— LECTURE NOTES —

Introduction to Computational Semantics

First of all, many people would contest the use of the term *semantics* to describe what we will be dealing with here. Our goal of course is to create natural language processing software that behaves as if it understood the natural language expressions it encounters or produces. "Understanding" depends on grasping the meaning of the expression, so, since semantics is the study of meaning, clearly it is important to what we are about.

The confusion stems from the following two distinct points of view on what semantics really is. First off we have the tradition most closely associated with Tarski, which is pervasive in mathematical logic, for example, as well as in natural language semantics (particularly "model theoretic semantics") and programming language semantics (particularly "denotational semantics"). The essential idea is that expressions in a formal language—endowed with a deductive calculus capable of generating new formulas from old—is to be connected to a system of abstract structures whose behavior we may only grasp intuitively, but where at least we have reliable intuitions about what follows from what, and what conclusions are valid.

Now, of course, we cannot actually talk about these abstract structures without naming them, drawing diagrams or writing other formulas that reliably communicate what we mean to say about them. So right away, just in order to talk about valid conclusions about the natural numbers, say, we wind up presenting what are in fact a pair of formula languages and a mapping between them. The fact that one of them is "formal" while the other is "ad hoc" is easy to lose sight of when the ad hoc formula language is quite systematic and reflects centuries of successful practice.

Here is where the second point of view comes in. This is the point of view that arises in particular from cognitivist linguistic theories, like Chomsky's Transformational Generative Grammar. The central claim, and this is a claim about cognition in general, not just about language, is that cognition in humans is a system of levels of representation with mappings between them and rules of inference within any given level. At the highest level these representations encode what we think and feel about the world around us and even about the worlds that exist only in our imaginations (worlds that are counter to fact—"What if?" worlds, worlds that are grounded in our false beliefs, etc.—as well as worlds where there are integers, truth, beauty and a really perfect pizza, which may exist without actually having any concrete referent in the "real world").

On this view, "semantics" is really a matter of translation from one system of representations—representations of syntactic structure—to another system or representations, which support inferences about the things that we are talking about. Now, these systems of representations are formal. Their main property is that we can do computations with them, transforming them automatically into other representations at the same level, which we may then translate back into syntactic structures and so ultimately into sound or writing. Now, this level of semantics, one of syntax's next door neighbors, is presumably where we encode much of the information that makes our behavior meaningful. Certainly there is nothing about syntax to suggest that we can generate meaningful responses to questions, for example, simply by computations directly on syntactic structures. So we are justified at least to some extent in calling these higher level representational systems Semantics. But, of course, formal "semantic" representations have, in the earlier model-theoretic sense, their own semantics in terms of things some of which at least have an existence quite independent of our own cognition.

And so we have two notions of "semantics", which I will from now on term *model theoretic* and *cognitive*. It should be clear that our concern here is with some sort of simulation of the cognitive variant. We actually want to construct systems of data structures and relations over them that will allow us to write programs that respond to natural language instructions. (We will not make much of an effort to get them to answer back in natural language: we don't really have time to cover Natural Language Generation in even cursory detail.)

1 Information We will assume from here on out, without nearly sufficient comment, that the semantics of natural language expressions are to be given in terms of some concrete *information state* of the receiver (which will always be the computer in our examples). Declarative sentences, in particular, will be interpreted in terms of their *information update potential*. That is, we will assume that they are uttered for the purpose of updating the information state of the computer. Questions will have the obvious interpretation of queries of the computer's information state. Imperatives will have the obvious interpretation in terms of commands to the computer to do something (presumably something the computer can actually do: "Create world peace!" should probably elicit the all purpose Prolog response "`no`" . . .)

2 Attention The first problem that we will sweep under the rug (actually, I guess it is already far too late for it to be the first) is the following. If semantics is really just about information, we run into the following problem very quickly. The amount of information the computer has just keeps growing and growing and growing. As it does, the problem of responding to queries, of inferring implicit knowledge from explicitly given facts, and so on becomes progressively harder.

