1991

A Knowledge-based approach to understanding natural language

Bernard J. Huber Jr.

Rochester Institute of Technology
Computer Science Department

A Knowledge-Based Approach to Understanding Natural Language

by

Bernard J. Huber, Jr.

A thesis, submitted to
The Faculty of the Computer Science Department,
in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science

Approved by:

Prof. Kevin Donaghy

Prof. John A. Biles

Prof. Peter G. Anderson

September, 1991

PERMISSION TO COPY

Title of thesis: A Knowledge-Based Approach to Understanding Natural Language.

I, Bernard J. Huber, Jr., hereby **grant permission** to the Wallace Memorial Library of RIT to reproduce my thesis in whole or in part. Any reproduction will not be for commercial use or profit.

Date: September 17, 1991.

ABSTRACT

Understanding a natural language requires knowledge about that language as a system of representation. Further, when the task is one of understanding an extended discourse, world knowledge is also required. This thesis explores some of the issues involved in representing both kinds of knowledge, and also makes an effort to arrive at some understanding of the relationship between the two.

A part of this exploration involves an examination of some natural language understanding systems which have attempted to deal with extended discourse both in the form of stories and in the form of dialogues. The systems examined are heavily dependent on world knowledge.

Another part of this exploration is an effort to build a dialogue system based on speech acts and practical arguments, as they are described in "Recognizing Promises, Advice, Threats, and Warnings", a Masters Thesis presented to Rochester Institute of Technology, School of Computer Science and Technology, in 1986 by Kevin Donaghy.

This dialogue system includes a deterministic syntactic parser, a semantic representation based on the idea of case frames, and a context interpreter that recognizes and represents groups of sentences as practical arguments. This Prolog implementation employs a frame package developed and described in "A Frame Virtual Machine in C-Prolog", a Masters Thesis presented to Rochester Institute of Technology, School of Computer Science and Technology, in 1987 by LeMora Hiss.

While this automated dialogue system is necessarily limited in the domain that it recognizes, the opportunity it allows to build a mechanism and a system of representation brings with it a range of issues from the syntactic, through the semantic, to the contextual and the pragmatic. Here, the focus of inquiry came to settle in the semantic representation, where the relationship between knowledge about language and knowledge about the world seems to be naturally resident.

TABLE OF CONTENTS

1. Introduction

1.1. History and Intention

"Recognizing Promises, Advice, Threats and Warnings in Natural Language Discourse", [DONA 86], demonstrated that certain speech acts could be recognized from purely linguistic cues. That system accepted natural language as input and produced, as output, an intelligent paraphrase, one which indicated the speech act involved. Four types of speech acts were recognized: advice, warnings, conditional promises, and conditional threats. The content of each speech act was represented as a practical argument composed of a premise and a conclusion. The user could either enter the speech act explicitly in this form, or a practical argument could be induced from an enthymematic input if it provided enough evidence to induce the major premise of the argument. The following are a few examples: ("S" indicates the speaker).

Input: If you finish your homework then I will let you watch television. So finish that math.
Output: S promised to let you watch television if you finish your homework.

Input: If you study for the exam then you will pass. So study hard.
Output: S advised you to study hard since if you study for the exam you will pass.

These examples explicitly include the premise of the argument ("If you study for the exam then you will pass") and the conclusion ("So study hard"). The following examples are enthymematic; notice that in these cases the paraphrase itself supplies the premise.

Input: The stove is hot. So do not touch the stove.
Output: S warned you not to touch the stove since if you touch the stove you will be burned.

Input: See the exhibition. It is excellent.
Output: S advised you to see the exhibition since if you see the exhibition you will be pleased.

The above paraphrases are produced through interactions with a knowledge base that knows that 'hot things cause burns when touched' and that 'excellent things are pleasing to see'. (For a more complete description of practical arguments see section 2.1; for a description of speech acts see section 2.3.)

[DONA 86] also proposed that recognizing speech acts such as advice and warnings, and recognizing the reasons (premises) that are given in support of advice and warnings, could be first steps toward constructing an automated dialogue system that might effectively communicate with a human being. The next steps would involve developing the capability to evaluate those reasons and to accept or reject advice as a result of those evaluations. In the course of further research, Dr. Donaghy proposed that a frame-based system would provide an effective means of performing this

evaluation. Each instance of a speech act could be represented by a frame that contained the premise and the conclusion of an argument, as well as information that would be produced by an evaluation of the argument.

The thesis presented here begins with these ideas for extending the system first developed in [DONA 86]. The initial intention was to extend the capabilities of the earlier system so that it could not only recognize certain speech acts, but could also evaluate the practical arguments involved, determining whether they are sound or unsound. While this intention remains intact as the background or field of research for this thesis, implementation features did not go so far as to fully explore this process of evaluation. Instead, the Prolog implementation that accompanies this study focuses on the representational issues that go with the use of frames to process natural language. The original intention of using frames to represent practical arguments has been extended to include the use of frames throughout the system, both as a representation of linguistic meaning, and as a way to organize linguistic and world knowledge. While [DONA 86] produces an intelligent paraphrase of its input, the implementation presented here attempts to use frames to more fully process its input. Such an implementation is necessary if the recognition and the processing of practical arguments is to become a part of a larger system, one that is able to carry on a dialogue with a human user. Beyond the implementation, a great many of the issues that present themselves to the effort to construct an automated dialogue system will be discussed. The implementation will also be handled as one way (more detailed than most) of discussing and exploring some of these issues.

The system presented here will accept natural language as input, recognize certain speech acts when they occur, represent those speech acts as practical arguments, organized and stored in a frame structure, and handle some other features of carrying on a dialogue, such as paraphrasing the input and replying to some user questions. Beyond this implementation, the discussion will also involve the further capability of performing reasoning functions toward an evaluation of the practical arguments it recognizes. The output of a fully realized system should include three types of replies to the user. If the system recognizes a speech act and finds that the argument involved is sound, then the system produces an intelligent paraphrase of the argument, indicating the speech act involved. If the system finds that the argument is unsound, that is, that the advice or warning given is in conflict with previous entries, then the system tries to resolve the conflict, and will report this resolution to the user. Lastly, the system is able to respond to user queries regarding its own reasoning processes.

1.2. A Sample Dialogue

The following is a sample dialogue intended to give a brief overview of how the capabilities mentioned above might be used. This dialogue is presented to demonstrate the higher level reasoning processes of a possible dialogue system and to provide material for implementation and discussion.

Input: A test is scheduled for Monday. You should study for the test.
Output: The speaker has advised the hearer to study for an upcoming test.
Input: Why?
Output: The speaker believes that the hearer will improve his chances of passing the test if the hearer studies for the test.
Input: Does the speaker believe that the hearer wishes to pass the test?
Output: Yes.
Input: Why?
Output: A test is a task. The normal goal of a person with a task is to wish to successfully complete that task.
Input: A lab is due on Monday. You should do the lab.
Output: Is the lab due the same day that the test is scheduled?
Input: Yes.
Output: The hearer should finish the lab, and then should study for the test.
Input: Why do the lab before studying for the test?
Output: Doing the lab is a necessary condition for finishing the lab. Studying for the test is not a necessary condition for passing the test.
Input: It will probably take all of the available time to finish the lab.
Output: The hearer should do the lab.
Input: Did you forget about the test?
Output: The hearer may be able to pass the test without studying for the test. The hearer cannot finish the lab without doing the lab.
Input: You could cheat and someone else could do the lab.
Output: Then the hearer risks failing the course and being expelled from the program.
Input: Why?
Output: It is a necessary condition for passing a course that you do your own work. It is a necessary condition for completing the program that you be enrolled in the program. Cheating entails no longer being in the program.
Input: You may not be caught.
Output: Then you contradict the motive for taking the course and pursuing the degree: you do not learn anything by doing the lab.

The preceding dialogue demonstrates the main features of the proposed system. This system is able to recognize two speech acts: advice and warnings. It is able to paraphrase the user's input and report which of those speech acts the input represents. When there is a conflict between goals, it tries to resolve that conflict. If it is not possible to resolve a conflict, it simply reports the conflict to the user. Lastly, the system is able to answer simple questions about its own reasoning process.

We will now step through the sample dialogue, indicating the capabilities that the system must have to accomplish each stage of the dialogue.

Input: A test is scheduled for Monday. You should study for the test.
Output: The speaker has advised the hearer to study for an upcoming test.

The system is able to recognize advice, and generates the missing antecedent and consequent necessary to represent that advice as a practical argument. It stores this information in a frame. The following is the English equivalent of the contents of the practical argument frame for this example.

Event: A test is scheduled for Monday.
Antecedent: If you study for the test,
Consequent: then you are more likely to pass the test,
Imperative: so, you should study for the test.

From the input, and from what the system knows about the structure of a practical argument, the system is able to infer the missing antecedent and consequent. The system also contains a knowledge base of plans, goals, and other facts. It compares the practical argument against the knowledge base, and determines that the inferred antecedent and consequent are in accord with what it knows about plans and goals. (The antecedent corresponds to a plan, and the consequent corresponds to a goal.) This inference process will be discussed in detail in Chapter 7. We should also note here that the system is capable of accepting the full form of the practical argument as input.

The last feature to be indicated at this point in the dialogue is that the system can distinguish advice from a warning. The importance of this distinction rests solely in what it tells us about the relation between the consequent and what the speaker believes are the interests of the hearer. If the antecedent and the imperative are both positive or both negative, then the statement is interpreted as advice, and the consequent is believed by the speaker to be a benefit to the hearer. Otherwise, the statement is interpreted as a warning and the consequent is believed by the speaker to be detrimental to the hearer. An example of a warning would be:

The stove is hot. Don't touch the stove.

For a warning, the practical argument frame will take the same form as it did for a piece of advice, but the value of the speech act slot will indicate that the consequent is to be interpreted as harmful to the hearer.

