— LECTURE NOTES —

Computational Semantics
– More on Questions –

1 We now know how to deal with questions given a language with only individual-denoting NPs. What happens if we are allowed complex NPs? In particular, what happens when the WhP itself can be complex? Let us consider the following example.

> *Which books did Waldo read?*

We have previously analyzed

> *What did Waldo read?/What is Waldo reading?*

as

```
wh( X, read(waldo,X) ).
```

It would make sense to assign to the more complicated sentence an analysis like

```
wh( X, book(X), read(waldo,X) )
```

We can paraphrase this as something like "(Tell me) for which x such that x is a book did Waldo read x." We can then revise our earlier logical form, for consistency, to read

```
wh( X, thing(X), read(waldo,X) )
```

where `thing` is a predicate which we stipulate holds true of all things in our universe.[1] Since `thing(X)` is true of every individual, it must be the case that `(thing(X),read(waldo,X))` is equivalent to `read(waldo,X)`.

2 So we see that Wh-phrases can—and probably have to—be treated as generalized quantifiers.

Since we now have to parse the WhP, we cannot assign a logical form to the S′ independently of the logical form we get from parsing the WhP. Indeed, the LF of the S′ *is* the LF of the WhP, with the nucleus bound to the logical form we get from parsing the S. Revised grammar rules are presented in Fig. 1.

In this grammar fragment I have also included a second gap channel for the verb, to handle subject-auxiliary inversion. I have used `nogap` as the symbol to mark an empty gap channel. Also I used just `gap` to indicate the presence of a verb in the gap channel, since no information about it is required at the gap

[1] Actually this is a poor design decision in real life, since it confuses a predicate which we assume is defined in a fixed way in all models with what might reasonably be the semantics of the word "thing" if "thing" were in our language fragment.

```
sbar( LF ) ---> /* declarative */
        [
         s( LF, nogap, nogap )
        ].
sbar( LF ) ---> /* wh question */
        [
         whp( LF, [X,Q] ),
         aux,
         s( Q, t(X), gap )
        ].

%%% s( LF, WhGap, VGap )
s( LF, nogap, nogap ) ---> /* declarative */
        [
         np( LF, [Index,Nucleus] ),
         vp( Nucleus, [Index], nogap, nogap )
        ].
s( LF, t(X), VGap ) --->   /* subject WhQ */
        [
         vp( LF, [X], nogap, VGap )
        ].
s( LF, t(X), VGap ) --->   /* object WhQ */
        [
         np( LF, [Index,Nucleus] ),
         vp( Nucleus, [Index], t(X), VGap )
        ].
```

Figure 1: New S′ and S rules.

site in this grammar. Note that the handling of the V-channel depends on what is in the Wh-channel.

As you can see, if the sentence is declarative, its LF is just the LF of the S. Otherwise, it is the LF of the WhP, with the nucleus argument of the WhP bound to the value of the S. Owing to the use of the gap channel, we are assured that the nucleus logical form contains an instance of `X` in it, so that the WH operator binds both instances of `X`.

3 We have already seen a rather simple semantic evaluator for generalized quantifiers, and a similar implementation can be given to the WH operator.

```
wh( X, P, Q ):-
        setof( X, P, Xs_that_are_P ),
        setof( X, Q, Xs_that_are_Q ),
        ord_intersection( Xs_that_are_P, Xs_that_are_Q, Answer ),
        format( "~nThe answer is: ~p.~n", Answer ).
```

Actually, since the intersection of the things that are P with the things that are Q is equivalent to the things which are $P \wedge Q$, we could still use the same semantics that we used before, only slightly modified:

```
wh( X, P, Q ):-
        setof( X, (P,Q), Answer ),
        format( "~nThe answer is: ~p.~n", Answer ).
```

4 We can also take advantage of the conservativity of natural language generalized quantifiers and give a definition which may be more efficient in some cases. Recall that the property of conservativity has as a consequence that all natural language determiners can be defined using only the size of the restrictor and the size of the intersection of the restrictor and nucleus. That is, we only need to know the number of things that are P and the number of Ps that are Q.

This suggests an implementation based on a recursive traversal of the list of things that are P, doing something in every case where Q succeeds for that particular member of P. There is a technical problem that has to be dealt with for such a thing to work, however. For every member `A` of P, we will want to instantiate the bound variable `X` to `A` *temporarily*, so as to check whether Q succeeds under that instantiation. Then we have to *undo the binding* of `X`, *without losing the solution*.

There are two ways to do this. One is as in Fig. 2. It uses the Prolog builtin `copy_term/2`, which makes a copy of its first argument such that (a) all variables in the copy are fresh, i.e., free of outside bindings, but (b) they exhibit the same pattern of sharing as we find in the first argument. This way we can make a copy of the goal `Q` together with `X`, then we instantiate the *copy* of `X` (`Y`, that is) and call the copy of `Q`, with the copy of `X` instantiated to the current candidate from `Xs_that_are_P`. For example, if `Next` is currently instantiated to `wendy`, and `Q` is currently instantiated to `late(X)`, then

```
?- copy_term( X^Q, Y^Goal ), Y = Next.
```

will first instantiate `Y^Goal` to something like `Y^late(Y)`, and then finally to `wendy^late(wendy)`. Observe that this has *no effect* on `X`. `X` is not bound to anything in the copy, and we only work with the copy. Note that we can simplify this to

```
?- copy_term( X^Q, Next^Goal ).

wh( X, P, Q ):-
        setof( X, P, Xs_that_are_P ),
        filter( Xs_that_are_P, X, Q, Answer ),
        format( "~nAnswer: ~p.~n", Answer ).

filter( [], _, _, [] ).
filter( [Next|Rest], X, Q, Out ):-
        copy_term( X^Q, Y^Goal ),
        Y = Next,
        (   call( Goal ) ->
            Out = [Next|RestOut]
        ;
            Out = RestOut
        ),
        filter( Rest, X, Q, RestOut ).
```

Figure 2: A "conservative" way of evaluating `wh`.

