— LECTURE NOTES —

Introduction to Computational Semantics
– Part 2 –

1 Evaluating Logical Forms

1 Let $\mathbf{2} = \{0, 1\}$ be the set of truth values. (Draw them anyway you like; last time we called them "true" and "false".) Let P and Q be variables of type $U \to \mathbf{2}$. We give the truth conditions for several English determiners in Fig. 1, for the specific case where they occur in subject NPs.

Many determiners, including *many* itself, depend for their definition on an indeterminate boundary quantity that is hard to specify precisely, even for specific cases. For example, what is the meaning of *almost twenty*? 18? Hard to say. Usually we give definitions for determiners of the *many* and *few* class with a named constant parameter ("k" or "η") which is understood to be set to a value within some fuzzy and situation specific interval. So, for example,

$$[\![\textit{almost twenty}]\!](P, Q) = 1 \iff 20 - \eta \leq |P \cap Q| < 20 \ .$$

You can see the effect of this fuzzy constant in the choice of adverbs that these determiners take: *very many*, *ridiculously few*, as opposed to *exactly five*, *almost every*. The adverbs *very* and *ridiculously* can at least intuitively be interpreted as sliding η around. Anyway, this is not a problem we will pursue any further here.

$$\begin{aligned}
[\![\textit{every}]\!](P, Q) = 1 &\iff P \subseteq Q \\
[\![\textit{some}]\!](P, Q) = 1 &\iff P \cap Q \neq \emptyset \\
[\![\textit{most}]\!](P, Q) = 1 &\iff |P \cap Q| > |P|/2 \\
[\![\textit{no}]\!](P, Q) = 1 &\iff P \cap Q = \emptyset \\
[\![\textit{five}]\!](P, Q) = 1 &\iff |P \cap Q| \geq 5
\end{aligned}$$

Figure 1: Some determiner denotations.

2 **The Blocks World Example** A classic schoolroom example for doing computational semantics is a *Blocks World*. The universe in this case is restricted to a surface on which rest a number of blocks of different shapes, sizes and colors. We are to imagine a robot arm manipulating these blocks according to commands given in natural language. (In Terry Winograd's milestone NLP application SHRDLU I believe there really was a robot arm, but that is another story.) The advantage of this kind of world is that its universe of individuals can be restricted to a very small finite size, so that we can actually effectively enumerate its elements and check them one by one to see if they have certain properties.

The state of a Blocks World can be represented in Prolog by a collection of predicates like `big/1`, `red/1`, `cube/1`, `on/2`, `left_of/2`, and so forth. The blocks themselves are implemented as arbitrary integer codes. Suppose for the moment that our robot has quit work for the day and gone to soak up a few cans of oil down at the corner Kneipe. So the positions of the blocks are fixed, and all we can do is to decide whether certain declarative sentences are true or false with respect to the current state of the Blocks World.

How would we go about deciding the truth of a sentence like *every red block is on a cube*? Presumably the logical form for such a sentence should look like the following Prolog term.

```
every(A,
      and(red(A),block(A)),
      some(B,
           cube(B),
           on(A,B)  ))
```

Now, the logical forms for simple predicates like *red* are already in the format of Prolog goals (recall that the definition of the Blocks World depends on having predicates like `red/1`). And our conjunction above (`and(red(A),block(A))`) has exectly the same truth conditions with respect to a Prolog Blocks World as the Prolog goal `(red(A),block(A))`. The only real problem is in defining predicates for the quantifiers. These will have to be meta-predicates, since they have goals for arguments, just like Prolog builtins like `call/1`, `assert/1` and `setof/3`. The following fragment of Prolog code should make it clear how we can translate the definitions in Fig. 1 pretty directly into Prolog code using `setof/3`, a library of set theoretic predicates (like SICStus' `ordsets` library, for instance), and some other list processing utilities like `member/2` and `length/2`.

```
every( X, R, N ):-
        setof( X, R, Xs_that_are_R ),
        setof( X, N, Xs_that_are_N ),
        ord_subset( Xs_that_are_R, Xs_that_are_N ).
```

Caution!!! This code violates a rule that normally applies for using `setof/3`, namely, that variables appearing in the template term (argument 1) should not occur anywhere else in the code that calls `setof/3`. In the above code fragment we are quite intentionally using the variable bound by the quantifier `every` as the template term. Indeed it is with just such a purpose in mind that we have kept track of this variable in the data structures for quantified logical forms.