Input: Why?
Output: The speaker believes that the hearer will improve his chances of passing the test if the hearer studies for the test.

Input: Does the speaker believe that the hearer wishes to pass the test?
Output: Yes.
Input: Why?
Output: A test is a task. The normal goal of a person with a task is to wish to successfully complete that task.

This section of the dialogue demonstrates the basic question answering capacity of the system. It resembles an explanation facility in an Expert System. With no indications to the contrary, the system assumes its last reply as the context, and answers the user's first question by drawing on the plan, the goal, and the relationship between them as an explanation of the speaker's advice to the hearer. In order to answer the second question, the system only needs to recognize that the speech act involved is advice; when the speech act is advice, the consequent (passing the test) is believed by the speaker to be a benefit to the hearer. However, the third question asks for an explanation of the system's previous reply, and this seems to call for something other than a description of the system's underlying reasoning process. This third question can be better answered by retrieving a goal directly from the database, and then matching that goal to the current context. The current context may simply involve a reference to the active practical argument frames, or it may involve the creation and maintenance of additional frames that keep track of the user's beliefs.

Input: A lab is due on Monday. You should do the lab.
Output: Is the lab due the same day that the test is scheduled?
Input: Yes.
Output: The hearer should finish the lab, and then should study for the test.
Input: Why do the lab before studying for the test?
Output: Doing the lab is a necessary condition for finishing the lab. Studying for the test is not a necessary condition for passing the test.

This section of the dialogue introduces a potential goal conflict. Once the system establishes that the time windows are the same for both plans, it assumes a potential conflict (perhaps we could call it an apparent conflict), and tries to order the plans so that the potential conflict (one person cannot do two things of the same order at the same time) disappears. In this case, it draws upon the difference in the relationships between the respective plans and goals, and puts the relationship that is necessary before the one that is not strictly necessary. Relationships between plans and goals can be necessary or not necessary, and sufficient or not sufficient.

Input: It will probably take all of the available time to finish the lab.
Output: The hearer should do the lab.
Input: Did you forget about the test?
Output: The hearer may be able to pass the test without studying for the test. The hearer cannot finish the lab without doing the lab.

Now the apparent conflict becomes a true conflict and can only be resolved by abandoning one of the plans. The system notices that abandoning the plan of doing the lab entails abandoning the goal of finishing the lab, while abandoning the plan of studying for the test does not mean abandoning the goal of passing the test, but only involves lessening the chances of achieving that goal. In this and the preceding exchange, the system goes beyond recognizing advice, and actually seems to generate advice. This capacity to generate advice is limited to ordering plans, and making choices between plan scenarios.

Recognizing logical forms and doing elementary reasoning are not the only abilities such a system must have. Here the system offers an explanation of its reasoning in response to a question from its user, but, strictly speaking, that question has been formed in such a way that it needs more than a literal interpretation to be recognized as a request for an explanation. By a purely literal interpretation, the user would appear to be questioning the memory of the system. The system, however, has access to enough information from the dialogue context to be able to recognize that it is far more likely that the user is seeking an explanation here, and so it answers accordingly. The user's question is a phenomenon of language that can be called an "indirect speech act", where the actual meaning is something other than the literal interpretation. This phenomenon will be discussed in Chapter 2 and Chapter 4.

Input: You could cheat and someone else could do the lab.
Output: Then the hearer risks failing the course and being expelled from the program.
Input: Why?
Output: It is a necessary condition for passing a course that you do your own work. It is a necessary condition for completing the program that you be enrolled in the program. Cheating entails no longer being in the program.

The speaker proposes an alternate solution to the goal conflict. The system, however, reports that this violates higher level goals. The system maintains a goal hierarchy and always checks to make sure that a given piece of advice does not contradict any higher level goals. Here the system does not refer directly to the lab or to the test, but refers to the higher level goals that may be overturned by the proposed plan.

Input: You may not be caught.
Output: Then you contradict the motive for taking the course and pursuing the degree: you do not learn anything by doing the lab.

The preceding dialogue fragment involved a contingency - getting caught. The system had assumed that the risk involved is sufficient, and so stopped at reporting that risk. When the user makes explicit reference to that risk, and suggests that the contingent element may not be realized, then the system reports on the situation if that is the case. This may mean that it moves one level higher in the goal hierarchy, and reports on the effects of cheating at that level. However, a purely hierarchical arrangement may not be flexible enough for that kind of exchange. Perhaps keeping track of the current state, and noting state changes would work better at this stage of the dialogue.

These last two exchanges highlight what is a concern throughout the process of engaging in a dialogue. Only a limited amount of information should be reported to the user, and that information should always be pertinent to the user's question or suggestion. There are a lot of choices to be made as to what the system can assume the user already knows or can be expected to know.

1.3. System Components

Components that are believed necessary to implement such a system are listed, and briefly described, below. These components can roughly be divided into two major areas: those components that deal with language issues, and those that deal with world knowledge issues.

The implementation will be written in C-Prolog and will employ a frame package developed and described in [HISS 87].

1.3.1. Modules

The proposed system should contain the following major modules.

A syntax-based parser accepts English sentences as input, and produces, as output, frames that serve as internal representations of those sentences.

A **case frame interpreter** accepts frames produced by the parser as input, and produces, as output, frames that represent the semantic interpretation of those sentence frames.

A **generator** accepts frames as input and produces English sentences as output. The output is either a paraphrase of a practical argument, the system's attempt to resolve a conflict, or the answer to a question concerning the system's reasoning process.

A predominantly **syntactic dictionary** serves the parser and the generator.

A **semantic dictionary** serves the case frame interpreter. This dictionary consists of a hierarchical network defining classes to which nouns and pronouns belong, and frame definitions of verbs, indicating cases that go with each defined verb, as well as what classes of nouns may serve as values for these cases.

A **context interpreter** examines a completed sentence frame (composed of both a syntactic and a semantic frame representation), and determines how to connect it with previous sentence frames in the effort to maintain an overall context. This interpreter works in conjunction with frame definitions created at the context level.

A **knowledge base** contains plans, goals, the relationships between them, and any conditions which affect these plans and goals.

A **planner** evaluates plans, detects goal conflicts, and attempts to resolve such conflicts. (This module will not be implemented here.)

1.4. Overview

Chapter 2 examines some of the theoretical issues involved in this thesis and reports on some of the research that has been done in the area of natural language understanding. Topics covered are the nature of practical arguments, the use of planning theory in natural language understanding systems, and the role of speech act theory in constructing dialogue systems.

Chapter 3 looks at some of the issues involved in constructing an automated dialogue system. It reports on an experimental system (UC) intended to function as a help facility for users of the UNIX operating system.

Chapter 4 briefly examines some of the theoretical issues involved in representing meaning at the sentence level. The center of attention here is the use of cases as a way of representing semantic content.

Chapter 5 gives a detailed description of a system that handles the syntactic and semantic issues introduced in Chapter 4. The implementation described in this chapter includes the parser, case frame interpreter, and the dictionaries listed in section 1.3 above.

Chapter 6 revisits the theoretical issues first presented in Chapter 4. It discusses those issues from the perspective of what has been learned from the effort to implement a natural language understanding system.

Chapter 7 describes an interpreter and accompanying frame system for recognizing and handling practical arguments as a part of a dialogue.

2. Theoretical Issues

2.1. Practical Arguments

The following discussion of practical arguments is adapted from [DONA 86], chapter 1.

A practical argument is "an argument whose conclusion specifies an action to be taken and whose premises provide reasons for that action." The following are two examples:

> The stove is hot, so don't touch the stove.
>
> A test is scheduled for tomorrow. You should study for it.

The conclusion of a practical argument may be identified as any of a number of speech acts. The first example above is a warning, and the second is advice. In the absence of a performative verb (e. g. "advise", "warn"), that identification depends upon the premises. In terms of dealing with a connected discourse, we can view these premises as supplying a context which enables us to identify the imperative conclusion, and so extend our ability to represent it beyond its more general linguistic identity as an imperative. Given the imperative, "don't go near the stove", any of a number of possibilities can be supplied by its premises or context:

1. If you touch the stove, you will burn yourself.
   So, don't go near the stove.
   (warning)

2. When they are convinced that you are not going to feed them, they will leave.
   So, don't go near the stove.
   (advice)

3. It's my turn to cook tonight.
   So don't go near the stove.
   (request)

4. I can't take another one of your casseroles. If you want to continue living,
   don't go near the stove.
   (threat)

5. If you can control your impulse to cook, I will take you out to dinner tonight.
   So, don't go near the stove.
   (promise)

6. I'm in charge here.
   So, don't go near the stove.
   (order)

2.1.1. A Form for Practical Arguments

One of the necessary functions of a natural language understanding system is the ability to make inferences, and so fill in text which the user might consider so obvious as to be unnecessary to state explicitly. A practical argument provides us with a fairly regular pattern by which such inferences can be made. Given the text,

The stove is hot, so don't touch it.

we can infer the full form of the argument:

1. The stove is hot.
2. Hot things cause burns when touched.
3. So, if you touch the stove, you will burn yourself. (1,2)
4. But you wish to avoid burning yourself. (agent's assumed interests)
5. So, don't touch the stove. (3,4)

We can see that the role of "the stove is hot" is simply to be combined with the assumed common knowledge stated in #2 in order to provide evidence for the major premise, #3. Also, the agent's interests in this matter are so obvious as to be readily assumable, and when that assumption is combined with the major premise, the conclusion, "don't touch the stove" is readily obtainable.

Working from this full form of a practical argument, [DONA 86] arrives at a way of expressing practical arguments, as follows:

If X then Y.
So (don't) do Z.