This approach does not ever try to construct or examine the full set of things that are Q. Therefore it may be more efficient than the previous simpler solution in case Q is relatively large compared to P. It can in principal be adapted to use with any generalized quantifier, as a consequence of the conservativity property.

A related technique that does not use `copy_term/2` uses double negation-as-failure instead. It works because we do not need any bindings from the `Goal` term. We only need to know if it succeeded or not. So we can write the recursive clause of `filter` as follows.

```
filter( [Next|Rest], X, Q, Out ):-
        (   \+ \+ (X=Next,call(Q)) ->
            Out = [Next|RestOut]
        ;
            Out = RestOut
        ),
        filter( Rest, X, Q, RestOut ).
```

The first negation will undo all variable bindings, and fail if the inner goal succeeded, and then the outer negation will turn failure back into success, without restoring any of the lost bindings. This is, to my way of thinking, a dirtier trick than using `copy_term/2`, but not all Prologs have `copy_term/2`, especially older ones.

5 With appropriate grammar rules our new semantics works for both subject and object questions. Indeed, once we get inside of the S the rules are not really any different from the rules for simple Wh phrases, since the relevant calculations are still just a matter of fitting the variable from the gap channel into the right slot of the main predicate.

6 Our grammars both for Wh phrases and quantified NPs have had the property that operators take scope where they appear in surface structure. So, for example, we would assign to a sentence like *Which red blocks is some blue cube on?* a logical form like that below

```
wh( A,
   ( block(A), red(A) ),
   some( B,
         ( cube(B), blue(B) ),
         on(B,A)
        )
   )
```

This means that, for example, a sentence like *Which books did every student read?* is taken to be unambiguous, and it must mean that we are interested in the list of books such that every book on it was read by every student. This is probably correct in this case. There are some quantifiers which insist on taking wide scope, notably *each*. So *Which books did each student read?* is normally taken to be inquiring after a list of lists of books, one list for each student, possibly different for each student.

In ordinary declarative sentences with more than one quantified NP it sometimes happens that (in English) the sentence has a reading in which quantifier scope does not reflect surface structure. For example, a sentence like *Some student read every book* seems to have two distinct logical forms. The one which reflects surface constituency is only true if there is one single particular student who has accomplished the amazing feat of reading every book (relevant to the discourse context, that is). On the other hand there is another reading which makes the sentence true in worlds in which for every book we can find a student that read it, but they may be different students for different books.

7 We now know at least one technique for getting out-of-order quantifier scopings. In a sense the quantifier scoping in an object question is out of order, since the object outscopes the subject. Of course, that is reflected in the surface structure, but the techniques we would need to construct the logical form for the "object wide scope" reading are essentially the same, even when the order of words is irrelevant. That is, consider the following two sentences.

(1) *Some student read every book.*
(2) *Which books did some student read?*

The logical forms (including only the object wide scope reading for (1)) are as follows.

```
(1) every( A, book(A), some( B, student(B), read(B,A) ))
(2) wh( A, book(A), some( B, student(B), read(B,A) ))
```

Notice in particular that the logical form of the nucleus, the formula under `some`, is identical in both cases. So we should be able to use something like a gap channel to pass information up to the scope position.

8 There is a very big difference here, however, namely that the lexical quantifier that determines the main functor appears late in the input string and low in surface structure. So we must pass not just the variable but now the entire 3-argument quantifier term in the gap channel. For instance:

```
vp( LF, Args, gap(Quant,[X,Q]) ) -->
        verb( LF, [X|Args] ),
        np( Quant, [X,Q] ).
```

When we decide to use this rule we have decided by the same token to scope the object NP wide, so we store all the information about its semantics in a quantifier gap channel. However, we also take the opportunity to bind the verb's current argument slot with the object's index variable. This is equivalent to θ-role assignment: this is what tells the world that the object NP is indeed the grammatical object. That is something independent of its quantificational scope.

We must also have a recursive S rule (or a variant of the S′ rule) which uses the contents of the quantifier channel (sometimes called *Cooper Storage* after an early advocate) to construct the LF for the entire sentence.

```
s( LF, nogap ) -->
        s( Q, gap( LF, [X,Q] ) ).
```

9 There is a terribly important efficiency issue here. With Wh phrases we know in advance that we are going to have to move stuff around through our gap channels. There is no reading that does not use the Wh gap channel. With quantifier scopings, *this is always an option*. So quantifier scope ambiguities can vastly increase the number of parses of a sentence.

Some important natural language processing projects, like the *Core Language Engine* of SRI International, avoid the question of quantifier scoping, or rather, postpone it. As a result, parsing derives "quasi logical forms", which are still ambiguous. In effect, quantified NPs are placed in a gap channel automatically, but not ever removed from it (in syntax). Then a separate processing stage acts to try out the consequences and test the likelihoods of the various alternative scopings.