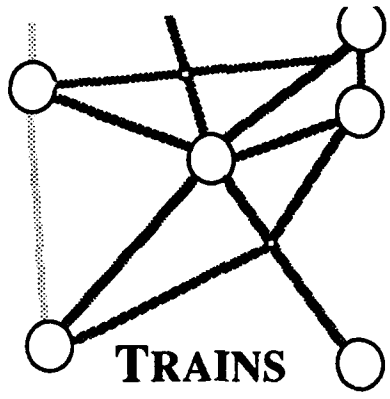
AD-A256 898



The Discourse Reasoner in TRAINS-90

David R. Traum

DTIC
ELECTE
OCT 16 1992
S D

TRAINS Technical Note 91-5
May 1991

92 1 049

92-27193

UNIVERSITY OF
ROCHESTER
COMPUTER SCIENCE

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

The Discourse Reasoner in TRAINS-90

David R. Traum

The University of Rochester
Computer Science Department
Rochester, New York 14627

TRAINS Technical Note 91-5

May 1991

Abstract

This note describes the Discourse Reasoner module of the TRAINS conversation system. The overall purposes and functioning of the module is described, as well as the general conversation tracking theory, and a guide to the code of the current system. A trace of the discourse reasoner in action on a sample dialogue fragment is also given.

This material is based upon work supported by ONR/DARPA under Research Contract number N00014-82-K-0193 and the National Science Foundation under Grant number IRI-9003841. The Government has certain rights in this material.

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: TN 91-5
2. GOVT ACCESSION NO.:
3. RECIPIENT'S CATALOG NUMBER:

4. TITLE *(and Subtitle)*: The Discourse Reasoner in TRAINS-90

5. TYPE OF REPORT & PERIOD COVERED: technical note

6. PERFORMING ORG. REPORT NUMBER:

7. AUTHOR(s): David R. Traum

8. CONTRACT OR GRANT NUMBER(s): N00014-82-K-0193

9. PERFORMING ORGANIZATION NAME AND ADDRESS:
Computer Science Dept.
University of Rochester
Rochester, NY, 14627, USA

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS:

11. CONTROLLING OFFICE NAME AND ADDRESS:
DARPA
1400 Wilson Blvd.
Arlington, VA 22209

12. REPORT DATE: May 1991

13. NUMBER OF PAGES: 23 pages

14. MONITORING AGENCY NAME & ADDRESS *(if different from Controlling Office)*:
Office of Naval Research
Information Systems
Arlington, VA 22217

15. SECURITY CLASS. *(of this report)*: unclassified

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE:

16. DISTRIBUTION STATEMENT *(of this Report)*:
Distribution of this document is unlimited.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*:

18. SUPPLEMENTARY NOTES:
None.

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*:
natural language understanding; speech acts; dialog systems

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*:

This note describes the Discourse Reasoner module of the TRAINS conversation system. The overall purposes and functioning of the module is described, as well as the general conversation tracking theory, and a guide to the code of the current system. A trace of the discourse reasoner in action on a sample dialogue fragment is also given.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

1 Introduction: The TRAINS Conversation System

The TRAINS project involves research in a wide range of areas of natural language and knowledge representation, including natural (spoken and written) language understanding, dialogue and discourse modelling, and real-time plan-guided execution. A centerpiece of this research is the design and implementation of a system which serves as a planning assistant to a human manager, communicating with the manager in an English conversation, with the duty of sending execution directives to agents in the simulated world to achieve the goal of the manager. More detailed information on the design and scope of the system can be found in [Allen and Schubert, 1991].

1.1 The Discourse Reasoner

The discourse reasoner is responsible for maintaining the flow of conversation, and making sure that the conversational goals are met. For this system, the main goal is that an executable plan which meets the manager's goals is worked out and agreed upon by both the system and the manager and then that the plan is executed.

The discourse reasoner must keep track of the manager's current understanding of the state of the dialogue, and determine the intentions of utterances of the manager, generate utterances back, and send commands to the planner and executor when appropriate.

1.2 Other Related Modules

The **Natural Language Subsystem** is responsible for producing a semantic interpretation of an input sentence. This interpretation is a statement or statements in Episodic Logic [Schubert and Hwang, 1989]. Parsing, semantic interpretation and reference resolution and inference are steps in the interpretation process. The NL Subsystem needs to process both the manager's and the system's utterances, in order to update reference and tense structures. The manager's processed utterances are passed over to the Discourse Reasoner for speech act analysis and further action, if necessary. See [Schubert, 1991; Light, 1991] for further information on the NL subsystem.

The **NL Generator** takes speech acts produced by the discourse system and converts them to Natural Language utterances which are then output to the user (and fed back to the NL subsystem). The current generator is just a stub module which can only produce a couple precomputed sentences. It is described in appendix A.

The **Domain Planner** does plan recognition and plan elaboration. Plan recognition is a crucial part of understanding utterances of the manager, and plan elaboration produces the new information that needs to get communicated to the manager. The domain planner is described in [Ferguson, 1991].

The **Executor** takes a plan and sends the necessary commands to the individual agents to have that plan carried out. It also monitors the progress of the plan to make sure it is successful. Plan execution occurs in the simulated TRAINS world, which is described in [Martin and Miller, 1991]

1.3 Overview

Section 2 describes the tasks of the Discourse Reasoner in detail. Section 3 describes the architecture used in this system for achieving these tasks. Section 4 describes the actual implementation. Section 5 describes the next stages needed to improve the current system. Section 6 gives a detailed trace of the system operating on a sample dialogue. The appendices describe the workings of closely related stub modules.

2 The Tasks of the Discourse Reasoner

The Discourse Reasoner has several tasks that it must fulfill. First, it must recognize what intentions are being communicated by an utterance of the manager. This involves analyzing the semantic interpretation of the utterance and trying to identify illocutionary acts that the manager meant to convey.

Once the acts are recognized, they must be responded to. At a minimum, the Discourse Reasoner must update the conversation knowledge context to include the new information, but further action may also be warranted. When a suggestion or request about a domain plan is recognized, plan recognition should also be performed, in order to capture any implicit suggestions. The Discourse Reasoner should also acknowledge utterances which it thinks it understands, and accept or reject suggestions and requests made by the manager.

In addition, the Discourse Reasoner must tell the executor to execute a plan when a complete plan has been agreed upon by the manager and the system, and must determine what needs to be said to the manager, based on the structure of the conversation and private beliefs and obligations.

