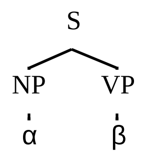
(11) ⟦$[_S[_{NP}$*Leopold*$][_{VP}$*sleeps*$]]$⟧ = {w| w is a possible world and LB sleeps in w}

However, for the time being we will be content with a simpler representation of sentence meanings. We will say that **a sentence has a truth value as its meaning**: It means "true" (or "1"), if it is true in the real world, and "false" (or "0"), if it is false. We will return to the more sophisticated version using possible worlds later. For the time being we will work under the following assumption:

(12) If Φ is a sentence, then ⟦Φ⟧ = 1 if Φ is true, and ⟦Φ⟧ = 0 if φ is false.

Now let us turn to intransitive verbs. Intransitive verbs β, like *snores*, can be used in the following syntactic configuration:

(13)



What form should the meaning of an intransitive verb have, then? Interestingly, we can come up with an answer by following the **compositionality principle**. This is because we already know the type of meaning an NP has -- it is an individual --, and we know the type of meaning a sentence has -- it is a truth value, 0 or 1. The compositionality principle tells us that the meaning of a complex expression results from the meaning of its immediate syntactic parts and the way they are combined. So we will best assume that the meaning of a VP is a function that, when combined with the meaning of a NP, will give us the meaning of a sentence.

Hence the meaning of a VP will be a function from individuals (NP meanings) to truth values (S meanings). In particular, we have the following:

(14)a. $[\![\textit{sleeps}]\!] = \lambda x \quad D_e[\text{x sleeps}]$,
the function from individuals to truth values such that for all individuals y,
$\lambda x[\text{x sleeps}](y) = [\text{y sleeps}] = 1$ if y sleeps, and = 0 if y does not sleep.
b. $[\![\textit{snores}]\!] = \lambda x \quad D_e[\text{x snores}]$.

Notice that VP meanings are **unsaturated**. They need an argument, here, an entity like a person, to form a sentence meaning, which again is a saturated meaning.

So far we have given the meanings of the NPs and the meanings of the intransitive verbs (or simple VPs) of Toy English. NP meanings are individuals, and VP meanings are functions from individuals to truth values. Now it is easy to give the meaning rule that corresponds to the syntactic rule S → NP VP:

(15) $[\![[_S[_{NP}\alpha][_{VP}\beta]]]\!] = [\![[_{VP}\beta]]\!]([\![[_{NP}\alpha]]\!])$

That is, whenever we have a syntactic structure that is a S consisting of an NP α and a VP β, then the meaning of this S can be computed by applying the meaning of the VP β (which is a function from individuals to truth values) to the meaning of the NP α (which is an individual). The result will be a truth value, of course.

We still need a rule that gives us the meaning of a tree that consists of one syntactic label dominating a word, like [NP*Leopold*], or [VP*snores*]. We simply assume that in this case the meaning of the tree will be the same as the meaning of the word:

(16)If α is a word and X is a syntactic label, then $[\![[_X\alpha]]\!] = [\![\alpha]\!]$

Let us now illustrate how the meaning of a complex sentence can be derived. Let us assume that Leopold snores. Then we can compute the meaning of the syntactic tree

$$[_S[_{NP}\textit{Leopold}][_{VP}\textit{snores}]]$$

as follows:

(17)a.	$[\![[_S[_{NP}\textit{Leopold}][_{VP}\textit{snores}]]]\!]$	
b.	$= [\![[_{VP}\textit{snores}]]\!]([\![[_{NP}\textit{Leopold}]]\!])$	(Rule (15))
c.	$= [\![\textit{snores}]\!]([\![\textit{Leopold}]\!])$	(Rule (16))
d.	$= \lambda x \quad D_e[\text{x snores}](\text{LB})$	(Rule (9), (14))
e.	= [LB snores]	(Function Application)
f.	= 1	(if LB sleeps)
	= 0	(if he doesn't).

We see that our rules indeed allow us to compute the meaning of a sentence.

