Research Report AI–1990–02

Handling Constrained Clauses in Discourse Representation Theory

Artificial Intelligence Programs
The University of Georgia
Athens, Georgia 30602

Handling Constrained Clauses in Discourse Representation Theory

William H. Smith

Piedmont College
Demorest, GA 30535

Artificial Intelligence Research Group
The University of Georgia
Athens, GA 30602

August 8, 1990

Introduction

This report describes a Prolog program that transforms a natural language input into a knowledge base of Prolog clauses. The key element of the program is its ability to handle constrained clauses—embedded clauses whose truth evaluation is different from that of independent clauses. Consider sentence (0.1.a):

(0.1.a) Bob, who is a farmer, believes that Carol regrets that she kissed Ted.
(b) Bob is a farmer.
(c) Carol regrets that she kissed Ted.
(d) Carol kissed Ted.

Clauses (0.1.b,c,d) are embedded in (0.1.a). (0.1.b) is not constrained; it is subject to the same truth evaluation that (0.1.a) is. (0.1.c) and (0.1.d) are constrained. (0.1.a) reports what Bob believes, but Bob may be wrong, so the truth of (0.1.a) should not depend on the truth value of (0.1.c). As we shall see in Section 3 of this report, (0.1.d) should be subjected to truth evaluation; the difference between (0.1.c) and (0.1.d) is determined by the difference between the constrainers *believe* and *regret.*

The theoretical basis of this program is an extension of Discourse Representation Theory (Kamp 1981). Kamp proposed Discourse Representation Theory (DRT) as a bridge between the output of a syntactic parse and model theoretic semantics, a bridge that would combine "a definition of truth with a systematic account of semantic representations." (277) The DRT algorithm produces a representation that can be used to determine the truth conditions of a discourse. Subsequent research has extended the original theory and has implemented it in computer programs.

Section 1 of this report presents a description of the theoretical framework of DRT, a description that will further clarify what is meant by 'constrained clause' and 'truth contribution.'

Section 2 examines suggested extensions to DRT that enable it to handle a wider range of sentences. Section 3 presents the linguistic theory underlying the difference between (0.1.c) and (0.1.d).

In the remaining sections the report shifts its focus from theory to implementation. Section 4 describes the Prolog program of which this is an extension, and Section 5 describes the extensions to that program. Section 6 evaluates the success of the program and suggests other extensions that are needed.

1 DISCOURSE REPRESENTATION THEORY

A model consists of two sets: a set of entities (the universe) and a set of properties of those entities and relations that hold among them. DRT seeks to provide a representation for discourse that will be suitable for truth evaluation in a model. DRT takes as input the output of a syntactic parse (e.g. the Logical Form of Government and Binding Theory) and produces a representation whose structure parallels that of the model.

The central notion of DRT is the Discourse Representation Structure (DRS). A DRS `K` is a pair <U,C>, where `U` is a set of reference markers (the universe) and `C` is a set of conditions (properties, relations, or complex conditions—negation, disjunction, or implication). The initial DRS, `K0`, contains none of the information in the discourse. As the discourse is processed, the DRS construction algorithm, taking the output of a syntactic parse as its input, produces a sequence of `K'` as it incorporates new material from the discourse into `K`. For example:

`(1.1)` Bob saw a woman.

```
K:<U:{R1, R2},
   C:{Bob(R1),
      woman(R2)
      saw(R1, R2) }>}
```

At any point in the construction, the current DRS may be evaluated for truth in the model. Discourse truth in model theoretic semantics is determined by a mapping from a representation of the discourse (in DRT, a DRS) to the model, a mapping that preserves the properties and relationships expressed in the discourse. A discourse is held to be true in a model if there is a mapping such that the set of referenced items (the discourse referents in DRT) maps to a subset of the universe of the model and each property or relation expressed by the discourse (the set of conditions in DRT) is true of the corresponding entities in the model. (For ease of exposition, I refer to the truth evaluation of a clause or discourse; it is in fact the DRS that is evaluated.)

In DRT, each proper noun and each indefinite noun phrase (NP) in the input is taken to have existential quantification. (In a DRS, quantifiers are implicit; with exceptions to be noted shortly, all discourse referents are understood as being existentially quantified.) Thus, when the construction algorithm encounters a proper noun or indefinite NP, a new discourse referent is added to `U` (`R1` and `R2` in (1.1)). Each definite NP (including each pronoun) in the discourse must be coreferential with a discourse referent. Antecedent assignment is accomplished by finding an item in `U` that agrees with the anaphoric expression (for pronouns, an entity that agrees in gender and number). (It should be noted that the fragment of natural language that can be handled by current versions of DRT is quite limited, excluding plural and generic NPs.) Thus, (1.2) shows an extension of (1.1).

`(1.2)` He kissed her.

```
K:<U,
   C := C + {kissed(R1, R2) }>
```

As in Pascal, := is an assignment operator and + indicates union of sets. Thus, the third line of (1.2) means that the new `C` is the union of the old `C` and the set `{kissed(R1, R2)}`.

The basic theory would add R3, R4 to `U`, and then set them equal to R1, R2, respectively. A later "clean-up" operation would eliminate these redundant discourse referents by equating coreferential discourse referents. In the implementation to be described here, resolution of anaphoric relations is accomplished during the syntactic parse, so the redundancy does not occur.

If `U` contains two or more possible antecedents, the conflict is resolved by selecting the most recent one. Consequently, `U` must be ordered, contrary to Kamp (1981) in which `U` is a set, and must be re-ordered each time an entity is referenced (see, for example, Goodman 1988).

Kamp (1985) describes the DRS construction algorithm as "a set of rules that operate, in a roughly top-down manner, on the nodes of the parse tree," (2) converting those nodes into the conditions of `C` and, when appropriate, introducing new discourse referents into U. The basic version of DRT is directed toward the role of NP nodes in the discourse—their relationship to `U`.

The special value of DRT is its ability to handle indefinite NPs that should not introduce new discourse referents. Consider the following extension of (1.2):

`(1.3.a)` She did not wear a ring.

We must extend the algorithm, because as described so far it would produce something like (1.3.b):

```
(1.3.b)  K:< U := U + {R3},
             C := C + {ring(R3),
                       not wear(R2, R3) }>
```

The truth contribution of (1.3.a) to the truth of the discourse will depend on the existence in the model of a ring that she did not wear. But that is not what (1.3.a) means; (1.3.a) is true in the model only if there is no ring in the model such that she wore it. That is, (1.3.c) must be false:

(1.3.c) She wore a ring.

Thus, the truth evaluation of a negative clause demands a special representation.

Kamp's solution is to add to `C` a sub-DRS for (1.3.a):

```
(1.3.d)  K:<U, C := C + {not(K1,
                              K1:<U1: {R3},
                                  C1: {ring(R3),
                                       wear(R2, R3) }> ) }>
```

Discourse referents in `U` are visible to conditions in C1, but those in U1 are not visible to C. Since the sub-DRS is controlled by negation, its truth contribution is the opposite of the truth value of `K1`; that is, if `K1` can be satisfactorily mapped to the model, the discourse is false.

I shall speak of clauses such as (1.3.c) in (1.3.a) as 'constrained clauses' because their truth contributions to the DRS in which they are embedded are constrained, in this case by negation. (This definition of 'constrained clause' may seem to be at odds with that offered in the introduction, since (1.3.c) is not, in traditional terms, an embedded clause. Some syntacticians, however, treat negation as a 'higher predicate' whose argument is the embedded clause to be negated. (e.g. McCawley 1988)).

Other constrained clauses handled by Kamp's original theory are those in disjunctions, implications, and universally quantified NPs. In each case, the constrained clauses are represented by sub-DRSs. The truth contribution of a disjunction is 'true' if one of the disjuncts evaluates to true; that of an implication is 'true' provided that in any mapping in which the antecedent is true the consequent is true also.

An important feature of DRT is that it treats sentences containing universally quantified NPs as implications. The information within the universally quantified NP **restricts** the set of entities in the universe to which the NP can refer, and the remainder of the sentence is the **scope** of the quantifier. For example, the scope of the subject NP is the VP, and the scope of the direct object is the verb. Kamp treats the information in the universally quantified NP as the antecedent of an implication whose consequent is the information in the scope of that NP. Thus, (1.4.a) is represented as (1.4.b).

`(1.4.a)` Every sad man loves a pretty woman.

```
    (b)  K:<U:{},
            C:{K1 ==> K2,
               K1: <U1:{R1},
                     C1:{man(R1),
                         sad(R1) }>
               K2: <U2:{R2},
                     C2:{woman(R2),
                         pretty(R2),
                         loves(R1, R2) }> }>
```

The effects of sentence connectives as constrainers is uncomplicated, as they are taken directly from formal logic. The treatment of other constrainers is more complex, as will be seen when extensions to the basic theory are considered.