2.1 Identifying Intentions from Utterances

The first job of the Discourse Reasoner is trying to recognize what the manager meant by a particular utterance. While the NL subsystem derives semantic interpretations of the utterances, it is up to the Discourse Reasoner to recognize how they apply to the conversation at hand and what to do about them. The first step in this process is to identify illocutionary acts from the interpreted utterance. The types of acts handled by the prototype system are shown in table 1.

suggest item	a proposal that can be related to a domain plan or to the conversation itself.
request item	like a suggestion, but introduces an obligation on the hearer to address it (either by performing a requested action, or accepting or rejecting the request explicitly).
accept item	accept a proposed item.
reject item	reject a proposed item.
release-turn	a signal of the end of the current turn and a willingness for the system to begin speaking.

Table 1: Speech Act Types

The items mentioned in Table 2 can be any of the types listed in Table 2 or **(adopt p)** which is a meta-plan item which means adopt plan **p** as shared between the participants.

The domain plan items have the following meanings: **(goal g p)**, where **g** can be a state to be achieved or an action to be performed, means that in order for **p** to be a successful plan, goal **g** must be met. **(action-in a p)** means that **a** is an action (e.g. move-oranges or unload) performed in the course of executing plan **p**. **(uses o p)** means that **o** is some domain object (e.g. a particular factory or car) which is to be used somehow in plan **p** (e.g. as a parameter in some action). **(fact c p)** means that **c** is some fact (e.g. (at (oranges o1) (city cityB))) about the world which is a constraint on plan p. See [Ferguson, 1991] for more on the meanings and uses of the domain plan items.

(goal g p)	**g** is a goal of plan **p**
(action-in a p)	**a** is an action in plan **p**
(uses o p)	**o** is an object used in plan **p**
(fact c p)	**c** is a constraint on plan **p**

Table 2: Domain Plan Items

Some example interpreted illocutionary acts would be:

- (suggest (fact (at (oranges o1) cityI) plan1))
- (request (action-in (move-oranges [parameters]) plan1))
- (accept (adopt plan1))
- (release-turn)

where *(at (oranges o1) cityI)* is a fact about the TRAINS world, saying that the group of oranges *o1* is at *cityI*, *move-oranges* is an action in that world, and plan1 is an abstract plan in the TRAINS domain. Further examples can be seen in the example traces in section 6.

Some parts of the speech act can be read directly from the utterance, such as the particular action or fact, and often the type of act. Other parts will often have to be filled in by the context, such as which plan is currently under discussion, and what an acceptance (e.g. "Okay") or a rejection (e.g. "no, let's not") refers to.

Multiple speech acts can be the result of a single utterance. For instance, the utterance, "Shall we ship the oranges?" in the trace given in Section 6 is interpreted as being a suggestion of (1) a particular action being part of the current plan, (2) a request to adopt the plan proposed by the speaker, and (3) a release of the current turn.

2.2 Tracking a Plan Through a Conversation: The Domain Plan Contexts

The Discourse Reasoner must maintain the knowledge about the state of the conversation and how it relates to the topic being discussed. For the TRAINS system, the main object of discussion is the TRAINS world, and plans for bringing about different situations therein. The manager must get the system to have a plan to carry out his goals, since the system has the only connections to the agents. Also, the system must try to satisfy the needs of the manager (its overall design motivation). It is thus important that the system and manager

Availability Codes
Dist | Avail and/or Special
A-1

must come to a shared plan of action. By "shared plan", we mean, roughly, shared belief in a system intention to execute the plan, rather than a more intricate definition such as that in [Grosz and Sidner, 1990].

2.2.1 Domain Plans

From the point of view of the Discourse Reasoner, Domain Plans are abstract entities which contain a number of parts. These include: the goals of the plan, the actions which are to be performed in executing the plan, objects used in the plan, and constraints on the execution of the plan. The composition of plans are negotiated by the System and the Manager to come up with an agreement on an executable plan, which can then be carried out. Seen this way, the conversational participants can have different ideas about the composition of a particular plan, even though they are both talking about the "same" plan. See [Ferguson, 1991] for details of domain plans.

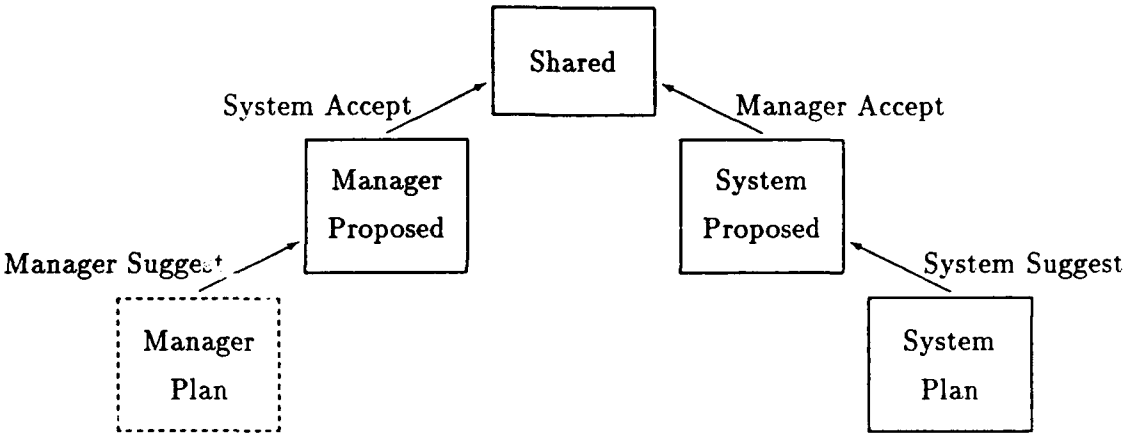


Figure 1: The Domain Plan Contexts

2.2.2 The Plan Contexts

In order to keep track of the negotiation of the composition of a plan during a conversation, a number of plan contexts are used. These are shown in Figure 1. The system's private knowledge about a plan is kept in the **System Plan** context. Items which have been suggested by the system but not yet accepted by the manager are in the **System Proposed** context. Similarly, items which have been suggested by the manager but not accepted by the System are in the **Manager Proposed** context. Items which have been proposed by one party and accepted by another are in the **Shared** context. The **Manager Plan** context is shown in dashed lines, because the system has no direct knowledge of the private reasoning of the manager. If we were trying to reason about and represent knowledge that the manager has but is not trying to communicate, then we would put it here. Spaces inherit from the spaces shown above them in the diagram. That is, everything in **Shared** will be in both **System Proposed** and **Manager Proposed**. Also, everything in **System**

Proposed will be in **System Plan**, and if it were used, **Manager Plan** would contain everything in **Manager Proposed**.