3.2.2. Transitive verbs

Let’s turn to transitive verbs. A transitive verb, like *loves*, expresses a relation between two entities. So we may treat it as a set of pairs, as before in our informal discussion. We could do so easily when we assume a syntactic rule that combines subject and object NP at the same time with a transitive verb, such as:

(18) S $\rightarrow$ NP V NP

However, there are syntactic arguments that tell us that the relation between a transitive verb and its object is closer than the relation between a transitive verb and its subject. The syntactic rules reflect that by having two rules instead of one:

(19) a. S $\rightarrow$ NP VP
b. VP $\rightarrow$ V NP

We have already developed the semantic rule that corresponds to the first syntactic rule. Ideally, it should be applicable to the case of complex VPs as well. Now let us turn to the semantic rule that corresponds to the second syntactic rule. The question here is: What is the form of the meaning for transitive verbs, V? Again, we find the answer when we follow the compositionality principle. Now we have the following situation:

(20)
```
          VP√
         /   \
       V?     NP√
```

We already know that NP-meanings are individuals, and VP-meanings are functions from individuals to truth values. We also assume compositionality, that is, the VP-meaning should be computable from the V-meaning and the NP-meaning. The most obvious way of analyzing V mean

ings then is as functions from individuals to VP-meanings. For example:

(21) $[\![$*loves*$]\!] = \lambda y \quad D_e[\lambda x \quad D_e[x \text{ loves } y]]$,
the function that maps every individual u to $\lambda y \quad D_e[\lambda x \quad D_e[x \text{ loves } y]](u)$
(which is $\lambda x \quad D_e[x \text{ loves } u]$),
that is, a function that maps every individual v to $\lambda x \quad D_e[x \text{ loves } u](v)$
(which is [v loves u], that is 1 if v loves u, and 0 if v does not love u).

Note that $\lambda y \quad D_e[\lambda x \quad D_e[x \text{ loves } y]]$ is an unsaturated meaning that, when combined with an individual u, yields another unsaturated meaning, $\lambda x \quad D_e[x \text{ loves } u]$. Only when $\lambda x \quad D_e[x \text{ loves } u]$ is combined with an individual do we get a saturated meaning.

What we have done is to express a function with two arguments (the first analysis of *loves*) by a function with one argument that yields another function with one argument as value.

We now can formulate the semantic rule that corresponds to the syntactic rule VP $\rightarrow$ V NP:

(22) $[\![[_{VP}[_V\alpha][_{NP}\beta]]]\!] = [\![[_V\alpha]]\!]([\![[_{NP}\beta]]\!])$

Analysis of an example; assume that Leopold loves Molly.

(23)a. $\llbracket[_S[_{NP}\textit{Leopold}][_{VP}[_V\textit{loves}][_{NP}\textit{Molly}]]]\rrbracket =$
b. $= \llbracket[_{VP}[_V\textit{loves}][_{NP}\textit{Molly}]]\rrbracket(\llbracket[_{NP}\textit{Leopold})\rrbracket)$ (Rule (15))
c. $= \llbracket[_V\textit{loves}]\rrbracket(\llbracket[_{NP}\textit{Molly}]\rrbracket)(\llbracket[_{NP}\textit{Leopold})\rrbracket)$ (Rule (22))
d. $= \llbracket\textit{loves}\rrbracket(\llbracket\textit{Molly}\rrbracket)(\llbracket\textit{Leopold}\rrbracket)$ (Rule (16))
e. $= \lambda y \;\; D_e[\lambda x \;\; D_e[\text{x loves y}]](\text{MB})(\text{LB})$ (Rules (9), (21))
f. $= \lambda x \;\; D_e[\text{x loves MB}](\text{LB})$ (Function application)
g. = [LB loves MB] (Function application)
h. = 1 (if LB loves MB)
= 0 (if LB doesn’t love MB)

3.2.3. Modularity and Type-Driven Interpretation

So far we have two separate semantic rules for the combination of an NP with a VP, and of a V with an NP, namely (15) and (22). They are repeated here:

(24)a. $\llbracket[_S[_{NP}\alpha][_{VP}\beta]]\rrbracket = \llbracket[_{VP}\beta]\rrbracket(\llbracket[_{NP}\alpha]\rrbracket)$
b. $\llbracket[_{VP}[_V\alpha][_{NP}\beta]]\rrbracket = \llbracket[_V\alpha]\rrbracket(\llbracket[_{NP}\beta]\rrbracket)$