3 Another thing to take care of in writing very high level code like this is that it becomes very difficult to debug. The addition of some well-timed printing statements is very helpful in keeping track of what the Prolog interpreter is doing. Keep in mind that just because a logical form is in fact true, and Prolog comes back and says `true?`, it does not follow that the program is working! The following shows the addition of some (rather a lot, actually) built in tracing.

```
every( X, P, Q ):-
        format( "Attempting ~p.~n", [every(X,P,Q)] ),
        setof( X, P, X_P ),
        format( "  ~p things: ~p.~n", [P,X_P] ),
        setof( X, Q, X_Q ),
        format( "  ~p things: ~p.~n", [Q,X_Q] ),
        ord_subset( X_P, X_Q ).
```

(The SICStus `format/2` builtin works analogously to the C `printf()` function.)

2 A Few More Constructions

4 Dealing with adjectives is, in the best case, relatively straightforward. As suggested above, a red cube is a thing which is both red and a cube. That is, many adjectives' semantics is basically *conjunctive* or *intersective*. Of course there are some tricky points. For example, the notion "tall thing" is not really well-defined. Someone could be a tall jockey without being a tall basketball player, even if they were both a jockey and a basketball player. Other notions like "imitation thing" are also somewhat ill-founded. A fake Rembrandt is not a Rembrandt, so we have problems with our intersective semantics right off the bat: that entails that nothing is a fake Rembrandt. It gets worse: a fake Rembrandt is not a fake thing. Indeed it is hard to imagine what a "fake thing" could possibly be, since it clearly cannot be a thing of any sort. Very probably adjectives like *imitation* and *fake* cannot be assigned to a type as simple as $U \to \mathbf{2}$, and certainly they defy our straightforward intersective semantics.

Still, in the context of the blocks world, it is probably okay to assign to a string like `[red,cube]` a semantics like `( red(X), cube(X) )`, with `X` a slot variable.

```
np( LF, [I,Q] ) -->
        det( LF, [I,P,Q] ),
        nbar( P, [I] ).

nbar( LF, [I] ) -->
        noun( LF, [I] ).
nbar( (P,Q), [I] ) -->
        adj( P, [I] ),
        nbar( Q, [I] ).

word( red, adj(red(X),[X]) ).
word( cube, noun(cube(X),[X]) ).
```

Note that the operator `','/2` is not the direct translation of any word occurring in the NP but rather a translation of the construction itself. According to the principle of compositionality the meaning of an expression should depend solely on the meanings of the subexpressions *and the manner of their combination*.

A more thoroughly lexicalized interpretation of adjectives is as functions which map a predicate Q to the conjunction $P \wedge Q$. However, this gets us into problems with the use of adjectives as predicates, as in *every block is red*, which does not have such an obviously conjunctive semantics.

5 Prepositional Phrases We have not dealt much with prepositional phrases so far, but in Blocks World sentences they appear all over the place.

When they appear as predicates (*it is on the blue cube*) they can be dealt with in the same way we deal with transitive verbs: they have the semantics of simple binary relations. What happens when they appear as modifiers? As in: *the pyramid next to the red cube*, for example.

In fact, if we treat them as being like transitive verbs, then our semantics for category $\bar{\mathrm{N}}$ is already set up to deal with them. This is true even if we have expressions like *on every green column*, with a quantifier.

green column	`(green(A),column(A))-[A]`
every green column	`every(A,(green(A),column(A)),B)-[A,B]`
on every green column	`every(A,(green(A),column(A)),on(C,A))-[C]`

```
pp( F, [Y] ) -->
        prep( Q, [X,Y] ),
        np( F, [X,Q] ).
```

The preposition takes two arguments. We will assume that the arguments are ordered as for verbs, with the direct object first and the subject last. Then the object NP acts to bind the prepositional object role, `X`, leaving the external role `Y` still unbound in `Q`, which in turn has now become a subterm of `F`, the generalized quantifier denoted by the NP.