As an example, in the trace in section 6, the domain planner decides in the course of the elaboration that part of the plan will include loading the oranges into a car, this action gets put into the **System Plan** space. When the Discourse Reasoner makes a suggestion to the manager to this effect in the utterance, "shall I start loading the oranges in the empty car at I?" this action is moved to the **System Proposed** space. Later, if the manager accepts this, as in Section 6.1.6, it gets moved to the **Shared** space.

2.3 Acting on the basis of Discourse Information

There are several types of actions the Discourse Reasoner may take. The most important are speaking, planning, and plan execution.

In order to accomplish its goals, the Discourse Reasoner may need to communicate with the manager. This communication would take the form of English utterances that are attempts at speech acts that the system would like to perform. These speech acts conform to the types listed in Section 2.1. Since the main goal of the system is to come up with a shared plan, these acts will be to propose new information and accept proposals made by the manager.

The system will also have to do some planning to come up with these plans. When the manager makes a proposal, the system must try to incorporate the proposal into what is already known about the Manager's idea of the plan, which involves plan recognition to get at implicatures of the proposal. The system must also do plan elaboration on its own, to check the plan and fill in any gaps. An unqualified acceptance by the system of the manager's plan carries the implicature that the plan is executable and will be carried out by the system.

Finally, when a plan is fully explicit and is agreed upon, the system must execute it. This involves sending off directives to the agents in the TRAINS world to perform actions.

3 Architecture of the Discourse Reasoner

The Discourse Reasoner is composed of 2 main programs, the **Speech Act Analyzer** and the **Discourse Actor**, as well as some context information maintained by these.

3.1 The Discourse Context

Information in the discourse context is available to both the Speech Act analyzer, and the Discourse Actor. Several types of information are represented in the Discourse Context, including information about turn-taking, discourse segmentation, discourse obligations, and discourse goals.

Turn-taking information is maintained in order to keep track of who is speaking. We use a scheme which is much simpler than, but in the spirit of [Sacks *et al.*, 1974]. We

hypothesize a **release-turn** action, which hands the turn over to the other participant. In multi-party conversations, the **release-turn** could take an argument, which would be the person (if any) directed to speak next. A **release-turn** could be recognized by any of several cues, including: a question or request, an unmarked pause, or ending a statement with rising intonation. A participant may also make a **take-turn** attempt. Any speaking while another conversant has the turn may be seen as a **take-turn** attempt, though it will only be successful if the turn is completed and acknowledged. Participants may also perform a **hold-turn** action to explicitly keep the turn, while not furthering the conversation. Examples of these would be things like "unhh" or "well, ..". These serve to either eliminate or mark the pause that would otherwise be seen as a **release-turn**.

Discourse Segmentation information is kept for a variety of reasons. Some of these have to do with linguistic interpretation and generation, such as the ability to determine the possible referents for a referring expression. Others have more to do with the relations between utterances, things like adjacency pairs or clarification subdialogues. The currently open segment structure will signal how certain utterances will be interpreted. Utterances like "yes" can be seen as an acceptance of the last question asked but unanswered, if one exists in an open segment. Certain utterances like "by the way", or "anyway", or "let's go back to .." or "let's talk about .." will signal a shift in segments, while other phenomena such as clarifications will signal their changes in structure just by the information content. Arguments for the importance of discourse segmentation structure can be found in [Grosz and Sidner, 1986].

A record of the system's **Discourse Obligations** is maintained so that the obligations can be discharged appropriately. A promise or an accepted offer will incur an obligation. Also a request or command by the other party will bring an obligation to perform or address the requested action. If these requests are that the system say something (as in a **release-turn** action) or to inform (as in a question), then a discourse obligation is incurred. Rather than going through an elaborate planning procedure starting from the fact that the question being asked means that the speaker wants to know something (e.g. [Allen and Perrault, 1980; Litman and Allen, 1990]), which should then cause the system to adopt a goal to tell them, meeting the request is registered directly as an obligation, regardless of the intent of the questioner, or the other goal structure of the system. If the system doesn't address the obligation, then it must deal with the usual social problems of obligations which have not been met. This should help distinguish things expected by convention (e.g. that a question be answered) from simple cooperative behavior (e.g. doing what another agent wants). Other parts of the system might also bring about discourse obligations. For example, in some circumstances if the execution of the plan goes wrong, this would bring an obligation to inform the user.

Discourse goals are maintained so that the system can use the conversation to satisfy its own goals. The over-riding goal for this domain is to work out an executable plan that is shared between the two participants. This will lead to other goals such as accepting things that the manager has suggested, doing domain plan synthesis, or proposing plans to the manager that the domain planner has figured out. Another top level goal is to fulfill all discourse obligations.

3.2 The Speech Act Analyzer

The Speech Act Analyzer is responsible for recognizing the illocutionary acts in an utterance. The types of illocutionary acts used in this system are described in section 2.1 above. The method for recognizing them is based on [Hinkelman, 1990]. What is needed is to combine linguistic information with plan reasoning to come up with the most plausible interpretation. We can use information based on the mood of the utterance, as well as certain clueword cues to get the range of possibilities, and then filter them based on plan reasoning. If we are talking about the domain, then we can get the domain reasoner to do plan recognition and see if the interpretation would make sense in the current context. Eventually, as the types of discourse acts considered become more numerous and complex, we could do plan recognition of some sort at the discourse level as well.

3.3 The Discourse Actor

The Discourse Actor is the central "agent" of the Discourse Reasoner, which controls the flow of the system. The Discourse Actor is implemented as a reactive executor which will react to Illocutionary Acts made by the manager (as provided by the speech act analyzer), information in the Discourse Context, responses to queries to the Domain Planner. and the relationship of contents in the Domain Plan Context Spaces. The actions of the discourse actor to different stimuli are given in Tables 3, 4, and 5.

Speech Act	Action
Suggestion	Incorporate suggestion in **Human Proposed** space. This will involve plan recognition to figure out implicatures of the suggestion.
Accept item	Move item from **System Proposed** to **Shared** Space.
Reject item	Don't move plan to **Shared**. This will also update the discourse structure so that further acts will not be seen as implicit acceptance.
Request	Add request to Discourse Obligations.
Release-turn	Take turn.

Table 3: Actions based on recognized Speech Acts

When the system gets the turn, it will first try to discharge discourse obligations, and then to achieve its goal of trying to get a shared executable plan to achieve the domain goals. This will involve first trying to balance the domain plan contexts as described in Table 4. After this, the planner will try to elaborate the current plan in the **System Plan** context. It will release the turn either when it begins execution of a finished plan or when it needs information back (generally after a request).