2 EXTENSIONS TO THE BASIC THEORY

Events and Propositions

The basic theory is confined to a very limited subset of natural language. In particular, it is limited to singular, non- generic NPs, to anaphoric reference (i.e. the referent is present in the discourse), and to sentences whose main verbs do not take propositions (i.e. DRSs) as arguments. Reducing the first two limitations requires a full theory of reference that must take into account grammar, pragmatics, and knowledge of the real world, and is beyond the scope of this study. Here we are concerned with reducing the third limitation.

Guenthner et al. (1986) extend the basic theory by adding two new types of discourse referent: event markers and time markers. These researchers also include meaning rules in the DRS construction algorithm that assign an event marker to each verb and to each noun that refers to an action (e.g. *accident*). Each time reference (i.e. time of day or extent of duration) is assigned to a time marker. Events are temporally ordered with respect to each other and to time references: an event may precede or overlap another event or time, it may be given an argument expressing its duration, or it may be a subset of another event. The addition of event markers makes it possible for predicates to take DRSs as arguments. Guenthner et al. (1986) do not include any examples of such a use of event markers, but Guenthner

(1987) does. In that article he also makes a notational distinction between events, which advance the time of discourse, and situations, which do not. Spencer-Smith (1987) does not use event markers, but adds a different type of discourse referent, a proposition marker. This extension makes it possible to include embedded predicates, such as infinitival complements and beliefs, in the DRS:

(2.1) Carol wants to kiss a rich man.

```
K:<U: {R1, P1
   C: {Carol(R1), want(R1, P1),
        P1: <U: {R2},
              C: {rich(R2),
                   man(R2),
                   kiss(R1, R2) }> }>
```

The embedded DRS of (2.1) is an example of a constrained clause, in this case constrained by *want*. (2.1) may evaluate to true even if no such R2 exists in the model. (Note that this reading gives narrow scope to *rich man*; Carol does not know which rich man she wants to kiss.)

Beliefs

Kamp (1985) also explored the representation of beliefs. His treatment requires two further additions to DRT: internal and external anchors. Anchors are used to connect discourse referents to entities in the world. External anchors are ordered pairs, <Entity, Marker>, that associate the two as they actually are, while internal anchors are DRS-like structures that associate items as a subject believes they are. The use of anchors makes it possible to represent propositions that are in fact contradictory but are not so in the subject's belief system because his internal anchors differ from the external anchors:

(2.2) Bob believes that Hesperus is pretty and Phosphorus is not pretty.

```
External Anchors: <R1,Bob>,
                  <R2, Venus>,
                  <R3, Venus>
Internal Anchors:
     L: <U: {R2, R3},
         C: {evening_star(R2),
             morning_star(R3)}>
     K: <U: {R1, P1},
         C: {Bob(R1),
             believes(R1, P1),
             P1: <U: {R2, R3},
                C: {pretty(R2),
                    ~K1,
                    K1: <C1: {pretty(R3)}> }> }>
```

The extension of DRT with internal and external anchors give the theory considerable power, but is far from giving it the power necessary to represent adequately the full range of meanings of natural language. It does not, for example, specify whose internal anchor is to be invoked. Asher (1986) combined the concepts of proposition markers and of anchors in order to handle the problem, again with considerable modification to DRT and the DRS construction algorithm.

Asher dealt with the problem of Pierre, who, as a child in Paris, saw Londres—London—in a picture book, and who, as an adult, lives in a London slum. It is now reported that

(2.3) Pierre believes that Londres is pretty and London is not pretty.

Sentences like (2.2) and (2.3) have been a serious problem for traditional logic, which holds that the set of true sentences is closed under substitution of coreferential terms (SC), such as *Londres* and *London*. The belief cannot be added to the toplevel DRS because under SC it is contradictory in the model and its truth value is false, although it is actually a true report of Pierre's belief. The belief must somehow be constrained in a sub-DRS whose truth is evaluated according to Pierre's cognitive state. But the belief remains contradictory in a simple sub-DRS such as that used for (2.2).

Asher made use of a proposition marker that refers to a sub-DRS rather than an entity, one that represents the content of the complement of *believes*. This sub-DRS, a 'delineated DRS', is more complex than those of the basic theory. It is augmented with DRS-like structure representing the cognitive state of the believer—entities and properties that are not explicit in the discourse but must be true in the possible world in which the belief is true. This internal anchor may contain a condition that distinguishes between two referents that are coreferential in the model, as in Pierre's case, or vice versa. In addition, Asher argues that SC does not hold in natural language; two NPs that are coreferential but different in form convey different information and are not substitutable.

In Smith (1989), I applied DRT, the basic theory augmented by the extensions suggested above, to a narrative that had been normed at sixth-grade comprehensibility in order to detect further extensions to DRT that would be necessary in order to represent adequately a natural language discourse. That study suggested the need for additional mechanisms for handling the truth contributions of constrained clauses; mechanisms that deal with the influence of the constrainer on the constrained clause and mechanisms that determine whether, on the basis of subsequent information, a constrained clause should be elevated to the toplevel. These topics are the focus of this study, and will be considered in the next section.

3 THE TRUTH-CONTRIBUTIONS OF CONSTRAINED CLAUSES

Clearly, beliefs are constrained clauses; their truth evaluation is constrained by believe to a method such as that described by Asher. However, while Asher's procedure answers a number of problems when truth is considered from the standpoint of philosophical logic, it is rather awkward when truth is considered from the point of view of information retrieval. For an information retrieval system, truth may simply be the ability, given (2.3), to answer such questions as (3):

```
(3.1.a)  What does Pierre believe?
    (b)  Does Pierre believe that London is pretty?
```

Although Pierre's belief is contradictory, the report of that belief is not. Perhaps the user does not care that the answer to (3.1.a) is contradictory (the user may even want to know if it is contradictory); or that the answer to (3.1.b) is one of two possible answers. Furthermore, the software necessary to construct a possible world in which Pierre's belief is true (if such a program can be constructed at all) comes at a considerable cost in production time, program size, and run time, and the world it describes might be even less acceptable than Pierre's contradictory belief system. We turn now to a method by which constrained clauses can be handled, from an information retrieval point of view, by a much less elaborate method.

Truth Evaluation in an Information Retrieval System

From the information retrieval point of view, truth is what the user claims to be true; input is taken to be true unless it is constrained, as in (2.3) (unless, perhaps, it is inconsistent with the knowledge base). However, constrained clauses are not always blocked from toplevel truth evaluation. I turn now to an examination of various constraining terms and their effects on truth evaluation. This examination is limited to constrained clauses whose truth value is available; that is, the event or state reported has already occurred or failed to occur. Consider the 'neutral' constrainer in (3.2):

(3.2.a) Ted said that Bob kissed Alice.
(b) Ted said, "Yecch!"
(c) She does not kiss yuppies.
(d) Ted was right.

(3.2.a) is true if Ted uttered the constrained clause (or words to that effect), even if the user knows Ted to be a pathological liar. Thus, the constrained clause should not be evaluated for truth at all. The simplest representation of (3.2.a) would assign a discourse referent to the constrained clause in its surface form, without analysis, as is necessarily the case for (3.2.b). However, there are at least four reasons for representing the constrained clause as a sub-DRS, albeit a DRS that is shielded from truth evaluation:

(3.3.a) The user might have applied SC to Ted's actual utterance. If the system includes a facility for calculating a confidence factor (i.e. How reliable is Ted?), it will be necessary to analyze the constrained clause in order to allow for SC in checking for corroboration.

(b) (3.2.a) might be followed by (3.2.c). If the constrained clause has not been analyzed, the discourse referent for *Alice* will not be available as an antecedent for *she*.

(c) If (3.2.a) is the first occurrence of either proper NP, a discourse referent for that entity should be added to the toplevel universe; that entity is 'pragmatically presupposed' by the proper NP (McCawley 1981). This addition can only be made if the constrained clause is first analyzed.

(d) If (3.2.a) is sometime later followed by (3.2.d), provided by the user, the discourse referents and conditions for the constrained clause should be added to the toplevel DRS, and that addition should be based on the context available at the time of (3.2.a), excluding anything that might have been added by intervening sentences. While the necessary analysis could be carried out at the time of (3.2.d), it would be much simpler if it were done at the time of (3.2.a).

Types of Constrainers

The procedure described above, which blocks the truth evaluation of the constrained clause, may, depending on the requirements of the information retrieval system, be satisfactory for a number of constrainers that report utterances or cognitive states, including the following:

(3.4) **SAY CLASS (OBJECT)**

say	suppose	allege	believe	deem
assert	assume	charge	fancy	conclude
maintain	intimate		deny	conjecture

(The list of constrainers in (3.4), as well as those lists that follow, is taken largely from Kiparsky and Kiparsky (1970), as is the analysis on which what follows is based.)