Now, both the modification construction ($\bar{\mathrm{N}}$) and the main S construction are looking for predicates with a single unbound slot, so the PP logical forms provided by this rule could serve either as modifiers or as predicates.

Note that the same goes for adjectives: the logical forms we have given them are suitable for both predicate and modifier roles.

6 Nonverbal Predicates For example *is a cube*, *is on a column*, *is big*. All of these examples involve a form of the copula, which carries the inflection but does not contribute much to the semantics. A sentence like *Few blocks are cubes* ought then to have a logical form like the following.

```
few( X, block(X), cube(X) )
```

Note that very very few determiners can appear in predicate NPs. Possibly, only the indefinite article is allowed. This is presumably because in this position *NPs are not interpreted as generalized quantifiers!* They are interpreted in fact as $\bar{\mathrm{N}}$s, and not as NPs at all. This in turn suggests that the indefinite article *a* is not a quantifier, and, in particular, not just a synonym for *some*.

```
vp( P, [X] ) -->
        copula,
        adj( P, [X] ).
vp( P, [X] ) -->
        copula,
        pp( P, [X] ).
vp( P, [X] ) -->
        copula,
        indef_np( P, [X] ).

indef_np( P, [X] ) -->
        ( [a] ; [] ),
        nbar( P, [X] ).
```

