

**TL—ASSIGNMENT 3**  
**Compositional Semantics for DCG**  
**due Nov 9th 2007 in class**

## 1 Relative Clauses Again

For the last homework you wrote a slash-passing structure-building definite clause grammar that built surface structures represented as list structures such as `[s, [[np, [harry]], [vp, [sees [np, [barry]]]]]]`. This homework makes the DCG build semantic representations represented as **prolog terms** rather than list structures. The advantage of building prolog terms is that they can be **evaluated as prolog queries** with respect to a simple database. That is, the result of evaluating the query `s(T, [harry, sees, barry], [])`. will now yield `T = sees(harry, barry)`, and the query `s(T, [harry, sees, barry], []), call(T)`. yields the answer `Yes`, or `No`, depending on the database. (The prolog predicate `call/1` does exactly what it says; it calls, ie. tries to prove, its argument)

The homework asks you to extend this semantics to cover quantification along lines developed in the lectures, building prolog queries as interpretations and evaluating them to answer questions about a simple prolog database.

## 2 DCG with Montague-Style Semantics

The following relative of the program you worked on in the last homework, which is to be found on the web page, is augmented with a semantics using the approach presented in the lectures, ultimately derived from the work of Montague and Woods.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%
%% (Program 3.12 with semantics                                     %%
%% This is similar to but different from Program 4.5 in P&S.)    %%
%%                                                                %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

:- op(500, xfy, &).      % A dummy logical conjunction operator
:- op(510, xfy, =>).    % Dummy implication, for when we want to do "every"

%% root S has no gap
s(S) --> s(S,nogap).

%% A predicate is a function NP->S; S is instantiated by 'partial
%% execution'. But the subject is now a function over predicates:
```

```

s(S1,Gap) --> np(VP^S1,nogap), vp(VP,Gap).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% write new sentence-level rules for questions here:
%% NB &/2 is a dummy term-building operator.
%s_whq(answer(X)&S1) --> ...
%s_whq(answer(X)&S1) --> ...
%
%s_ynq(S1) --> ...
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% In fact, ALL NPs, even proper names, are functions over properties
%% (so betrand becomes np((bertrand^S)^S), while walks as always yields
%% vp(X^walks(X)). MAKE SURE YOU UNDERSTAND HOW THIS MAKES THE ABOVE
%% S RULE YIELD S1 = walks(bertrand) BEFORE YOU READ BEYOND THE NEXT LINE):
np((E^S)^S,nogap) --> pn(E).

%% The real point of this is to make full NPs do the right thing with
%% quantifiers (look at the definition of determiners etc
%% below. They turn it via partial execution into the equivalent of
%% the following:
%%np((X^S2)^S3,nogap) -->
%%      det((X^S1)^S2)^S3, n(X^S0),optrel((X^S0)^S1)).
np(NP,nogap) -->
      det(N2^NP), n(N1),optrel(N1^N2).

%% ... And gaps
np((X^S)^S,gap(np,X)) --> [].

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% define wh_nps like who and which frog
%%wh_nps
%wh_np(WH) --> whn(WH).
%
%wh_np(NP) --> ...
%wh_np(NP) --> ...
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

vp(X^S1,Gap) -->                                % NB Here we depart from P&S
      tv(Y^X^S),                                % There is more to this than meets
      np((Y^S)^S1,Gap).                          % the eye !

vp(VP,nogap) -->
      iv(VP).

%% The following bind S0, S1, S2, and S3 in the NP definition:
optrel(N^N) --> [].

```

```
optrel((X^S1)^(X^(S1&S2))) -->
    relpron(X), vp(X^S2,nogap).
```

```
optrel((X^S1)^(X^(S1&S2))) -->
    relpron(X), s(S2,gap(np,X)).
```

```
det(LF) --> [Det], {det(Det,LF)}.
det(a, (X^S1)^(X^S2)^exists(X,S1&S2)).
det(the, (X^S1)^(X^S2)^def(X,S1&S2)).
det(some, (X^S1)^(X^S2)^exists(X,S1&S2)).
det(every, (X^S1)^(X^S2)^all(X,S1=>S2)).
```

%%define wh determiners whdet and whpronouns whn here

%%

```
n(LF) --> [N], {n(N,LF)}.
n(author,X^author(X)).
n(book,X^book(X)).
n(man,X^man(X)).
n(program,X^program(X)).
n(programmer,X^programmer(X)).
n(professor,X^professor(X)).
n(student,X^student(X)).
n(subject,X^subject(X)).
```

```
pn(E) --> [PN], {pn(PN,E)}.
pn(begriffsschrift,begriffsschrift).
pn(principia,principia).
pn(lunar,lunar).
pn(shrdlu,shrdlu).
pn(bertrand,bertrand).
pn(gottlob,gottlob).
pn(gilbert,gilbert).
pn(george,george).
pn(terry,terry).
pn(bill,bill).
pn(mathematics,mathematics).
```

```
tv(LF) --> [TV], {tv(TV,LF)}.
tv(concerns,Y^X^concerns(X,Y)).
tv(met,Y^X^met(X,Y)).
tv(ran,Y^X^ran(X,Y)).
```

% NB Here we depart from P&S  
% Note that this is a more  
% traditional semantics for TV  
% See the transitive vp defn above

```

tv(wrote,Y^X^wrote(X,Y)).
tv(write,Y^X^wrote(X,Y)).
tv(concern,Y^X^concerns(X,Y)).

iv(LF) --> [IV], {iv(IV,LF)}.
iv(halted,X^halted(X)).
iv(walks,X^walks(X)).

relpron(LF) --> [RelPron], {relpron(RelPron,LF)}.
relpron(that,LF).
relpron(who,LF).
relpron(whom,LF).

```

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

```

```

answer(Term).
'&'(X,Y) :- X,Y.

