An Implementation of Discourse Representation Theory

Michael A. Covington
Nora Schmitz
Advanced Computational Methods Center
University of Georgia
Athens, Georgia 30602

April 1988
Minor corrections May 1989

Abstract

This paper documents a computer program that constructs discourse representation structures (DRSes) from ordinary English input. A source listing of the program is included. This program is a much extended version of the one developed by Johnson and Klein (1986) and was built to serve as a basis for other research. It is implemented in Prolog using GULP, a locally developed system for translating feature-structure notation into Prolog terms. The reader of this paper is expected to be familiar with discourse representation theory, Prolog, and GULP.

1 Introduction

This paper briefly documents an implementation of discourse representation theory (DRT) that was built to serve as a foundation for further research.

The reader is assumed to be familiar with DRT (Kamp 1981, Spencer-Smith 1987, Guenthner 1987), with Prolog, and with the extension of Prolog known as GULP (Covington 1987).[1]

The implementation relies on a top-down parser written in definite clause grammar (DCG) notation. The parser incorporates a unification-based grammar that builds discourse representation structures (DRSes). These have the form

```
drs([X1,X2,X3...],[C1,C2,C3...])
```

where `X1`, `X2`, `X3` ... are discourse markers (entities) and `C1`, `C2`, `C3`... are conditions (predications). The entities are represented by integers. Thus the sentence

```
A farmer owns a donkey.
```

is translated into a DRS such as

$$\begin{array}{|l|}\hline \mathit{1\ 2} \\ \hline \mathit{farmer(1)} \\ \mathit{gender(1,m)} \\ \mathit{donkey(2)} \\ \mathit{gender(2,n)} \\ \mathit{owns(1,2)} \\ \\ \hline \end{array}$$

which is represented in this system as

```
drs([1,2],[farmer(1),gender(1,m),donkey(2),gender(2,n),
           owns(1,2)].
```

[1]This work was supported by National Science Foundation Grant Number IST–85–02477. Opinions and conclusions expressed here are solely those of the authors. We thank Donald Nute, David Goodman, and Marvin Belzer for suggestions.

(The gender information for farmer and donkey is looked up in a table; this will be discussed below.)

Note that in the originally created DRS, the discourse markers and conditions appear in the opposite of the order in which they are encountered in the discourse; this does not affect the truth conditions. A utility predicate, `display_drs`, is provided which undoes this reversal and displays the final DRSes in a neat indented format with all their elements occurring in the same order as in the original discourse.

2 Parsing

2.1 Basic syntax

The syntax accepted by the parser is briefly summarized by the following PS-rules:

```
discourse --> (statement ; question), endpunct, discourse.
discourse --> [].

endpunct --> ['.'];['?'];['!'].

question --> [does], np, vp.
question --> [is], np, adj.
question --> [is], np, np.

statement --> s.

s --> np, vp.
s --> np, [does,not], vp.
s --> np, [is], adj.
s --> np, [is,not], adj.
s --> np, [is], np.
s --> np, [is,not], np.
```

```
s --> [if], s, [then], s.

np -->
np --> n(class:proper).
np --> det, n2.
np --> [he]; [him]; [she]; [her]; [it].

n1 --> n(class:common).
n1 --> adj, n1.

n2 --> n1.
n2 --> n1, relcl.

relcl --> ([who]; [whom]; [which]; [that]), s.

det --> [a]; [an]; [every]; [no]; [not,every].

vp --> v, np.
vp --> v.
```

Many syntactic niceties, such as subject-verb agreement, are neglected.

2.2 The hold mechanism

When a relative pronoun is encountered during processing, it is pushed onto a stack, from which it is retrieved when a gap is found (i.e., when an NP is needed but not present). Thus

```
The donkey which the farmer feeds.
```

is parsed as if its structure were:

the donkey $[_s$ the farmer feeds which $]$

That is, which is pushed onto the stack when encountered, then carried along until a direct object for feeds is needed.

The pushdown stack handles nested relative clauses correctly:

$$\textit{The donkey}\ \underbrace{\textit{which the man}\ \underbrace{\textit{whom the woman loved}\_\_\_}\ \textit{feeds}\_\_\_}.$$

The relative pronouns are retrieved from the stack in the opposite of the order in which they are stored.

This hold mechanism is implemented by giving two additional arguments to each phrasal node. For example, in the rule

```
    vp(VP,H1,H2) --> v(V), np(NP,H1,H2).
```

`H1` is the input to the hold mechanism and `H2` is the output. More precisely, upon entry to the rule `H1` is instantiated to a (usually empty) list representing the contents of the holding stack before parsing the verb phrase. Upon exit from the rule, `H2` will be instantiated to the contents of the holding stack after parsing the verb phrase. In this case, `H1` and `H2` are simply passed to the noun phrase, which makes whatever change is required. The verb does not receive arguments for the holding stack because it is not a phrasal node (a verb is a single word).

This technique is described more fully by Covington, Nute, and Vellino (1988: 417–422).

Note crucially that in the current implementation the holding stack contains, not the relative pronouns themselves, but structures of the form `rel(Index)` where `Index` is a discourse marker. Thus `[rel(14),rel(10)]` is what the holding stack looks like when it contains relative pronouns modifying discourse markers 10 and 14.

2.3 Feature structures

The bulk of the work of the parser is done by feature structures, implemented using GULP. The first argument of each node, phrasal or nonphrasal, contains a feature structure of the general form:

$$\begin{bmatrix} \textit{syn:} & \begin{bmatrix} \textit{index: } \ldots \\ \textit{class: } \ldots \\ \textit{arg1: } \ldots \\ \textit{arg2: } \ldots \end{bmatrix} \\ \\ \textit{sem:} & \begin{bmatrix} \textit{in: } \ldots & \\ \textit{out: } \ldots & \\ \textit{res:} & \begin{bmatrix} \textit{in: } \ldots \\ \textit{out: } \ldots \end{bmatrix} \\ \textit{scope:} & \begin{bmatrix} \textit{in: } \ldots \\ \textit{out: } \ldots \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

Not all of the features are instantiated for every node. The roles of the features are:

`index` is the discourse marker (a number) associated with the node. It is instantiated only on nouns and nodes that predicate them (e.g., adjectives), dominate them (e.g., noun phrases), or unify with them (e.g., pronouns). $\underline{\text{Unique}}$ indices are only created for nouns (common and proper). The other nodes that have indices obtain them by copying.