Item in context	but not in context	Action
System Plan	**System Proposed**	Generate utterance (suggestion) to convey the new items to the human, move new items to **System Proposed** context
System Proposed	**Shared**	generate request to adopt items in plan
Human Proposed	**Shared**	either generate acceptance of items, move items to **Shared**
		or generate rejection and an alternative suggestion, put alternative in **System Proposed** context

Table 4: Actions based on the Domain Plan Contexts

Domain Call	response	Action
Elaborate	okay	Execute plan in **Shared** context
	new items	Generate utterance (suggestion) to convey the new items to the human, move new items to **System Proposed** context
	choice items	choose an item, **incorporate** choice into **System Proposed** context, generate utterance to convey the new item to the human
Incorporate	success	Generate acknowledgment that suggestion was understood
	failure	Initiate request for clarification or correction

Table 5: Actions based on responses from the Domain Reasoner

4 The Current Implementation

This section describes the programs of the demonstation system that was implemented over the summer of 1990. Some slight simplifications have been made to the model described above, and some of the actions described have been left implicit in the program flow of control. Currently, any question or request is interpreted as a *release-turn* action, and this is the only facility for turn-taking. The current system does not concern itself with discourse segmentation, handling only conversations within a single segment, although it does keep track of the requests, to judge interpret answers. The discourse obligations and goals are implicit in the control structure of the current discourse actor, rather than being explicitly represented.

4.1 Interface to Other Modules

There are two top-level routines of the current Discourse Reasoner. These are `init-discourse` and `discourse`. `init-discourse` gets called once at the beginning of the system, to initialize local data structures. It is called with no arguments and doesn't return a meaningful value. Currently it is called by the domain planner function `init-planner`. `discourse` gets called by the top-level module with a list of Semantic Interpretations in the form of Episodic Logic statements. It returns a stream which includes any utterances it has made while processing. This is so that these utterances can be interpreted by the NL subsystem to update it's local state.

The Discourse Reasoner will make several calls to other modules. These include: `(create-plan)`, `(is-a-plan item)`, `(incorporate plan-part space)`, `(elaborate p space)`, `(execute p space)`, `(move-plan p oldspace newspace)`, and `(generate speech-act)`. All of the arguments of these calls are either symbols or lists of symbols, except for the domain plan contexts, which take the actual context data structures.

`(create-plan)` takes no arguments and returns an identifier for a new domain plan. This plan can then have items added to it in various domain plan contexts. A dummy version of `(create plan)` is currently in the file **discourse**.

`(is-a-plan item)` is a predicate which takes one argument and returnst if this argument is a valid plan. A dummy version of `is-a-plan` is in the file **discourse**.

Name	Context
`*hprop*`	Manager Proposed
`*sh*`	Shared
`*sprop*`	System Proposed
`*splan*`	System Plan

Table 6: Internal Names for Domain Plan Contexts

`(incorporate plan-part space)` takes two arguments, the first of which is one of the plan-items described in Table 2, the second of which is one of the domain plan contexts

described in Section 2.2.2. This command will cause the domain planner to add the item to the plan in the specified context, and perform plan recognition to add anything else necessary for the item to make sense as part of the plan. This command will return an error if it can't incorporate the item in that space. The most common call to `incorporate` will be in the **Manager Proposed** context, as a result of a suggestion or request by the manager. Calls may also be made in the **System Proposed** context as a result of checking the likely implicatures of an utterance made by the system. The system internal names of the Domain Plan Contexts are given in Table 6.

`(query plan-part space)` takes two arguments, the first of which is one of the plan-items described in Table 2, the second of which is one of the domain plan contexts described in Section 2.2.2. This command will return true if the plan-part makes sense in that space. `query` is very similar to `incorporate`, except it does not change the context. `query` is currently unimplemented, but should be used by the Speech Act Analyzer to evaluate and choose between possible speech acts. It could also be used by the NL generator when trying to come up with felicitous utterances.

`(elaborate p space)` takes two arguments, the first of which is a plan, and the second is one of the domain plan contexts described in Section 2.2.2. `elaborate` is a command to the domain planner to try to complete the plan in that space. `elaborate` returns the new items, including keys signalling where it can not go further. `elaborate` is usually called in the **System Plan** space.

`(execute p space)` takes two arguments, the first of which is a plan, and the second is one of the domain plan contexts described in Section 2.2.2. `execute` tells the domain planner to ship off the plan in that context to the Executor to start performing the plan. `execute` is currently called in the **System Plan** space.

`(move-plan p oldspace newspace)` takes three arguments, the first of which is a plan, and the next two of which are domain plan contexts, as described in Section 2.2.2. This moves a plan from one space to another, essentially copying all information about the plan in one space to another. This is called to move plans from **System Proposed** or **Manager Proposed** to **Shared** and from **System Plan** to **System Proposed**. This should probably be extended to allow moving individual plan-items, rather than just whole plans.

`(assert-plan-item plan-part space)` takes two arguments, the first of which is a plan, and the second is one of the domain plan contexts described in Section 2.2.2. `assert-plan-item` adds an item to the space, as in `incorporate`, but does not do plan recognition or consistency checking as `incorporate` would. It is currently unused, but would be another way to move items from **System Plan** to **System Proposed**.

`(verify proposition)` takes as its argument a proposition about the world as an argument and checks to see if this is true. It may involve sending queries to the domain agents. Results of this may also modify the plan in the **System Plan** context if changes are noticed. This command is currently unimplemented.

`(generate speech-act)` takes as its argument a speech act of the form described in Section 2.1. It generates a natural language utterance to the manager which conveys the speech act, and adds a string with the utterance to the variable `*output*` for the function `discourse` to return. See Appendix A for more details.

4.2 The Code of the Current System

The code for this version of the Discourse Reasoner is divided into three files, plus additional related material described in the appendices. The file **discourse** contains the top level routines as well as some context information and general information. The file **disc-actor** contains the Discourse Actor. **get-acts** contains the Speech-Act Analyzer.

4.2.1 File: discourse

The file **discourse** contains top level definitions for the Discourse Reasoner. It contains the two top level functions `discourse` and `init-discourse` which get called from outside, as described in Section 4.1. `discourse` calls the Episodic Logic Translator (see Appendix B) on each episodic logic statement. It then calls the Speech Act Analyzer on each translation, gathering up the Speech Acts which have been recognized, and then calls the Discourse Actor to act in response to each of these. Finally, `discourse` returns a stream of all utterances the system has made, as stored by the generator in the global variable `*output*`.

