— LECTURE NOTES —

Computational Semantics 1.5
– Questions, Mainly –

1 Disclaimer Keep in mind that virtually everything I say here about natural language semantics is highly oversimplified! No professional semanticist is going to be happy with the presentation of semantics given here or elsewhere in these notes. The techniques we develop, however, will remain useful.

2 In order to assign semantic values to natural language expressions, it is crucial to have in mind a *model* against which such expressions will be evaluated. This remains true *even if the expressions are never actually evaluated.* For example, our primary focus may be on *inference* done directly on logical forms in such a way that it leads to computational behaviors that appear to the user to be reasonable or intelligent. In such a case we are only interested in translating sentences to their logical forms and then finding and applying inference rules to derive new logical forms that can be used to generate responses or even possibly physical actions, in a robot for example. Nonetheless, these inference rules will not lead to intelligent behavior unless they, and the logical forms they operate on, are constructed with a sensible model in mind. Put another way, sometimes the *truth conditions* of an expression (the conditions a model would have to satisfy in order to make the sentence true) are more important than its actual truth with respect to any pre-selected particularly privileged model.

Actually, you may want to have several models in mind. For example, in our case we are most likely to evaluate logical forms against (some part of) the Prolog database, but the design of that data or knowledge base is already influenced by the relation it bears to the real world.

3 We will assume from now on that our models are Prolog databases, and not worry too much about whether they represent anything sensible or consistent with any known (meta)physics. We have in mind a particular application, namely data- or knowledge-base querying, or question answering systems. (Rather than, say, autonomous agents or translation systems.) As Prolog databases, our models will be primarily characterized by collections of terms and relations, where the relations may be characterized either *extensionally*, by a collection of unit clauses acting as the records in an ordinary relational database, or they may be characterized *intensionally*, by a collection of clauses some of which are rule clauses, which may even be recursive, in which case they may characterize an infinite relation.

If our models are Prolog databases, then it makes sense for us to make our logical forms *goals*. After all, that gives us a maximally simple mechanism for evaluating them: simply call them, as in the following.

```
?- parse( [...], LF ), call( LF ).
```

4 This means that we can describe semantic values of the following types very easily. First off the type $\mathcal{U}$ of individuals is simply described by constant terms. Actually, given the standard interpretation of an arbitrary Prolog database in first order logic, *any term* denotes an individual. For the moment at least we will focus on constants to keep things simple. Virtually the only natural language expressions that can have this type are *proper nouns*, i.e., *names*. (It may also be possible to treat *definite descriptions*—mainly NPs with a definite article like *the*—as denoting individuals.)

Secondly, we can represent expressions whose values are relational or predicative, i.e., expressions of type $\mathcal{U}^n \to \mathbf{2}$ (or, equivalently, $\mathcal{U} \to \cdots \to \mathcal{U} \to \mathbf{2}$). Many natural language expressions are of this type. For example, common nouns, simple adjectives, prepositional phrases and verb phrases may be assigned to the type $\mathcal{U} \to \mathbf{2}$. Prepositions and transitive verbs may be assigned to $\mathcal{U}^2 \to \mathbf{2}$ or $\mathcal{U} \to (\mathcal{U} \to \mathbf{2})$.[1]

5 Suppose for the moment that this is all we have: all our noun phrases name individuals. How are we to interpret questions like

What is Waldo reading?

The answer might be *Waldo is reading "Gone With the Wind,"* which could be a true answer if our model contained the assertion

```
read(waldo,gone_with_the_wind).
```

Suppose for the moment that we don't worry about how to generate a full sentence answer like *Waldo is reading "Gone With the Wind."* We will content ourselves with any kind of ordinary Prolog output from which we can infer such an answer with very minimal effort.

6 How would we go about getting such an answer? Well, given that the database includes clauses like

```
read(waldo,gone_with_the_wind).
```

it should be pretty clear that in order to get our answer we must have asked a query like

```
?- read(waldo,X).
```

In particular, we do not want to know what Wendy or William are reading, so the subject argument must be bound to `waldo`. So, in the simplest case, we can take `read(waldo,X)` as the semantic value of the input string

```
[what,is,waldo,reading].
```

[1] Actually, it is very common to treat verbs as having an extra argument which ranges over *events* or *states*, in which case we have to add 1 to the arity that we would normally assign to a verb.

7 How do we build up this logical form during the recognition of the sentence? Well, the subterm `waldo` clearly gets built during the process of recognizing the subject NP. In fact, to say that it "gets built" is overstating things: we expect that it will come right out of the lexicon. That is, the lexical entry for the proper noun `waldo` will be something like the following.

```
word( waldo, np( waldo ) ).
```

Here we have defined our grammatical categories to be complex symbols made up from a major category symbol (here `np`) plus the semantic "feature" `waldo`.

