Semantics II

1. Resume

Previously we saw how we could provide a compositional semantics for simple sentences. In the first instance, we associated a semantic value with each phrase and a rule for combining semantic values with each rule for combining phrases. Then the notion of logical form was introduced. Instead of pairing a phrase directly with its semantic value, it was paired with an expression of a logical language that had the same truth conditions.

In this lesson we will examine an encoding of the procedure for translation into logical form in Prolog, and a possible approach to ambiguities in the scope of quantifiers.

2. Encoding the fragment in Prolog

leave.

We will need to augment each phrase in a DCG with an additional argument to carry the logical form. Then we will add an additional predicate to each rule to specify the relation between the logical forms of the rhs constituents and that of the lhs. Then we will partially execute the resulting program, that is, replace certain computations that would normally be performed at run-time by changes to the program itself.

For instance, in the fragment for handling quantified noun phrases given above, the LF for the sentence was obtained by applying the LF of the subject noun phrase to that of the verb phrase. Thus our DCG rule is:

```
s(S) --> np(NP),vp(VP),{apply(NP,VP,S)}.
```

We will encode an expression of the form λX E as X^ E. E will (usually) be an expression that contains one or more instances of the bound variable X. Full evaluation of the result of applying a lambda-expression to an argument requires lambda-binding the argument and then simplifying the resulting expression recursively. In what follows we will be interested only in the initial binding step. lambda-binding this function w.r.t to an argument a involves replacing each occurrence of X in E with a. In this case, we will obtain the following simple definition for bind:

```
bind(Arg^ Expr,Arg,Expr).
```

Note how logic variables are encoded as Prolog variables, so that variable binding is realised by Prolog unification. The function is encoded as a 'difference structure', that is, as an incomplete data structure whose unbound variable(s), though arbitrarily deeply nested within the structure, share with a variable at the top level. The effect of executing the bind procedure is thus to bind the variable positionally at the top level, and thus simultaneously to bind all occurrences of it in the function body. For example:

```
bind(X^ f(3,X,g(X,Y)),2,f(3,2,g(2,Y))).
```

Since the bind procedure is deterministic, we can partially execute it, giving the following grammar rule for s:

```
s(S) --> np(VP^ S), vp(VP).
```

This encoding of functions as difference structures means that we need to code λ expressions which return λ expressions as follows:

λX λY E = Y^ X^ E

In our original grammar determiners were given translations such as:

λQλPall(X)[P(X)->Q(X)]

but notice that we cannot render this in Prolog as:

P^ Q^ all(X,P(X) -> Q(X))

since this is not legal Prolog (predicate variables are not allowed). Therefore we will have to make things slightly more complex. Consider an instance of the above s rule, as in parsing "every man is_stupid". The vp can be straightforwardly represented as:

X^ is_stupid(X)

The translation of the s will be

all(X,man(X) -> is_stupid(X))

Therefore the translation of "every man" will have to be:

Q^ all(X,man(X) -> R)

where bind(Q,X,R) holds. Partially executing this we obtain

(X^ R)^ all(X,man(X) -> R)

as the translation for "every man". By a similar process we can arrive at a final Prolog representation for "every":

(X^ P)^ (X^ R)^ all(X,P -> R)

Now we need the following rule for combining a verb with an object noun phrase:

vp(X^ P) --> tv(X^ P1),np(P1^ P).

Notice what is going on here is analogous to the appending of difference lists that represent the string arguments of a DCG. The translation of the noun phrase is a difference structure which is the difference between the translation of the verb phrase and that of the verb. The verb in turn is a difference structure with a variable corresponding to the individual variable that will be introduced by the subject.

The final version of the entire program is as follows:

```
:- op(100,xfy,'->'). :- op(100,xfy,&).

s(S) -->
   np(VP^ S),
   vp(VP).

np(NP) -->
   det(N1^ NP),
   noun(N1).

vp(IV) -->
   iv(IV).

vp(X^ S) -->
   tv(X^ IV),
   np(IV^ S).

det((X^ R)^ (X^ S)^ exists(X,R&S)) -->
   [a].

det((X^ R)^ (X^ S)^ all(X,R->S)) -->
   [every].

noun(X^ man(X)) -->
   [man].

noun(X^ woman(X)) -->
   [woman].

tv(X^ Y^ loves(X,Y)) -->
   [loves].

iv(X^ is_stupid(X)) -->
   [is_stupid].

test(P) :-
   s(P,[every,man,loves,a,woman],[]).
```

alternative quantifier scopes.

det((X^R)^(X^S)^a(P^Q^exists(P&Q),R,S))
→ [a].