`class` is `common` or `proper`, instantiated on nouns, or `transitive` or `intransitive`, instantiated on verbs.

`arg1` and `arg2` are the discourse markers of subject and direct object, instantiated only on verbs and the nodes that dominate them.

`sem:in` is the discourse representation structure as it exists before processing the current node. When processing a discourse consisting of several sentences, the feature `sem:in` will provide the previous context within which to interpret the new sentence.

`sem:out` is the discourse representation structure after processing the current node. The final semantic representation of a sentence will be crucially influenced by its determiners.

`sem:res` and `sem:scope` are used for passing semantic information to other constituents that modify the logical structure of the sentence; they will be discussed further below (see 3.3).

3 DRS construction

3.1 Form of DRSes

A DRS is a set of discourse markers `U` and a set of conditions `Con`, represented in Prolog as `drs(U,Con)`, where both `U` and`Con` are lists. Corresponding to each discourse marker there is a condition giving its gender for pronoun reference. For example, the sentence

```
Pedro owns a donkey. He feeds it.
```

having the following DRS

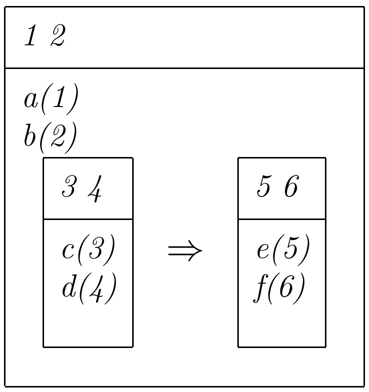
1 2 3 4
named(1,pedro)
gender(1,m)
donkey(2)
gender(2,n)
owns(1,2)
feeds(1,2)

is represented as

```
drs([1,2],
    [named(1,pedro),gender(1,m),donkey(2),gender(2,n),
     owns(1,2),feeds(1,2)]).
```

In practice, the DRS construction algorithm always works with a list of DRSes. This list begins with the DRS currently under construction, followed by all superordinate DRSes, so that all accessible discourse markers can be found. This is not a cyclical data structure because each superordinate DRS is represented as it was before the current DRS was embedded in it.

Consider for instance the hypothetical DRS



at the moment when `f(6)` is being added. The active DRS is the one containing `5` and `6`, and two other DRSes are superordinate to it. Thus the DRS list, as seen by the construction rules at that moment, will be:

```
[drs([6,5],[f(6),e(5)]),
 drs([4,3],[d(4),c(3)]),
 drs([2,1],[b(2),a(1)])].
```

After the innermost DRS is built, construction rules for the outer DRSes will perform the embeddings, resulting in the final structure

```
[drs([2,1],
     [ifthen(drs([4,3],[d(4),c(3)]),
             drs([6,5],[f(6),e(5)])),
      b(2),
      a(1)])].
```

which will be displayed by display_drs as:

```
[1,2]
a(1)
b(2)
IF:
  [3,4]
  c(3)
  d(4)
THEN:
  [5,6]
  e(5)
  f(6)
```

restoring the original order in which elements were encountered.

3.2 Meaning = DRS change

Crucially, the meaning of a sentence, or of any constituent, is the change in the DRS that occurs when that sentence or constituent is processed. Thus every constituent's `sem:in` and sem:out features differ in some way, and the difference represents the meaning of that constituent. For example, if the noun donkey were handled by a single rule, that rule would be

```
n(N) --> [donkey],
   { unique(I),
     n = syn: (index:I ::
```

```
                 class:common) ::
           sem: (in:  [drs(U,Con)|Super] ::
                 out: [drs([I|U],[donkey(I)|Con])|Super]) }.
```

This rule would do the following things:

- Generate a unique number and instantiate `I` to it. This number becomes the discourse marker (index) of the noun.
- Unify the current DRS list with `[drs(U,Con)|Super]`. Here `drs(U,Con)` is the DRS currently under construction and `Super` contains zero or more superordinate DRSes.
- Add `I` to the list `U`, and add `donkey(I)` to the list `Con`, thus adding the meaning of <u>donkey</u> to the DRS under construction.

In the implementation presented here, however, this rule is formulated in a more general way: it applies to the entire category of common nouns. The particular semantics for donkey is looked up in a table:

```
     common_noun(donkey, lambda(X,[gender(X,n),donkey(X)])).
     common_noun(farmer, lambda(X,[gender(X,m),farmer(X)])).
     etc.
```

The rule for a verb is similar, but instead of adding an index to `U`, it uses the indices that are passed to it as syntactic arguments. Here are pseudo-rules for one- and two-argument verbs (again, the real rules look up the semantics in a table):

```
v(V) --> [brays],
   { V = syn: (class:intransitive ::
                 arg1: A) ::
           sem: (in:  [drs(U,Con)|Super] ::
                 out: [drs(U,[brays(A)|Con])|Super]) }.
```

```
v(V) --> [feeds],
   { V = syn: (class:transitive ::
                arg1: A1 ::
                arg2: A2) ::
         sem: (in:  [drs(U,Con)|Super] ::
                out: [drs(U,[feeds(A1,A2)|Con])|Super]) }.
```

The verb rules rely on other rules to pass them valid values of `arg1` and `arg2`. Of course, since this is a unification-based process, it is order-independent; "passing a value" may merely mean unifying `A1` and `A2` with variables that exist elsewhere and will later be instantiated.

3.3 A note on proper nouns

Kamp (1981) originally treated proper nouns as a device for direct reference, hence introducing different DRS-representations for proper nouns or 'constants' on the one hand, and 'predicates' (e.g. common nouns, adjectives, verbs, etc.) on the other. Thus

```
23 = john
```

identifies discourse referent 23 with John, whereas

```
boy(23)
```

says discourse referent 23 is a boy.

This account seems counter-intuitive since the same proper name may very well refer to different individuals. As such, proper names should not be interpreted as logical constants.