Note that if a sentence whose main verb is *deny* is later confirmed, it is the negation of the constrained clause that must be added to the toplevel DRS. The same would be true of (3.2.a) followed by (3.2.d) with *right* replaced by *wrong*. Assuming this fact to be self-evident, I shall not make special note of other constrainers of this type.

The representation described above will allow an infor- mation retrieval system to answer questions such as those in (3.1). However, there remains an implementation problem to be resolved. Should the system respond only when the constrainer in the inquiry is identical to that in the original input, or should it respond when one constrainer in (3.4) is replaced by another? If the answer is the latter, which constrainers are substitutable? The answer depends on the needs of the particular system, and will not be examined further here.

A group of constrainers whose truth evaluation should be treated in the same fashion is illustrated in (3.5):

(3.5) It seems that Bob kissed Alice.

Constrainers such as seem question or hedge on the truth of their subjects, which are often extraposed, as in (3.5). Similar constrainers are shown in (3.6):

(3.6) **SAY CLASS (SUBJECT)**

seems	is likely	is possible
appears	is probable	

Kiparsky and Kiparsky point out the fact that some of these constrainers allow non-finite *for-to* clauses (e.g. ‘for Bob to kiss Carol’) as their subjects, but do not allow *poss-ing* (e.g. ‘Bob’s kissing Carol’) subjects. I shall return to this fact shortly.

Many of the Say Class of constrainers have adverbial forms:

(3.6.a) It is probable that Ted kissed Carol.

(b) Ted probably kissed Carol.

Although the two sentences in (3.6) are synonymous, the constrained clause in (3.6.a) is dependent while the same clause in (3.6.b) is independent. Nevertheless, synonymous clauses should be treated in the same manner; that is, the constrained clause should not be evaluated for truth in the toplevel DRS.

The group of constrainers shown in (3.7) is syntactically identical to those in (3.6), but the truth contribution is quite different.

(3.7) **TRUE CLASS (SUBJECT)**

true	happens	certain
	chances	sure
false	turns out	

These constrainers assert the truth (or falsity) of the constrained clause. Thus, the constrained clause should not be shielded from the truth evaluation of the toplevel DRS. The simplest representation of a sentence containing one of these constrainers would ignore the constrainer and add the representation of the constrained clause to the toplevel DRS. Such a representation would make it impossible to refer to the constrained clause (cf. 3.2.d), but such reference seems unlikely. If such references must be allowed, sentences containing those constrainers can be handled in the manner of (3.2.a) followed by (3.2.d).

Note that if one of the True Class of constrainers appears in negated form, it is the negation of the constrained clause that is added to the DRS. (However, the negation of *certain* is a Say Class constrainer. The converse is true of *possible*.)

A fourth group of constrainers, although syntactically identical to the Say and True Classes, makes a truth contribution that is different from either.

(3.8) **REGRET CLASS (SUBJECT)**

surprising	exciting	amuses	matters
significant	relevant	counts	suffices
tragic	odd	bothers	makes sense

These constrainers ('factives' for Kiparsky and Kiparsky) semantically presuppose the truth of their complement clauses. That is, the complement clause is held to be true, whether the constrainer appears in positive or negative, declarative or interrogative form. Therefore, the constrained clause should not be shielded from toplevel truth evaluation. In addition, these constrainers have semantic content of their own, unlike those in the True Class. (e.g. The user may want to know **what** the significance is.) It is thus necessary to assign a discourse referent to the constrained clause so that it can appear as an argument to the constrainer. These considerations suggest that a clause constrained by one of these terms should be represented twice, once as an independent sentence and once in the manner of a clause constrained by a Say Class constrainer.

The fifth group of constrainers functions syntactically like the Say Class (Object), but makes a truth contribution like that of the Regret Class (Subject).

(3.9) **REGRET CLASS (OBJECT)**

regret	forget (about)	be aware (of)
resent	make clear	bear in mind
deplore	ignore	take into account/consideration
mind	comprehend	
care (about)	grasp	

These verbs also have semantic content, so discourse referents for the constrained clauses must be added to the toplevel DRS. These verbs allow *the fact that* and *poss-ing* complements, but not *for-to* complements (at least, not with presuppositional value). Some (*regret*, *forget*) have presuppositional value under *equi-NP deletion*, but others do not.

Kiparsky and Kiparsky note that the truth contribution of a clause depends only on the immediate constrainer, no matter how deeply it is embedded:

(3.10.a) Carol appears to believe that Bob regrets that he kissed Alice.
(b) Bob kissed Alice.

Thus, (3.10.a) presupposes (3.10.b), because it is constrained by *regret*, in spite of the fact that the intervening *appear* and *believe* are Say Class constrainers.

The next class of constrainers is labeled 'indifferent' by Kiparsky and Kiparsky because the truth of the constrained clause may or may not be presupposed.

(3.11) **SUSPECT CLASS (OBJECT)**

suspect	acknowledge	announce	emphasize
anticipate	admit	report	remember
deduce			

With *poss-ing* complements, these verbs are factives, Regret Class constrainers. With finite and *for-to* complements, however, it is unclear whether they are Say Class or True Class constrainers.

It may have been noted that there is no list of True Class (Object) constrainers. This gap is particularly striking when one considers that two of the verbs in (3.7), *certain* and *sure*, can constrain object clauses.

(3.12.a) It was certain that Bob kissed Alice.
b) Ted was certain that Bob kissed Alice.
c) Ted was certain of Bob's kissing Alice.
d) Bob kissed Alice.
e) I am certain that Bob kissed Alice.

Sentence (3.12.a) asserts the user's belief that (3.12.d) is true, but (3.12.b) only asserts Ted's belief; the user may know Ted to be wrong. Sentence (3.12.c) is even worse; Ted may have believed that (3.12.d) had already occurred, or that it was potential. Thus, while these verbs are True Class (Subject) constrainers, they are Say Class (Object) constrainers.

Almost paradoxically, this class shift holds even with a first-person subject. Sentence (3.12.e) seems to assert what (3.12.a) asserts, the user's belief that (3.12.d) is true in the model. However, the user's choice of (3.12.e) rather than (3.12.a) is a hedge on that assertion.

Another complex group of constrainers is that shown in (3.13):

(3.13) **KNOW CLASS (OBJECT)**

know	comprehend	learn
see	understand	realize

When one of these verbs is used in a past tense, it is factive; the truth of its constrained clause is presupposed, regardless of the subject or the polarity (affirmative or negative). With a third-person subject, the constrained clause is presupposed in the present tense. (It is assumed here that the constrainer is provided by the user and is not his ironic quotation of the subject.)

It is when the verb has a first-person subject in a present tense that the complexity arrives. In this case, the verbs in (3.13) must be treated on an individual basis. To do so, it will help to use the template in (3.14), where the blank is to be filled in by a (possibly negated) constrainer.

(3.14) I __________ a. (that) Bob kissed Alice.
b. (of/about) Bob's kissing Alice.
c. (for) Bob (to) kiss Alice.

The constrainer *know* allows (3.14.a,b). In either case, it is like *certain*: in the affirmative it is a True Class constrainer, but in the negative it shifts to the Say Class. As in the case of*certain*, the *poss-ing* complement is not presupposed.

In (3.14.b), *comprehend* and *understand* presuppose their complement clauses. The verb *understand* allows (3.14.a), but only in the affirmative, in which case it is a Say Class constrainer, with a meaning something like "Someone said that Bob kissed Alice." The constrainer *realize* is acceptable only in (3.14.a) and presupposes its complement clause; thus, if it is negated, it denies its own presupposition and the sentence makes no sense.

The simple present tense of learn does not work in (3.14) except in special contexts (e.g. an actor referring to his role as himself), and the present progressive requires that the learning process be extended, so that what is learned is (usually) not a simple fact. While in some special contexts the constrained clause is presupposed, it is more practical to treat first-person present tense *learn* as a Say Class constrainer. The verb *see* is ambiguous between an act of perception and a state of cognition. In the former case it allows (3.14.c) (without prepositions) and is a True Class constrainer. In (3.14.a), *see* reports a cognitive state; in the affirmative it acts as does *understand*, but in the negative it is a Say Class constrainer.

This section has examined clauses whose truth contribu- tions to a DRS are constrained by a certain group of predicates (verbs and adjectives). It has found five classes of constrainer, some with subclasses according to whether the constrained clause is the subject (perhaps extraposed) or the object of the predicate:

(3.15.a) SAY CLASS: The constrainer makes no claim about the truth contribution of the constrained clause, so that clause should be shielded from toplevel truth evaluation. However, the clause should be identified by a discourse referent because it may later be confirmed or contradicted.