The intention is to capture the argument in the briefest form possible without losing meaning or structure. The form adopted settles for expressing the major premise and the conclusion. (Just why the agent's interests can be assumed will be explained shortly.) The system which used this form had the ability to recognize the major premise (If X then Y) if it were entered explicitly, and also had the ability to infer the major premise if enough evidence were supplied. Since the only concern was the recognition of the speech act involved, the evidence (statements 1 and 2 in the full form) could be ignored once the major premise was established. Since our concern is now extended to include the evaluation of practical arguments, it becomes necessary to retain the evidence and so expand the form by which we represent practical arguments. This is such a minor change in the representation scheme that the algorithms developed in [DONA 86] can be retained and used here. It remains true that we can safely assume the agent's interests (see below).

2.1.2. Two Algorithms

[DONA 86] and this system will accept input of practical arguments either in the explicit form described above, or in an enthymematic form which can then be filled in by making inferences. The enthymematic form must, of course, present enough evidence to allow the program to induce the major premise of the argument. If the user says,

> A test is scheduled for Monday.
> You should study for the test.

the system will be able to arrive at the major premise form:

> If you study for the test, then you are likely to pass the test.
> So, study for the test.

In the process, this system will produce a frame representation of the argument as follows:

```
argument1(
        event1(
                event:  A test is scheduled for Monday.
                goal:  A test is to be passed.
                )
        antecedent:  If you study for the test,
        consequent:  then you are more likely to pass the test,
        conclusion:  so, study for the test.
        speech_act:  advice.
        ).
```

Two algorithms which allow the reduction to major premise form are presented below. In order to produce the full form of the argument it is only necessary to save the evidence as well as the inductions.

Example 1.

> The stove is hot. Don't touch the stove.
> (and the knowledge base contains the fact that "hot things cause burns when touched")

Algorithm 1.

1. Assume that the real premise (RP) of the argument is of the form "If X then Y" where
   X = the negation of the propositional content of the conclusion
   Y = some as yet unspecified harm to the hearer

RP: If you touch the stove, <something bad will happen>

2. Also assume that the role of the stated premise (SP) is to provide evidence for RP.
3. The consequent of RP (viz. you will burn yourself) can now be deduced from SP and the known fact that hot things cause burns when touched.

Algorithm 1 applies to cases where an agent is being urged to do something to avoid an undesirable state of affairs. Algorithm 2 deals with cases where an agent is being urged to do something in order to bring about a desirable state of affairs.

Example 2.

short version:

The movie is excellent. So see it.
(and the knowledge base contains the fact that "excellent things are pleasing to see")

long version:

1. The movie is excellent.
2. Excellent things are pleasing to see.
3. So if you see the movie you will be pleased. (1,2)
4. (You want to be pleased. (agent's assumed desires))
5. So see the movie. (3,4)

Algorithm 2.

1. Assume that the real premise is of the form "If X then Y", where
   X = the propositional content of the conclusion
   Y = some as yet unspecified benefit to the hearer.
2. Also assume that the role of the stated premise (SP) is to provide evidence for RP.
3. The consequent of RP (viz. "you will be pleased") can now be deduced from SP and the known fact that excellent things are pleasing to see.

These two algorithms differ from one another in the relation between the antecedent (X) and the conclusion (Z), and the resulting relation between the consequent (Y) and the interests of the hearer (actually, what the speaker believes are the interests of the hearer). In algorithm 1, X = not(Z), and Y is marked as harmful to the hearer. In algorithm 2, X = Z and Y is marked as beneficial to the hearer. Both algorithms mark Y as harmful or beneficial even before the content of Y has been deduced. The two algorithms express this relation as a conjunction of two boolean variables: as long as we know the relation between X and Z, we know the relation between Y and the interests of the hearer. Before we know the content of Y, we know whether that content is beneficial or harmful to the hearer. This is why we do not have to explicitly state the interests of the hearer in our representation of a practical argument: those interests are already implicitly represented in the form we have adopted for expressing practical arguments:

If X then Y.
So (don't) do Z.

It remains to demonstrate that these two algorithms include all the cases where the propositional content of the antecedent (X) and the conclusion (Z) are the same.

Given that the propositional content of X and Z are the same (so that either X = Z or X = not Z), and that the relation between Y and the agent's interests can be either beneficial or harmful, but not both, it follows that two more algorithms are possible.

Algorithm 3:

X = the negation of the propositional content of the conclusion
Y = some as yet unspecified benefit to the hearer.

Algorithm 4:

X = the propositional content of the conclusion
Y = some as yet unspecified harm to the hearer.

No examples have been included with Algorithms 3 and 4 simply because there are no cases to which they apply.

> This follows from a purely conceptual point about the nature of arguments. Consider the following argument schemata.
>
> If you do X then Y.
> So do X.
>
> If you do X1 then Y1.
> So don't do X1.
>
> In the first instance, the fact that doing X will lead to Y is cited as a sufficient or decisive reason for you to do X. But that fact could not possibly count as a sufficient reason for you to do X unless Y is some desirable state of affairs. (Algorithm 4 is ruled out.) In the second instance, the fact that doing X1 will lead to Y1 is cited as a sufficient reason for you not to do X1. But that fact could count as a sufficient reason for not doing X1 only if Y1 is some undesirable state of affairs. (Algorithm 3 is ruled out.) [DONA 86] (p. 6)

The above considerations are so intuitively obvious that it is difficult to do more by way of demonstration to prove that only the first two algorithms apply to actual cases. The principle involved cannot appeal to any outside knowledge concerning the benefit or harm of an action, so we cannot produce any examples for algorithms 3 and 4. If we try:

If you touch the stove, you will burn yourself. So touch the stove.

then we are forced to assume that the speaker believes that being burned is a benefit to the hearer. We should remember that all we are concerned with here are the speaker's beliefs concerning the interests of the hearer. In a sense, the logical restrictions applied here define the good faith of the speaker, something we must assume in order to retain the logical form we have adopted. The form itself provides us with such an extraordinary consistency that bad faith on the part of the speaker is unthinkable. At the stage of recognizing the speech act and the practical argument, this is not a fault or flaw in our reasoning process, since we are not concerned yet with evaluating the practical argument, but only with representing it in a strictly consistent form.

2.1.3. Applications and Limitations

[DONA 86] is actually a good deal more complicated than the preceding description indicates. It deals with cases where the propositional content of the antecedent and the conclusion are not identical, and it recognizes four different

speech acts: advice, warnings, conditional promises, and conditional threats. This project, however, will limit itself to the basic issues addressed in that study. It will only deal with practical arguments where the propositional content of the antecedent and the conclusion are the same, and it will only recognize two speech acts, advise and warnings. Limiting our study to these conditions, allows us to retain the basic features of the previous study and to retain the most useful generalizations contained therein. We can thus concentrate more attention upon the task of evaluating practical arguments, and we can allow that attention to be a guideline for building a system that more completely processes its input.

The proposed system will be limited to recognizing only advice and warnings because these two speech acts correspond to algorithms 1 and 2 described above. A warning is the speech act recognized by algorithm 1 (where X = not Z), and advice is the speech act recognized by algorithm 2 (where X = Z). Thus we can easily distinguish the two speech acts from one another, and we can use that distinction to establish the interests of the hearer, determining whether the consequent is believed to be beneficial or harmful to the hearer. Also these two speech acts are the ones that are most likely to be of use in a dialogue system.

2.1.4. Summary

A practical argument specifies an action to be taken, and provides reasons for that action. It serves as a bridge between informal logic and common discourse, providing us with both a logical representation of beliefs, and a way to supply context to the kind of informal communication that a dialogue system might be expected to handle. Two speech acts, advice and warning, can be easily expressed as practical arguments, and these speech acts offer the most promise as we attempt to build a natural language dialogue system.

2.2. Plans and Goals

Most of the current research in natural language understanding and dialogue systems revolves around the application of planning theory to goal directed discourse. The basic strategy involves recognizing the goals of an agent and generating inferences by knowing the plans that agent might employ in trying to achieve his goals. [ALLE 83] gives a good brief description of the world view underlying planning theory:

> ...the world is modeled as a set of propositions that represent what is known about its static characteristics. The world is changed by actions which can be viewed as parameterized procedures. Actions are described by preconditions, conditions that must hold before the action can execute, and by effects, the changes that the action will make to the world. Given an initial world state W and a goal state G, a plan is a sequence of actions that transforms W into G.

Planning theory, as it first evolved in the area of problem solving, was not appropriate for the needs of natural language understanding. A problem solver such as GPS, [NEWE 59], or STRIPS, [FIKS 71], never had to be concerned about the nature or source of its goal. The problem solver would simply accept a goal supplied by the user, and would attempt to construct a plan to achieve that goal. The process of understanding connected discourse requires a different kind of control structure: here the system may have to recognize many interrelated goals, but more than likely will not need to evolve highly detailed plans for achieving those goals, opting instead for the use of "canned" plan-goal pairs sufficient for making connections and supplying context. These early problem solvers dealt with robot worlds that tended to be highly constrained and physical, so that a great amount of detail would be necessary in the construction of a plan; a natural language system deals with a human world and only needs an appropriate level of detail (see [WILE 78]). Despite these differences, the basic issues and the world view elucidated by Allen underlie both problem solving and natural language understanding, so that [WILE 83] aims at developing a theory of planning in common sense situations alongside a theory of understanding; such a comprehensive approach requires three theories:

> 1. The theory of planning, which describes the process by which an intelligent agent determines and executes a plan of action.
> 2. The theory of understanding, which describes the process by which an understander comes to comprehend the behavior of another.
> 3. The theory of plans, which describes the knowledge about planning used for both these tasks.
> ([WILE 83], p.6).

Ever since its introduction as a way to approach text understanding in [SCHA 77], planning theory has retained its central place in the research directed at understanding connected discourse. Both Wilensky (first at Yale, and later at UC Berkeley) and Allen (at the University of Rochester) have relied heavily upon it to make the task of understanding natural language a tractable one. Unlike the language driven approach outlined in the previous section, planning theory approaches are strictly domain driven, concentrating their attention upon the situations that surround the use of language, rather than upon the structure of language itself. In this way they fit into an overall pattern of development in the world of intelligent systems: "It has become axiomatic to most computer scientists that no intelligent system can be constructed unless it employs large quantities of world knowledge" ([WILE 83], p. 1).