In later work, Kamp (1983) and Guenthner (1986) revise the original DRT account of proper names, but by introducing unary predicates:

```
john(23)
```

In the implementation presented here, proper nouns are related to their discourse marker(s) by means of a predicate `named`, e.g., `named(23,'John')`. Different individuals with the same name and occurring within the same discourse no longer create a problem. Further, the same individual can have more than one name (e.g., Hesperus and Phosphorus).

A second aspect of the proper noun problem is their quantificational interpretation. In Kamp (1981), all proper nouns are given an existential interpretation in that they automatically rise to the universe of the topmost DRS. This makes them accessible (as antecedents of anaphors) from anywhere in the discourse.

Since DRT is not concerned with the outside world but rather with mental models and their embeddings in a possible world model, this approach does not have any trouble interpreting unicorns or Santa Clauses: the discourse markers in DRT do not have real world referents and their existence is only 'mental'.

3.4 The crucial role of determiners

The fundamental insight of the Johnson and Klein implementation is that determiners, despite their minor syntactic role, are the most important constituents for establishing the logical structure of the sentence.

Syntactically, a determiner has only one argument: a noun (or NP minus determiner). Semantically, however, we will follow Johnson and Klein in saying that a determiner has two arguments:

> restrictor: the remaining information within the NP.
> scope: the predicate outside the NP.

For example, the sentence

```
A donkey brays.
```

can be translated into predicate logic as

```
(some X: donkey(X)) brays(X)
```

where `some X` is the quantifier, `donkey(X)` is the restrictor (an extra condition attached to the quantifier), and `brays(X)` is the scope.

Accordingly, the DRS for A donkey brays is constructed by passing the restrictor donkey and the scope brays to the determiner a. This is achieved by several rules working together.

Consider first the rule for the simple sentence:

```
s(S) --> { NP = sem:A,
           S  = sem:A,
           VP = sem:B,
           NP = sem:scope:B,
           NP = syn:index:C,
           VP = syn:arg1:C }, np(NP), vp(VP).
```

(Here and in what follows, the hold mechanism is left out for brevity.)

This rule says that:

- All the semantic features of the S are passed to the NP.
- The scope of the NP (and eventually of its determiner) is the semantics of the VP (which comprises `in` and `out` features although we do not see them here). Because of (1), this is also the scope of the S.
- The index of the NP – that is, its discourse marker – is the same as the subject (`arg1`) of the verb.

In this case the features of the VP are simply those of the verb. But what happens in the NP?

The answer is given by the rule

```
np(NP) --> { N   = syn:A,
             NP  = syn:A,
             NP  = sem:B,
             Det = sem:B,
             N   = sem:C,
             Det = sem:res:C }, det(Det), n(N).
```

(Again, this is simplified slightly; the actual grammar includes the syntactic categories `n1` and `n2` intermediate between noun and noun phrase.)

This rule says that an NP has the syntactic features of the noun (crucially including index) but the semantic features of the determiner. Further, the semantics of the noun becomes the restrictor of the determiner.

Finally, the determiner, receiving a scope and restrictor, must incorporate them into the overall DRS in the correct way. The task of the determiner a is very simple:

```
det(Det) --> [a],
       { Det = sem:in:A,
         Det = sem:res:in:A,
         Det = sem:res:out:B,
         Det = sem:scope:in:B,
         Det = sem:scope:out:C,
         Det = sem:out:C }.
```

That is, pass the semantics to the restrictor, then to the scope, letting each of them do its work. The determiner a is implicit in DRT and has no special representation of its own.

Other determiners manipulate the DRS in more complex ways, e.g. every:

```
det(Det) --> [every],
       { Det = sem:in:A,
         Det = sem:res:in:[drs ([],[])|A],
         Det = sem:res:out:B,
         Det = sem:scope:in:[drs([],[])|B],
         Det = sem:scope:out:[Scope,Res,drs(U,Con)|Super],
        Det = sem:out:[drs(U,[ifthen(Res,Scope)|Con])|Super] }.
```

This order-independent process is hard to describe in a step-by-step manner. It is recommended that the interested reader try working out some examples of feature structure unifications by hand.

3.5 "If-then" sentences

As explained in Kamp (1981), "if-then"[2] sentences give rise to a DRS-split: every way in which the IF is true carries with it a way of the THEN being true. For example, the sentence

```
If a man loves a woman then a boojum sees a bandersnatch.
```

goes into DRT as

$$\begin{array}{|l|}\hline \mathit{1\ 2} \\ \hline \mathit{man(1)} \\ \mathit{gender(1,m)} \\ \mathit{woman(2)} \\ \mathit{gender(2,f)} \\ \mathit{loves(1,2)} \\ \\ \hline \end{array} \Rightarrow \begin{array}{|l|}\hline \mathit{3\ 4} \\ \hline \mathit{boojum(3)} \\ \mathit{gender(3,n)} \\ \mathit{bandersnatch(4)} \\ \mathit{gender(4,n)} \\ \mathit{sees(3,4)} \\ \\ \hline \end{array}$$

[2]We call these "if-then sentences" rather than "conditionals" because in DRT, "conditions" are the predicates in a DRS, i.e., the truth-conditions of the DRS, and also because some DRT "if-then" structures arise from sentences that do not look superficially like conditionals in the logical sense.

and creates conditions of the form

```
ifthen(drs(U1,Con1), drs(U2,Con2)).
```

The embedding is performed by the S rule for sentences of the form if X then Y, and by the Det rule for every.

3.6 Negated sentences

Negated sentences are handled by embedding a structure of the form `neg(drs(U,Con))` as a condition of a higher DRS. For example, the S rule for sentences containing does not is:

```
s(S) --> { S  = sem:in:A,
           NP = sem:in:[drs([],[])|A],
           VP = sem:C,
           NP = sem:scope:C,
           NP = syn:index:D,
           VP = syn:arg1:D,
           NP = sem:out:[X,drs(U,Con)|Super],
           S  = sem:out:[drs(U,[neg(X)|Con])|Super] },
                    np(NP), [does,not], vp(VP).
```

Crucially, this rule does not pass `sem` of S directly to `sem` of NP. Instead, it intercepts `sem:in` of S and adds a new, empty DRS at the beginning of it before passing it to NP. The processing of the sentence then adds information to this new DRS. Then this rule intercepts `sem:out` of NP, removes the new DRS (`X`), and embeds it as a condition in the DRS that was previously at the beginning of the list.