The file **discourse** also defines several auxiliary functions including `hprop`, `sh`, `sprop`, and `splan` which take no arguments and return the spaces `*hprop*`, `*sh*`, `*sprop*`, and `*splan*`, respectively.

4.2.2 File: get-acts

This file contains the Speech Act Analyzer. The main function is `get-speech-acts`, which is called by the function `discourse` with a translated logical form as argument. This function returns a list of all speech acts found to be associated with this utterance. The current implementation does not check to see if the speech acts are plausible with respect to plan based information, or choose among a set of possible acts. It merely checks the surface features (e.g. mood of utterance, main verbs, keywords) and determines the acts based on these. It will also set the discourse-global `*request*`, which keeps track of the last request made.

4.2.3 File: disc-actor

The main function is `actor`, which is called by function `discourse` with a speech act as argument. This function works more or less as described in Table 3, above, branching to a subfunction for each of the types of speech acts. One simplification is that the discourse obligations are not explicitly represented. Instead, when a request is recognized, it is acted upon immediately. Also, the system currently always adopts the user's plan whenever it gets the turn, and never tries to interrupt before it thinks the user has handed over the turn.

5 Next Stages for Implementation

There are several extensions which should be made to subsequent versions of the Discourse Reasoner. These involve bringing the implementation (as discussed in Section 4) in accord

with the architecture described in Section 3, as well as extending that architecture. The elements of the Discourse Context should be made more explicit. Discourse goals and obligations should be explicitly represented and the Discourse Reasoner should be given the ability to do some actual planning, or other types of inference, rather than simply reacting as speech acts come in. In addition, as more complex problems and conversations are considered, a capability of handling more than one discourse segment and more than one plan at a time must be added.

This in turn will necessitate a more sophisticated Speech Act Analyzer. Different Speech acts will appear more likely depending on which discourse segment the utterance is part of, and which plan it is talking about. The Speech Act analyzer will have to make queries to the domain planner, and choose the most likely act among a set of alternatives. More types of speech acts should be allowed, including some which would function strictly as informs of background information, without being related to any particular plan. The Discourse Reasoner would also have to keep a more elaborate record of rhetorical relations between utterances than the current system, which just keeps track of the last request.

A more realistic NL generation module must be added, to form a tighter coupling between the information provided by the planner, and the utterances made to the user. There also needs to be some way to distinguish between items added to the contexts explicitly (through identification of the intentions of an utterance) and items which have been assumed as implicitly communicated. It is generally a possibility in conversation to make things more explicit, but different forms are used to reconfirm things already explicit in the conversation.

Some other advances to the system should eventually be made. The system should have a little more autonomy, with the ability to decide to accept a suggestion from the manager only when it would come to a better plan. In addition, we may also want to add a couple more domain contexts for items which have been acknowledged by the other party, but not accepted. This would form the basis for more complex types of negotiation.

Finally, a change from the current sequential system to one in which the separate modules can run concurrently will merit further changes. When the planner can operate even while the system does not have the turn, there will be times when the system will want to take the turn instead of waiting passively for it to be handed over. This architecture will also allow the system to recognize and respond to statements from the manager while the planner is operating.

6 Example Traces of the Discourse Reasoner

We present two variants of a conversation to meet the same goal.

6.1 Example 1

First the System is set up. The Discourse System sets up a new domain plan, called `PLAN1` which will be filled out in the course of the conversation. The plan is empty in all the Domain Plan contexts.

6.1.1 The First Sentence

The first sentence uttered by the user is,

```
HUMAN SAYS-> We have to make OJ.
```

The Episodic Logical form of this sentence is,

```
(E E10 ((E10 AT-ABOUT NOW1) AND (E9 ORIENTS E10))
       ((HUM TELL SYS
            (THAT (E E11 ((E11 AT-ABOUT E10) AND
                          (E8 ORIENTS E11))
                         (((K (L LE1 (((SET-OF HUM SYS)
                                       (MAKE OJ)) ** LE1)))
                           MUST-OCCUR) ** E11)))) ** E10))
```

The translation of that statement is:

```
(TELL (AGENT HUMAN (*VAR* V_1071)) (AGENT SYSTEM (*VAR* V_1072))
      (OBLIGATION AGENT (MAKE_OJ (*VAR* V_1069) (*VAR* V_1070))))
```

From the "OBLIGATION", the speech act analyzer determines that this is a goal of the manager. The default act that a "TELL" represents is a suggestion. The current plan under discussion is PLAN1. So the Speech act analyzer recognizes the following act:

```
(SUGGEST (GOAL (MAKE_OJ (*VAR* V_1069) (*VAR* V_1070)) PLAN1))
```

Now the discourse actor reacts to this speech act. This involves a call to incorporate in the **Manager Proposed** context:

```
(incorporate (GOAL (MAKE_OJ (*VAR* V_1069) (*VAR* V_1070)) PLAN1)
             *HPROP*)
```

This will cause the domain planner to do plan recognition in the **Manager Proposed** context. The planner returns okay, indicating a successful incorporation, and the next sentence comes through:

6.1.2 The Second Sentence

```
HUMAN SAYS-> There are oranges at I and an OJ factory at B.
```

The Episodic Logical form for the first conjunct is:

```
(E E12 ((E12 AT-ABOUT NOW2) AND (E10 ORIENTS E12))
       ((HUM TELL SYS
             (THAT (E E13 ((E13 AT-ABOUT E12) AND (E11 ORIENTS E13))
                          (((V117 (PLUR ORANGE)) AND
                            (V117 LOC-AT STN-I)) ** E13)))) ** E12))
```

which is translated as:

```
(TELL (AGENT HUMAN (*VAR* V_1083)) (AGENT SYSTEM (*VAR* V_1084))
      (AT (ORANGES 01 (*VAR* V_1085)) (CITY CITYI (*VAR* V_1086))))
```

producing the speech-act:

```
(SUGGEST (FACT (AT (ORANGES 01 (*VAR* V_1085))
                   (CITY CITYI (*VAR* V_1086))) PLAN1))
```