```

```

exists(X,TermInX) :- TermInX.

```

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% all etc defined here

```

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

```

Given an appropriate database, including facts like the following –

```

halted(lunar).
program(lunar).

```

– we can not only build queries:

```

| ?- s(T,[some,program,halted],[ ]).

```

```

T = exists(_A,program(_A)&halted(_A)) ?

```

```

yes
| ?- s(T,[some,professor,halted],[ ]).

```

```

T = exists(_A,professor(_A)&halted(_A)) ?

```

```

yes
| ?-

```

— We can also (given appropriate prolog definitions of exists, & etc.) *run* them to yield truth values:

```

| ?- s(T,[some,program,halted],[ ]), call(T).

T = exists(lunar,program(lunar)&halted(lunar)) ?

yes
| ?- s(T,[some,professor,halted],[ ]), call(T).

no
| ?-

```

As noted in the lectures, this program still does not capture the semantics of quantifiers completely. Consider the following example, of the kind you are asked to handle in the homework:

```

| ?- s(T,[every,student, wrote, some,program],[ ]).

T = all(_A,student(_A)=>exists(_B,program(_B)&wrote(_A,_B))) ? ;

no
| ?-

```

Your program will only yield *one* interpretation, whereas there should be two. **You are not expected to solve this problem.**

To begin to make the thing into a query answerer, the exists predicate has been defined as follows:

```
exists(X,TermInX) :- TermInX.
```

This clause succeeds if there is an instantiation of the Term in X, binding X. (The definition `exists(X,TermInX) :- \+ \+ TermInX.` would have the effect of unbinding X and other variables.)

The grammar as it is can be used to test the truth of statements (against the prolog database). So if the facts 'halted(lunar).' and 'program(lunar).' are in the database, as they will be if you consult the testdata file, the following prolog query will succeed.

```

| ?- s(LF, [a, program, halted], [ ]), call(LF).

LF = exists(lunar,program(lunar)&halted(lunar)) ?

yes
| ?-

```

Notice that this treatment of exists gives you the instance(s) that made the call succeed, which seems likely to be useful for answering questions.

The first part of the homework asks you to extend this treatment to universal quantification, so you can determine the truth-value of sentences like ‘All programs halted.’

The simplest way to make this work is to write a rule embodying the quantifier itself, starting like this:

```
all(X, P=>Q) :- ....
```

—where “...” is a definition in terms of ‘P’, and ‘Q’, and possibly the standard Prolog relation `findall` and/or Prolog negation, such that `all(X, program(X) => halted(X))`—the translation of ‘every program halted—is true, if every X for which `program(X)` is true `halted(X)` is also true. (Notice that `=>` is just acting as punctuation, like a comma, in a rule which implements a generalised quantifier. So `all(X, P => Q)` unifies with for example `all(X, program(X) => halted(X))`. The logicians in the class may be able to see that they can in fact define `all/2` very elegantly without using `findall` at all.

The second part of the homework is about extending this grammar beyond declarative sentences like the ones above to wh-questions and yes/no- questions, so you can ask PROLOG questions and get answers.

As an example, what you want to achieve is to be able to ask ‘Who walks?’ instead of having to type the query `| ?- walks(X)`.

This exercise has two (interdependent) parts:

- Getting the *syntax* of these types of questions right.
- Getting the *semantics* right, ie. getting it in such a format that you can support wh-question (and auxiliary verbs)

As far as output from the program goes, you don’t have to do anything special, although you may want to do some pretty printing using `write`. Prolog will work in the normal prolog backtracking way to gives you all values of a variable corresponding to the wh-phrase, just as in the following:

```
| ?- program(X).  
  
X = shrdlu ? ;  
  
X = hw3c ? ;  
  
no
```

### 3 Homework 3

Make sure you understand how the program works using the appropriate examples from the file `hw3.testdata.pl` of test data. Notice that `exists/2` has been defined as `exists(X, TerminX) :- TerminX.`, rather than `exists(X, TerminX) :- \+ \+ TerminX.`

1. Define a correct interpretation for the determiner *every* including the predicate *all* and the relation  $\Rightarrow$  (which the program already declares as an operator with an appropriate precedence), and test it out on the database in the file `hw3.testdata.pl`.  
(Hint: you may want to find out about the prolog built-in predicate `findall` from the sicstus manual on the web page in order to define *all*. `&` is trivial but the summary of built-in predicates tells you how to do that as well.)
2. Extend the grammar to make the DCG handle wh-questions and yes no questions with the auxilliary *did*, as in *Who wrote Principia?*, *Which book did Bertrand write?* and *Did Bertrand write Principia?*.  
*Hint: think about questions as a different type of sentence (so far we have only seen declarative sentences identified by the predicate `s`), and of wh-questions and y/n-questions as different types of questions identified by the predicates `s_whq` and `s_ynq`. This has already been started for you. Then, think about the similarities between relative clauses and questions—notice that both are unbounded. (You can make the semantics of the auxilliary trivial, simply passing the semantics of the main verb).*
3. Run the program on the relevant examples in the file on the web page, calling the interpretation as a query to get an answer to the questions in the file, and putting the results in your copy of the file in the usual way.
4. Warning: this program and the exercise simplify the problem of finding answers to wh-questions: there are some parsable questions like “What does every book concern” that give good-looking logical forms but which will not yield an answer without major modifications. In fact this is where we encounter the real limitations of simulating higher-order logic with prolog.
5. As well as handing in the assignment on the due date in class, send it to the TA via email, with the file named `'hw3-YOURNAME.pl'`.