2.2.1. Making Inferences: MARGIE, SAM, and PAM

A central issue in the task of understanding connected discourse is the need to fill in information that the speaker believes is common knowledge and so does not need to be mentioned explicitly. For instance, suppose an automated

system were required to understand the following:

> John remembered that Monday is Mary's birthday.
> He drove to the drugstore to buy a birthday card.

We really cannot say that a system understands this simple text unless it knows that John intends to send Mary the birthday card, and that sending a card is a way that people have of recognizing and celebrating the birthday of a friend.

Just how to go about making such inferences was the subject of a series of natural language story understanding projects undertaken at Yale University. The first of these, MARGIE, [SCHA 73], contains a memory component that is capable of generating such needed inferences. Possible inferences are divided into sixteen classes, a few of which are listed below.

1. Resultative inference.
   Input: John gave Mary a car.
   Inference: Mary has a car.

2. Motivational inference.
   Input: John hit Mary.
   Inference: John probably wanted Mary to be hurt.

3. Functional inference.
   Input: John wants a book.
   Inference: John probably wants to read the book.

4. Feature inference.
   Input: Andy's diapers are wet.
   Inference: Andy probably is a baby.

(taken from [WILE 78], p. 7)

An inference procedure is associated with each class of inference; if the input matches the criteria specified for that class, then the associated procedure generates the inference; inferences are, in turn, generated for other inferences. A strength value is maintained for each inference; this value grows weaker with each generation of an inference until the strength value falls below an arbitrary value, and the process halts.

Reiger's system for generating inferences has only two formal limits set upon the way it defines meaning. The first of these is the sixteen classes of inference that can be made; the second is the waning strength value maintained as the inferences chain. There is a set of items to look for, and there is a time to stop looking. In terms of the flow of control in a computer system, Reiger's method is essentially bottom-up, or data driven. Although the sixteen classes of inference could be considered as hypotheses, they are constructed at a very low level, as close to the data as they can be,

and only represent a minimal amount of knowledge. If these classes comprehend all the kinds of inferences needed to understand connected discourse, then the system is theoretically ready to handle any story.

The problem with this method is two-fold. First, it is computationally expensive, generating between 500 and 1,000 inferences for a single utterance. We would like to think that this is unnecessary; while a human being may indeed possess that much background knowledge about a single utterance, it is unlikely that he would need to actually apply that much knowledge in order to understand a simple narrative or in order to maintain a dialogue. If we look at the 'birthday card' example given previously, we can see that the important connections are supplied by the high level inferences which specify an action that an agent is likely to take (John intends to send Mary the birthday card), and which give a reason for taking that action (to celebrate her birthday). We suspect that we can represent these connections sufficiently without generating a lot of details about birthdays, drugstores, and cars. The second problem is that some inferences that might be necessary for understanding may not be reachable by Reiger's method. [WILE 78] gives the example of a policeman giving a motorist a ticket. In order to understand a story with this event in it we need to know some specific knowledge about the role of a policeman in relation to a motorist. It seems that we need a higher level of more specific knowledge to make this kind of inference as quickly and as cheaply as possible.

SAM (see [SCHA 77]), which stands for Script Applier Mechanism, was an attempt to overcome the difficulties and shortcomings of Reiger's system. SAM works in a top down fashion. Various hypotheses concerning the subjects the system will process are stored in memory in the form of scripts. These scripts deal with highly stereotyped situations, specifying the actions that can be expected to occur in a particular situation. Once an input matches the subject matter in one of these scripts, then the script can take over the task of processing the rest of the story. Explicit statements can be matched against the contents of the script, and any information which is needed but is not explicitly mentioned can be found among the actions contained in the script.

The advantages of SAM's processing method are that the proliferation of inferences has been eliminated, and specific knowledge about roles and institutions is readily available for making the kind of high level inferences that are likely to be useful for making connections between the successive statements in a story. One disadvantage is that a great many scripts, each containing rather extensive knowledge about a specific situation, are likely to be needed to understand even very simple stories. While Reiger's system is computationally expensive, Cullingford's system is expensive in terms of the memory requirements it makes on a system. That is, MARGIE takes too long to do the job, while SAM

consumes too much space to be practical as a general purpose story understander. A second problem with SAM is that it has no flexibility. If an event does not match a script entry, then SAM is lost in an effort to process that event. [WILE 78] gives the following example to highlight that inflexibility:

> ex:
>
> John went to a restaurant. The waiter came over and said they were out of menus. John asked the waiter what he recommended.
>
> SAM could not have understood this because SAM does not know why a waiter brings the menu. From SAM's viewpoint, reading the menu in a restaurant is a ritual, just as if prayer were required before ordering. ([WILE 78], p. 13)

PAM is an effort to overcome the shortcomings of MARGIE and SAM by incorporating a knowledge of intentionality (of what the speaker intends) into its knowledge of events. PAM breaks the large chunks of data found in the scripts of SAM into smaller pieces representing specific actions or plans. Each of these plans is associated with a goal which expresses the intentions of an agent who might carry out that plan. Thus PAM does know why the waiter brings the menu, and it can process alternate plans (asking the waiter what he recommends) in order to understand how a goal (getting something to eat) is achieved.

The theoretical basis underlying PAM is expressed in *Scripts, Plans, Goals, and Understanding*, [SCHA 77]. Here a theory of intentionality is advanced: themes, plans, and goals are the elements that make up that theory. Themes deal with broad issues in life and are divided into several categories. Preserving health, for instance is one such theme. Themes give rise to goals. Goals, in turn, are achieved by executing plans. Often plans have subplans and a goal associated with a plan may be a necessary condition or subplan for achieving a higher level goal. These ideas are incorporated into PAM and form the foundation for a reasonably sophisticated approach to the problem of understanding connected discourse.

PAM achieves a plateau in these experimental efforts to find a way to generate inferences and so make connections between successive statements in a connected discourse. It is unlikely that either the memory component of MARGIE or the scripts of SAM were ever taken seriously as a practical way to process a narrative. They did however explore ways to approach the problem, and provided researchers with a good understanding of the difficulties involved. The basic ideas underlying PAM provide a reasonable approach to the problem, and so the new direction becomes one of studying and attempting to improve the ways that those ideas are to be implemented. The basic ideas of PAM retain their place in the later work of Wilensky, and are similar to the overall approach of Allen. Whether or not these ideas represent an insight into human intentionality (as Schank and Wilensky claim), they do nevertheless offer a good foundation for a

computer system. Knowledge is broken up into small chunks, an arrangement that has the proper aspects of modularity and flexibility that we expect in a viable computer system. The flow of control is reasonably sophisticated that is, it matches the task at hand. Processing can begin in a bottom-up, data driven manner. Once a goal has been detected, then expectations can be formed as to the next input (expectations mean that we are prepared for something to happen, not that we predict that something will happen). If these top-down expectations are not met, then the system can return to a bottom-up method. The same basic ideas that work for story understanding can also be adapted for the needs of a dialogue system, as indeed they are by Wilensky for UC (see chapter 3), a dialogue system that attempts to answer questions about the UNIX operating system ([WILE 84]).

2.2.2. Goal Conflicts

Having arrived at a reasonable way to generate inferences, the next stage in the task becomes apparent. As [WILE 78] points out, the simple detection of a goal and a matching plan is hardly an interesting enterprise. A story that is composed only of situations where an agent has a goal, and formulates a plan to achieve that goal, is not much of a story. Any real story has a point of interest, or a succession of such points of interest, and these elements of interest can be described in terms of the ways in which goals interact with other goals. It is precisely these points of interest that generate the need to tell a story or which cause a dialogue to continue. Most of PAM is concerned with exploring these goal interactions, focusing attention upon cases where the goals of an agent are in conflict with one another, or are in conflict or in concord with the goals of another agent. Allen's focus is similar, only the emphasis shifts slightly in that his system looks for impediments to carrying out a plan, and generates expectations that the dialogue will focus upon recognizing and removing such impediments ([ALLE 86]).

2.2.3. Application

The structure of a practical argument closely resembles the structure of a plan-goal pair, so we should be able to draw upon this area of research. A practical argument specifies an action to be taken, and supplies reasons for taking that action. In terms of plans and goals, the action to be taken corresponds to a plan, and a reason for taking that action corresponds to a goal. Even a language driven system must have access to world knowledge in order to evaluate arguments; the structural similarity between a practical argument and a plan-goal pair will allow this world knowledge to be

stored in terms of plans and goals, and will allow this stored knowledge to be easily accessed by a system that manipulates antecedents and consequents. We would like to think that a language driven system that can evaluate advice could be part of a larger, more general dialogue system; since most of the current work being done to develop such larger systems makes use of planning theory, we should at least be able to demonstrate that a system based on practical arguments can be incorporated into a planning system.

While we can make use of the research in task-oriented dialogues, we should realize at the outset that there is a theoretical difference between a practical argument and a plan-goal pair. The relationship between a plan and a goal is psychological, while the relationship between a reason and an action, as expressed in a practical argument, is intended to be logical. In a broader context, the kind of system that is built from the recognition of plans and goals is likely to be domain dependent and to have a belief system of its own, one which needs to express a great deal of world knowledge. The ideal that lies behind the use of practical arguments aims at a language driven system which uses facts to recognize the beliefs of the user, rather than having beliefs of its own. Nevertheless, the practical argument system is going to need world knowledge in order to evaluate the soundness of arguments, and it may not make any difference in the final product what we call that knowledge. In any case, the paths of the two approaches are parallel, and the difference that separates them does not so much represent a conflict as it represents a small tension, one which should tune awareness to what we are doing at each step of the process of building a system.