The determiner no performs a similar manipulation by intercepting `sem:in` and `sem:out` of Det.

We do not account for all possible syntactic positions or semantic interpretations of negation. For instance, not can have as its syntactic argument an adjective or adverb:

```
Pedro's donkey is not brown but gray.
```

This is a structure we do not account for.

Further, negated sentences containing quantifiers are often ambiguous; for example,

```
Every man does not love a woman.
```

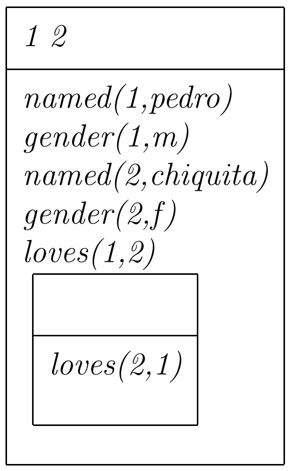
can mean either "every man fails to love" or "not every man loves," and we account for only the latter reading.

3.7 Questions

Discourse representation theory does not provide a way to handle questions. This system treats them like negated sentences except that the embedding functor is `query(...)` rather than `neg(...)`. Thus the discourse:

```
Pedro loves Chiquita. Does she love him?
```

goes into DRT as



or, in the internal representation,

```
drs([1,2],[named(1,pedro),gender(1,m),named(2,chiquita),
           gender(2,f),loves(1,2),
           query(drs([],[loves(2,1)]))]).
```

4 Anaphora

4.1 The basic mechanism

DRT claims that an anaphoric pronoun can only refer to a discourse marker in the current DRS or in a DRS immediately superordinate to it. A DRS that encloses the current DRS is superordinate to it; in addition, the left side of an "if-then" structure is superordinate to the right side, but not vice versa.

As explained in section 3.3, proper names overrule these accessibility conditions for anaphoric linking in that their discourse markers always rise to the universe of the topmost DRS, i.e., an NP can refer to any individual that has been previously mentioned by proper name, regardless of the syntactic structure of the discourse.

Recall that the `sem:in` feature passed to each constituent is not a single DRS, but rather a list containing the current DRS plus all DRSes that are superordinate to it. Accordingly, the antecedent of an anaphoric pronoun (he, him, she, her, or it) is found by searching the list of discourse markers for every DRS in the list, looking for a discourse marker for which the correct gender is recorded. This is done by rules such as:

```
np(NP) --> [he],
     { NP = sem:in:DRSlist,
       member(drs(U,Con),DRSlist),
       member(Index,U),
       member(gender(Index2,m),Con), Index==Index2,
       NP = syn:index:Index,
       NP = sem:scope:in:DRSlist,
       NP = sem:scope:out:Result,
       NP = sem:out:Result }.
```

That is: Choose a `drs(U,Con)` in the DRS list; choose a discourse marker `Index` in `U`; and check that `gender(Index2,m)` occurs in `Con` and that `Index` and `Index2` are already instantiated to the same value. If this is the case, set the index of the current NP equal to the index of the antecedent just found; pass the DRS list to the scope of this NP (i.e., the VP); and take the result as output.

This strategy will always find the most recent possible antecedent first, because it begins searching the lists at the most recently added item.

The implementation presented here does not establish equivalence relations between different individuals with the same proper name, nor between different proper names for the same individual.

4.2 Why some rules are upside down

The alert reader will have noticed that rules for phrasal nodes have the unifications before the expansions, while rules for single words have the unifications last, thus:

```
np(...) --> {...unifications...}, det(...), n(...).
```

versus

```
n(...) --> [donkey], {...unifications...}.
```

There is a good reason for this. The calls to `member` that are used in resolving anaphors require that the DRS list be instantiated at the time `member` is called. In this, the anaphora resolver deviates from strict unification-based grammar. By performing the unifications for all phrasal nodes before parsing the subordinate nodes, we ensure that the instantiations will have taken place. When parsing a non-phrasal node, on the other hand, we know that the anaphora resolver will not be called; the first thing the parser should do is look at the actual form of the word, so that if it has guessed wrong it can back out immediately.

References

[1] Covington, M. A. (1987) *GULP 1.1: An extension of Prolog for unification-based grammar.* ACMC Research Report 00–0021, University of Georgia.

[2] Covington, M. A.; Nute, D.; and Vellino, A. (1988) *Prolog programming in depth.* Glenview, Ill.: Scott, Foresman.

[3] Guenthner, F. (1986) A theory for the representation of knowledge. *IBM Journal of Research and Development* 30.1:39– 56.

[4] Guenthner, F. (1987) Linguistic meaning in Discourse Representation Theory. *Synthese* 73: 569–598.

[5] Johnson, M., and Klein, E. (1986) *Discourse, anaphora, and parsing.* CSLI Research Report 86–63, Stanford University.

[6] Kamp, H. (1981) A theory of truth and semantic representation. In Groenendijk et al. (eds.) *Formal methods in the study of language*, 277–322. University of Amsterdam.

[7] Kamp, H. (1983) *SID without time or questions.* Manuscript, University of Texas, Austin.

[8] Spencer-Smith, R. (1987) Semantics and discourse representation. *Mind and Language* 2.1: 1–26.