The Episodic Logical Form for the second conjunct is

```
(E E14 ((E14 AT-ABOUT NOW3) AND (E12 ORIENTS E14))
       ((HUM TELL SYS
             (THAT (E E15 ((E15 AT-ABOUT E14) AND (E13 ORIENTS E15))
                          ((E m~1082 (((m~1082 ((ATTR-N OJ) FACTORY))
                                       & (m~1082 LOC-AT STN-B)) * E15))
                           ** (F E15)))))
        ** E14))
```

which is translated as:

```
(TELL (AGENT HUMAN (*VAR* V_1087)) (AGENT SYSTEM (*VAR* V_1088))
      (AT (OJFACTORY F1 (*VAR* V_1089)) (CITY CITYB (*VAR* V_1090))))
```

producing the speech-act:

```
(SUGGEST (FACT (AT (OJFACTORY F1 (*VAR* V_1089))
                   (CITY CITYB (*VAR* V_1090))) PLAN1)
```

These will cause the following calls:

```
(incorporate (FACT (AT (ORANGES 01 (*VAR* V_1085))
                       (CITY CITYI (*VAR* V_1086))) PLAN1)
     *HPROP*)
```

```
(incorporate (FACT (AT (OJFACTORY F1 (*VAR* V_1089))
                           (CITY CITYB (*VAR* V_1090))) PLAN1)
             *HPROP*)
```

6.1.3 The Third Sentence

Next,

```
HUMAN SAYS-> Engine E3 is scheduled to arrive at I at 3PM.
```

The Episodic Logical Form is:

```
(E E16 ((E16 AT-ABOUT NOW4) AND (E14 ORIENTS E16))
       ((HUM TELL SYS
           (THAT (E E17 ((E17 AT-ABOUT E16) AND (E15 ORIENTS E17))
                        ((E m~1112 (((m~1112 SCHEDULE) &
                                     (m~1112 SPECIFIES
                                       (K (L LE2 (((ADV-E (AT-TIME 3PM))
                                                   (ENG3 ARRIVE-AT STN-I))
                                                  ** LE2)))))
                                     * E17))
                    ** (F E17)))))
         ** E16))
```

Which translates to:

```
(TELL (AGENT HUMAN (*VAR* V_1113)) (AGENT SYSTEM (*VAR* V_1114))
      (AT (ENGINE ENG3 (*VAR* V_1115)) (CITY CITYI (*VAR* V_1116))))
```

Which produces the speech act:

```
(SUGGEST (FACT (AT (ENGINE ENG3 (*VAR* V_1115))
                   (CITY CITYI (*VAR* V_1116))) PLAN1)
```

and the planner call:

```
(incorporate (FACT (AT (ENGINE ENG3 (*VAR* V_1115))
                       (CITY CITYI (*VAR* V_1116))) PLAN1)
             *HPROP*)
```

6.1.4 The Fourth Sentence

```
HUMAN SAYS-> Shall we ship the oranges?
```

which has the Episodic Logical Form:

```
(E E18 ((E18 AT-ABOUT NOW5) AND (E16 ORIENTS E18))
       ((HUM ASK SYS
             (ANSWER-TO (Y-N-Q (E E19 ((E19 AT-ABOUT E18) AND (E7 ORIENTS E19))
                                      ((SHALL-OCCUR ((SET-OF HUM SYS) SHIP V117))
                                       ** E19)))))
         ** E18))
```

which translates to

```
(ASK (AGENT HUMAN (*VAR* V_1130)) (AGENT SYSTEM (*VAR* V_1131))
     (SHALL (MOVE_ORANGES (*VAR* V_1121)
                ((AGENT (*VAR* AGENT1122) (*VAR* V_1123))
                 (ORANGES 01 (*VAR* V_1132))
                 (CAR (*VAR* CAR1124) (*VAR* V_1125))
                 (CITY (*VAR* CITY1126) (*VAR* V_1127))
                 (OJFACTORY (*VAR* OJFACTORY1128) (*VAR* V_1129))))))
```

Which produces the following speech acts:

```
1) (SUGGEST (ACTION_IN (MOVE_ORANGES (*VAR* V_1121)
                             ((AGENT (*VAR* AGENT1122) (*VAR* V_1123))
                              (ORANGES 01 (*VAR* V_1132))
                              (CAR (*VAR* CAR1124) (*VAR* V_1125))
                              (CITY (*VAR* CITY1126) (*VAR* V_1127))
                              (OJFACTORY (*VAR* OJFACTORY1128) (*VAR* V_1129))))
                        PLAN1))

2) (REQUEST (ADOPT PLAN1))

3) (RELEASE-TURN)
```

1) is recognized since a question with *shall* as the head verb indicates a suggestion. 2) and 3) are recognized, since all questions are seen as **release-turn** actions, and implicit requests to adopt the plan just mentioned. The speech acts will cause the following actions to be taken:

```
1) (incorporate (ACTION_IN (MOVE_ORANGES (*VAR* V_1121)
                                 ((AGENT (*VAR* AGENT1122) (*VAR* V_1123))
                                  (ORANGES 01 (*VAR* V_1132))
                                  (CAR (*VAR* CAR1124) (*VAR* V_1125))
                                  (CITY (*VAR* CITY1126) (*VAR* V_1127))
                                  (OJFACTORY (*VAR* OJFACTORY1128) (*VAR* V_1129))))
                            PLAN1)
                *HPROP*)

2) (generate (ACCEPT (ADOPT PLAN1)))
   (move-plan from *HPROP* to *SHARED*)
```

Now, the plan as recognized is shared by both parties.

```
3) (take-turn)
```

The system takes the turn.

6.1.5 The System's First Turn

Now the system has the turn. It tries to elaborate the plan in the **System Plan** context.

```
(elaborate PLAN1 *SPLAN*)
```

This returns the following new items, which have been added to the **System Plan** context:

```
(ACTION_IN (MOVE_ENG ACT1133
                     ((ENGINE ENG3 PARM1092)
                      (CITY (FACTORY-CITY F1) PARM1074)))
           PLAN1)

(ACTION_IN (RUN ACT1134 ((OJFACTORY F1 PARM1074))) PLAN1)

(ACTION_IN (UNLOAD_ORANGES ACT1135
                           ((OJFACTORY F1 PARM1074)
                            (CAR (*VAR* C) PARM1093)
                            (CITY CITYB PARM1077) (ORANGES 01 PARM1075)))
```

```
          PLAN1)

(ACTION_IN (LOAD_ORANGES ACT1136
                         ((CAR (*VAR* C) PARM1093)
                          (ORANGES O1 PARM1075) (CITY CITYI PARM1094)))
          PLAN1)

(ACTION_IN (RUN ACT2268 ((OJFACTORY F1 PARM2205))) PLAN1)
```

In addition, the planner also returns,

```
(CHOICE (CAR C2 PARM1093) (CAR C1 PARM1093))
```