(b) TRUE CLASS: The constrained clause is added to the toplevel DRS because the constrainer asserts the truth of its complement. (If the constrainer is negated, it is the negation of the constrained clause that is added.) Such clauses might not require discourse referents, although no harm is done if referents are assigned.

(c) REGRET CLASS: The constrainer presupposes the truth of the complement clause, so that clause is added to the toplevel DRS. This addition is independent of the form of the matrix clause (i.e. affirmative-negative, declarative-interrogative).

(d) SUSPECT CLASS: Presupposition depends on the syntactic form of the constrained clause. *poss-ing* complements are presupposed, but other complements are unclear about presupposition and are to be treated as those under Say Class constrainers.

(e) KNOW CLASS: The constrained clause is presupposed unless the constrainer is first-person, present tense. In the latter case, each verb requires its own treatment.

It was noted that some True Class constrainers have corresponding adverbs. Other adverbs can act as constrainers, and their truth contributions must be examined. Modal auxiliaries and verbs with similar semantic value (e.g. *ought to*, *want to*) also require such analysis. Associated with the modal class is the case of constrained clauses whose truth is not yet known; should they be left unevaluated, or should they become 'demons' whose truth must eventually be evaluated? These questions remain to be answered.

In the next section we examine a computer implementation of DRT that is able to handle certain of the constrained clauses described here.

4 AN IMPLEMENTATION OF DRT

In this section and the next we describe a computer implementation of DRT. It takes as input a 'discourse' of one or more English sentences, parses the input and constructs a DRS for it, and then asserts the set of conditions in that DRS to a Prolog knowledge base. The language it accepts includes both the types of sentence described in Section 2 and a subset of those described in Section 3. Specifically, it handles noun clause complements of a selected group of Say, True, and Regret Class Constrainers.

This implementation is an extension of *From English to Prolog via Discourse Representation Theory* (Covington, Nute, Schmitz, & Goodman, 1988; henceforth CNS&G). That program is "a set of techniques for translating the discourse representation structures of Kamp (1981) into semantically equivalent clauses in a slightly extended form of Prolog. (1)" That program is in turn an extension of *An Implementation of Discourse Representation Theory* (Covington & Schmitz, 1988; henceforth C&S), "a program that constructs discourse representation structures from ordinary English input. (1)" It, in turn, is an extended version of the program described in Johnson & Klein (1986). (The extension to be described in Section 5 of this report is designated CONSTRAIN. Note that the abbreviations C&S and CNS&G will be used to refer both to the programs and to the documents describing them.)

Phrase Structure Rules

C&S is a top-down parser, written in definite clause grammar (DCG) notation, for a phrase structure (PS) grammar. The PS rules are augmented with a hold mechanism that allows the parsing of certain empty categories, and with a feature structure that allows syntactic and semantic information to be passed from one node of the parse tree to another. Although Kamp employs a top-down procedural algorithm, CNS&G view the PS-to-DRT mapping as a static relation defined in terms of unification of feature structures. (The CNS&G view follows that of Johnson & Klein (1986) and is in the spirit of Zeevat (1989)).

The addition of a hold mechanism and a feature structure, although necessary to give the program its power, can make the description of the parser complicated; the reader who is only concerned with the syntactic rules is likely to be confused by the presence of the feature unification rules. To avoid that confusion, this description will be presented in three passes. The first pass will treat the parser as if it consisted of DCG rules only, the second will add the hold mechanism to that description, and the final pass will treat the feature structure. Because of the interaction between the three, this neat simplification is not entirely possible. For the first two passes the reader must accept without explanation the fact that the semantic features carry the DRS for the portion of the discourse already processed.

C&S accepts discourses that conform to the PS rules shown below. (Parentheses indicate optional elements; the parser actually has separate rules for these possibilities. Words in brackets are terminal symbols.)

(4.1)	discourse	→	statement, (discourse).
	discourse	→	question, (discourse).
	statement	→	sentence.
	question	→	[does], np, vp.
	question	→	[is], np, adj.
	question	→	[is], np, np.
	sentence	→	np, vp.
	sentence	→	np, [does, not], vp.
	sentence	→	np, [is], adj.
	sentence	→	np, [is, not], adj.
	sentence	→	np, [is], np.
	sentence	→	np, [is, not], np.
	sentence	→	[if], sentence, [then], sentence.

The PS rules for VP and Adj are straightforward, as is shown in (4.2). (Form is a variable representing a terminal symbol.)

(4.2.a)	vp	→	v, np.
(b)	vp	→	v.
(c)	v	→	[Form].
(d)	adj	→	[Form].

As we shall see below, syntactic features prevent (4.2.c) from selecting a transitive verb for (4.2.b), or an intransitive for (4.2.a).

While the rules for VP are essentially those of Chomsky (1965), the NP rules make use of X-bar theory. NP is bar-3, and numeric suffixes indicate the other bar levels. As is the case for VP, syntactic features subcategorize Form for lexical insertion—proper or common noun. (Pronoun is a notational device; each gender of pronoun requires a separate rule. C&S is not sensitive to pronoun case.)

(4.3.a)	np	→	n.	(Proper noun)
(b)	np	→	[Pronoun].	
(c)	np	→	[].	(Empty category)
(d)	np	→	det, n2.	
(e)	n2	→	n1.	
(f)	n2	→	n1, relcl.	(NP with relative clause)
(g)	n1	→	n.	
(h)	n1	→	adj, n1.	
(i)	n	→	[Form].	

Rules (4.3.c) and (4.3.f) will be discussed with the hold mechanism. Rule (4.3.b) searches the incoming DRS for the most recent discourse referent that matches Pronoun in gender. (Remember that this version of DRT does not handle plural NPs.) Rule (4.3.h) is recursive, so n1 may contain any number of adjectives.

In order to handle empty categories, C&S adds to each phrasal rule two arguments, an input and an output hold list. When rule (4.3.f) is applied, the discourse referent of n1 is placed at the head of the input list and that list is passed to relcl. When (4.3.c) is applied while processing relcl, the first referent on the hold list is removed and assigned to the empty category. This procedure insures that empty categories are instantiated on a last-in-first-out basis. The rules for **discourse**, **statement** and **question** require that the hold lists be empty, since empty categories cannot be bound across sentence boundaries.

Unification of Feature Structures

The C&S parser incorporates a unification based grammar (Schieber 1986) that passes feature structures from one node of the parse tree to another. Since features are passed by means of unification, it is possible to pass a feature from one node to another before it has been instantiated. For example, rule (4.2.a) unifies the discourse referent of `np` with the second argument of `v` before either (4.2.c) or (4.3) is applied.

The use of feature structures is facilitated by the use of the GULP extension to Prolog (Covington 1987). GULP makes it possible for the programmer to refer to features by name, rather than by position in the feature structure. In consequence, a parser written in GULP is much easier to read than is one written in standard Prolog notation. The C&S parser uses syntactic features that constrain the parse procedure and semantic features that construct DRSs during the parse procedure.

The syntactic features employed by C&S are `syn:index`, `syn:class`, `syn:arg1` and, `syn:arg2`. `syn:index` is a unique integer that is generated for each noun in the discourse; its value is the discourse referent that is used to bind pronouns and empty categories. `syn:arg1` and `syn:arg2` are also integers; they are unified with the `syn:index` features of the nouns in a sentence and become the arguments of its predicates. The `syn:class` feature of a noun (proper or common) or verb (transitive or intransitive) is a subcategorization that serves to constrain the parse procedure, as was noted above.

The semantic features are `sem:in`, `sem:out`, `sem:res:in`, `sem:res:out`, `sem:scope:in`, `sem:scope:out`. GULP makes it possible to address an individual feature or a bundle of features; for example, a variable may be bound to `sem:res:in`, to `sem:res` (both `in` and `out` are bound), or to `sem` (all features are bound). Features may be cross-unified, so that `sem:res` of one node may be bound to `sem` of another node.

The value of a semantic feature is a list of one or more structures of the form `drs(U, Con)`, where `U` is a list of discourse referents and `Con` is a list of conditions. The first `drs/2` on the list is the representation currently being processed. Not all of the features are instantiated for every node; in fact, most nodes use only the `sem:in/out` features. Each parent node instantiates the `sem:in` of its daughter(s) to the DRS for the portion of the discourse already processed. The daughter then adds its own discourse referents to `U` and its conditions to Con, and returns the expanded `drs/2` as `sem:out` to its parent. (The initial `sem:in` for a discourse is `[drs([], [])]`.)

Consider, for example, (4.4), which gives the full form of (4.2.d) along with the rules that it calls.

```
(4.4.a)  adj(Adj) --> [Form],
                        {adjective_features(Form, Adj)}.
    (b)  adjective_features(Form,Adj):-
                append(Semantics, Con, NewCon),
                Adj = syn: (index:I) ::
                        sem: (in : [drs(U, Con)|Super] ::
                        out: [drs(U, NewCon)|Super]).
    (c)  adjective(big, lambda(I, [big(I)])).
```

The N1-rule that calls (4.4.a) unifies the `syn` features of `Adj` with those of the noun that the adjective modifies. It also unifies the `sem:in` feature of `Adj` with the appropriate feature of that noun (which feature it is unified with depends on the calling rule). The entire feature structure of `Adj` is then passed to (4.4.b).