When all is said and done there really isn't that much Syntax in any one given language. But the amount of Semantics there is is as far as we know unbounded. Certainly, in order to respond intelligently to a wide range of situations the computer must be endowed with a considerable encyclopedia of information. The difficulty of knowing what information is *relevant* and may be *useful* is an extremely important one.

So any realistic application of any scope is going to need to be able to manage *attention* as well as raw information. There are aspects of grammar which help with this. Word order is often a great clue to what knowledge is assumed to be old and what is supposed to be new. The definite/indefinite distinction in determiners is another classic way of managing "information packaging". In an application with speech input, intonation can be invaluable in determining what parts of the sentence are meant to wake up background knowledge already possessed by the hearer and what parts are meant to provide new information.

As noted, all of this will be swept under the rug. We will find quite enough to worry about just taking our first steps.

3 The Dictionary and the Encyclopedia One more thing that we will sweep under the rug, for the most part, is the problem of *real-world knowledge*. In fact, we will not worry too much about lexical knowledge at all. Typically we will use some typographical device to indicate the meanings of words, e.g., DOG is the meaning of *dog*. This does not contribute very much to the theory of lexical meaning. However, if we adopt the assumption (the principle of *compositionality*) that the meaning of a whole is dependent solely on the meanings of the parts and the manner of their combination, then we will find that we can study very many aspects of sentence meanings even if we must at a certain point shrug and say, "Well, the meaning of that is clearly PAST(BARK(DOG))." A more sophisticated account of lexical meaning is offered by the lexical database *WordNet*, but even here the attempt is just to identify word-forms with meanings where "meanings" are little more than codes for sets of synonymous word-forms. That is, the relationship of the word *dog* to other words is illustrated in some detail, but there is still a long stretch between that information and the information you would need in order to construct an automatic dog-recognizer machine.

4 Types The foundation of the theory of sentence meaning, which we will concentrate on here, is the theory of *types*. In ordinary usage "type" means something like "feature" or "property". But in our more restricted usage we mean something a little more focused, and a little more like the use of the term in computer programming. The type of a word is meant to convey information about the ways in which it can combine with other words. The idea is that, given the types of the words in a sentence, and the way they combine syntactically, we should be able to work out a structure that would tell us exactly what to do with the actual word meanings, once someone was kind enough to supply us with those.

1 Difference Structures for Partial Types

The following example uses difference structures to represent incomplete types. Here the difference structures may have more than one extracted variable. Indeed, the number of extracted variables varies quite a bit depending primarily on the particular choice of lexical head for any given phrase. The essential logic of this example is simple and follows the form of the corresponding context free grammar very closely. A string is of category `s` if the first part of it is of category `np` and the second part of it is of category `vp`. The meaning of the sentence is then just the meaning of the verb phrase (a predicate with a single free argument) applied to the meaning of the noun phrase (which takes the place of its argument). Thus, if the meaning of the noun phrase is `waldo` and the meaning of the verb phrase is `chase(Agent,wilma)`, then solving `waldo = Agent` will result in the sentence-meaning `chase(waldo,wilma)`. Similarly, if the meaning of the verb phrase is `laugh(Agent)`, and the meaning of the noun phrase is `wilma`, then solving `wilma = Agent` will result in the sentence-meaning `laugh(wilma)`.

```
analyze( String, Meaning ) :-
        s( Meaning, String, [] ).

s( Proposition ) -->
        np( Subject ),
        vp( [Argument], Predicate ),
        {Argument = Subject},
        {Proposition = Predicate}.

vp( Args, Predicate ) -->
        verb( [Arg1|Args], Predicate ),
        np( Arg1 ).
vp( Args, Predicate ) -->
        verb( Args, Predicate ).

verb( [Theme, Agent], chase(Theme,Agent) ) --> [chased].
verb( [Agent], laugh(Agent) ) --> [laughed].

np( waldo, [waldo|L], L ).
np( wilma, [wilma|L], L ).
```

Figure 1: A simple example of semantic analysis.