Signalling that PARM1093, the car which the oranges must be loaded into and unloaded from in the above actions, may be either C1, an empty car already at city I, or C2, the car which is attached to Engine E3. The Discourse Reasoner arbitrarily chooses C1, making the following call,

```
(incorporate (FACT (AT (CAR C1 PARM1093) (CITY CITYI PARM1094))
                   PLAN1)
             *SPROP*)
```

In order to communicate the new information to the manager, it passes the following speech acts to the NL generator:

```
(SUGGEST (AT (CAR C1 PARM1093) (CITY CITYI PARM1094)))

(SUGGEST (ACTION_IN (LOAD_ORANGES ACT1136
                          ((CAR (*VAR* C) PARM1093)
                           (ORANGES O1 PARM1075)
                           (CITY CITYI PARM1094)))
                    PLAN1))

(SUGGEST (ACTION_IN (UNLOAD_ORANGES ACT1135
                          ((OJFACTORY F1 PARM1074)
                           (CAR (*VAR* C) PARM1093)
                           (CITY CITYB PARM1077)
                           (ORANGES O1 PARM1075)))
                    PLAN1))

(SUGGEST (ACTION_IN (RUN ACT1134 ((OJFACTORY F1 PARM1074)))
                    PLAN1))

(SUGGEST (ACTION_IN (MOVE_ENG ACT1133
                          ((ENGINE ENG3 PARM1092)
                           (CITY (FACTORY-CITY F1) PARM1074)))
                    PLAN1)
```

Since it has nothing further to say it will also pass along:
(RELEASE-TURN) and (REQUEST (ADOPT PLAN1)).
Now the generator will cause the following to be uttered,

```
Yes, shall I start loading the oranges in the empty car at I?
```

The Discourse Reasoner will move these items from **System Plan** to **System Proposed**

6.1.6 The Sixth Sentence

The manager now replies,

```
HUMAN SAYS-> Yes, and we'll have E3 pick it up.
```

This produces the following episodic logical forms,

```
(E E23 ((E23 AT-ABOUT NOW8) AND (E21 ORIENTS E23))
       ((HUM TELL SYS YES) ** E23))


(E E24 ((E24 AT-ABOUT NOW9) AND (E23 ORIENTS E24))
       ((HUM TELL SYS
             (THAT (E E25 ((E25 AT-ABOUT E24) AND (E22 ORIENTS E25))
                          ((E E26 ((E26 AFTER E25) AND (E5 ORIENTS E26))
                                  (((SET-OF HUM SYS)
                                    BRING-ABOUT
                                    (K (L LE3 ((ENG3 PICK-UP C1) **LE3))))
                                   ** E26))
                           ** (F E25)))))
         ** E24))
```

Which lead to the following translations:

```
(TELL (AGENT HUMAN (*VAR* V_1153)) (AGENT SYSTEM (*VAR* V_1154)) (YES))

(TELL (AGENT HUMAN (*VAR* V_1158)) (AGENT SYSTEM (*VAR* V_1159))
      (CAUSE SYS_HUM (COUPLE (*VAR* V_1155)
                             ((ENGINE ENG3 (*VAR* V_1160))
                              (CAR C1 (*VAR* V_1161))
                              (CITY (*VAR* CITY1156) (*VAR* V_1157))))))
```

Which lead to the following speech acts:

```
(ACCEPT (ADOPT PLAN1))

(SUGGEST (ACTION_IN (COUPLE (*VAR* V_1155)
                            ((ENGINE ENG3 (*VAR* V_1160))
                             (CAR C1 (*VAR* V_1161))
                             (CITY (*VAR* CITY1156) (*VAR* V_1157))))
                    PLAN1))
```

The first will cause the discourse actor to move the items in **System Proposed** to **Shared**. The second leads to the call,

```
(incorporate (ACTION_IN (COUPLE (*VAR* V_1155)
                                ((ENGINE ENG3 (*VAR* V_1160))
                                 (CAR C1 (*VAR* V_1161))
                                 (CITY (*VAR* CITY1156) (*VAR* V_1157))))
                            PLAN1)
             *HPROP*)
```

6.1.7 The Seventh Sentence

The manager now utters,

```
HUMAN SAYS-> ok?
```

which produces the Logical Form,

```
(E E27 ((E27 AT-ABOUT NOW10) AND (E24 ORIENTS E27))
       ((HUM ASK SYS
            (ANSWER-TO (Y-N-Q (E E28 ((E28 AT-ABOUT E27) AND
                                      (E4 ORIENTS E28))
                                     ((E34 OK) ** E28)))))
        ** E27))
```

and the following translation:

```
(ASK (AGENT HUMAN (*VAR* V_1164)) (AGENT SYSTEM (*VAR* V_1165))
     (OK))
```

Which leads to the following speech acts:

```
(REQUEST (ADOPT PLAN1))

(RELEASE-TURN)
```

The *first one* causes the Discourse Actor to accede to the request and move these items from **Manager Proposed** to **Shared** and generate `(ACCEPT (ADOPT PLAN1))` The second causes the Discourse Actor to take up the turn.

6.1.8 The System's Second Turn

First the Discourse actor will call

```
(elaborate PLAN1  *SPLAN*)
```

Which this time does not return any new items: the plan is complete. Thus the Discourse Actor will call

```
(execute PLAN1  *SPLAN*)
```

And give up the turn. The generator will produce,

```
OK.
```

and the executor will begin to carry out the plan.

6.2 Example 2

Example 2 is identical to Example 1 until the manager's first response (the sixth sentence).