Appendix: Program listing

```
/*******************************************************************
      AN IMPLEMENTATION OF DISCOURSE REPRESENTATION THEORY
 ******************************************************************

/*******************************************************************
    Experimental implementation of Discourse Representation Theory
    modeled on that of Johnson and Klein (CSLI Report 86-63).
    Programmed by Michael Covington and Nora Schmitz, U. of Georgia.
    Supported by National Science Foundation Grant IST-85-02477.
 ******************************************************************/

/*******************************************************************
                           DECLARATIONS
 ******************************************************************/

g_features([in,out,syn,sem,index,scope,res,class,arg1,arg2]).
```

```
/*****************************************************************
                      COMPUTATIONAL UTILITIES
 *****************************************************************/

/*
 * reverse(List,Result)
 *   Fast list reversal with stacks.
 *   NOTE: If the tail of the list is uninstantiated, this procedure
 *   will instantiate it to nil. So don't use this procedure on
 *   open lists that need to remain open.
 */

reverse(List,Result) :-
     nonvar(List),
     reverse_aux(List,[],Result).

reverse_aux([],Result,Result).

reverse_aux([H|T],Stack,Result) :-
     reverse_aux(T,[H|Stack],Result).


/*
 * unique_integer(N)
 *    Unifies N with a different integer every time it is called.
 */

unique_integer(N) :- retract(unique_aux(N)),
                     NN is N+1,
                     assert(unique_aux(NN)),
                     !. /* Cut needed by Quintus, not Arity */

unique_aux(0).


/*
 * add_to_topmost_drs(I,Semantics,DRSList,NewDRSList)
 *    Used to let the discourse markers for proper nouns rise to the
 *    universe part of the topmost DRS.
 *    I is an atom (the index); Semantics is a list of DRS-conditions.
 */

add_to_topmost_drs(I,Semantics,[drs(U,Con)],[drs([I|U],NewCon)]) :-
    append(Semantics,Con,NewCon).

add_to_topmost_drs(I,Semantics,[H|T],[H|NewT]) :-
    add_to_topmost_drs(I,Semantics,T,NewT).


/*********************************************************************
                            I/O UTILITIES
 *********************************************************************/

/*
 * display_drs(X)
 *  Outputs a readable representation of a DRS.
 */

display_drs(X) :- display_drs_indented(X,0).

display_drs_indented(X,N) :-
     var(X),
     !,
     write(X),nl.

display_drs_indented(ifthen(X,Y),N) :-
     !,
     tab(N), write('IF:'), nl,
     NN is N+2,
     display_drs_indented(X,NN),
     tab(N), write('THEN:'), nl,
     display_drs_indented(Y,NN).

display_drs_indented(neg(X),N) :-
```

```
     !,
     tab(N), write('NOT:'), nl,
     NN is N+2,
     display_drs_indented(X,NN).

display_drs_indented(query(X),N) :-
     !,
     tab(N), write('QUERY:'), nl,
     NN is N+2,
     display_drs_indented(X,NN).

display_drs_indented(drs(X,Y),N) :-
     !,
     reverse(X,RX),
     tab(N), write(RX), nl,
     reverse(Y,RY),
     display_drs_indented(RY,N).

display_drs_indented([H|T],N) :-
     !,
     display_drs_indented(H,N),
     display_drs_indented(T,N).

display_drs_indented([],_) :- !.

display_drs_indented(Cond,N) :-
     tab(N), write(Cond), nl.


/*********************************************************************
                             DRS-BUILDER
 *********************************************************************/

/***************************************
 * Lexicon and lexical insertion rules *
 ***************************************/
```

```
/*
 * Proper nouns.
 */

n(N) --> [Form],
         { proper_noun_features(Form,N) }.
         /*  add_to_topmost_drs }.  */

proper_noun_features(Form,N) :-
           proper_noun(Form,lambda(I,Semantics)),
           append(Semantics,Con,NewCon),
           unique_integer(I),
           N = syn: (index:I  ::
                     class:proper) ::
               sem: (in:  DRSList ::
                     out: NewDRSList),
           add_to_topmost_drs(I,Semantics,DRSList,NewDRSList).

proper_noun(pedro,     lambda(X,[gender(X,m),named(X,pedro)])).
proper_noun(chiquita,  lambda(X,[gender(X,f),named(X,chiquita)])).

/*
 * Common nouns.
 */

n(N) --> [Form],
         { common_noun_features(Form,N) }.

common_noun_features(Form,N) :-
           common_noun(Form,lambda(I,Semantics)),
           append(Semantics,Con,NewCon),
           unique_integer(I),
           N = syn: (index:I ::
                     class:common) ::
               sem: (in:  [drs(U,Con)|Super] ::
                     out: [drs([I|U],NewCon)|Super]).
```

```
common_noun(bandersnatch,lambda(X,[gender(X,n),bandersnatch(X)])).
common_noun(boojum,      lambda(X,[gender(X,n),boojum(X)])).
common_noun(man,         lambda(X,[gender(X,m),man(X)])).
common_noun(woman,       lambda(X,[gender(X,f),woman(X)])).
common_noun(donkey,      lambda(X,[gender(X,n),donkey(X)])).
common_noun(farmer,      lambda(X,[gender(X,m),farmer(X)])).


/*
 * Adjectives.
 */

adj(Adj) --> [Form],
          { adjective_features(Form,Adj) }.

adjective_features(Form,Adj) :-
            adjective(Form,lambda(I,Semantics)),
            append(Semantics,Con,NewCon),
            Adj = syn: (index:I) ::
                  sem: (in:  [drs(U,Con)|Super] ::
                        out: [drs(U,NewCon)|Super]).