Note that we are overloading the Prolog atom `waldo`, which now serves as both an orthographic word-form and a semantic value. We might prefer to use something more abstract to denote our individuals, like `individual_473` or `ind(473)` or some such. Hopefully in what follows context will make it clear which aspect of the Prolog atom we are talking about.

8 Likewise we expect that the meaning of the word `reading` will be the relation `read/2`. But here we have a choice to face. Our syntactic rules can be set up to assure that the word-form `reading` only ever occurs with two arguments. In that case we can assign as the semantic value of `reading` just the functor symbol `read`, and let our parser glue that to two arguments. But this requires the use of meta-logical predicates in Prolog, and if for no other reason than to keep our programs short and simple we prefer to stick to pure Prolog as much as we can. So we will represent the semantic value of the word-form `reading` as the structure `read(A,B)`, where `A` will be filled in with the value of the subject and `B` with the value of the object.

So minimally the lexical entry for `reading` will have to look like the following.

```
word( reading, verb( read(A,B) ) ).
```

9 Now we know what the semantic values for the terminal elements of the syntax are. Next we have to figure out what the semantic values are for the nonterminal categories. For example—setting questions aside for the moment—we get the semantic value of the VP by combining the semantic value of the V with the semantic value of the direct object NP. By "combining" we usually mean something like functional application. That is, we take one of the two input values to represent a function and the other to represent its argument (or one of its arguments). We apply the function to the argument and take the result to be the value of the whole expression.[2] In our particular case, the application of the verb logical form `read(A,B)` to the NP logical form `gone_with_the_wind` should be:

[2] If we follow this scheme rigorously then it will turn out that the syntactic category of a nonterminal expression will always be irrelevant to the semantics. All nonterminal categories are interpreted as "apply the functor to the argument". We will not be so radical. We can make the lexicon a little simpler by allowing the syntax to carry a little more weight, and the trade-off seems worth it to me.

```
read(A,gone_with_the_wind)
```

"Applying" the verb to the object is clearly just a matter of performing the correct unification, namely:

```
B = gone_with_the_wind
```

Of course, in order to do that the VP rule

```
vp(...) --> verb(...), np(...).
```

will have to have access to the second argument position of the semantic value of the verb, no matter what the verb is.

One way to do this is to use difference structures. That is, we associate with the semantic value of the verb a list which contains all the variables in that logical form. So we now assume that the lexical entry for `reading` looks like this:

```
word( reading, verb( read(A,B), [B,A] ) ).
```

Note the order of variables in the "slot list": `B` is the one we will bind first, so we put that first on the list. Now we can fill in the VP rule as follows.

```
vp( LF, RemainingSlots ) -->
        verb( LF, [B|RemainingSlots] ),
        np( NP_LF ),
        { B = NP_LF }.
```

This in turn simplifies to

```
vp( LF, RemainingSlots ) -->
        verb( LF, [B|RemainingSlots] ),
        np( B ).
```

10 An alternative, and fairly common, way to represent difference structures when they are used for logical forms is to "abstract" the variables one at a time (as in λ-calculus). For example, the logical form for `reading` could be represented as `B^A^read(A,B)`. The Prolog builtin operator `'^'` (which is normally used to bind variables in the goal arguments of `setof/3` and `bagof/3`) associates to the right, so this term is equivalent to `B^(A^read(A,B))`, which is what we want. Then the VP rule looks like the following.

```
vp( LF ) -->
        verb( B^LF ),
        np( B ).
```

This resembles the λ-calculus a lot, where the application of *read* to *GWtW* (short for *"Gone With the Wind"*) would be written

$$(\lambda b.\lambda a.read(a,b))(GWtW) = \lambda a.read(a,GWtW)$$

This Prolog "poor man's lambda calculus" notation is very convenient in this simple case, but becomes rather cumbersome and hard to read when the types get more complex.

11 There is yet another way to do this that does not require the use of difference structures. If we turn `read` from a functor into an argument, for instance in a structure like `read/[A,B]` or `lf(read,A,B)`, then at least under some circumstances we can access the desired argument directly by unification, since all verb logical forms now have the same functor.

```
vp( lf(F,A,B), RemainingSlots ) -->
        verb( lf(F,A,B) ),
        np( B ).
```

This restricts this VP rule to verbs with exactly 2 arguments, where we would otherwise have been able to use it for verbs with 3 or even more arguments. A simple modification fixes that.

```
vp( lf(F,[A|Rest]), RemainingSlots ) -->
        verb( lf(F,[A,B|Rest]) ),
        np( B ).
```

Note that you can't play games with the order of the arguments here: it is awkward to take the arguments in a different order from that in which they are represented in the finished logical form.

12 The value of entire sentence can be got in much the same way as the value of the VP (and with the same options for different data structures).

```
s( LF ) -->
        np( A ),
        vp( LF, [A] ).
```

Indeed, given our approach to semantics a sentence is just a zero-place VP. (And a VP is just a one-place V . . .)