6.2.6 The sixth sentence

Here, the manager rejects the system's suggestion to use c1 and instead says,

```
HUMAN SAYS-> No, load the oranges into the car attached to E3.
```

which has the logical forms,

```
(E E23 ((E23 AT-ABOUT NOW8) AND (E21 ORIENTS E23))
       ((HUM TELL SYS NO) ** E23))

 (E E24 ((E24 AT-ABOUT NOW9) AND (E23 ORIENTS E24))
        ((HUM INSTRUCT SYS
              (K (L A2 (((FST A2) LOAD-INTO V117 C2)
                        ** (RST A2)))))
         ** E24))
```

and leads to the following translations:

```
(TELL (AGENT HUMAN (*VAR* V_1152)) (AGENT SYSTEM (*VAR* V_1153)) (NO))

(REQUEST (AGENT HUMAN (*VAR* V_1157)) (AGENT SYSTEM (*VAR* V_1158))
         (LOAD_ORANGES (*VAR* V_1154)
               ((CAR C2 (*VAR* V_1159))
                (ORANGES O1 (*VAR* V_1160))
                (CITY (*VAR* CITY1155) (*VAR* V_1156)))))
```

Leading to the speech acts:

```
1) (REJECT (ADOPT PLAN1))

2) (REQUEST (DO (ACTION_IN (LOAD_ORANGES (*VAR* V_1154)
                             ((CAR C2 (*VAR* V_1159))
                              (ORANGES O1 (*VAR* V_1160))
                              (CITY (*VAR* CITY1155) (*VAR* V_1156))))
                           PLAN1)))

3) (RELEASE-TURN)
```

Act 1) leaves the system's suggestion in the **System Proposed** context, rather than moving it to shared. Act 2) causes the system to make the call,

```
(incorporate (ACTION_IN (LOAD_ORANGES (*VAR* V_1154)
                            ((CAR C2 (*VAR* V_1159))
                             (ORANGES O1 (*VAR* V_1160))
                             (CITY (*VAR* CITY1155) (*VAR* V_1156))))
                          PLAN1)
             *HPROP*)
```

Act 3) causes the system to take the turn again.

6.2.7 The System's Second Turn

First the system will accept the suggestion, moving the new items from **Manager Proposed** to **Shared** (and giving up on the plan in **System Proposed**). Next the system makes the call,

```
(elaborate PLAN1  *SPLAN*)
```

to see if there are any gaps left in the Manager's plan. The Domain Planner returns the new items (which have been added to the **System Plan** context):

```
(ACTION_IN (MOVE_ENG ACT1163
                ((ENGINE ENG3 PARM1092)
                 (CITY (FACTORY-CITY F1) PARM1074)))
           PLAN1)

(ACTION_IN (RUN ACT1164 ((OJFACTORY F1 PARM1074))) PLAN1)

(ACTION_IN (UNLOAD_ORANGES ACT1165
              ((OJFACTORY F1 PARM1074)
               (CAR C2 PARM1093)
               (CITY CITYB PARM1077)
               (ORANGES O1 PARM1075)))
           PLAN1)
```

Since none of these involve a choice, the system can go ahead and execute the completed plan, confident that this is okay with the manager. It has nothing more to say, so it sends a `release-turn` *to the generator, producing the utterance,*

```
OK.
```

It al‹ › makes the call,

```
(execute PLAN1 *SPLAN*)
```

Which causes the executor to carry out the plan using the attached car this time.

A The NL Generator

The current NL Generator module is just a stub which produces appropriate utterances for the demonstration system. The NL Generator is in file **generator**. It consists of two functions: **generate**, and **make-utterance**. **generate** gets called by the Discourse Reasoner with a speech act as argument. It saves the speech acts in a list, and then calls **make-utterance** on that list when it has enough to make a coherent statement. Currently, **generate** does not make judgements on the form of the speech-acts, it just makes an utterance when-ever it receives a **release-turn** act. A more sophisticated version might look at other things such as timing and the relationship of the different acts so as to make several utterances within a turn.

make-utterance takes a list of acts as argument and produces an utterance. It has two prewritten utterances set up for this demonstration, and chooses strictly on the number of speech acts it receives.

make-utterance calls the system function **utter** with a string containing the utterance as argument. **utter** is a function of the executor package which actually makes the text appear on screen to the user. **make-utterance** also sets the discourse global variable ***output*** to the utterance, so this can be passed back as the result of the discourse call. This is an artifact of the current control strategy. Probably make-utterance should directly call the language sub-system when it makes an utterance.

B The Episodic Logic Translator

Episodic Logic [Schubert and Hwang, 1989] is a general purpose knowledge representation suitable for Natural Language story understanding in a wide variety of domains. The domain reasoner, however, is a planner specially built for the TRAINS world. In order to shift from the generality of Episodic Logic to this particular domain, a translator is the first step in the interpretation process. The translator is a pattern matcher contained in the file **matcher** which uses rules in the file **rules**. The function **translate** gets called by the function **discourse** with an episodic logic statement, and returns a statement in the TRAINS planning representation.

References

[Allen and Perrault, 1980] James F. Allen and C. R. Perrault, "Analyzing Intention in Utterances," *Artificial Intelligence*, 15:143–178, 1980.

[Allen and Schubert, 1991] James F. Allen and Lenhart K. Schubert, "The TRAINS Project," TRAINS Technical Note 91-1, Computer Science Dept. University of Rochester, 1991.

[Ferguson, 1991] George Ferguson, "Domain Plan Reasoning in TRAINS-90," TRAINS Technical Note 91-2, Computer Science Dept. University of Rochester, 1991.

[Grosz and Sidner, 1986] Barbara Grosz and Candice Sidner, "Attention, Intention, and the Structure of Discourse," *CL*, 12(3):175–204, 1986.

[Grosz and Sidner, 1990] Barbara J. Grosz and Candace L. Sidner, "Plans for Discourse," In P. R. Cohen, J. Morgan, and M. Pollack, editors, *Intentions in Communication*. MIT Press, 1990.

[Hinkelman, 1990] Elizabeth Hinkelman, *Linguisic and Pragmatic Constraints on Utterance Interpretation*, PhD thesis, University of Rochester, 1990.

[Light, 1991] Marc Light, "Semantic Interpretation in TRAINS-90," TRAINS Technical Note 91-3, Computer Science Dept. University of Rochester, 1991.

[Litman and Allen, 1990] Diane J. Litman and James F. Allen, "Discourse Processing and Common Sense Plans," In P. R. Cohen, J. Morgan, and M. Pollack, editors, *Intentions in Communication*. MIT Press, 1990.

[Martin and Miller, 1991] Nat Martin and Brad Miller, "The TRAINS-90 Simulator," TRAINS Technical Note 91-4, Computer Science Dept. University of Rochester, 1991.

[Sacks *et al.*, 1974] H. Sacks, E. A. Schegloff, and G. Jefferson, "A Simplest Systematics For the organization of Turn-Taking for Conversation," *Language*, 50:696–735, 1974.

[Schubert and Hwang, 1989] L. K. Schubert and C. H. Hwang, "An Episodic knowledge representation for Narrative Texts," In *1st Inter. Conf. on Principles of Knowledge Representation and Reasoning (KR89)*, pages 444–458, Toronto, Canada, May 15-18, 1989.

[Schubert, 1991] Lenhart K. Schubert, "Language processing in the TRAINS Project," Trains technical note, Computer Science Dept. University of Rochester, forthcoming 1991.