Rule (4.4.b) unifies I with the `syn:index` feature of the modified noun and passes it to (4.4.c), which in turn unifies it with the argument of `Form`, the adjective being processed. The resulting structure is appended to `Con`, the condition list on the input DRS, and the result, `NewCon`, is passed in the output DRS.

The `sem:res` and `sem:scope` features are needed in order to handle universally quantified NPs. DRT treats a sentence with a universally quantified NP as an implication. The antecedent, the `sem:res` feature, consists of the information in the NP itself, information that `restricts` the set to which the NP refers. The consequent, the `sem:scope` feature, consists of the information in the remainder of the sentence, the scope of the NP. Scope is determined by the left-to-right ordering of NPs; the verb falls within the scope of all NPs, and the object NP is within the scope of the subject NP.

C&S follows Johnson & Klein (1986) in treating the determiner as the key element in determining quantification. Thus, the `sem` of a sentence is the `sem` of the subject NP, which in turn is the `sem` of the determiner of that NP. Consider (4.5).

```
(4.5)  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] }.
```

Let us assume that (4.5) is called by an NP-rule that is called directly by an S-rule (i.e. the NP in question is the subject of the sentence), and let us trace the feature unifications through the parse tree. At this point, the only feature that has been instantiated is `sem:in` (`A` in rule (4.5)), which contains the DRS for the preceding portion of the discourse. The NP-rule unifies its entire `sem` feature with that of Det, and the S-rule likewise unifies its `sem` feature with that of `NP`. The S-rule also unifies the `sem` of `VP` with the `sem:scope` of `NP`, from which it is passed to the `sem:scope` of `Det`. The NP-rule also unifies the `sem` of `N2` with sem:res of Det.

Rule (4.5) unifies `A` with `sem:in`, the incoming `DRS` list, and conses to `A` a DRS with empty universe and condition lists. The resulting list becomes `sem:res:in`, and therefore `sem:in` of `N2`. The N2-rule fills those empty lists and returns them as `B`, the `sem:res:out` of `Det`. (4.5) then prefixes an empty DRS to `B`, producing the `sem:scope:in` of `DET` and `NP`, and the `sem:in` of `VP`. The VP-rule fills the empty DRS and returns it as `sem:scope:out` of `NP` and `DET`. (4.5) extracts from `sem:scope:out` the first three DRSs: `Scope`, the now filled DRS from `VP`, `Res`, the now filled DRS from `N2`, and the first DRS in `A` (the input DRS list), `drs(U, Con)`. Finally, `Res` and `Scope` are made arguments to the functor `ifthen`, which is prefixed to `Con`. The result is the `sem:out` of `DET`, `NP`, and `S`.

Rule (4.5) is more complicated than most of the C&S rules, but all phrasal rules work in the same general fashion. The reader is referred to C&S for complete specification of the grammar; here we are concerned with the extensions to that program necessary in order to handle constrained clauses.

Prologization of a DRS

CNS&G adds to C&S procedures that translate the DRS in the `sem:out` feature of a discourse into Prolog clauses (Prologization) and then assert/process those clauses. Again, we are concerned primarily with those portions that must be modified or extended for the present implementation. The reader is referred to CNS&G for a complete specification of that program.

Before translating the set of conditions of the output DRS into Prolog, two "clean-up" steps must be performed. The lexical insertion rules of the parser provide two conditions for each noun in the discourse: the property denoted by the noun and the gender of the noun. The latter is needed only for anaphora resolution and is discarded before that set is processed.

The second clean-up step involves unifying equated discourse referents. The original C&S

parser assigns to each noun a unique integer, even in sentences such as (4.6):

`(4.6)` Pedro is a farmer.

The parser would then add a condition that equates the discourse referent of *Pedro* with that of *farmer*. The parser in CNS&G is modified to defer the assignment of integers to discourse referents to the Prologization module. That delay makes it possible to unify the discourse referents for (4.6) so that only one integer is assigned to them. With that unification accomplished, the universe of the DRS is no longer needed, and Prologization converts the list of properties/relations in the conditions into a list of Prolog clauses.

Prologization works its way through the list of conditions and produces a new list that is suitable for asserting or querying. If a condition is a simple property or relation, the output of a lexical rule, it is simply added to the new list. If the condition is `query(DRS)`, `DRS` itself has to be cleaned up, Prologized, and converted into a conjunction of clauses. If the condition is `ifthen(DRSA, DRSC)`, `DRSA` (the antecedent) and `DRSC` (the consequent) must be cleaned up and Skolemized before they can be Prologized and converted into conjunctions of goals. Skolemization involves binding each uninstantiated variable in DRSC that does not appear in `DRSA` to a list whose head is a unique integer and whose tail is the universe of `DRSA`. Skolemization insures that such variables have existential import and narrow scope.

Processing Prologized Clauses

The output of Prologization is a list of clauses, each in one of the following forms:

`(4.7.a)` Clause
`(b)` neg(Clause)
`(c)` (ClauseList1 ::- ClauseList2)
`(d)` query(Clause)

Processing (4.7.a) in the assert/query module is a simple matter of asserting Clause. CNS&G does not support negation; (4.7.b) is processed simply by noting that negation is not supported. The symbol ::- in (4.7.c) is a functor corresponding roughly to :- in Prolog; processing involves converting the list of clauses into a conjunction of Prolog terms and asserting them. Similarly, Clause in (4.7.d) is converted to a conjunction of Prolog terms and called, and the result is reported to the user.

Actually, CNS&G does not assert or query anything; it simply announces what it would do if it did. One of the first extensions to that implementation made by CONSTRAIN is to make the processing step work. In essence, this extension is a matter of making the program do what the CNS&G program says it would do. In the case of (4.7.b), `Clause` is not converted to a conjunction of terms; it remains a list, the single argument to `neg`.

This treatment of negation is rather cursory, but the treatment of negation is not the focus of this study. CONSTRAIN extends the C&S grammar to include disjunction, to which the processing step pays similar lip service. This limitation is due to the fact that a disjunction such as (4.8.a) is not a Horn clause and cannot be represented directly in Prolog (although a disjunction may appear in the body of a rule). (4.8.a) could be represented as (4.8.b) and (4.8.c), but it would be unwise to do so before the details of `neg/1` are worked out.

(4.8.a) p OR q
(b) p ::- neg(q)
(c) q ::- neg(p)
(d) query(p)

Further extensions to the grammar require more significant refinement of the assert/query module especially in the case of questions; we shall return to these after examining the extensions to the parser.

5 EXTENDING THE IMPLEMENTATION OF DRT

CONSTRAIN is divided, for ease of development, into eleven modules:

(5.1) DRT_LOAD.GLP
DRT_UTIL.GLP
DRT_PS1.GLP
DRT_PS2.GLP
DRT_PS3.GLP
DRT_LEX.GLP
DRT_RED_GLP
DRT_PRO1.GLP
DRT_PRO2.GLP
DRT_TRY.GLP
DRT.TEST.GLP

DRT_LOAD loads the program. Files that include feature structures make use of the GULP `load/1` command, which converts GULP feature notation into Prolog form. Those that do not use feature structures are `consult`ed because that procedure is faster. DRT_LOAD also defines the goal `loadred/0`, which is used to load the lexical redundancy rules in DRT_RED, a process that must be repeated every time the knowledge base is cleared with `newkb/0`. Placing all of the load commands in one file facilitates relocating the program. If, for example, the program is to be run from the `A`: drive, only DRT_LOAD need be edited.

DRT_UTIL contains general utility routines, such as list manipulation procedures, and procedures for displaying a DRS in a readable form.

DRT_PS1, DRT_PS2, and DRT_PS3 contain the phrase structure rules. DRT_PS1 contains lexical insertion rules: rules that make direct calls to the lexicon (DRT_LEX). DRT_PS3 contains rules that deal with whole clauses (discourse, question, statement). DRT_PS2 contains rules that rewrite intermediate structures.

DRT_LEX and DRT_RED contain the lexicon. The former contains the lexicon proper, while DRT_RED contains lexical redundancy rules, such as the rule that allows the program to infer that Bob is a man.

DRT_PRO1 and DRT_PRO2 contain the rules that apply the DRS output by the parser to a Prolog knowledge base. DRT_PRO1 contains the Prologization module that translates the conditions of the DRS into Prolog clauses, and DRT_PRO2 contains the assert/query module that asserts these clauses to the knowledge base or calls them as queries to that knowledge base.