5 **What Russell Noticed** This picture of sentence semantics is appealingly simple, but it has been known for almost a century that it is inadequate. For example, consider the sentence

Every villain was chasing Waldo

as compared to

A villain was chasing Waldo

Leaving out of consideration problems of tense and aspect, we can assign the first sentence a logical form like

$$(\forall x)[\mathit{villain}(x) \Rightarrow \mathit{chase}(x, \mathit{waldo})]$$

while the second could be assigned a logical form like

$$(\exists x)[\mathit{villain(x)} \wedge \mathit{chase}(x, \mathit{waldo})]$$

There are really two problems here. First off, the constituency is all wrong, from the point of view of the syntax. The structure of these formulas associates the noun meaning with the verb meaning more closely than it does with the determiner meaning. Secondly, the syntactic determiner determines not only the semantic quantifier ($\forall$ vs. $\exists$) but also the connective ($\Rightarrow$ vs. $\wedge$).

Also, though some natural language determiners have nice first order definitions (*every*, *some*, *none*, etc.), some have meanings which are provably not first order. A classic example is *most*. To see that there is a problem here, try to translate a sentence like *Most villains chase Waldo* into a logical form with a meaning like *most things are X* for any choice of *X*. It cannot mean the same thing: the meaning of *most* is crucially tied to the size of the set of *villains*, not the set of all things. For example, "most things are such that if they are villains then they chase Waldo," does not have the right meaning: very few things are villains in the first place, compared to the number of things that there are. So more than half of the villains still account for much less than most of the things that there are.

6 What Montague Noticed To cut a long store short, the way out of this is to turn the "natural" interpretation of a simple declarative sentence on its head. Instead of thinking of the VP as a function applying to the subject NP, we think of the subject NP as a function applying to the VP. We can always do this. For example, we can think of the number 2 as a function that maps the function $x + 3$ to 5, and the function x^2 to 4 and so forth. In the case of natural language NPs, since the VP denotes a property that individual things can have, an NP denotes a higher order function that maps the set of possible properties that things can have to the set of just those properties that the NP actually has. In the simplest case, the case of proper names, we now think of *Waldo* as not simply an individual in the domain of discourse but as the set of properties that that individual has. *Most villains* can now be assigned a formal meaning: it is true of a property P just in case the set of villains that have P is larger than the set of villains that don't.

7 Working Out the Higher Types Tightening the screws a little, if we think of properties (like what VPs denote) as functions from individuals to truth values then we can assign them to the type $U \to \mathbf{2}$, where $\mathbf{2}$ is just a conventional name for the set $\{true, false\}$ of truth values and U is just our universe of individual things. Recall that a function to $\mathbf{2}$ is equivalent to a predicate, and in turn is equivalent to a set. The function is called the *characteristic function* of that set. So, a higher order function that picked out a particular set of properties would have the type $(U \to \mathbf{2}) \to \mathbf{2}$.

Now hold on to your seats! Determiners take noun denotations as their arguments and return an entity of this higher type $(U \to \mathbf{2}) \to \mathbf{2}$. Now, nouns denote the same things as verb phrases: properties of individuals: $U \to \mathbf{2}$. So we derive the type of determiners as

$$(U \to \mathbf{2}) \to ((U \to \mathbf{2}) \to \mathbf{2}) \tag{1}$$

So *every* would denote a function which maps *student* to the set of properties that every student has, while it would map *teacher* to the set of properties characteristic of teachers.

8 Conservativity We could equally well think of a determiner as taking a pair of properties (N, V) and returning *true* just in case $N \subseteq V$. *Most* would map (N, V) to *true* just in case $|N \cap V| > (|N|/2)$: the number of Ns who are V is greater than half the number of Ns. It turns out to be a general fact about natural language determiners (with only very few exceptions) that their meanings depend only on the size of N and the size of $N \cap V$. So the only individuals in V that we care about are the individuals that are also in N. This property of determiners is sometimes called *conservativity*, sometimes the *lives-on* property.

9 Transitive Clauses Actually, even the monster type for determiners given in (1) is somewhat inadequate in the face of transitive verbs. The proper generalization about NPs is that they are *arity reducers*. If we represent one place predicates as P^1, two-place predicates as P^2 and so on then NPs denote functions from P^n to P^{n-1}. The trouble is, for $n > 1$ there is more than one way in which the arity can be reduced: by filling in the first argument or by filling in the second, for example, in the case where there are just two arguments. The standard mechanism for coping with this generalization is to use *variable binding* rather than just the set theoretic notation that we used above, but it is also possible to develop a variable-free semantics that uses basically set theoretic notational devices.