Note that in defining indefinite NPs we have no way of assuring that there is no (dummy) determiner if and only if they are plural, since we have not attempted to include any syntactic features in this grammar.

```
%%% blocks2.pl

:- ensure_loaded(library(ordsets)).
:- ensure_loaded(library(lists)).


start_symbol( s(LF,_), LF ).

s( LF, [] ) --->
        [
         np( LF, [Index,Nucleus] ),
         vp( Nucleus, [Index] )
        ].

np( GenQuant, [Index,Nucleus] ) --->
        [
         det( GenQuant, [Index,Restrictor,Nucleus] ),
         nbar( Restrictor, [Index] )
        ].

np_indef( Predicate, [Index] ) --->
        [
         art_indef,
         nbar( Predicate, [Index] )
        ].

vp( Nucleus, ThetaGrid ) --->
        [
```

```
         copula,
         pp( Nucleus, ThetaGrid )
        ].
vp( Nucleus, ThetaGrid ) --->
        [
         copula,
         adj( Nucleus, ThetaGrid )
        ].
vp( Nucleus, ThetaGrid ) --->
        [
         copula,
         np_indef( Nucleus, ThetaGrid )
        ].


pp( Nucleus, RestGrid ) --->
        [
         prep( Predicate, [NextIndex|RestGrid] ),
         np( Nucleus, [NextIndex,Predicate] )
        ].

nbar( Term, Slots ) ---> [noun( Term, Slots )].
nbar( (NPred,APred), [Index] ) --->
        [
         adj( APred, [Index] ),
         nbar( NPred, [Index] )
        ].


word( block, noun(klotz(X),[X]) ).
word( cube, noun(cube(X),[X]) ).
word( blocks, noun(klotz(X),[X]) ).
word( cubes, noun(cube(X),[X]) ).

word( red, adj(red(X),[X]) ).
word( blue, adj(blue(X),[X]) ).
word( green, adj(green(X),[X]) ).
word( big, adj(big(X),[X]) ).
word( little, adj(little(X),[X]) ).

word( is, copula ).
word( are, copula ).

word( on, prep(on(X,Y),[Y,X]) ).

word( a, art_indef ).
```

```
word( every, det(every(X,P,Q),[X,P,Q]) ).
word( some, det(some(X,P,Q),[X,P,Q]) ).


%%% In this world
%%%
%%%     [every,red,block,is,on,some,cube]
%%%
%%% but not
%%%
%%%     [some,red,block,is,on,every,cube]

klotz(1).   klotz(2).   klotz(3).   klotz(4).
klotz(5).   klotz(6).   klotz(7).   klotz(8).

red(1).     red(2).     red(3).

green(4).   green(5).

blue(6).    blue(7).    blue(8).

cube(1).    cube(4).    cube(6).    cube(8).

on(1,4).    on(2,6).    on(3,8).    on(4,5).


every(X,P,Q):-
        format("Attempting ~p.~n",[every(X,P,Q)]),
        setof(X,P,X_P),
        format("  ~p things: ~p.~n",[P,X_P]),
        setof(X,Q,X_Q),
        format("  ~p things: ~p.~n",[Q,X_Q]),
        ord_subset(X_P,X_Q).
some(X,P,Q):-
        format("Attempting ~p.~n",[some(X,P,Q)]),
        setof(X,P,X_P),
        format("  ~p things: ~p.~n",[P,X_P]),
        setof(X,Q,X_Q),
        format("  ~p things: ~p.~n",[Q,X_Q]),
        ord_intersection(X_P,X_Q,[_]).
```

```
%%% earley_dcg.pl
%%%
%%% Tom Cornell
%%% Jan. 13, 2000
%%%
%%% Earley's Algorithm for Augmented PSGs
%%% ------------------------------------
%%%
%%% Agenda based algorithm:
%%% 1. Chart <- empty; Agenda <- initial items
%%% 2. Repeat until the Agenda is empty:
%%%    a. Pop the Agenda.
%%%    b. If the item is new, then apply all appropriate rules
%%%        predict if item active, complete-left if active,
%%%        just complete-right if passive).
%%%    c. Push all immediate consequences onto the Agenda.
%%% 3. If a final item is in the chart, the sentence
%%%    has been accepted.
%%%
%%% In practice, we can handle all of the consequences of each
%%% rule at once, before applying the next rule, so that we
%%% do not have to concatenate all the new consequences and
%%% the old Agenda.
%%%
%%% Items: item( Category, AfterTheDot, StartPos, EndPos ).
%%%
%%% Chart: The current definition of the dynamic predicate
%%%        item/4.
%%%
%%% Agenda: A list of items. (Or several lists, tied together
%%%        in the current resolvent.
%%%
%%% This parser implements the subsumption check, for use
%%% with augmented PSGs, but it does not implement
%%% restriction.


%%% Grammars are expected to provide definitions of the
%%% following predicates.
%%%
%%%    <category> ---> <list of categories>.
%%%    word( <wordform>, <category> ).
%%%    start_symbol( <category>, <parse-value> ).
%%%
%%% <parse-value> can really be anything (or nothing). It
%%% is usually either a parse tree or a logical form.
```

```
:- dynamic item/4.


%%% This tests first to see if the required operator already
%%% exists.
:- ( current_op(X,Y,--->) -> true ; op( 1100, xfx, ---> ) ).



%%% parse( String, Tree )
%%% ---------------------
%%%
%%%    Build the chart. Then call goal_in_chart/2 to
%%%    construct a final item. It also extracts the
%%%    parse value ``feature'' from the final item
%%%    and returns it in Tree. In fact this depends