DRT_TRY and DRT_TEST contain the user interface. DRT_TRY contains the rules that accept an input and send it to the program for processing, while DRT_TEST contains a test suite that calls DRT_TRY for processing.

Knowledge Representation

Most of the changes to C&S involve the addition of PS rules to DRT_PS2 or DRT_PS3. These additions, of course, entail concomitant modifications to the other modules. These modifications, however, are based on a more fundamental modification, a change in the form of knowledge representation in DRT_LEX.

In Kamp's version of DRT, conditions are represented as predicate-argument structures: `Pred(Arg)` or `Pred(Arg1, Arg2)`. `Pred` is the natural language word (noun, verb, or adjective), and `Arg` is a discourse referent. C&S follows that representation, with one modification: rather than being treated as predicates, proper nouns are second arguments to the predicate `named/2`. Thus, (5.2.a) is represented by the conditions in (5.2.b).

(5.2.a) Bob kisses Alice.
(b) [named(1, bob), named(2, alice), kiss(1, 2)]

This representation poses several problems for the design goals of CONSTRAIN: First, with such a representation it is awkward to distinguish between events and states. Although that distinction plays a relatively minor role in this implementation, it could play a major role in an extension that incorporates Guenthner's (1987) proposals.

Second, as was noted in Section 3 of this report, some constrainers behave differently depending on the tense of the sentence, so it is necessary to include tense in the representation. It would, of course, be possible to add an argument for tense to the structure shown above, but such a solution would be, at best, unesthetic.

Third, in order to query a knowledge base using the format shown above, it is necessary to know the predicate in advance. One can ask who Bob kisses, but one cannot ask what Bob does. Finally, the use of redundancy rules is very awkward with this format. Given (5.3.a), one would like to get an affirmative response to (5.3.b).

(5.3.a) Bob knows that Ted kissed Alice.
(b) Does Bob believe that Ted kissed Alice?

Using the knowledge representation in (5.2.b), it is necessary to have a separate rule for each verb that entails *believe.*

In order to overcome these difficulties, this program reifies natural language predicates so that they become arguments to Prolog predicates. (This, in effect, is the reverse of Kamp (1981), who uses proper nouns as predicates.) Thus, the conditions in a DRS have one of the four forms in (5.4).

(5.4.a) named(Index, Name)
(b) isa(Noun, Index)
(c) event(Tense, Event, [Arg1|Rest])
(d) state(Tense, State, [Arg1|Rest])

Event is an action verb, while *State* is a stative verb or an adjective. Representing the argument(s) to a natural language predicate as a list makes it unnecessary to have separate rules for addressing one- and two-place predicates.

Representations (5.4.a,b) are adequate for the current implementation, which assumes that the name and/or class of an entity does not change during the time covered by a discourse. If the program is to cover greater periods of time, tense arguments must be added to these predicates in order to account for (5.5.a, b), for example.

(5.5.a) Bob was a boy.
(b) Bob is a man.

For affirmative statements, the addition of time reference to the knowledge representation requires, at first glance, only a simple modification of the lexical rules. That is, the C&S rule in (5.6.a) need only be changed to that in (5.6.b) and an additional rule be added for saw.

```
(5.6.a)  transitive_verb(sees,
                lambda(A1, A2, [see(A1,A2)]))
      (b)  transitive_verb(sees,
                lambda(pres,A1, A2,
                     [event(pres, see, [A1,A2])])])
```

For negatives and questions, however, tense is determined by the form of the auxiliary, not that of the main verb. It is necessary, therefore, to add a syntactic feature, syn:tense, to the GULP feature structure in order to pass tense from the clause-level rule to the lexical rules. Thus, for example, the question rule of (4.1), repeated as (5.7.a), is modified to the form of (5.7.b). (For clarity of exposition, hold arguments are not shown, and only those feature unification rules that are relevant to the present discussion are displayed.)

```
   (5.7.a)  question(Q) --> [does],np(NP),vp(VP).
       (b)  question(Q) --> {DO = syn:tense:Tense,
                             VP = syn:tense:Tense,
                             NP = syn:tense:Tense},
                              do(DO), np(NP), vp(VP).
```

(5.7.b) requires the addition of a lexical insertion rule for `do` and lexical rules for *do* and *does*. The variable `Tense` is bound in the `do` rule, and its value is passed, by way of `VP`, to the lexical rule for the verb. Similar modifications are needed for the PS rules that include the copula *is*, passing tense to the subject complement.

(5.7.b) does not pass `syn:tense` only to `VP`; it also unifies that feature with the corresponding feature in `NP`. Although nouns do not have tense (at least, in the present system, but see the discussion of `isa/2` above), nouns may be modified by attributive adjectives—states that require tense arguments—which receive their tense from the matrix clause. Thus, every clause-level rule must pass a `tense` feature to its subject, and a VP rule must pass that feature to the direct object.

Even within the limits noted above, the addition of time reference to the knowledge representation introduces a further difficulty: the frame problem. Given that a state obtained in the past, does it continue into the present? For the human processor of natural language, the frame problem is rarely a problem. He knows which states can be assumed to be permanent (in the absence of evidence to the contrary) and which can be assumed to be momentary. For those states that fall in between, he is able to assign a probable length of duration. This assignment is based on real-world knowledge of the state and the participants. Acquiring and programming such knowledge, however, is a daunting problem for artificial intelligence. As is the case for negation, CONSTRAIN adopts a simplistic approach: a state that was true in the past remains true in the present unless its negation in the present has been asserted. This approach is encoded in a meaning postulate in the file DRT_RED:

```
   (5.8)  state(pres, Verb, Args) :-
```

```
                    state(past, Verb, Args),
                    not neg([state(pres, Verb, Args)])
```

Proper Nouns

A further extension that is independent of processing constrained clauses is the treatment of proper nouns. Kamp (1981) assigns a discourse referent to each proper noun, and then equates those referents that are arguments to the same name; thus, there can be only one individual for each name. CNS&G take the opposite approach, allowing for more than one individual with a particular name. Each proper noun is assigned a unique integer, and there is no way to show that two instances of *Bob* refer to the same individual.

CONSTRAIN takes a middle road. Within a discourse (a list of words and punctuation marks), all instances of a proper noun are assumed to refer to the same individual. The first rule for a proper noun treats it as a pronoun, seeking a match in the input universe. If a match is found, the new instance is assigned the same discourse referent; otherwise a new discourse referent is generated.

Between discourses that apply to the same knowledge base, however, the program allows for the possibility that a new instance of a name refers to a different individual. Before the input string is sent to the parser, the rule `preprocess/2` creates a list of the proper nouns in the input, eliminates duplication, and queries the user about each of the names that appears in the knowledge base. If the user responds that this is the same (i.e. last mentioned) or the previous individual of that name, the appropriate `named/2` and `gender/2` propositions are placed in the initial DRS, `K0`; otherwise the parser will generate a new discourse referent for the name. To allow correct application of this procedure, when the assert/query module of DRT_PRO2 encounters a `named/2` clause it first retracts that clause, if it is present in the knowledge base, and then adds it with `asserta/1` to insure that it will be the next such clause to be accessed.

(Note that pronominal anaphora can only be resolved within a discourse. In order to address a discourse referent in the knowledge base, it is necessary to refer to that individual by name.)

Answering a Query

A final modification to CNS&G that is necessary for the handling of constrained clauses but also applies to other clauses is an extension of the mechanism for querying the knowledge base. A query in CNS&G can only ask a *yes/no* question: given (5.9.a), one may query (5.9.b), but not (5.9.c,d).

(5.9.a) Bob kissed Carol.
(b) Did Bob kiss Carol?
(c) Who kissed Carol?
(d) Whom did Bob kiss?

The first problem is to find a place to put the answer. The CNS&G program would handle questions by converting the list of queried propositions to a conjunction in the Prologization step and passing that conjunction to `assert_or_process/1`, That predicate in turn calls `test(Goal, Result)` and prints `Result`. `test/2` uses the built-in predicate `call/1` to test that conjunction. `Result` is bound to 'yes' if the call succeeds and 'no' if it fails.

The first step in this extension to that program is to expand the possibilities for `Result`. `test(Goal, Result)`, after displaying the goal that is being queried, calls `test_list(Goal, Result)`. If the latter call fails, `Result` is bound to 'INSUFFICIENT DATA'; otherwise it returns the value assigned by `test_list/2`.