In both cases, the meaning of one subexpression is applied to the meaning of the other one, and we have specified explicitly which is to be applied to which. But this is perhaps not necessary. For example, we could not have applied the meaning of a name to the meaning of a VP, because this is not the way how these meanings can be combined. So it is perhaps more general to have just one general rule for branching nodes:

(25)General rule for branching structures:
$\llbracket[\alpha\ \beta]\rrbracket = \llbracket\alpha\rrbracket(\llbracket\beta\rrbracket)$ or $\llbracket\beta\rrbracket(\llbracket\alpha\rrbracket)$, whichever makes sense, that is, whichever is possible.

This rule does not mention any specific categories like NP or VP or V. It applies to all binary structures [α β]. It leaves it up to the semantics whether we apply the meaning of the first constituent to the meaning of the second, or vice versa. We call the general nature of the meaning of an expression — whether it stands for an entity, a truth value, a function from entities to truth values etc. — the semantic **type** of this expression. An interpretation rule like (25) is called **type-driven** because it depends on the types of the parts. For example, we can combine a function from entities to truth values and an entity only in one way, by applying the first to the second.

For non-branching structures we simply assume that the interpretation of the mother node is the interpretation of the daughter node:

(26)General rule for non-branching structures: $\llbracket[\alpha]\rrbracket = \llbracket\alpha\rrbracket$

Furthermore, we can now simplify our phrase structure rules a bit. They should not make a distinction between intransitive verbs like *snores* and transitive verbs like *loves*. After all, this appears to be a genuinely semantic difference. So we have the following rules:

(2) j. VP → V (NP)
k. V → {*sleeps, snores, loves, knows*}

The rule format for (j) is an abbreviation for the two rules VP → V and VP → V NP; that is, the parentheses around NP indicate that this node is **optional**.

It is evident that we can derive the sentences like *Leopold snores* or *Leopold loves Molly* with rule (25). But perhaps this rule gives us too much. For example, we can now derive things like **Leopold loves* and, even worse, ***Leopold snores Molly*. But even though we can derive this syntactically, what we get doesn't make much semantic sense:

(27)a. $\llbracket[_S[_{NP}\ \textit{Leopold}]\ [_{VP}\ [_V\ \textit{loves}]]]\rrbracket$
b. = $[[_{VP}\ [_V\ \textit{loves}]]]([[_{NP}\ \textit{Leopold}]])$
or $[[_{NP}\ \textit{Leopold}]]([[_{VP}\ [_V\ \textit{loves}]]])$, whichever makes sense (Rule (25))
c. = $[\textit{loves}]([\textit{Leopold}])$
or $[\textit{Leopold}]([\textit{loves}])$, whichever makes sense (Rule (26))
d. = $\lambda y\ \ D_e[\lambda x\ \ D_e[x \text{ loves } y](LB)$
or $LB(\lambda y\ \ D_e[\lambda x\ \ D_e[x \text{ loves } y])$, whichever makes sense (Rules (9), (21))
e. = $\lambda y\ \ D_e[\lambda x\ \ D_e[x \text{ loves } y](LB)$ (only this makes sense)
f. = $\lambda x\ \ D_e[x \text{ loves } LB]$ (function application)

We arrive at a meaning a meaning allright, but it is not a sentence meaning — there is still an argument that needs to be satisfied.

(28)a. $\llbracket[_S[_{NP}\ \textit{Leopold}]\ [_{VP}\ [_V\ \textit{snores}]\ [_{NP}\ \textit{Molly}]]]\rrbracket$

b. = $\llbracket[_{VP}\ [_V\ \textit{snores}]\ [_{NP}\ \textit{Molly}]]\rrbracket(\llbracket[_{NP}\ \textit{Leopold}]\rrbracket)$
or $\llbracket[_{NP}\ \textit{Leopold}]\rrbracket(\llbracket[_{VP}\ [_V\ \textit{snores}]\ [_{NP}\ \textit{Molly}]]\rrbracket)$, whichever makes sense