13 Finally we are in a position to talk about questions. The main problem is the problem of what logical form we are to assign to a VP with a gap in object position.

In our simple example, intuitively, we can simply allow the verb to "forget" its object argument slot if there is something in the Wh gap register. Furthermore, since we are not keeping track of syntactic features, the only thing we need to know about the gap register is whether there is anything in it or not.

So the VP rules, augmented with a gap register, might look like the following (where a `0` in the gap slot indicates no gap and a `1` indicates a gap).

```
vp( LF, Rest, 0 ) -->  % same as before
        verb( LF, [B|Rest] ),
        np( B ).
vp( LF, Rest, 1 ) -->
        verb( LF, [_|Rest] ).
```

The same trick applies to the S rule. (On the assumption that our start symbol is now `sbar`, and that it takes care of the inverted auxiliary verb . . .)

```
sbar( LF ) --> % ordinary declarative clause
        s( LF, 0 ).
sbar( LF ) --> % question
        wh_pron,
        aux,
        s( LF, 1 ).

s( LF, 0 ) -->
        np( A ),
        vp( LF, [A] ).
s( LF, 1 ) --> % object questions
        np( A ),
        vp( LF, [A], 1 ).
s( LF, 1 ) --> % subject questions
        vp( LF, [_] ).
```

14 No attempt is made here to keep track of the unfilled argument position. The Prolog interpreter will nonetheless respond to the goal

```
?- parse([what,is,waldo,reading],LF), call(LF).
```

with an answer like

```
LF = read(waldo,gone_with_the_wind)
```

and this is a pretty good approximation the full sentence answer *"Waldo is reading GWtW."*

The problem becomes more serious if we want to ask something like *"Which book is Waldo reading?"* where an answer to the effect that *"Waldo is reading der Spiegel"* would be a little funny. (It is still meaningful if we take Pragmatics into account: it is roughly equivalent to the answer *"None. He is reading a magazine, namely der Spiegel."*) In particular, in case Waldo may be reading more than one thing (for example if the question were *Which books is Waldo reading these days?*) we want the answer to pick out from among the things Waldo is reading just the ones that are books. In this case the "index variable" will need to be the same for both `book` and `read`, so it is no longer an anonymous singleton variable.

Furthermore, even if Waldo is reading only books, if he is reading more than one of them we might want a more complete answer. In ordinary conversation the answer *"Well, Waldo is reading GWtW,"* would generally be sufficient, even in that case. But in a computational application, completeness is generally a good thing. So we would want to be able to construct a call to `setof/3` using `read(waldo,X)` as the goal argument, with `X` as the template argument. That is, if the meaning of the S is

```
read(waldo,X)
```

then the meaning of the S′ should probably be something like

```
setof(X, read(waldo,X), Answer).
```

Again, the object slot is no longer a singleton variable, so it cannot remain anonymous.

15 The first problem we face in getting such a logical form is the problem of keeping track of the argument slot variable that we were simply throwing away above. We find out what variable that is in the VP rule, but we need it in the S′ rule. So we will need to use the gap channel to keep track of it. We will put the term `t(X)` in the gap channel when there is a gap, and continue to use `0` to signify no gap.

```
sbar( LF ) --> % ordinary declarative clause
        s( LF, 0 ).
sbar( setof(X,LF,_) ) --> % wh question
        wh_pron,
        aux,
        s( LF, t(X) ).

s( LF, t(X) ) -->
        vp( LF, [X], 0 ).
s( LF, t(X) ) -->
        np( A ),
        vp( LF, [A], t(X) ).

vp( LF, Rest, t(X) ) -->
        verb( LF, [X|Rest] ).
```

If we have in our database clauses like the following,

```
read(waldo,gone_with_the_wind).
read(waldo,king_lear).
```

then the query

```
?- parse([what,is,waldo,reading],LF), call(LF).
```

will return the answer

```
LF = setof(_A,read(waldo,_A),[gone_with_the_wind,king_lear])
```

16 We might wish to process this raw response a little further in order to produce something a little more user friendly. In such a case we could define a Prolog predicate `wh`, and have the S′ rule build a call to `wh`, with `wh` then calling `setof`. For example:

```
sbar( wh(X,LF) ) -->
        wh_pron,
        aux,
        s( LF, t(X) ).

wh( X, Goal ):-
        setof(X,Goal,Answer),
        print(Answer),
        nl.
```

Then calling the goal with a "semi-anonymous" variable like `_LF`:

```
?- parse([what,is,waldo,reading],_LF), call(_LF).
```

will suppress printing the value of `_LF`, but if it is a `wh` goal, then it will have the side effect of printing the value we are really interested in.

More importantly, if we want to see the whole LF, it will now have something more like the structure of a variable-binding WH operator applied to the main sentence.