In terms of our simple type system based on U and $\mathbf{2}$, the version of NP that applies to 1-place predicates has type

$$(U \to \mathbf{2}) \to \mathbf{2} \ .$$

2-place predicates have type $U \to (U \to \mathbf{2})$, so the version of NP that applies to 2-place predicates to turn them into 1-place predicates has type

$$(U \to (U \to \mathbf{2})) \to (U \to \mathbf{2}) \ .$$

In general, an NP has a type described by the template

$$(U \to X) \to X \ . \tag{2}$$

Using unification in Prolog we can take advantage of this regularity and describe the inifinite possible types that NPs can have with a single Prolog term.

10 Prolog Data Structures Given a verb with one or more arguments (often conventionally identified with the θ-roles of generative syntax), we think of a sentence denotation as applying the right number of NP denotations to that core to eventually bind all of the verb's variables. So let us represent a sentence like *several villains laughed* as

```
several(X,villain(X),laughed(X))
```

The part of this term contributed by the subject NP is

```
several(X,villain(X),_1)
```

and the part of the term contributed by the VP is

```
laughed(_2)
```

To put it all together we need to be able to execute the unifications `_1=laughed(_2)` and `_2=X`.

To do this, we need to keep track of what slot variables (like `_1` and `_2` above) are available in each constituent's logical form, that is, we have to use *difference structures*, which we will represent generally as a pair consisting of a structure and a list of the slots available for binding in that structure. So the NP's logical form is represented by the pair of Prolog terms that follows.

```
[X,Q], several(X,villain(X),Q)
```

The VP will be assigned a difference structure like the following.

```
[Y], laughed(Y)
```

The sentence logical form is constructed by

1. Unifying `X` and `Y`, so that the quantifier binds the VP's argument variable.
2. Unifying `Q` and `laughed(Y)` (or, equivalently, `laughed(X)`).

You should make certain you see how these unifications will yield the logical form above.

The relevant S-rule will look like this:

```
s( R ) -->
        np( [X,Q], R ),
        vp( [X], Q ).
```

11 These logical form data structures do not exactly match up with the types we have been assigning to our expressions. The VP is all right, but the NP should just be a 1-place function, and it appears to have 2 variables. In fact, the `X` really belongs with the `Q`. With some fancy use of infix operators we might have written this rule as

```
s( R ) -->
        np( (X=>Q)=>R ),
        vp( X=>Q ).
```

or something of the sort. That leads very quickly to quite unreadable logical forms for lexical items, however, and so we will simplify and flatten out our logical forms whenever possible. The underlying structure is left implicit in the structure of the grammar rules.

12 **Building up NPs** We have represented simple predicates in Prolog as follows.

```
verb( [Agent], laugh(Agent) ) --> [laughed].
```

Since common nouns also denote properties, we expect them to have similar semantics.

```
noun( [Index], villain(Index) ) --> [villains].
```

A determiner takes the two parts of a noun denotation and combines them to form a difference structure as follows.

```
det( [X,P,Q], several(X,P,Q) ) --> [several].

np( [Index,Q], Det ) -->
        det( [Index,P,Q], Det ),
        noun( [Index], P ).
```

13 The logical form corresponding to transitive sentences is complicated by the fact that in English, though not in all languages, some pairs of quantifiers exhibit scope ambiguities. So, for example, the logical form for *some villain chased every hero* is ambiguously one of the following terms.

```
some(X,villain(X),every(Y,hero(Y),chased(X,Y)))
every(Y,hero(Y),some(X,villain(X),chased(X,Y)))
```

The following VP rule will derive the first of the readings above. Getting the second reading is more complicated, and requires essentially the same techniques that we have used to deal with syntactic movement constructions.

```
vp( Rest, Predicate ) -->
        v( [Theme|Rest], Nucleus ),
        np( [Theme,Nucleus], Predicate ).
```