2.3. Speech Acts

In section 2.1 we indicated that practical arguments could be identified as various speech acts. The same imperative ("don't go near the stove") could be any one of a number of speech acts, depending upon the context supplied by the premises of the argument in which that imperative could stand as a conclusion. Among the examples given, two speech acts, advice and warnings, were singled out as playing a dominant role in the system being developed here. In the course of that discussion, it was taken for granted that we can all recognize a promise or a threat in our normal, common experience of language. While we went to greater lengths to formally define, by means of algorithm 1 and algorithm 2, the difference between a piece of advice and a warning, still, we left it to common sense to arrive at an understanding of how these speech acts fit into our normal experience of a dialogue. Now we would like to turn our attention to a more formal consideration of just what a speech act is and how a theory of speech acts has influenced research in constructing automated dialogue systems. We hope that these considerations will provide us with a bridge between the language

driven logic system represented by practical arguments, and the domain driven psychological system represented by planning theory.

Speech acts, by their very nature, belong to the language-logic aspect of our considerations, and were given their fullest development along the lines of a theory in [SEAR 69] (*Speech Acts: An Essay in the Philosophy of Language*). At the same time, they have been adopted by researchers in the area of automated dialogue systems because their formal properties have striking similarities to the characteristics possessed by plan-goal theories. From this latter point of view, they are perceived as a planning theory approach whose domain is language itself. We will attempt to work our way back from the considerations of plan-goal theory to our first consideration, which is the ideas presented in [DONA 86]. Along the way, we can hopefully arrive at a better sense of the strengths and weaknesses of the system proposed here. We will also take a brief look at the notion of an indirect speech act, a problem child that should help us to see clearly that any automated dialogue system must be able to construct its notion of meaning at several different levels.

2.3.1. Domain Goals and Language Goals

The following dialogue and commentary is taken from [ALLE 82], a brief report on the early stages in the development of ARGOT: The Rochester Dialogue System. It is described as "a slightly cleaned up version of an actual dialogue between a computer operator and a user communicating via terminals", with the implication that an automated system might be expected to play the role of the operator.

(1) User: Could you mount a magtape for me?
(2) It's T376.
(3) No ring please.
(4) Can you do it in five minutes?
(5) Operator: We are not allowed to mount that magtape.
(6) You will have to talk to the head operator about it.
(7) User: How about tape 241?

> One of the major advances made in ARGOT is that it recognizes multiple goals underlying utterances. For example, consider the user's goals underlying utterance (2). From the point of view of the task domain, the user's goal is to get the tape mounted (by means of identifying it). From the point of view of the dialogue, the user's goal is to elaborate on a previous requests (sic), i.e. the user is specifying the value of a parameter in a plan that was recognized from the first utterance. In the ARGOT system we recognize both these goals and are investigating the relationship between them.

In order to handle these multiple goals, ARGOT attempts to employ two high level goal reasoners. One is concerned with task domain goals such as "mounting a magtape, reading files, etc."; the other is concerned with communicative goals such as "introducing a topic, clarifying or elaborating on a previous utterance, modifying the current topic, etc.". It is still a matter of research, at this early stage of development, as to just how these two sets of high level goals will

interrelate, but the general plan is to have the communicative goals serve as input to the task level goals.

> The high level communicative goals reflect the structure of English dialogue and are used as input to the task level reasoner. In other words, these goals specify some operation (e.g., introduce goal, specify parameter) that indicates how the task level plan is to be manipulated.

[ALLE 83] provides us with a good general description of why speech acts are interesting to those constructing a plan-goal based dialogue system. First he describes how a world model looks to planning theory; then he draws parallels between this world model and a theory of speech acts.

> ... the world is modeled as a set of propositions that represent what is known about its static characteristics. The world is changed by actions which can be viewed as parameterized procedures. Actions are described by preconditions, conditions that must hold before the actions can execute, and by effects, the changes that the action will make to the world. Given an initial world state W and a goal state G, a plan is a sequence of actions that transforms W into G.
>
> Austin [Austin 62] suggested that every utterance is the result of several actions or speech acts. We are particularly interested in the class of speech acts that includes requesting, warning, asserting, informing, and promising. These speech acts are appropriate only in certain circumstances. In particular, they may require the speaker and the hearer to have certain beliefs and intentions. For example, to sincerely INFORM you that I am tired, I must believe that I am tired and I must intend to get you to believe that I am tired.
>
> Cohen [Cohen 1978] demonstrated that speech acts such as requesting and informing can be modeled successfully as actions in a planning system. He showed how speech acts may be planned in order to achieve specific (typically non-linguistic) goals. ([ALLE 83], p. 11)

The concept of a speech act is easily incorporated into Allen's system because it can be modeled on the already well known model of planning theory. Language is viewed as an act and like any other act it has preconditions which make it possible to perform that action, and like any other act it is intended to achieve an effect. Also, the concept of a speech act is useful because it greatly expands the capabilities of the system. In [ALLE 86] the plans associated with speech acts do not directly affect the world model based on domain goals and plans, but rather affect the beliefs of the hearer. This adds complexity and sophistication to the system, since there is now room to represent the possibility that the hearer may reject the communicative goals of the speaker. This distinction between the beliefs of the hearer and his goals can lead to ever more sophisticated representations of the relation between speech act goals and domain related goals, depending upon the needs of the system.

2.3.2. Language as Action

Allen's understanding of speech acts seems consistent with the general tenor of Searle's study ([SEAR 69]). According to Searle, the basic units of language are speech acts, and a theory of language is to be contained within a theory of action.

> The reason for concentrating on the study of speech acts is simply this: all linguistic communication involves linguistic

> acts. The unit of linguistic communication is not, as has generally been supposed, the symbol, word or sentence, or even the token of the symbol, word or sentence, but rather the production or issuance of the symbol or word or sentence in the performance of the speech act ... speech acts ... are the basic or minimal units of linguistic communication. ([SEAR 69], p. 16)

> It might be objected to this approach that such a study deals only with the point of intersection of a theory of language and a theory of action. But my reply to that would be that if my conception of language is correct, a theory of language is part of a theory of action, simply because speaking is a rule governed form of behavior. Now, being rule governed, it has formal features which admit of independent study. But a study purely of those formal features, without a study of their role in speech acts, would be like a formal study of the currency and credit systems of economics without a study of the role of currency and credit in economic transactions. ([SEAR 69], p. 17)

Searle's hypothesis is "that speaking a language is engaging in a rule-governed form of behavior." ([SEAR 69], p. 22) Whether this hypothesis holds true or not, it nevertheless represents an opinion that exactly serves our purposes: that is, anything which is not rule governed is not likely to be captured in a computer system towards any useful end. These rules, however, have nothing in common with the rules of syntax and semantics; that is, they do not describe language from the inside, regulating how lexical items may go together to form a sentence or utterance. The rules generated by speech act theory are applied to language from the outside; language is described in terms of the intentions of the speaker, the physical and social realities that are referred to by the speaker's words, and the effects that those words can be expected to have upon the hearer. These rules are conditions that must be present for a given speech act to be effective as a serious communication. The following example of how these conditions apply is taken from [DONA 86].

> Suppose that a speaker S utters a sentence St, "Please open the door". Is S requesting that some hearer H open the door? Not necessarily. S, for example, may be reciting a line in a play or engaged in a language lesson. When S's utterance is intended as a serious request, there are at least six situational features that are normally present.
>
> 1. There is a hearer(s).
> 2. There is a door which is singled out by the context.
> 3. The door is not already open.
> 4. It is possible for H to open the door.
> 5. S has some interest in getting H to open the door.
> 6. The words are uttered in an attempt to get H to open the door.
>
> Note that these situational features together serve as a template or schema for non-defective requests to open a particular door. In situations in which one or more of these features is not present, the request misfires.
>
> .... Searle's contention amounts to this. The set of conditions C under which a speech act Sp is performed non-defectively creates the possibility of performing Sp by serving as its set of constitutive rules. ([DONA 86], p. 30)

These conditions then go together to "constitute" the speech act in question. They are not regulative in the sense that rules of etiquette, for instance, regulate social behavior. Rather the conditions are the act in question, in the same way that the rules of chess are the game of chess. If two players abide by any different rules (say, for instance, that a rook can move diagonally) then they are playing a different game from chess. If any of the conditions listed above are

missing, then the speech act is not a serious request "to open the door". In addition, both the speaker and the hearer must fully accept the rules involved, and if in subsequent conversation one of them indicates something which contradicts these conditions, then the other must assume that the speech act was not what he originally believed. (For a fuller discussion of these issues see [DONA 86], chapter 3.)

2.3.3. Application

We can define the conditions which constitute the two speech acts of advice and warning. The following definitions are taken from [DONA 86], chapter 4.

Advice:

1. The propositional content of S's utterance is a future act A of H (viz. "H will do A").
2. S believes that A is in H's best interests.
3. The speaker's utterance is an undertaking to the effect that A is in H's best interests.

A Hypothetical Warning (for an utterance of the form "If X then Y"):

1. The propositional content of Y is some event, state, etc. E.
2. S believes that if X is true E will occur (has occurred).
3. S believes that E is not in the best interests of H.
4. S' utterance counts as an undertaking to the effect that E is not in H's best interests.
5. The propositional content of the imperative conclusion is a future action, viz., H's (not) doing Z.

2.3.4. Indirect Speech Acts

While the phenomenon of an indirect speech act is probably beyond the scope of this thesis, both from a theoretical and from a practical point of view, it nevertheless does represent a theoretically interesting phenomenon, and it does highlight some of the real difficulties that confront efforts to build automated dialogue systems. [ALLE 86] provides a good description of the phenomenon, and of the practical difficulties involved in handling it.