c. = $\llbracket[_{VP}\ [_V\ \textit{snores}]]\rrbracket(\llbracket[_{NP}\ \textit{Molly}]\rrbracket)(\llbracket[_{NP}\ \textit{Leopold}]\rrbracket)$
or $\llbracket[_{NP}\ \textit{Molly}]\rrbracket(\llbracket[_{VP}\ [_V\ \textit{snores}]]\rrbracket)(\llbracket[_{NP}\ \textit{Leopold}]\rrbracket)$
or $\llbracket[_{NP}\ \textit{Leopold}]\rrbracket(\llbracket[_{VP}\ [_V\ \textit{snores}]]\rrbracket(\llbracket[_{NP}\ \textit{Molly}]\rrbracket)$
or $\llbracket[_{NP}\ \textit{Leopold}]\rrbracket(\llbracket[_{VP}\ [_V\ \textit{snores}]]\rrbracket)(\llbracket[_{NP}\ \textit{Leopold}]\rrbracket)$, whichever makes sense

d. = $\lambda x\ \ D_e[x \text{ snores}](MB)(LB)$, = $[MB \text{ snores}](LB)$
or $MB(\lambda x\ \ D_e[x \text{ snores}])(LB)$
or $LB(\lambda x\ \ D_e[x \text{ snores}](MB))$, = $LB([MB \text{ snores}])$
or $LB(MB(\lambda x\ \ D_e[x \text{ snores}]))$, whichever makes sense

The problem with this is that nothing makes sense at all. We can reduce the first and the third formula a bit, but in none of the four cases do we end up with a sentence meaning. In fact, all four cases are totally meaningless.

We see here that even though the syntax does allow for the formation of expressions like *Leopold loves*, or *Leopold snores Molly*, such sentences cannot be interpreted. In the way how we sketched the collaboration of syntax and semantics in this subsection, they together determine whether an expression is well-formed or not. An expression may be ill-formed because it cannot be generated by the syntax, or it may be ill-formed because it cannot be interpreted.

We can think of syntax and semantics as representing two different **modules** that jointly determine what is a good sentence of English. Syntax tells us, for example, that **loves Leopold Molly* is no good (it would be good, with different words of course, in a so-called VSO language, like Irish). Semantics tells us that, for example, **Leopold snores Molly* is no good. A theoretical framework that distributes the load of explanation to different subtheories that are themselves simple and general is called **modular**. We will come back to the issue of modularity later.

The particular phenomenon that a verb like *snores* allows only for one NP, whereas a verb like *loves* needs two has been called the **theta criterion**. It states that for each semantic argument of a verb there must be exactly one syntactic constituent that fills that argument, and vice versa, for each syntactic constituent that stands in a certain syntactic configuration with a verb there must be one semantic argument of the verb that it fills.

Type-driven interpretation, as introduced in this section, is simpler than a **rule-by-rule** interpretation, as used in the previous sections. In particular, one global semantic rule replace several construction-specific rules. Furthermore, type-driven interpretation is more **restrictive**. For example, in a rule-by-rule interpretation we could allow for “active” VPs and “passive” VPs that differ just in the way how the NPs are mapped to the semantic arguments:

(29)a. $S \rightarrow NP\ VP_{active}$,
$S \rightarrow NP\ VP_{passive}$,
$VP_{active} \rightarrow V\ NP$,
$VP_{passive} \rightarrow V\ NP$

b. $[\![[_{VPactive}\ [_V\ \alpha]\ [_{NP}\ \beta]]]\!] = [\![[_V\ \alpha]]\!]([\![[_{NP}\ \beta]]\!])$

c. $[\![[_{VPpassive}\ [_V\ \alpha]\ [_{NP}\ \beta]]\!] = \lambda x\ \ D_e[[\![[_V\ \alpha]]\!](x)([\![\beta]\!])]$

Notice that the interpretation rule (c) leads to a switch of argument positions so that a sentence like *Leopold* $[_{VPpassive}$ *loves Molly*] would be true if Leopold is loved by Molly. However, we could achieve this only by a very rule-specific intepretation, which is not an option when we adhere to type-driven interpretation. In that framework we will have to analyze passive sentences differently.