`test_list/2` tries three methods for assigning a value to `Result`. The first method is that of CNS&G, converting the list of propositions to be queried into a conjunction and calling that conjunction; if the call is successful, `Result` is bound to 'AFFIRMATIVE.' The next method checks the knowledge base for `neg(Goal)`; if that check succeeds, `Result` is bound to 'NEGATIVE.' Thus, this implementation replaces Prolog's negaQtion-as-failure with true negation. However, this procedure is not complete; a query will return 'NEGATIVE' only if an identical list of propositions exists as an argument to `neg`. Consequently, the response to (5.10.b) will be 'INSUFFICIENT DATA.'

```
(5.10.a)  Bob did not kiss a woman.
          named(p0, bob),
          neg([isa(X, woman), event(past, kiss, [p0, X])])
     (b)  Did he kiss a pretty woman?
          query([isa(Y, woman), state(past, pretty, [Y]),
                event(past,kiss,[p0, Y]) ])
```

If neither of the above methods succeeds, `test_list/2` enters a recursive loop that calls `query(Goal, Result)` on the head of the list of queries and than calls `test_list/2` on the remainder of the list. This recursion is necessary in order to handle *wh*-questions.

At first glance, the treatment of *wh*-questions might seem a trivial matter. One need only leave the discourse referent for the interrogative word unbound, so that `query/2` will seek to bind it. However, the CNS&G parser leaves the discourse referents of `all` common nouns unbound until the clean-up step is reached, at which point it is impossible to distinguish the referents that should be bound from those that should remain free. To overcome this difficulty, the CONSTRAIN parser unifies the pertinent `syn:arg` with the atom `wh`. A query containing `wh` will not unify with any clause in the knowledge base, so the first two methods for `test/2` will fail for such a clause. (Note that the treatment of interrogative *who/who* is quite distinct from that for relative *who/whom*, which unifies the `syn:index` of the relative pronoun with that of its antecedent.) When `query/2` encounters such a clause, it uses the built-in predicate `set_of/3` on a copy of the clause with `wh` replaced by a variable. It then uses `get_id_list/2` to produce a list of names of those discourse referents or, if they are not named, of their classes. `Result` is then bound to that list.

The recursion on `test_list` terminates when one of three conditions is met:

(5.11.a) The input list is empty and `Result` is bound.

(b) The input list is empty and `Result` is free (it is neither 'NEGATIVE' nor a discourse referent). `Result` is then bound to 'AFFIRMATIVE.'

(c) The input list can be satisfied by one of the non-recursive methods. `Result` is then bound to that `Result`.

Note that if `Result` is bound by `query/2`, `test_list(Goal, Result)` cannot unify with the non-recursive methods.

Constrained Clauses: Say Class

The extensions to CNS&G described above provide the tools necessary for the addition of constrained clauses to the language handled by the extended program. The first step in adding such clauses is the development of a representation of the embedded clause, an implementation of the proposition markers used by Spencer-Smith (1987) and Asher (1986). The representation chosen is an additional knowledge base predicate, `prop(Index, Clause)`. This structure is created by the following PS rule:

```
(5.12)  nouncl(NC, [], []) -->
             { NC = syn:index:Index,
               NC = sem:in:A,
               S  = sem:in:[drs([],[])|A],
               S  = sem:out:[DRS,drs(U, Con)|Super],
               NC = sem:scope:in:[drs([Index|U],
                                      [prop(Index,DRS)|Con])|Super],
               NC = sem:scope:out:DRSList,
               NC = sem:out:NewDRSList },
               [that], s(S, [], []),
             { add_to_topmost_drs(Index,[prop(Index, DRS)],DRSList,
                                  NewDRSList) }.
```

The hold lists in (5.12) must be empty in order to prevent an empty category's referring to an element outside its governing category. Like the rules for quantified NPs and negated sentences, (5.12) first `cons`es an empty DRS to the input DRS; that DRS is filled in as `S` is processed, becoming `DRS`. The remaining NC unification rules convert `Index` and `DRS` into a `prop/2` structure and create the output. After the DCG rule has instantiated those bindings, the rule `add_to_topmost_drs/4` elevates `Index` and the `prop/2` structure to the toplevel DRS.

The *nouncl* rule is called by the rules VP → V NC and S → it BE (not) Adj NC, which bind the `syn:class` feature of V to 'mental.' The 'mental' feature contrasts with the features 'transitive' and 'intransitive,' in the case of verbs, and 'common,' in the case of adjectives. The `syn:class` feature in a lexical insertion rule determines which lexical rule will be called, and the distinction must be reflected in the lexicon.

These modifications allow the program to handle clauses constrained by the Say Class of constrainers. Such clauses are simply added to the DRS and are shielded from further truth evaluation. Thus, the program at this point will properly handle the discourse in (5.13). (Responses to queries are shown below the queries.)

`(5.13.a)` Bob said that Ted kissed Alice.

`(b)` Did Bob say that Ted kissed Alice?
AFFIRMATIVE

`(c)` Who said that Ted kissed Alice?
bob

`(d)` Did Ted kiss Alice?
INSUFFICIENT DATA

`(e)` What did Bob say?
event(past, kiss, [ted, alice])

Regret Class Constrainers

In order to extend the set of constrainers that the program handles to the Regret Class, the PS rule that calls the nouncl-rule must call `check_factive(NC, X)` after the DCG rule completes feature bindings. NC is the feature structure of the constrained clause (actually, only the `sem:out` feature is used), and `X` is that of the constrainer's category (verb or adjective). `check_factive/2` tests the constrainer with `factive(Constrainer)`, a redundancy rule in DRT_LEX. If the test succeeds, `add_to_topmost_drs/4` elevates the discourse referent(s) and condition(s) of the constrained clause to the toplevel DRS and the `sem:out` feature of `X` is bound to the result. If the test fails, the `sem:out` feature of `X` is bound to that of NC. Crucially, `check_factive/2` makes no distinction between positive and negative constrainers, although different rules are necessary to handle the two structures.

Given this addition to the program, if *say* in (5.13) is replaced by *regret*, the response to (5.13.c) is AFFIRMATIVE. Thus, the program is able to handle Regret Class constrainers properly. Since the Know Class differs from the Regret Class only for first-person subjects, and since this parser does not handle first-person NPs, the latter class is subsumed by the former; in fact, three of the five factive constrainers included in the program are effectively in the Know Class. The parser is also limited to constrained clauses in the *that S* form, and Suspect Class constrainers are factive only in the *poss-ing* form, so that class is omitted from consideration.

Implicational relationships between *know* on the one hand and *think* and *believe* on the other are captured in lexical redundancy rules in DRT_RED. Another redundancy rule takes a rather optimistic view of learning: that if someone has learned something, he knows it. With these additions, the program will handle the following discourse:

`(5.14.a)` Carol learned that Bob kissed Alice.

`(b)` Did Bob kiss Alice?
AFFIRMATIVE

`(c)` Does Carol know that Bob kissed Alice?
AFFIRMATIVE

`(d)` Does Carol believe that Bob kissed Alice?
AFFIRMATIVE

True Class Constrainers

As Kiparsky and Kiparsky (1970) note, the truth contribution of a clause embedded under a factive verb depends only on the immediate constrainer; depth of embedding is irrelevant. It is for this reason that `check_factive/2` can work in the parser, even though depth of embedding is not available at the time the rule is called. In fact, if the treatment of such clauses were deferred, the constrainer might be buried in a list (e.g. within `prop/2`) and not available for examination. That independence does not hold for clauses constrained by True Class constrainers. Such clauses are true only if the constrainer is true (i.e. a toplevel assertion or the complement of a factive), and the constrainer's truth value is not available when the constrainer is being parsed. Furthermore, the effect of a True Class constrainer is dependent on its polarity, and that informa tion also may not be available when the constrainer is being parsed. Thus, the treatment of complements of such constrainers must be deferred to the processing step of the program.

CNS&G uses the predicate `note/1` to assert the output of the Prologization step to the knowledge base. That rule would simply assert its argument. CONSTRAIN expands `note/1` considerably so that it tests its argument before asserting it. One modification that has gone unremarked is that `note/1` first attempts to retract the clause before asserting it, then uses `asserta/1` rather than `assertz/1`, as in CNS&G. These steps eliminate duplication in the knowledge base and insure that the most recently evoked clauses are at the head of the knowledge base. It is the latter feature that allows proper nouns to be identified across discourses. A further modification that is made necessary by the addition of True Class constrainers is the elimination of double negation; if the argument to `neg/1` is itself a `neg/1`, the argument of the inner negation is processed and asserted.

The only clauses that reach `note/1` for processing are those that are not shielded from truth evaluation—initially, toplevel assertions and conditions that have been raised to toplevel by `check_factive/2`. Implications, including those created from universally quantified NPs, have been converted into proper Prolog rules and are subject to the constraints imposed by the interpreter. Clauses that constitute the conditions of negations and of embedded clauses remain in a list, an argument to `neg/1` or `prop/1`, and are not sent to `note/1` on their own.