> In particular there are many cases when the system will not be able to compute speech act descriptions directly from the input. Consider the widespread use of indirect speech acts (Searle[75]), utterances where the speaker, if taken literally, says one thing yet actually means another. For example, "Do you know the time?" literally is a yes-no question ... but it is usually used as a request for the time. In some settings, where the speaker knows the time and the hearer does not, it can even be meant as an offer to tell the hearer the time! Thus instead of computing the speech act from the actual sentence, we will assume that the system will compute a surface speech act form encoding the literal meaning of the sentence out of context .... the complicated issue of computing the possible indirect speech act will then be left to the plan recognition model. (p. 943)

The practical difficulties involved in handling indirect speech acts (or semantic indirection) do seem to be considerable. At the most general level, we have to consider whether there is any way of predicting where and when a given

speech act cannot be taken literally. If such predictions cannot be made (and it is difficult to imagine any way that they could be made), then the system must maintain two levels of meaning. All speech acts must be handled by both levels, and the interpretive level must be able to apply a series of tests or conditions to each speech act, even if we can expect that in the majority of cases the literal interpretation will be the correct one.

The work of Searle and others provide us with some initial guidelines as to what those tests and conditions might be. To begin with, [SEAR 75] believes that communication by means of semantic indirection is made possible by the following conditions:

> (1) mutual awareness of the constitutive rules of illocutionary acts,
> (2) non-linguistic background information shared by both parties,
> (3) the hearer's powers of inference.

Certainly all of these conditions have been recognized here as necessary to carry on a dialogue, and we have discussed them previously as they have appeared in various forms. Given this background, a system needs specific rules which can test and interpret a given speech act. Such rules might be applied as follows.

A person seated at a dinner table says to another person at the table, "Can you pass the salt?". Literally this is a question seeking a "yes" or a "no" answer, and refers to the hearer's ability to "pass the salt". However, we would expect this utterance to be interpreted by the hearer as a request. This interpretation could be made by testing the sincerity of the question: is the hearer ignorant of the expected answer? Such a test could be one of the conditions for recognizing a speech act that is a simple question. In this case the speaker is not likely to be ignorant of the answer to a literal interpretation of this utterance as a question, so the literal interpretation can be signaled as faulty. Since this bottom-up analysis of the speech act signals a fault, a top down analysis could then be performed. Suppose that one of the conditions for fulfilling a request were that the hearer is able to do what the speaker requests. A top-down analysis would eventually arrive at a "request" as its current hypothesis and would find the utterance in question did fulfill one of the conditions of a normal request. The system could then return to a bottom-up analysis and test for the rest of the conditions that make up a request. This, of course, is a very simple example, but the conditions supplied by [SEAR 75], and the processing method applied here seem to offer a way of approaching the problem of handling semantic indirection.

What is really of concern to us, however, is just the presence of this phenomenon in language. How is that we can literally say one thing, yet mean another? Are indirect speech acts an aberration? Searle believes that semantic indirection involves at least two speech acts, Sp2 and Sp1. Sp2, the secondary speech act, is the literal interpretation; Sp1, the

primary speech act, is the actual interpretation. Sp1 is derivable from Sp2 by such means as we have described previously. On these terms, an indirect speech act is a special case, requiring special mechanisms to handle it. However, if we attempt to handle such phenomena in a computer system we have no way of knowing when these special cases are likely to occur. A computer system would have to subject all speech acts to at least the bottom-up tests as previously described. Such a system would also have to preserve the meaning of every speech act at the level of Sp2 before determining that it deserves to be accepted at the level of Sp1. As we indicated previously, a computer system must be able to recognize two levels of meaning, and must treat every speech act as if it could involve indirection. While we have no way to guarantee that the kind of consistency necessary for a computer system is also present in natural language, we should at least consider the possibility that natural language operates in the same manner. This would allow that every utterance has two levels of meaning: a literal meaning and an interpretive meaning. In most cases these levels coincide and we never notice that there are two levels of meaning. But if there are two levels, then it would seem reasonable that they do not always coincide, and that when this happens an indirect speech act occurs. Perhaps we simply like to demonstrate to ourselves, every once in a while, that language does have both a literal and an interpretive meaning. On these terms, an indirect speech act would not be an aberration, but only an exception, one which points directly at the way in which language conveys meaning. We will return to these considerations in chapter 4, when we consider in more detail the whole issue of representing meaning.

2.3.4.1. Application

Almost assuredly the system developed here will not achieve the sophistication necessary to handle indirect speech acts adequately. We have included the notion of semantic indirection in our discussion primarily because it indicates rather clearly that any system that processes natural language ought to be able to represent meaning at two distinct levels, and it is well that we are convinced of this before we begin to build a dialogue system. We could have approached this issue by examining other systems, but the idea involved is not so clearly present there. [CARB 83], for instance, presents these two levels as a need for modularity. That is, his system, XCALIBUR (a natural language interface to an expert system), has two levels of interpretation: a case frame tied to linguistic analysis, and a second case frame cast in the form of the data base query language of the application. He puts forward the claim that such a modularity will allow one natural language system to be used by differing applications. (This is a claim that is subject to debate, to say the least.) In contrast, the phenomenon of an indirect speech act is a clear and dramatic demonstration that we need two

levels of meaning and representation merely to handle natural language as a carrier of meaning. When we consider that pragmatic issues have such a direct role to play in deriving the meaning of an utterance, then we are less likely to jump to conclusions about the potential modularity of natural language systems.

3. A Dialogue System: UC

In order to demonstrate and test some of the ideas put forth in [WILE 83], a dialogue system called UC was constructed at the University of California at Berkeley by Wilensky and his colleagues ([WILE 84]). UC is intended to be an online help facility for the UNIX operating system. A relatively naive user can query UC for information about UNIX commands. The following are a few simple examples.

USER: How do I delete a file?
UC: Typing 'rm filename' will remove the file with the name filename from your directory.
USER: a directory?
UC: Typing 'rm -r directoryname' will recursively delete the directory with the name directoryname and all the files and directories contained in the directory.
USER: What do you know about UNIX? (or What can you tell me about UNIX?)
UC: I currently know about the UNIX file system.

UC provides the opportunity to test theories of common sense reasoning and natural language understanding in a dialogue setting focused upon a restricted domain. It is intended to be a complete system, so that natural language issues and reasoning issues are integrated in a relatively complex and interesting way. On the one hand, as a stand alone system, UC gives one the chance to see how well its various modules go together to build a working system. On the other hand, most of the modules that make up UC are also being developed to do work in the domain of everyday situations and common sense reasoning. Thus the work that surrounds UC is able to address many issues not likely to be generated by the domain of this particular dialogue system.

3.1. UC Modules

The following is a brief description of the modules that go together to make up UC, together with some of the issues it hopes to address.

PHRAN (PHRasal ANalyzer) reads English language sentences, matches them against patterns stored in a data base, and produces an internal, frame based representation of their meaning.

PHRED (PHRasal English Diction) transforms internal representations of meaning (usually in the form of plans) into English language sentences. PHRAN and PHRED share a database of facts about the English language.

A goal analyzer (apparently unnamed) interprets the goals of the user. One of the cornerstones of Wilensky's previous work is the recognition of an agent's goals in order to facilitate the task of understanding connected discourse. In this particular application one might expect the user to explicitly state his own goal, but it is still desirable to be able to

respond to indirect requests where information that is explicitly requested may require information not explicitly asked for in order to form a truly helpful reply. Along much the same lines, the goal analyzer is intended to do a kind of "request correction" as outlined in the following example and explanation.

> USER: How can I prevent someone from reading my files?
>
> the direct response to this question would be to use a protection command. However, an additional response in UNIX would be to use an encryption command, that is, a command which encodes the file so that one needs a key in order to decode it. The problem with offering the latter suggestion is that it does not literally fulfill the user's request. Encryption does not prevent someone from reading a file, but merely from understanding it when it is read. In order to decide to inform the user about encryption, then, the interface must assume that the user misstated his request. Presumably, the user is really interested in preventing others from learning the contents of his files. Having made this reinterpretation of the actual utterance, both methods of protection would be applicable. ([WILE 84], p. 577)

Another task of the goal analyzer is to keep track of all the goals of the user. Consider the following example:

> USER: I'm trying to get some more disk space.
> NI: Type 'rm *'.
> (Taken from [WILE 84]; "NI" stands for "naive interface".)

While the above reply does seem a little bit extreme, we know that this reply is unlikely to be helpful to the user because we know that the user has the unstated goal of preserving his files. The system should know these implicit goals too.

A context model, which is really an extension of PHRAN, keeps track of the subject of the discourse, and is intended to help resolve references and ambiguities. Currently, this component has to be told explicitly that the context has changed.

PANDORA (PLan ANalyzer with Dynamic Organization, Revision, and Application) is a highly experimental model that is designed to accept a goal as input and find a plan for achieving that goal. Where UC itself is concerned, PANDORA will probably never need to do more than a database lookup for a plan explicitly associated with a given goal. For a description of more complex cases, ones which require formulating plans, see section 3.3.

UC Teacher is the learning component of the system. One of the advantages of using a declarative representation of knowledge is that the system can be easily extended by simply adding facts to its database. No changes have to be made to the procedural part (or inference engine) of the system. The role of UC Teacher is to accept new information in the form of English sentences, and to encode that information in the form of patterns recorded in the system's database; these patterns can thereafter be accessed by PHRAN, PHRED, and the other components of UC. At the time [WILE 84] was written, UC Teacher was at an early stage in its development, able to handle only very simple constructions, and

unable to properly index the items it adds to the database.

Unlike the other components of the system, this component does not have any direct role to play in the dialogue between a user of UNIX and UC. Rather, it seems to play the role of a shell to be used by the system's developers in order to more easily extend the system. Thus, while it can make use of some of the linguistic knowledge contained in PHRAN, we would expect it to be more constrained in the range of English sentences it will accept; this is a constraint which we would expect a more expert user to easily accept. More serious demands are put upon UC Teacher by its interaction with the knowledge base itself. It would seem that UC Teacher must possess a meta-level knowledge about the knowledge base in order to properly index the items that it places there. Thus developing this component would possibly help the system's developers to gain a better understanding of the system's underlying organization, and would help the system itself to be more "intelligent" in the sense that it would encode knowledge about its own workings inside UC. If nothing else, UC Teacher is intended to demonstrate the extensibility of the system (which is the only real claim that the system's developers make for it.)