3.2.4. Sentential Operators

Let us now discuss the semantic side of rules having to do with the sentential operators *and*, *or* and *it-is-not-the-case-that.*

First, let’s have a look at the **negation** modifier *it-is-not-the-case-that*, which is a **one-place sentence operator**. It’s semantic contribution is obviously that it reverses the truth value of a sentence: If the original sentence is true, the resulting sentence will be false, and vice versa. Hence we should assume that the meaning of this modifier is a function that maps 1 to 0 and 0 to 1. This function can be rendered as a set of pairs, { 1,0 , 0,1 }. Can we also render it as a lambda term? Yes. Let us use “D_t” for the set of truth values, {0, 1}.

(30) D_t = the domain of **t**ruth values = {0, 1}

Then we can render negation by the following lambda term:

(31) $[\![$*it is not the case that*$]\!] = \lambda t\ \ D_t[1—t]$

When we apply this function to the argument 1, we get 1—1 = 0 as a result, and when we apply it to 0, we get 1—0 = 1. (So it turns out that it was quite useful to work with 1 and 0 as truth values...)

With the general interpretation rule for branching structures (25) we don’t need any specific rule for sentences consisting of a negation and another sentence. As an example, we have the following derivation:

(32) a. $\llbracket[_S[_{Mod}\textit{it-is-not-the-case-that}][_S\textit{Molly snores}]]\rrbracket$

b. $= \llbracket[_{Mod}\textit{it-is-not-the-case-that}]\rrbracket(\llbracket[_S\textit{Molly snores}]\rrbracket)$ (Rule (25); only this order makes sense)

c. $= \llbracket\textit{it-is-not-the-case-that}\rrbracket(\llbracket\textit{snores}\rrbracket(\llbracket\textit{Molly}\rrbracket))$ (Rules (25), (26))

d. $= \lambda t \;\; D_t[1—t](\lambda x[x \text{ snores}](MB))$ (lexical rules)

e. $= \lambda t \;\; D_t[1—t]([MB \text{ snores}])$ (function application)

f. $= [1 — [MB \text{ snores}]]$ (function application)

g. $= 1$, if $[MB \text{ snores}] = 0$,
$= 0$, if $[MB \text{ snores}] = 1$

We get the right result: The sentence has the truthvalue 0 if Molly snores, and the truthvalue 1 if Molly doesn’t snore.

Let us now consider the **two-place sentence operators,** the so-called **conjunction** *and* and the **disjunction** *or*. Recall that we have assumed the following syntactic rule:

(33) $S \rightarrow S \text{ Coor } S$

This creates a structure that is not a simple binary branching, and our interpretation rule (25) consequently will not apply. We can introduce interpretation rules for such structures in a number of ways. One is to assume the following rule:

(34) $\llbracket[_S\,[_S\,\alpha]\,[_{Coor}\,\beta]\,[_S\,\gamma]]\rrbracket = \llbracket[_{Coor}\,\beta]\rrbracket(\,\llbracket[_S\,\alpha]\rrbracket,\,\llbracket[_S\,\gamma]\rrbracket\,)$

This says that the meaning of sentence $[_S\,[_S\,\alpha]\,[_{Coor}\,\beta]\,[_S\,\gamma]]$ consists in applying the meaning of the coordination $[_{Coor}\,\beta]$ to the pair of the meanings of the two conjuncts, $[_S\,\alpha]$ and $[_S\,\gamma]$. What then is the meaning of a coordination? Take *and*. When *and* combines two sentences, it should give us a true sentence if both sentences are true, else it gives us a false sentence. We can express this by using the function MIN that gives us the smallest number of a set. In particular, we have $\text{MIN}(\{1, 1\}) = 1$ and $\text{MIN}(\{1, 0\}) = \text{MIN}(\{0, 1\}) = \text{MIN}(\{0, 0\}) = 0$.