%%%    on the exact definition of start_symbol/2 in the
%%%    grammar file; it may not actually be a parse
%%%    tree.
%%%
parse( String, Tree ):-
        build_chart( String, End ),
        goal_in_chart( End, Tree ).


%%% build_chart( String, EndPosition )
%%% ---------------------------------
%%%
%%%    Erase the old chart, call assert_string/3 to construct
%%%    the definition of connects/3 from the input string,
%%%    call initialize_agenda/4 to construct the initial agenda
%%%    out of the categorizations of string elements, and finally
%%%    call close_chart/1 to build the chart from the Agenda.
%%%
%%%    The two predicates assert_string/3 and initialize_agenda/4
%%%    could probably be combined into a single procedure, with
%%%    items being added to the agenda for every category of a
%%%    word as that word is being asserted into the definition
%%%    of connects/3.
%%%
build_chart( String, End ):-
        erase_chart,
        assert_string( String, 0, End ),
        initialize_agenda( Agenda0 ),
```

```
        Tab = 0,
        close_chart( Agenda0, Tab ).


%%% assert_string( String, CurrentPos, EndPos )
%%% -----------------------------------------
%%%
%%%    This is a recursion on String that asserts clauses
%%%    into the definition of connects/3, one for each word
%%%    in String.
%%%        It also returns the final sentence position
%%%    in EndPos.
%%%
assert_string( [], N, N ).

assert_string( [Word|Words], I, N ):-
        J is I + 1,
        assertz( connects(Word,I,J) ),
        assert_string( Words, J, N ).


%%% initialize_agenda( ListOfItems )
%%% ------------------------------
%%%
%%%    Initial items are of the form item(S,RHS,0,0),
%%%    for S the start symbol of the current grammar,
%%%    and RHS some expansion of S. This assumes there
%%%    is only one start symbol in the grammar. It
%%%    would be simple to extend it to the case where
%%%    there were more, by embedding the call to findall/3
%%%    in another call to findall/3.
%%%
initialize_agenda( Agenda0 ):-
        start_symbol( S, _T ),
        findall( item(S,RHS,0,0),
                 ( S ---> RHS ),
                 Agenda0
               ).


%%% goal_in_chart( LastStringPosition )
%%% ----------------------------------
%%%
%%%    A goal item is an item of the form
%%%    item(S,[],0,LastStringPosition) for S the start
%%%    symbol of the current grammar.
```

```
%%%
%%%    This program assumes that the predicate
%%%    start_symbol/2, always supplied with the grammar,
%%%    extracts the tree argument of the start symbol
%%%    in its second argument. (In fact, this need not
%%%    be a tree; it can even be the anonymous variable,
%%%    but whatever it is will be reported to the
%%%    parse/2 predicate as the ``value'' of the parse.)
%%%
goal_in_chart( End, Tree ):-
        start_symbol( S, Tree ),
        item( S, [], 0, End ).


%%% close_chart( Agenda )
%%% ---------------------
%%%
%%%    Items are tested for freshness upon being popped off
%%%    of the agenda. If an item is not old, then it is added
%%%    to the chart, and its consequences are added to the agenda.
%%%    If the item is passive, it only has consequences if it is
%%%    the righthand antecedent of the Complete rule. If it is
%%%    active, then it may have consequences following from both
%%%    the Predict rule and as the lefthand antecedent of the
%%%    Complete rule. Note that all three rules contain calls
%%%    to close_chart/1, making close_chart/1 multiply recursive,
%%%    even though the interior calls do not show up directly in
%%%    the definition.
%%%
%%%    Very important is the fact that we add NextItem to the chart
%%%    before deriving its consequences. This is critical to stopping
%%%    endless searches in the face of left recursive grammars.
%%%
close_chart( [], _Tab ).

close_chart( [NextItem|OldItems], Tab ):-
        (   old( NextItem ) ->
            skip
        ;
            tab(Tab), format("Adding to chart: ~p~n", NextItem ),
            add_to_chart( NextItem ),
            (   passive( NextItem ) ->
                complete_right( NextItem, Tab )
            ;
                (   scan( NextItem, Tab ) ->
                    skip       % Predict rule doesn't apply.
```

```
                ;
                    predict( NextItem, Tab )
                ),
                complete_left( NextItem, Tab )
            )
        ),
        close_chart( OldItems, Tab ).



%%%___/ Tests \___________________


%%% old( Item )
%%% -----------
%%%
%%%    An item is old if it is already in the chart, that is,
%%%    already part of the current definition of item/4.