adjective(big,     lambda(X,[big(X)])).
adjective(green,   lambda(X,[green(X)])).
adjective(rich,    lambda(X,[rich(X)])).
adjective(old,     lambda(X,[old(X)])).
adjective(happy,   lambda(X,[happy(X)])).

/*
 * Transitive verbs.
 */

v(V) --> [Form],
         { transitive_verb_features(Form,V) }.

transitive_verb_features(Form,V) :-
           transitive_verb(Form,lambda(A1,A2,Semantics)),
           append(Semantics,Con,NewCon),
           V = syn: (class:transitive ::
                     arg1:A1 ::
                     arg2:A2) ::
               sem: (in:   [drs(U,Con)|Super] ::
                     out:  [drs(U,NewCon)|Super]).

transitive_verb(see,    lambda(X,Y,[sees(X,Y)])).
transitive_verb(sees,   lambda(X,Y,[sees(X,Y)])).
transitive_verb(love,   lambda(X,Y,[loves(X,Y)])).
transitive_verb(loves,  lambda(X,Y,[loves(X,Y)])).
transitive_verb(own,    lambda(X,Y,[owns(X,Y)])).
transitive_verb(owns,   lambda(X,Y,[owns(X,Y)])).
transitive_verb(have,   lambda(X,Y,[has(X,Y)])).
transitive_verb(has,    lambda(X,Y,[has(X,Y)])).
transitive_verb(beat,   lambda(X,Y,[beats(X,Y)])).
transitive_verb(beats,  lambda(X,Y,[beats(X,Y)])).
transitive_verb(feed,   lambda(X,Y,[feeds(X,Y)])).
transitive_verb(feeds,  lambda(X,Y,[feeds(X,Y)])).


/*
 * Intransitive verbs.
 */

v(V) --> [Form],
         { intransitive_verb_features(Form,V) }.

intransitive_verb_features(Form,V) :-
           intransitive_verb(Form,lambda(Arg,Semantics)),
           append(Semantics,Con,NewCon),
           V = syn : (class:intransitive ::
                      arg1:Arg) ::
               sem : (in:  [drs(U,Con)|Super] ::
                      out: [drs(U,NewCon)|Super]).

intransitive_verb(bark,   lambda(X,[barks(X)])).
intransitive_verb(barks,  lambda(X,[barks(X)])).
intransitive_verb(eat,    lambda(X,[eats(X)])).
intransitive_verb(eats,   lambda(X,[eats(X)])).
intransitive_verb(bray,   lambda(X,[brays(X)])).
intransitive_verb(brays,  lambda(X,[brays(X)])).

/*
 * Determiners, each with its own semantics.
 */

det(Det) --> ([a] ; [an]),
               { Det = sem:in:A,
                 Det = sem:res:in:A,    /* Pass 'sem:in' to 'res'.        */
                 Det = sem:res:out:B,
                 Det = sem:scope:in:B,  /* Pass 'res:out' to 'scope:in'. */
                 Det = sem:scope:out:C,
                 Det = sem:out:C }.     /* Whatever comes out of
                                           'scope:out' is the final
                                           result for the whole
                                           sentence.                      */

det(Det) --> [every],
               { Det = sem:in:A,
                 Det = sem:res:in:[drs([],[])|A],
                 Det = sem:res:out:B,
                 Det = sem:scope:in:[drs([],[])|B],
                 Det = sem:scope:out:[Scope,Res,drs(U,Con)|Super],
                 Det = sem:out:[drs(U,[ifthen(Res,Scope)|Con])|Super] }.

det(Det) --> [no],
               { Det = sem:in:A,
                 Det = sem:res:in:[drs([],[])|A],
                 Det = sem:res:out:B,
                 Det = sem:scope:in:B,
                 Det = sem:scope:out:[DRS,drs(U,Con)|Super],
                 Det = sem:out:[drs(U,[neg(DRS)|Con])|Super] }.
```

```
det(Det) --> [not,every],
               { Det = sem:in:A,
                 Det = sem:res:in:[drs([],[])|A],
                 Det = sem:res:out:B,
                 Det = sem:scope:in:[drs([],[])|B],
                 Det = sem:scope:out:[Scope,Res,drs(U,Con)|Super],
                 Det = sem:out:
                  [drs(U,[neg(drs([],[ifthen(Res,Scope)]))|Con])|Super] }.

/*************************
 * Phrase structure rules *
 *************************/

/*
 * n1: a common noun preceded by zero or more adjectives.
 */

n1(N1,H,H) --> n(N1).

n1(N1A,H1,H2) -->
          { N1A = syn:A,
            Adj = syn:A,
            N1B = syn:A,     /* Indices are syntactic, not semantic. */
            N1A = sem:in:B,
            N1B = sem:in:B,
            N1B = sem:out:C,
            Adj = sem:in:C,
            Adj = sem:out:D,
            N1A = sem:out:D },
          adj(Adj), n1(N1B,H1,H2).

/*
 * n2: a common noun of type n1 optionally followed by relative clause.
 */

n2(N2,H1,H2) --> n1(N2,H1,H2).
```

```
n2(N2,H1,H3) -->
           { N2 = syn:Syn,
             N1 = syn:Syn,
             RC = syn:Syn,    /* Pass index to 'RC'. */
             N2 = sem:in:S1,
             N1 = sem: (in:S1 :: out:S2),
             RC = sem: (in:S2 :: out:S3),
             N2 = sem:out:S3 },
           n1(N1,H1,H2), relcl(RC,H2,H3).
                              /* A noun phrase ending with a
                                 relative clause.    */

/*
 * Noun phrases. */

np(NP,H,H) -->
           { N   = syn:class:proper,
             N   = syn:A,
             NP  = syn:A,       /* 'NP' gets its syntax from 'N'.     */
             N   = sem:B,
             NP  = sem:res:B,   /* 'NP' gets its restrictor from 'N'. */
             NP  = sem:in:C,
             NP  = sem:res:in:C,     /* Pass 'sem:in' through 'res'.  */
             NP  = sem:res:out:D,
             NP  = sem:scope:in:D,   /* Pass on through 'scope'       */
             NP  = sem:scope:out:E,
             NP  = sem:out:E },
           n(N).                     /* Proper names do not take
                                        determiners. */

np(NP,H1,H2) -->
           { N2  = syn:class:common,
             N2  = syn:C,