(35) $\llbracket\textit{and}\rrbracket = \lambda\;\; t, t' \;\; [\text{MIN}(\{t, t'\})]$

This is a function from pairs of truth values t, t' to a truth value, namely, the minimum of t and t′. We then have derivations like the following:

(36) a. $\llbracket[_S[_S\textit{Molly sleeps}]\,[_{Conj}\textit{and}]\,[_S\textit{Leopold snores}]]\rrbracket$

b. $= \llbracket[_{Conj}\textit{and}]\rrbracket(\,\llbracket[_S\textit{Molly sleeps}]\rrbracket,\,\llbracket[_S\textit{Leopold snores}]\rrbracket\,)$ (Rule (35))

c. $= \lambda\;\; t, t' \;\; [\text{MIN}(\{t, t'\})](\,[MB \text{ sleeps}], [LB \text{ snores}]\,)$ (lexical rules, application)

e. $= \text{MIN}(\{[MB \text{ sleeps}], [LB \text{ snores}]\})$ (application)

f. $= 1$, if $[MB \text{ snores}] = [LB \text{ snores}] = 1$,
$= 0$, else.

This gives us the correct result. However, we again have assumed a specific semantic rule for a syntactic rule, (35). Also, in the interpretation of a coordination like *and* we have used a rather complex notation for variables, namely, the pair notation t, t'. We have seen with transitive verbs that we can reduce a function from pairs to functions from simple entities. We can now do the same here and assume the following meaning for *and*:

(37) $\llbracket\textit{and}\rrbracket = \lambda t \;\; D_t[\lambda t' \;\; D_t[\text{MIN}(\{t, t'\})]]$

Here, the meaning of *and* is a function that first takes one sentence meaning, and yields a function that then takes the other sentence meaning.

The general semantic rule (25) still does not fit for structures with three daughter nodes. Can we generalize it for the case at hand? The basic idea is that categories should combine in whatever way is semantically possible. This suggests a rule like the following:

(38) $\llbracket[\alpha\ \beta\ \gamma]\rrbracket = \llbracket\alpha\rrbracket(\llbracket\beta\rrbracket)(\llbracket\gamma\rrbracket)$ or $\llbracket\beta\rrbracket(\llbracket\alpha\rrbracket)(\llbracket\gamma\rrbracket)$ or $\llbracket\gamma\rrbracket(\llbracket\alpha\rrbracket)(\llbracket\beta\rrbracket)$, whichever makes sense.

There might be even more possible combinations, like $\llbracket\alpha\rrbracket(\llbracket\gamma\rrbracket)(\llbracket\beta\rrbracket)$, which I did not list here to keep things simple.

There is one potential problem with this generalization: It may be that more than one combination is possible. This is certainly the case for coordinations. For example, the meaning of *and* could first be combined with the first sentence, and then with the second, or vice versa. Luckily, this doesn't matter — we will always get the same semantic result, for MIN({t, t′}) is of course the same as MIN({t′, t}). So we can work with the generalized semantic rule indicated in (34).

Consider the following example derivation:

(39) a. ⟦[$_{S}$[$_{S}$*Molly sleeps*] [$_{Conj}$*and*] [$_{S}$*Leopold snores*]]⟧
b. = ⟦[$_{Conj}$*and*]⟧(⟦[$_{S}$*Molly sleeps*]⟧)(⟦[$_{S}$*Leopold snores*]⟧) (Rule (38))
c. = ⟦*and*⟧(⟦*sleeps*⟧(⟦*Molly*⟧))(⟦*snores*⟧(⟦*Leopold*⟧)) (Rules (25), (26))
d. = $\lambda t\ \ D_t[\lambda t'\ \ D_t[MIN(\{t, t'\})]]$([MB sleeps])([LB snores]) (lexical rules, application)
e. = MIN({[MB sleeps], [LB snores]}) (application)
f. = 1, if [MB snores] = [LB snores] = 1,
= 0, else.

What about disjunction, *or*? There are two possible approaches. One is to say that two sentences conjoined with *or* are true if either one sentence is true, but not both. We indeed use *or* often in this way:

(40) a. John (either) went to Paris or he went to London.
b. You may have candy, or you may have ice cream.

However, notice that this meaning element can easily be cancelled:

(41) a. John (either) went to Paris or he went to London, perhaps he went to both cities.
b. You may have mustard or mayonnaise on your sandwich (of course, you can have both).

Hence it presumably is just an **implicature**, as discussed in section #. And the real meaning of *or* should give us a true sentence if both sub-sentences are true. This allows us to model its meaning with the help of the function MAX, which gives us the maximal number of a set of numbers. In particular, we have MAX({1, 1}) = MAX({0, 1}) = MAX({1, 0}) = 1, and MAX({0, 0}) = 0.