3.2. Problems and Current Research

The most significant and continuing problem in developing UC involves the representation of its domain and language knowledge in a formal language. This is really the central issue in the development of any such dialogue system, and the authors note that their system of representation "has continually changed as the system has matured." Presently "a rather stable body of concepts has emerged."

Practical problems with UC include the slowness of its response to a user query (about 15 seconds of real time on a VAX), and the fact that the system has not yet been tested with actual users. The developers feel that UC's knowledge is not yet extensive enough to be able to reply to a sufficient percentage of possible user queries. They are fairly confident that they can address these practical issues, and estimate that it will take from three to five years to produce a tested and usable system; another three to five years should see the emergence of commercial versions of such natural language dialogue systems.

Probably the most significant shortcoming of UC is that it is an entirely passive system. That is, UC does nothing to help direct the flow of the dialogue the way that a real human consultant would. Given the previously mentioned difficulties, it is probably too early to know whether this kind of full, active participation in a dialogue is feasible for an

automated system. The authors do, however, sketch plans for an ego component of UC, something which would allow UC to question the user for further information, and which would help control the flow of the dialogue in other useful ways. The following is a sample "envisioned dialogue" which they provide.

UC: What can I do for you?
USER: Hi, I have a problem. I can't print anything.
UC: Have you checked to see if the lineprinter is on line?
USER: Yes, everything seems alright (sic).
UC: Hold on for a minute. I am checking things. OK, the lineprinter should work now. Let me know if it doesn't work.
USER: What did you do?
UC: First, I checked to see if the lineprinter demon was working by typing 'ps -ag | fgrep lpd'. Since it wasn't working, I restarted it by first removing the lock file, /usr/spool/lpd/lock, and then running the demon, /usr/lib/lpd.
USER: Great, now I have something else that you might be able to help me with.
UC: Sure, what can I do for you?
USER: Well, I want to copy an entire tree of directories over to another machine using the ethernet. How can I do that without having to rcp every file over one after another?
UC: You can copy trees of directories by using the -r option of rcp. With rcp, you can type 'rcp -r dir1 dir2 ... destinationdir' in order to copy the directories, dir1, dir2, etc. into destinationdir.
USER: Thanks. I never noticed that rcp had such an option.
UC: You're welcome. Do you have any other questions?
USER: No, you've been a lot of help. Bye.
UC: Good-bye.

3.3. PANDORA

PANDORA (Plan ANalyzer with Dynamic Organization, Revision, and Application) is the highly experimental common sense reasoning and planning component of UC ([FALE 82]). The system designers intend that UC call on PANDORA whenever more complicated planning is required. PANDORA is also being developed to work in the domain of everyday situations. Its theoretical basis is founded on the ideas expressed in [WILE 83].

PANDORA's knowledge-base consists of frames that describe various situations. Associated with each frame is a set of goals that normally apply in that situation, and a set of inference rules that might be necessary. (These frames seem reminiscent of the scripts employed by SAM, with procedural attachment adding flexibility to deal with a wider range of events.) PANDORA is able to detect its own goals by reacting to events. Its planning process is described as hierarchical. This means that plans may have subplans, and these subplans must be accomplished before the parent plan can execute. In addition, plans are generated in a dynamic fashion. Thus the execution of one plan may cause a new goal to be generated, which will then be planned for. Controlling this process are a Plan Selector, a Plan Projector, and a Plan Executor. A potential plan is recorded in a separate database of future events (currently, only the top level plan is projected in this manner); PANDORA can then examine that future or hypothetical world to detect possible goals. The

system, however, does not really plan ahead; it can only deal with one goal conflict at a time (at the point where a plan about to be executed causes a conflict). The following example from the UNIX domain demonstrates this dynamic, step by step process of solving a problem. PANDORA has been told that her sister component, PAMELA (a story understander) has a new electronic mailing address. While attempting to write this information into a file, she is told that the file cannot be written since she is out of disk space. (While the following represents PANDORA's reasoning capabilities, the links to UNIX appear to be simulated by explicitly telling PANDORA the situation.)

```
;Edit the Addresses file.
(StartUp(Program Editor))
(MakeChanges (File Addresses)
                (Changes (AddAddress())))

;Attempt to write the file but fail.
(Type (Text (String (":w"
                        (FileName (File Addresses))
                        (Return)))))
(Verify Success))); -- Reads error message from editor.

;Plan to get rid of the problem.
;Try to delete unneeded files but fail to find enough.
(DeleteUnneededFiles (Actor Ego))

;Temporarily save file in temporary space.
(Type (Text (String (":w"
                        (FileName (File ("usr/tmp/pandora")))
                        (Return)))))
(Verify Success)))

;Save file more permanently.
(Mail (To Ego) (Content (File "/usr/tmp/pandora")))

;Ask system manager for more disk space.
(Mail (To System)
        (Content (Message
                (Request (MoreDiskSpace))))
(Quit (Program Editor))
```

Each goal that PANDORA recognizes has a normal plan associated with it; in the majority of cases it is enough to execute that plan. In the above case the goal is to obtain more disk space, and the normal plan is to delete unneeded files. When this plan fails, the Plan Selector looks for an alternative, which is to write the file to temporary space. Since all users have access to this temporary space, this solution is not permanent, so PANDORA mails the file to herself. Once these immediate concerns are taken care of, she addresses the real problem, and asks the system manager for more disk space.

PANDORA also employs meta-plans, probably the most interesting idea advanced in [WILE 83]. Meta-plans and meta-goals encode the reasoning process itself, but are processed like any domain level plan and goal. For example, if two goals are in conflict, then PANDORA generates the meta-goal, Resolve Goal Conflict; this meta-goal has as its normal meta-plan, Replan, which looks for a plan which can resolve the conflict. Thus the reasoning process itself can be encoded in the system declaratively and all knowledge resides at the same level in the system. There is however one difference between the treatment of meta-plans and domain plans. The system maintains a queue of plans to execute, and a normal (domain) plan is put at the end of the queue to wait its turn. Meta-plans however must be executed immediately since the knowledge they encode involves arriving at the next step in the reasoning process.

The use of meta-plans and meta-goals simplifies the system from a procedural point of view. Each domain goal has a normal plan associated with it. If that plan fails, a meta-goal comes into existence. Each meta-goal has a normal plan associated with it, one which involves finding an alternative to the failed plan. Thus the majority of the system's planning knowledge is stored in the knowledge base itself. This reflects Wilensky's overall belief that a natural language system should employ declarative knowledge wherever possible. This also makes a large knowledge base necessary for this kind of planning. However it is Faletti's claim that this planning knowledge is applicable to a wide range of plans and goals.

4. Representing Meaning

Previously we have been concerned with functions such as reasoning, generating inferences, and designing systems. Now we will look at the process of transforming English language sentences into more formal internal representations that can be accepted as input by the planning or reasoning components of a system. None of the systems we have discussed so far actually accept natural language as input; rather, separate components such as ELI and PHRAN produce a relatively formal representation that serves as input to PAM and PANDORA. In general, an automated system needs to reduce the variety of possible expressions in a natural language into some more constrained representation of what those expressions can mean to the system.

4.1. Semantics

In section 2.3 we examined the idea of describing language in terms of speech acts. The meaning of an utterance or series of utterances can be portrayed as a set of rules or conditions which govern a common understanding of the situation to which the utterance or series of utterances refers. These conditions are said to be constitutive of the speech act and its meaning. Whether or not a theory of speech acts is adequate to describe language and meaning, we nevertheless found that such a theory could be useful in constructing an automated dialogue system where we must have some way to formally represent meaning. Such a formal description of meaning as speech act theory proposes could be classified as pragmatic, since it refers not to the structure or features of an utterance, but rather to the world or the situation that an utterance describes. We can also say that the formal representations supplied by practical arguments and planning theory are pragmatic in nature.

Since none of these pragmatic rules do anything to describe an utterance or sentence as it is composed of linguistic elements, we must have some other set of rules that can govern the composition and recognition of a meaningful utterance. Generative grammar provides the most commonly used source of such a set of rules. Given a set of terminal elements and a set of rules for the placement of those elements, one can write a formal grammar composed of terminal elements, nonterminal classifications, and a set of rules or transformations. Any legal sentence in a language described by the grammar can be generated by the grammar or can be recognized by the grammar, depending upon which order is followed in applying the transformations. Such a system has proved to be extremely useful for the description of formal languages such as computer languages, where each terminal element has one and only one meaning or place in the

overall system or language. In such an application, every legal string or sentence will be recognized by the grammar, and every string or sentence not recognized by the grammar will be rejected by the grammar, and reported as an error.

When applied to natural language, a formal grammar is less effective as a description, but is still useful. We can construct a grammar that is able to accept legal sentences (we may not want to restrict this to grammatically well formed sentences), but the complexity of the task makes it less easy to construct the grammar in such a way that it will reject ill formed input. This is not a real problem since we can allow the pragmatic functions in the understanding process to simply ignore what has no meaning at this level. In general, we can say that the grammar will allow a great many sentences (many of them well formed grammatically) that do not have a meaning to the higher level functions. (For a grammar constructed along these lines, and a further discussion of the issues involved, see section 5.1.3.3.)