             NP  = syn:C,       /* 'NP' gets its syntax from 'N'.      */
             Det = sem:A,
             NP  = sem:A,       /* 'NP' gets its semantics from 'Det'. */
             N2  = sem:B,
              Det = sem:res:B }, /* 'Det' gets its restrictor from 'N'. */
          det(Det), n2(N2,H1,H2).

/*
 * Trace (gap) from moved relative pronoun.
 */

np(NP,[rel(Index)|Rest],Rest) --> [],   /* Trace from moved relative
                                           pronoun.                  */
          { NP = sem:in:B,              /* This kind of NP has no semantics
                                           and hence no restrictor.   */
            NP = sem:scope:in:B,
            NP = sem:scope:out:C,
            NP = sem:out:C,
            NP = syn:index:Index }.

/*
 * Anaphoric pronouns, with anaphora resolving routine.
 */

np(NP,H,H) --> ([he];[him]),
           { NP=sem:in:DrsList,
             member(drs(U,Con),DrsList),
             member(Index,U),
             member(gender(Index2,m),Con),  Index == Index2,
             NP=syn:index:Index,
             NP=sem:scope:in:DrsList,
             NP=sem:scope:out:DrsOut,
             NP=sem:out:DrsOut }.

np(NP,H,H) --> ([she];[her]),
           { NP=sem:in:DrsList,
             member(drs(U,Con),DrsList),
             member(Index,U),
             member(gender(Index2,f),Con),  Index == Index2,
             NP=syn:index:Index,
             NP=sem:scope:in:DrsList,
             NP=sem:scope:out:DrsOut,
             NP=sem:out:DrsOut }.

np(NP,H,H) --> [it],
           { NP=sem:in:DrsList,
             member(drs(U,Con),DrsList),
             member(Index,U),
             member(gender(Index2,n),Con),  Index == Index2,
             NP=syn:index:Index,
             NP=sem:scope:in:DrsList,
             NP=sem:scope:out:DrsOut,
             NP=sem:out:DrsOut }.

/*
 * Verb phrases.
 */

vp(VP,H1,H2) -->
           {  V  = syn:class:transitive,
              V  = syn:D,
              VP = syn:D,          /* 'VP' gets its syntax from 'V'.        */
              NP = sem:A,
              VP = sem:A,          /* 'VP' gets its semantics from 'NP'.    */
              NP = syn:index:C,
              VP = syn:arg2:C,     /* 'VP' gets its object index from 'NP'. */
              V  = sem:B,
              NP = sem:scope:B },  /* 'NP' gets its scope from 'V'.         */
           v(V), np(NP,H1,H2).

vp(VP,H,H) --> v(VP),
            { VP = syn:class:intransitive }.

/*
 * Relative clauses.
 */

relcl(RC,H1,H2) -->   { RC = syn:index:Index,
                       RC = sem:Sem,
                       S  = sem:Sem },
     ([who];[whom];[which];[that]), s(S,[rel(Index)|H1],H2).

/*
 * Simple sentences.
 */

s(S,H1,H3) -->
         { NP = sem:A,
           S  = sem:A,         /* Pass 'NP=sem' to 'S=sem'.             */
           VP = sem:C,
           NP = sem:scope:C,   /* Pass 'VP=sem' to 'NP=sem:scope'.      */
           NP = syn:index:D,
           VP = syn:arg1:D },  /* Pass 'NP=syn:index' to 'VP=syn:arg1'. */
     np(NP,H1,H2), vp(VP,H2,H3).

s(S,H1,H3) -->
          /*
           * Note: ``does not'' is given sentential scope here.
           * That is, ``Every man does not love a woman'' is
           * taken to mean ``It is not the case that every
           * man loves a woman.''
           */
         { S  = sem:in:A,
           NP = sem:in:[drs([],[])|A],
           VP = sem:C,
           NP = sem:scope:C,
           NP = syn:index:D,
           VP = syn:arg1:D,
           NP = sem:out:[DRS,drs(U,Con)|Super],
           S  = sem:out:[drs(U,[neg(DRS)|Con])|Super] },
     np(NP,H1,H2), [does,not], vp(VP,H2,H3).

s(S,H1,H2) -->
          { S   = sem:A,
            NP  = sem:A,
            NP  = sem:scope:B,
            Adj = sem:B,
            NP  = syn:C,      /* Pass along the syntax.  */
            Adj = syn:C },
     np(NP,H1,H2), [is], adj(Adj).

s(S,H1,H2) -->
          { S   = sem:in:A,
            NP  = sem:in:[drs([],[])|A],
            NP  = sem:out:[DRS,drs(U,Con)|Super],
            S   = sem:out:[drs(U,[neg(DRS)|Con])|Super],
            NP  = sem:scope:B,
            Adj = sem:B,
            NP  = syn:C,
            Adj = syn:C },
     np(NP,H1,H2), [is,not], adj(Adj).

s(S,H1,H3) -->
         {  S   = sem:A,
            NP1 = sem:A,
            NP2 = sem:B,
            NP1 = sem:scope:B,
            NP1 = syn:index:A1,
            NP2 = syn:index:A2,
            NP2 = sem:scope: (in:  [drs(U,Con)|Super] ::
                              out: [drs(U,[(A1=A2)|Con])|Super]) },
     np(NP1,H1,H2), [is], np(NP2,H2,H3).

s(S,H1,H3) -->
         {  S   = sem:in:A,
            NP1 = sem:in:[drs([],[])|A],
            NP1 = sem:out:[DRS,drs(U1,Con1)|Super1],
            S   = sem:out:[drs(U1,[neg(DRS)|Con1])|Super1],
            NP2 = sem:B,
            NP1 = sem:scope:B,
            NP1 = syn:index:A1,
            NP2 = syn:index:A2,
           NP2 = sem:scope: (in:  [drs(U2,Con2)|Super2] ::
                             out: [drs(U2,[(A1=A2)|Con2])|Super2]) },
     np(NP1,H1,H2), [is,not], np(NP2,H2,H3).


/*
 * Complex sentences.
 */

s(S,H,H) -->
         { S  = sem:in: A,
           S1 = sem:in: [drs([],[])|A],
           S1 = sem:out:B,
           S2 = sem:in: [drs([],[])|B],
           S2 = sem:out:[S2DRS,S1DRS,drs(U,Con)|Super],
           S  = sem:out:[drs(U,[ifthen(S1DRS,S2DRS)|Con])|Super] },
     [if], s(S1,[],[]), [then], s(S2,[],[]).
                                /* Empty hold lists enforce Coordinate