(42) ⟦*or*⟧ = $\lambda t\ \ D_t\lambda t'\ \ D_t[MAX(\{t, t'\})]$

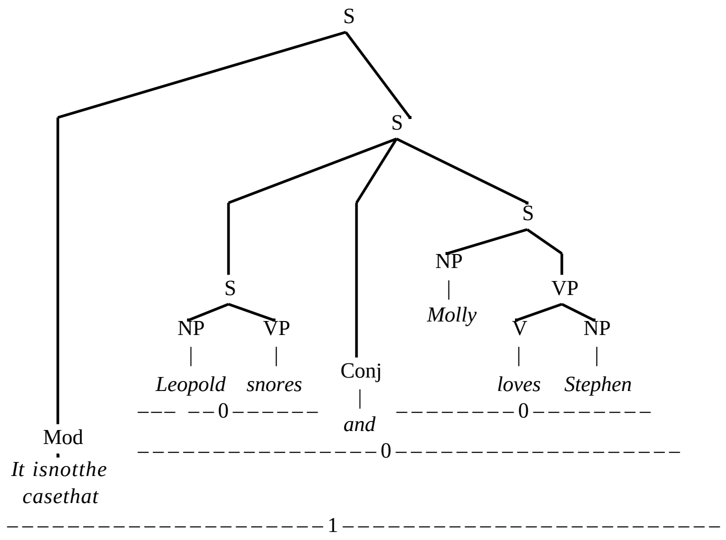
And we have derivations like the following:

(43) a. ⟦[S[S*Molly sleeps*] [Conj*or*] [S*Leopold snores*]]⟧
b. = ⟦[Conj*or*]⟧(⟦[S*Molly sleeps*]⟧)(⟦[S*Leopold snores*]⟧) (Rule (38))
c. = ⟦*or*⟧(⟦*sleeps*⟧(⟦*Molly*⟧))(⟦*snores*⟧(⟦*Leopold*⟧)) (Rules (25), (26))
d. = $\lambda t \; D_t[\lambda t' \; D_t[MAX(\{t, t'\})]]$([MB sleeps])([LB snores]) (lexical rules, application)
e. = MAX({[MB sleeps], [LB snores]}) (function application)
f. = 0, if [MB snores] = [LB snores] = 0,
= 1, else.

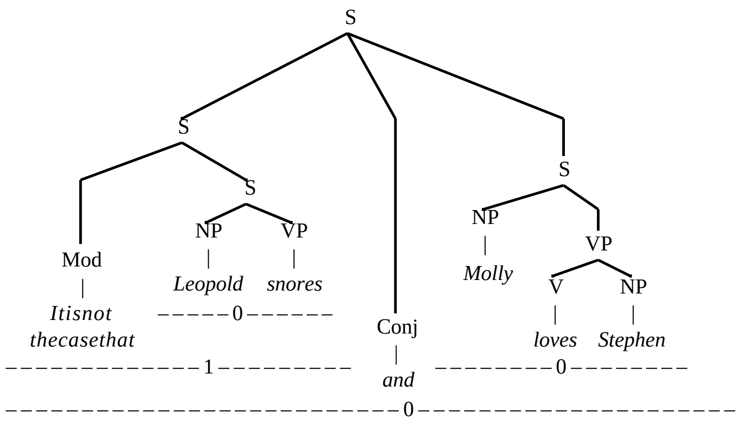
3.2.5. Syntactic and Semantic Ambiguity

We have seen that the syntax of Toy English allows for structural ambiguities. Now, it is interesting to notice that a structurally ambiguous sentence may be true on one reading, but false on another one. Take our previous example, and assume that Molly does not sleep (hence the meaning of *Molly sleeps* is 0), and that Molly does not love Stephen (hence the meaning of *Molly loves Stephen* is 0). We get the following truth values for the two structures:

(44)



(45)



We see that under the first syntactic analysis, the sentence *it is not the case that Molly sleeps or Leopold snores* is true, and under the other analysis it is false.