The representation of a True Class constrainer has the form `state(Tense, State, [X])`, where `State` is the constrainer and `X` is the discourse referent for the embedded `prop/1`. When `note/1` encounters a clause of that form, it calls the disjunctive query `(true(State); certain(State))`. These predicates, similar to `factive/1`, identify those constrainers whose complements are to be asserted to the knowledge base when the constrainer is positive. If the call succeeds, `prop(X, PropList)` is called and PropList is Skolemized and processed. The original argument to `note/1` is also asserted.

Processing `PropList` may bring a new True Class constrainer to `note/1`. Thus, a clause embedded successively under True Class constrainers will eventually come to `note/1` to be asserted. However, if a Say Class constrainer intervenes, the sequence will be blocked. Thus, (5.15.c) will receive AFFIRMATIVE if (5.15.a) has been entered, but INSUFFICIENT DATA if only (5.15.b) has been entered.

(5.15.a) It is true that it is certain that Ted kissed Carol.
(b) It is true that it is possible that Ted kissed Carol.
(c) Did Ted kiss Carol?

Since `PropList` is a list of conditions, there is no universe to provide the set of discourse referents to be Skolemized, so `set_of_args(PropList, Args)` must be called to collect those referents. Actually, this predicate was already necessary. This program, like C&S, raises the discourse referents of proper nouns to toplevel during parsing. CNS&G does not do so, so these referents are available in the subordinate universe when an implication is Skolemized. Thus, this program must call `set_of_args/2` before Skolemizing an implication.

`note/1` follows a similar procedure in the case of `neg([state(Tense, State, [X])])`. In this case, however, the disjunctive query is `(true(State); possible(State))`; which identifies those constrainers for which, when negated, the negation of the complement should be asserted.

`note/1` is also used to handle the discourse shown in (5.16), the last of the extensions to CNS&G.

```
(5.16.a)  Alice thinks that Ted kissed Carol.
     (b)  Alice is right.
     (c)  Did Ted kiss Carol?
          AFFIRMATIVE
```

Because *think* is a Say Class constrainer, (5.16.c) will receive INSUFFICIENT DATA if it only has (5.16.a) to work on. In order to handle (5.16.b), it is necessary to find what *she* is right about and process that proposition appropriately. The following clause for `note/1` accomplishes that.

```
(5.17)  note(state(Tense, Truth, [X])):-
               (Truth == right ; Truth == wrong),
               (event(Tense, _, [X,P]) ;
                    state(Tense, _, [X,P])),
               prop(P, Prop),
               note_truth(Prop, Truth).
```

If Truth is bound to 'right' or 'wrong,' `note/1` seeks an event or state whose subject is `X` and whose complement is an embedded clause. It then passes `Prop` and `Truth` to `note_truth/2`, which asserts `Prop` or its negation, according to the value of `Truth`. If the clause to be asserted already exists in the knowledge base, `note_truth/2` simply advises the user; if the opposite of the clause exists there, it is retracted before the new clause is asserted. This is a rather simple-minded treatment of belief revision, but CONSTRAIN assumes that anything entered by the user is true unless it is constrained.

6 CONCLUSION

This report has presented an extension of Discourse Representation Theory that allows the representation of embedded noun phrases whose truth evaluation demands special treatment (constrained clauses), treatment that is determined by the constrainer (verb or adjective) of the embedded noun phrase. It has also described an implementation of DRT, CONSTRAIN, that takes an English language input (which may contain such clauses) and transforms it into a Prolog knowledge base that can be queried. Both the theory and the implementation remain at the experimental stage; both must be extended considerably before they can be used as a natural language interface for practical programs. In this section we consider some of the necessary extensions.

The most needed extension to the implementation is a proper treatment of negation. CONSTRAIN replaces the negation-as-failure of Prolog with true negation, but in so doing it trades one form of unsoundness for another. CONSTRAIN will return NEGATIVE as the response to a query only if the knowledge base contains a clause `neg(Query)` such that the elements in the query exactly match those in `Query`. Thus, CONSTRAIN may return INSUFFICIENT DATA to a query when the knowledge base contains information that entails the response NEGATIVE.

With a more complete treatment of negation, proper treatment of disjunction may be added. CONSTRAIN can parse a disjunction and add it to the knowledge base, but it is unable to use those disjunctions in order to infer conclusions. As was noted in Section 4, a more complete treatment of negation will make possible a more adequate treatment of disjunction.

Improved treatment of negation and disjunction are matters of the DRT-to-Prolog portion of CONSTRAIN. An extension needed at the other end of CONSTRAIN, the parser, is the ability to handle other syntactic types of constrained clauses: *poss-ing* and *for-to* complements. That extension is made difficult by the fact that either construction may involve *equi-NP deletion*. Given such an extension to the parser, it should be possible to add other constrainers that take VP complements to the language handled by CONSTRAIN.

Other needed additions to the parser are the facilities to handle subordinating conjunctions and sentence adverbials (e.g. *probably, certainly*). The former require a knowledge representation that allows non-constrained clauses to be addressed in a manner like those that are arguments to `prop/2`. One possibility for such a representation is the event markers of Guenthner (1987). Handling sentence adverbials is a relatively simple but time-consuming matter. The machinery for DRS construction already exists in CONSTRAIN, but it will be necessary to add to each sentence rule a new version for each possible adverbial position.

Extensions of the implementation will, of, course, be required by extensions to the theory. The current version of DRT is limited to a very small subset of natural language. Many extensions to the theory are required before it can handle a useful range of natural language. The most obviously needed extensions are the ability to handle definite NPs and plural NPs. Although definite NPs are usually anaphoric and thus could be treated in the same fashion as pronouns, the head noun of a definite NP is often not identical to that of its antecedent (e.g. the two may be synonyms). Furthermore, definite NPs are not infrequently exophoric or generic, and an implementation would need to have some means of distinguishing between the possibilities.

Another needed extension to the theory is a formalism for specifying arguments that are not syntactically indicated (e.g. *sell* requires, conceptually, a *price*, but that argument is not syntactically obligatory). Handling this argument requires default specifications and a means of determining whether the default value applies.

These extensions, both to the implementation and to the theory, probably require user interaction. The implementation may need to query the user about the scope of negation, non-anaphoric definite NPs, and default values. With these extensions, DRT can provide a useful natural language interface to a knowledge base.

References

[1] Asher, N. 1986. Belief in discourse representation theory. *Journal of Philosophical Logic* 15:127–89.

[2] Chomsky, N. 1965. *Aspects of the Theory of Syntax.* Cambridge, MA: MIT Press.

[3] Covington, M. 1987. *GULP 1.1: An extension of Prolog for unification-based grammar.* ACMC Research Report 01–0021, The University of Georgia, Athens.

[4] Covington, M., D. Nute, N. Schmitz, and D. Goodman. 1988. *From English to Prolog via Discourse Representation Theory.* ACMC Research Report 01–0024, the University of Georgia, Athens.

[5] Covington, M. and N. Schmitz. 1988. *An Implementation of Discourse Representation Theory.* ACMC Research Report 01–0023, The University of Georgia, Athens.

[6] Goodman, D. 1988. An Implementation of and extension to discourse representation theory: Translating natural language to discourse representation structures to Prolog clauses. Unpublished master's thesis, The University of Georgia, Athens.

[7] Guenthner, F. 1987. Linguistic meaning in discourse representation theory. *Synthese* 73:569–98.

[8] Guenthner, F., H. Lehman, and W. Schonfeld. 1986. A Theory for the representation of knowledge. *IBM Journal of Research and Development* 30:1.39–56.

[9] Johnson, M., and Klein, E. 1986. *Discourse, Anaphora, and Parsing.* CSLI Research Report 86–63, Stanford University.

[10] Kamp, H. 1981. A Theory of truth and semantic representation. In J. Groendendijk, T. Janssen, and M. Stokhof (eds.) *Formal methods in the Study of Language,* 277–322. University of Amsterdam.

[11] Kamp, H. 1985. Unpublished discourse representation theory project description, University of Texas, Austin.

[12] Kiparsky, P., and C. Kiparsky. 1971. Fact. In D. Steinberg and L. Jakobovits (eds.), *Semantics*, 345–369. New York: Cambridge University Press.

[13] McCawley, J. 1981. *Everything that Linguists have Always Wanted to know about Logic.* Chicago: The University of Chicago Press.

[14] McCawley, J. 1988. *The Syntactic Phenomena of English.* Chicago: The University of Chicago Press.

[15] Shieber, S. 1986. *An Introduction to Unification-based Approaches to Grammar.* CSLI Lecture Notes No. 4, Stanford University.

[16] Spencer-Smith, R. 1987. Semantics and discourse representation. *Mind and Language* 2:1.1–26.

[17] Smith, W. 1989. *Problems in applying discourse representation theory.* Research Report AI–1989–04, The University of Georgia, Athens.

[18] Zeevat, H. 1989. A compositional approach to discourse representation theory. *Linguistics and Philosophy* 12:95–131.