                                   Structure Constraint.               */

/*
 * Statements, i.e., top-level, non-embedded sentence.
 */

statement(S) --> s(S,[],[]).

/*
 * Questions.
 */

question(Q) -->
         { Q  = sem:in:A,
           NP = sem:in:[drs([],[])|A],
           VP = sem:C,
           NP = sem:scope:C,
           NP = syn:index:D,
           VP = syn:arg1:D,
           NP = sem:out:[DRS,drs(U,Con)|Super],
           Q  = sem:out:[drs(U,[query(DRS)|Con])|Super] },
     [does], np(NP,[],H2), vp(VP,H2,[]).


question(Q) -->
          { Q   = sem:in:A,
            NP  = sem:in:[drs([],[])|A],
            NP  = sem:out:[DRS,drs(U,Con)|Super],
            Q   = sem:out:[drs(U,[query(DRS)|Con])|Super],
            NP  = sem:scope:B,
            Adj = sem:B,
            NP  = syn:C,
            Adj = syn:C },
     [is], np(NP,[],[]), adj(Adj).


question(Q) -->
         {  Q   = sem:in:A,
            NP1 = sem:in:[drs([],[])|A],
            NP1 = sem:out:[DRS,drs(U1,Con1)|Super1],
            Q   = sem:out:[drs(U1,[query(DRS)|Con1])|Super1],
            NP2 = sem:B,
            NP1 = sem:scope:B,
            NP1 = syn:index:A1,
            NP2 = syn:index:A2,
            NP2 = sem:scope: (in:  [drs(U2,Con2)|Super2] ::
                              out: [drs(U2,[(A1=A2)|Con2])|Super2]) },
     [is], np(NP1,[],[]), np(NP2,[],[]).

/*
 * Discourse
 *   [a,discourse,is,a,series,of,consecutive,sentences,
 *   separated,by,endpuncts,like,this,'.',note,that,an,
 *   endpunct,is,required,after,the,final,sentence,'!']
 */
```

```
discourse(D1) -->
          { D1 = sem:in:A,
            S  = sem:in:A,
            S  = sem:out:B,
            D2 = sem:in:B,
            D2 = sem:out:C,
            D1 = sem:out:C },
     ( statement(S) ; question(S) ), endpunct, {!}, discourse(D2).

discourse(D) --> [],
          { D = sem:in:A,
            D = sem:out:A }.

/*
 * Endpunct (sentence terminator).
 */

endpunct --> ['.'] ; ['?'] ; ['!'].



/*********************************************************************
                              TEST SUITE
 *********************************************************************/

try(String) :- append(String,['.'],Discourse),
               tryd(Discourse).

tryd(String) :- write(String),nl,
                Features = sem:in:[drs([],[])],
                phrase(discourse(Features),String),
                Features = sem:out:DRS,
                DRS = [Current|Super],
                display_drs(Current).

test1 :- try([a,man,sees,a,donkey]).
test2 :- try([a,donkey,sees,a,man]).
```

```
test3 :- try([every,man,sees,a,donkey]).
test4 :- try([every,man,sees,every,donkey]).
test5 :- try([if,a,man,loves,a,woman,then,a,boojum,sees,a,bandersnatch]).
test6 :- try([if,every,man,loves,a,woman,
             then,every,boojum,sees,a,bandersnatch]).
test7 :- try([no,man,loves,every,woman]).
test8 :- try([every,man,loves,no,bandersnatch]).
test9 :- try([no,woman,loves,a,bandersnatch]).
test10 :- try([no,woman,loves,no,man]).
test11 :- try([a,woman,does,not,love,a,man]).
test12 :- try([a,man,does,not,love,every,woman]).
test13 :- try([every,boojum,does,not,see,every,bandersnatch]).
test14 :- try([pedro,owns,a,donkey]).
test15 :- try([pedro,loves,chiquita]).
test16 :- try([a,man,sees,pedro]).
test17 :- try([pedro,has,a,big,green,donkey]).
test18 :- try([no,man,has,a,green,donkey]).
test19 :- try([if,pedro,has,a,big,green,donkey,then,pedro,has,chiquita]).
test20 :- tryd([a,man,loves,a,woman,'.',pedro,owns,a,donkey,'.']).
test21 :- tryd([if,pedro,owns,a,donkey,then,pedro,owns,a,big,donkey,'.',
               chiquita,loves,a,man,'.',
               chiquita,does,not,love,pedro,'.']).
test22 :- tryd([a,donkey,brays,'.']).
test23 :- tryd([if,pedro,owns,a,donkey,then,every,donkey,brays,'.']).
test24 :- tryd([pedro,is,big,'.']).
test25 :- tryd([no,donkey,is,green,'.']).
test26 :- tryd([every,big,green,donkey,is,old,'.']).
test27 :- tryd([pedro,is,a,man,'.']).
test28 :- tryd([every,big,green,donkey,is,an,old,donkey,'.']).
test29 :- tryd([pedro,is,not,big,'.']).
test30 :- tryd([every,donkey,is,not,big,'.']).
test31 :- tryd([pedro,is,not,a,donkey,'.']).
test32 :- tryd([every,donkey,is,not,a,man,'.']).
test33 :- tryd([not,every,man,is,big,'.']).
test34 :- tryd([if,not,every,man,is,big,then,pedro,is,not,big,'.']).
test35 :- tryd([not,every,man,is,a,bandersnatch,'.']).
test36 :- tryd([every,man,who,owns,a,bandersnatch,is,rich,'.']).
```

```
test37 :-
   tryd([every,man,who,does,not,own,a,bandersnatch,that,brays,is,old,'.']).
test38 :-
   tryd([a,man,who,owns,a,bandersnatch,that,does,not,bray,is,happy,'.']).
test39 :-
   tryd([a,man,whom,a,bandersnatch,that,does,not,bray,loves,is,happy,'.']).
test40 :- tryd([pedro,owns,a,donkey,'.',he,is,happy,'.']).
test41 :- tryd([if,pedro,owns,a,donkey,then,he,is,happy,'.']).
test42 :- tryd([every,woman,whom,pedro,loves,is,happy,'.']).
test43 :- tryd([pedro,is,a,man,'.',chiquita,loves,him,'.']).
test44 :- tryd([chiquita,is,a,farmer,'.',she,feeds,a,donkey,'.']).
test45 :- tryd([every,farmer,who,owns,a,donkey,beats,it,'.']).
test46 :- tryd([if,a,farmer,owns,a,donkey,then,he,beats,it,'.']).
test47 :- tryd([is,pedro,a,man,'?']).
test48 :- tryd([does,pedro,own,a,donkey,'?']).
test49 :- tryd([does,every,farmer,own,a,donkey,'?']).
test50 :- tryd([does,every,farmer,who,owns,a,donkey,beat,it,'?']).
test51 :- tryd([chiquita,is,a,woman,'.',is,she,happy,'?']).
test52 :- tryd([pedro,loves,chiquita,'.',does,she,love,him,'?']).
```