%%%    There are reasons why this test is too simple, when
%%%    the chart contains non-ground items, but they need
%%%    not concern us here. (Ask me, if you want to know.
%%%    I've written a couple of sample grammars that illustrate
%%%    the problem cases.)
%%%
old( item(Cat,After,P0,P) ):-
        item( GenCat, GenAfter, P0, P ),
        subsumes( $(GenCat,GenAfter), $(Cat,After) ).


%%% subsumes( GeneralTerm, SpecificTerm )
%%% ------------------------------------
%%%
%%%    In general A subsumes B if an arbitrary ground instance
%%%    of B unifies with A. The builtin numbervars/3 has the effect
%%%    of grounding Specific. The double negation (an infamous
%%%    Prolog Dirty Trick) has the effect of undoing all of the
%%%    variable bindings that happen as a result of the call to
%%%    numbervars/3 and the unification with General, but it
%%%    succeeds just in case the innermost goal succeeds. In that
%%%    sense it is like logical double negation.
%%%
subsumes( General, Specific ):-
        \+ \+ (   numbervars( Specific, 0, _ ),
                  General = Specific
              ).
```

```
%%% passive( Item )
%%% ---------------
%%%
%%%    An item is passive (i.e., inactive) if there is nothing
%%%    to the right of the dot.
%%%
passive( item(_,[],_,_) ).



%%%___/ Actions \____________________

%%% skip
%%% -------
%%%
%%%    Equivalent to true. We just give it a special name for
%%%    documentation purposes. It is what we do if an item
%%%    is found to be old.
%%%
skip.


%%% In all of the nontrivial parsing rules, the bag of consequences
%%% is processed immediately. That is, the chart is closed (completely)
%%% under the immediate consequences of a rule before any older items
%%% are processed. This saves us concatenating lists of newly derived
%%% items onto the agenda, only to pop them right off again.

%%% predict( Item )
%%% ---------------
%%%
%%%    Implements the Predict rule:
%%%
%%%    [A --> alpha * B beta, i, j]
%%%    ---------------------------- ?- rule(B,gamma).
%%%        [B --> * gamma, j, j]
%%%
predict( item(_Left,[Next|_Rest],_I,J), Tab ):-
        findall( item(Next,NextRHS,J,J),
                 %%rule( Next, NextRHS ),
                 ( Next ---> NextRHS ),
                 PredictedItems
               ),
        tab( Tab ), format("Predicting...~n",[]), Tab1 is Tab + 2,
        close_chart( PredictedItems, Tab1 ).
```

```
%%% scan( Item )
%%% ------------
%%%
%%%    The Scan Rule:
%%%
%%%        [A --> alpha * B beta, i, j]  [w,j,k]
%%%        ------------------------------------ ?- word(w,B).
%%%             [A --> alpha B * beta, i, k]
%%%
scan( item(Mom,[Kid|Kids],I,J), Tab ):-
        connects( W, J, K ),
        word( W, Kid ),
        %%preterminal( Kid, W, KidCat ), % this builds the term
        tab( Tab ), format("Scanning...~n",[]), Tab1 is Tab + 2,
        close_chart( [item(Kid,[],J,K),item(Mom,Kids,I,K)], Tab1 ).


%%% The Complete rule is as follows:
%%%
%%%    [A --> alpha * B beta, i, k] [B --> gamma *, k, j]
%%%    --------------------------------------------------
%%%             [A --> alpha B * beta, i, j]
%%%
%%% It is implemented in two predicates, depending on whether
%%% the current agenda item is the left or right antecedent.



%%% complete_left( Item )
%%% ---------------------
%%%
%%%    In this case Item is the left antecedent, the active item
%%%    whose dot we are trying to move.
%%%
complete_left( item(Mom,[Kid|Kids],I,K), Tab ):-
        findall( item(Mom,Kids,I,J),
                 item( Kid, [], K, J ),
                 DotMovedItems
               ),
        tab( Tab ), format("Complete Left...~n",[]), Tab1 is Tab + 2,
        close_chart( DotMovedItems, Tab1 ).



%%% complete_right( Item )
```

```
%%% ----------------------
%%%
%%%    In this case Item is the right antecdent, the passive item
%%%    we are using to move the dot in the left antecedent.
%%%
complete_right( item(Kid,[],K,J), Tab ):-
        findall( item(Mom,Kids,I,J),
                 item( Mom, [Kid|Kids], I, K ),
                 DotMovedItems
               ),
        tab( Tab ), format("Complete Right...~n",[]), Tab1 is Tab + 2,
        close_chart( DotMovedItems, Tab1 ).




%%%___/ Utilities \____________________________________


%%% add_to_chart( Item )
%%% --------------------
%%%
%%%    Asserts Item as the last clause in the definition of item/4.
%%%
add_to_chart( Item ):- assertz( Item ).



%%% erase_chart
%%% -----------
%%%
%%%    Just an nullary wrapper around retractall/1 to retract
%%%    all items from the database.
%%%
erase_chart:-
        retractall( item(_,_,_,_) ),
        retractall( connects(_,_,_) ).
```