det((X^R)^(X^S)^a(P^Q^all(P->Q),R,S)).
→ every.

```
? - trace,test(LF).
    (   2)  1 call:  test(_0) ?
    (   3)  2 call:  s(_0,[every,man,loves,a,woman],[]) ?
    (   4)  3 call:  np(_22^_0,[every,man,loves,a,woman],L88) ?
    (   5)  4 call:  det(_26^_22^_0,[every,man,loves,a,woman],L99) ?
    (   6)  5 call:  C([every,man,loves,a,woman],a,L99) ?
    (   6)  5 fail:  C([every,man,loves,a,woman],a,L99)
    (   7)  5 call:  C([every,man,loves,a,woman],every,L99) ?
    (   7)  5 exit:  C([every,man,loves,a,woman],every,[man,loves,a,woman])
    (   5)  4 exit:  det((_29^_30)^(_29^_31)^all(_29,_30->_31),
                     [every,man,loves,a ,woman],[man,loves,a,woman])
    (   8)  4 call:  noun(_29^_30,[man,loves,a,woman],L88) ?
    (   9)  5 call:  C([man,loves,a,woman],man,L88) ?
    (   9)  5 exit:  C([man,loves,a,woman],man,[loves,a,woman])
    (   8)  4 exit:  noun(_29^man(_29),[man,loves,a,woman],[loves,a,woman])
    (   4)  3 exit:  np((_29^_31)^all(_29,man(_29)->_31),
                     [every,man,loves,a,woman],[loves,a,woman])
    ( 10)  3 call:  vp(_29^_31,[loves,a,woman],[]) ?
    ( 11)  4 call:  iv(_29^_31,[loves,a,woman],[]) ?
    ( 12)  5 call:  C([loves,a,woman],is_stupid,[]) ?
    ( 12)  5 fail:  C([loves,a,woman],is_stupid,[])
    ( 11)  4 fail:  iv(_29^_31,[loves,a,woman],[])
    ( 13)  4 call:  tv(_29^_51,[loves,a,woman],L146) ?
    ( 14)  5 call:  C([loves,a,woman],loves,L146) ?
    ( 14)  5 exit:  C([loves,a,woman],loves,[a,woman])
    ( 13)  4 exit:  tv(_29^_53^loves(_29,_53),[loves,a,woman],[a,woman])
    ( 15)  4 call:  np((_53^loves(_29,_53))^_31,[a,woman],[]) ?
    ( 16)  5 call:  det(_61^(_53^loves(_29,_53))^_31,[a,woman],L175) ?
    ( 17)  6 call:  C([a,woman],a,L175) ?
    ( 17)  6 exit:  C([a,woman],a,[woman])
    ( 16)  5 exit:  det((_53^_65)^(_53^loves(_29,_53))^
                     exists(_53,_65&loves(_29,_53)),[a,woman],[woman])
    ( 18)  5 call:  noun(_53^_65,[woman],[]) ?
    ( 19)  6 call:  C([woman],man,[]) ?
    ( 19)  6 fail:  C([woman],man,[])
    ( 20)  6 call:  C([woman],woman,[]) ?
    ( 20)  6 exit:  C([woman],woman,[])
    ( 18)  5 exit:  noun(_53^woman(_53),[woman],[])
    ( 15)  4 exit:  np((_53^loves(_29,_53))^
                     exists(_53,woman(_53)&loves(_29,_53)),[a,woman],[])
    ( 10)  3 exit:  vp(_29^exists(_53,woman(_53)&loves(_29,_53)),
                     [loves,a,woman],[])
    (   3)  2 exit:  s(all(_29,man(_29)->exists(_53,woman(_53)&loves(_29,_53
                     [every,man,loves,a,woman],[])
    (   2)  1 exit:  test(all(_29,man(_29)->exists(_53,woman(_53)&loves(_29,_
))

LF = all(_29, man(_29) -> exists(_53, woman(_53) & loves(_29, _53)))
```

1. Generating Alternative Quantifier Scopes

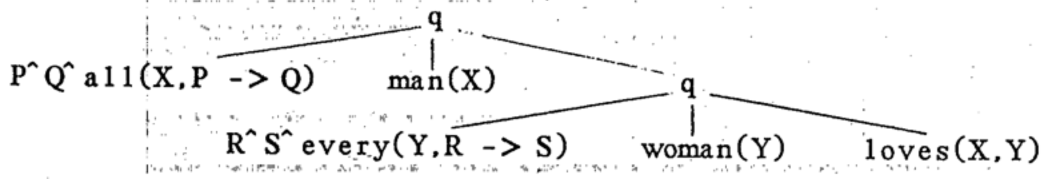
The program presented above assumes a fixed relative scoping for the quantifiers in a sentence, in which that introduced by the subject np always outscopes that introduced by the object np. In the above sentence this seems the most plausible. However, in general, the intended scoping depends not only on syntactic structure, but on inherent properties of quantifiers and on real-world knowledge. For instance, Woods (in Readings in NLP) discusses the likely intended scopings for the questions:

What is the average silicon content for all samples?
What is the average silicon content for each sample?

In the first case, the answer wanted is probably a single figure, while in the second, it is probably a set of sample-content pairs.

Therefore we will change the program above so that it does not produce a fixed quantifier scoping, but rather (under)represents all possible scopings simultaneously in a structure from which all fully specified scopings can be generated non-deterministically.

This intermediate structure is called a quantifier tree, and is built from quantifier nodes (internal) and predication nodes (leaves). Thus the quantifier tree for the above sentence would be:



So a quantifier node has the form q(D,R,S), where D is the expression representing the determiner meaning, and R and S are sub-trees from which the range and scope of D will be obtained. We can build such a quantifier tree merely by replacing the above determiner rules:

```
det((X^ R)^ (X^ S)^ exists(X,R&S)) --> [a].
det((X^ R)^ (X^ S)^ all(X,R->S)) --> [every].
```

with the following

```
det((X^ R)^ (X^ S)^ q(P^ Q^ exists(P&Q),R,S)) --> [a].
det((X^ R)^ (X^ S)^ q(P^ Q^ all(P -> Q),R,S)) --> [every].
```

The top-level sentence rule now reads:

```
s(T) --> np(VP^ S),vp(VP),{pull(S,T)}.
```

where S is a quantifier tree, T is a fully scoped formula, and pull is a procedure which generates all fully scoped formulae compatible with a given quantifier tree. For instance, from the above quantifier tree, the following two logical forms will be generated:

```
LF = all(_29, man(_29) -> exists(_53, woman(_53) & loves(_29, _53))) ;
LF = exists(_29, woman(_29) & all(_53, man(_53) -> loves(_53, _29))) ;
```