One consequence of this is that we cannot just give the interpretation of the string of words *it is not the case that Molly sleeps or Leopold snores*, but we have to specify the meaning of this sentence with respect to a syntactic analysis. This is the reason why our meaning rules give meanings to syntactic trees instead of sequences of syntactic labels or sequences of words.

3.2.6. Flexible Types for Negation

The rule that we have proposed for negation is quite clumsy. Instead of (46.a) we normally express things as in (b).

(46)a. It is not the case that Molly snores.
b. Molly doesn't snore.

From a syntactic point, it is remarkable that in (46.b) the verb is an auxiliary, *does*, with a negative suffix, *n't*. This is a peculiarity of modern English; Shakespeare would have said, *Molly snores not* (and that's still the way how things are in Dutch and German). If we take syntax seriously, then we cannot anlyze negation as a sentence modifier. The negative operator *doesn't* rather forms a constituent with the non-finite verb phrase:

(47)a. [Molly [doesn't snore]]
b. [Molly [doesn't [love Leopold]]]

Let us add the following syntactic rules to our rule system:

(2) l. $\text{VP} \rightarrow \text{Aux VP}_{\text{inf}}$
m. $\text{VP}_{\text{inf}} \rightarrow \text{V}_{\text{inf}}$ (NP)
n. $\text{V}_{\text{inf}} \rightarrow$ {*sleep, snore, love, know*}
o. Aux → *doesn't*

Obviously the rules (2.j), VP → V (NP), and (2.m), are versions of the same general pattern, and our grammar should capture that. But for the time being we assume that they are two distinct rules. Also, the lexical rules (2.n) correspond to the lexical rules for the finite verbs. The lexical rules for infinite verbs are just the same as the ones for finite verbs.

All we have to add now is the interpretation rule for *doesn't*: It should be a function from VP-meanings to VP-meanings. How should we specify the domain of these functions? We could, of course, introduce a new symbol, perhaps D_{VP}. But notice that VP-meanings are functions from entities in D_e to entities in D_t, and it would be good to indicate that in the name for this domain. We commonly call this domain D_{et}, to indicate that it comprises functions from D_e to D_t.

(48) D_{et} = the domain of functions from D_e to D_t.

I will use variables like P, Q for this domain. We then have the following interpretation rule for *doesn't*:

(49) ⟦*doesn't*⟧ = $\lambda P\ \ D_{et}\lambda x\ \ D_e[1—P(x)]$

This function takes a VP-meaning P and gives a function from entities x to truth values. In particular, it assigns the truth value of 1 minus P applied to x. To see how this works, consider the following example:

(50) a. ⟦[$_S$ [$_{NP}$ *Molly*] [$_{VP}$ [$_{Aux}$ *doesn't*] [$_{VPinf}$ [$_{Vinf}$ *snore*]]]]⟧
b. = ⟦[$_{Aux}$ *doesn't*] [$_{VPinf}$ [$_{Vinf}$ *snore*]]]]⟧(⟦[$_{NP}$ *Molly*]⟧) (Rule (25))
c. = ⟦*doesn't*⟧(⟦*snore*⟧)(⟦*Molly*⟧) (Rules (25), (26))
d. = $\lambda P\ \ D_{et}\lambda x\ \ D_e[1—P(x)](\lambda x\ \ D_e[x \text{ snores}])(MB)$ (lexical rules)
e. = $\lambda P\ \ D_{et}\lambda y\ \ D_e[1—P(y)](\lambda x\ \ D_e[x \text{ snores}])(MB)$ (variable renaming)
f. = $\lambda y\ \ D_e[1—\lambda x\ \ D_e[x \text{ snores}](y)](MB)$ (function application)
g. = $\lambda y\ \ D_e[1—[y \text{ snores}]](MB)$ (function application)
h. = [1—[MB snores]] (function application)
i. = 1, if [MB snores] = 0,
= 0, if [MB snores] = 1

We get the same result as before, but we arrived at that in a slighty different way. a way that is truer to the ordinary syntax of English. In step (e) I have renamed the variable x in the meaning of *doesn't* to y so that we do not mix it up with the variable x in the meaning of *snores*. While this was not absolutely necessary in this case, it is wise to do so to avoid confusion.