Another shortcoming of a formal grammar is of more concern to us. The complexity of natural language is such that a formal grammar is not likely to be able to handle anything but syntactic issues. Beyond this, the formality of the grammar is likely to break down into a host of semantic exceptions rather than rules. Beyond this, the distance between the issues of syntax and the issues of pragmatics is such that those who are concerned with building experimental natural language systems seldom mention grammars or syntactic issues at all. This is not to say that syntactic issues are not present, and even handled, after a fashion, by these systems; rather, the issues themselves are so far away from the focus of research (pragmatic issues) that they seldom make an appearance in any significant way in the course of describing this research. In systems that are avowedly domain dependent this is probably to be expected. The system being developed here, however, is intended to be as language dependent as possible. In order to accomplish this we need to construct a bridge between a formal syntactic grammar on one side, and those rules and knowledge that belong to pragmatics on the other side. Such a bridge seems necessary if only to keep both sets of issues in focus in the course of designing a well balanced system.

The bridge between syntax and pragmatics is, of course, semantics, which is traditionally considered to deal with issues of meaning in so far as those issues belong to the structure of language itself. That is, most studies of semantics begin where descriptions or studies of syntax are not adequate to describe language adequately. On those terms, semantics has always remained a somewhat murky issue. Perhaps, if we can look at issues of semantics from both a syntactic and a pragmatic viewpoint, we can improve this situation, however slightly. Our primary concern, however, is in constructing a practical system. From the point of view of that concern, we can see that we have two very good ways to

formally describe natural language (syntactic grammars, and domain dependent pragmatic rules), and we need to find an intermediate ground that can hold these two together, allowing us to take advantage of these formalities without allowing them to dissolve into a host of exceptions. The truly interesting thing about semantics is that when we approach it from the side of syntax, it looks like an extension of syntax, and when we approach it from the side of pragmatics, it looks like an extension of pragmatics, yet syntax and pragmatics look completely foreign to one another.

4.1.1. Case Frames

The most common and the most promising way to handle semantic issues is to represent the meaning of a sentence as a case frame analysis. A sentence is represented as a frame whose slots are semantic roles (cases) and whose values are the words and syntactic structures that fill those roles in that sentence. The following is a simple example.

Input: John gave Mary a book.
Frame:
    Sentence1:
        Actor: John
        Dative: Mary
        Object: a book
        Action: gave

In this example, the cases are actor, dative, and object. The verb, or action, is defined in such an analysis by the cases that it will accept. Different verbs may take different cases. The dative role, for instance, would not go with the verb 'to run'.

The original study of this system for representing semantic roles is [FILL 68]. That paper proposed the following set of cases (taken here from [WINO 83], p. 317).

Agentive	Animate instigator of action
Instrumental	Inanimate force or object involved
Dative	Animate being affected by action
Factitive	Object resulting from action
Locative	Location or orientation
Objective	Everything else

The following are a few examples to highlight this case set.

John opened the door. The door was opened by John.
    Agentive: John; Objective: the door

John opened the door with the key.
    Agentive: John; Objective: the door; Instrumental: the key

John used the key to open the door.
Agentive: John; Objective: the door; Instrumental: the key

The key opened the door.
Objective: the door; Instrumental: the key

Chicago is windy.
Locative: Chicago

We made him a jacket.
Agentive: we; Dative: him; Factitive: a jacket

4.1.2. Case Frames as an Extension of Syntax

One way to approach case frame analysis is to see it as an extension of syntactic analysis. (See [WINO 83] for a good brief presentation of this approach.) The surface structure of an English sentence depends primarily on the order of its constituent elements. The only way we know the role of John as agentive in the previous example ("John gave Mary the book") is by the position of the noun 'John' in the sentence. If we said "Mary gave John the book" then the roles of John and Mary would be reversed. Many languages, however, do not depend on word order to determine these roles. Instead, a very elaborate surface case structure exists in such languages as Latin and Russian. In languages which employ a case structure the ending of the noun indicates the role that it plays in a sentence. In Russian, for instance, "Professor uchenika tseloval" would be translated into English as "the professor kissed the student", while "Professora uchenik tseloval" is rendered in English as "the student kissed the professor". The case structure for a given language may be elaborate; Latin, for instance, defines case endings for nominative, accusative, dative, genitive, and ablative cases, and many languages are more elaborate in the number of cases they recognize.

It is tempting to portray case frame analysis as an extension of the very formal, rule defined case systems as they appear in such languages as Latin (and once appeared in English). But this is really not the derivation of case frame analysis as we would use it here. The surface case structures of the languages which employ them are not consistent across languages, and are not semantically consistent within individual languages. In the former difficulty, languages differ in the number of cases they distinguish. In the latter difficulty, entirely different semantic roles may be accepted by the same case. The kind of surface case structure present in Latin, Russian, and other languages is basically a syntactic phenomenon. It represents a way of connecting conceptual relations to surface structure, but it is not representative of any systematic effort to clearly distinguish different semantic roles from one another.

English has dropped most of the linguistic formalisms associated with such surface case systems. Its surface structure uses word order to recognize such syntactic categories as subject, object and indirect object. As a result, the mapping of semantic roles to syntactic structures is a very loose affair in English; that is, the syntactic role of a noun phrase in English tells us very little about its semantic role. The following are a few examples that [WINO 83] uses to highlight this phenomenon.

> We baked every Wednesday evening.
> The pecan pie baked to a golden brown.
> This oven bakes evenly.
>
> Cyril liked spaghetti.
> Cyril ate a meatball.
> Cyril got a bellyache.
>
> (p. 312)

In the first set of sentences, the subject of the verb 'to bake' plays in turn the semantic roles of actor, affected, and instrument. In the second set of sentences, the subject, Cyril, plays entirely different roles in each. First he is in a certain state, then he performs an action, then he undergoes something. In all six examples, merely recording the subject of the sentence does little or nothing to indicate the role that it actually plays in the structure and meaning of the sentence.

Since the surface structure of an English sentence is without any case system, some effort has been made to reintroduce the idea of case into English by tying it to notions of deep structure as developed in generative grammar. At the same time, this artificial rendering of case allows it to attempt to be much more semantically consistent in its organization than was ever the situation for a natural case system. Such efforts have not been entirely successful from a theoretical point of view, but in limited circumstances the idea involved has been employed in practical systems. The following sentences give a simple example of what is involved.

> The boy threw the dog a bone.
> A bone was thrown to the dog by the boy.
> The dog was thrown a bone by the boy.

All three sentences represent the same event and set of relations. According to deep structure theory we can represent the meaning portrayed in all three sentences by a single structure, one which can be derived by a set of rules or transformations from the different surface structures of the sentences. This conceptual deep structure should be expressed independently of subject, object, and indirect object, since the role of each varies over the different surface structures. A case frame analysis seems a good candidate for expressing this underlying concept. In all three sentences we can represent the boy as the agentive, the dog as the dative, and the bone as the objective (after Fillmore's system presented

above).

The connection between a deep structure syntactic analysis and a case frame representation of meaning allows us the opportunity to extend a syntactic grammar and parser to fill a structure that is semantic in nature. But the problems of doing so are many. First of all, the transformations depend entirely upon the verb; that is, the cases to be filled vary with the verb. While attempts have made to classify verbs according to the kinds of cases they will take (see [FILL 68]), truly general semantic principles are difficult to find. As Winograd notes, "... there seems to be a great deal of irreducible idiosyncrasy" here ([WINO 83], p. 316). A second problem is that a single verb may have many different meanings, with a different case structure for each of its potential meanings. Lastly, there has never been a successful attempt to define a complete set of cases for the English language:

> It should be remarked that in all of the formalisms and systems described here, there has never been a large-scale satisfactory coverage of English verbs. In systems that deal with a limited vocabulary and limited domain, case structure has proved useful, but there is no candidate for a comprehensive case grammar of English, and the attempt to integrate it into transformational grammars did not succeed well enough to be adopted in the mainstream of research. ([WINO 83], p. 324)

Actually, the situation is probably a little bit worse than Winograd describes it: if there ever were a satisfactory coverage of English verbs, we would probably have no way of recognizing it as such.

A syntactic approach to case frame analysis leaves us with a method to apply, but we have to look to a "limited vocabulary and a limited domain" if that method is to have any hope of being applied successfully to the English language. While this is certainly true, we would like to have a better understanding of the phenomena involved in applying these limits. This condition may not, after all, be a fault in formal descriptions of language; it may not even be a fault that can be traced to the limits of computers and the automated processing of natural language. The phenomenon of only being able to adequately describe a portion of the English language at any one time may well be a part of the nature of language itself. Natural language is an extremely difficult system to learn, and it takes all of us a great many years to even become mildly proficient at it. Furthermore, natural language is subject not to one formalism, but to a host of formalisms. We all speak differently than we write, and we all speak and write differently from one another, and speak and write differently depending upon the situation. Defining these situations seems to be the natural domain of a pragmatic study of language.

4.1.3. Case Frames as an Extension of Pragmatics

Besides the syntactic approach, [WINO 83] also recognizes that there is another approach to understanding case frame analysis, one that begins from a consideration of meaning.

> One way to look at these phenomena is to see the sentence structure as a window into an underlying scenario. The speaker has in mind some interrelated set of actions, perceptions, or properties and wants the hearer to know about them. Phrases that refer to participants in the underlying scenario must be arranged into clauses, each of which has three primary positions for noun phrases (subject, object, and indirect object) and optional places for any number of prepositional phrases. The grammar must provide a systematic way to carry out this mapping so the hearer will know which object plays which role and what kind of scenario is intended.
>
> It could be argued that this problem is not properly a part of syntax and instead should be viewed as semantic interpretation. Indeed, the whole issue of transitivity roles (systemic grammar's term for cases) lies in the fuzzy region between those things that can be dealt with using the mechanisms of syntax and those that cannot.... ([WINO 83], p. 313)

Winograd begins here with a description that closely resembles the way that speech act theory describes language: "as a window into an underlying scenario". But he quickly loses touch with this viewpoint when he proposes that a linguistic grammar handle the phenomenon involved. Lastly, he admits that on these terms the whole issue of case frame analysis is a murky one. Perhaps the issues become less murky if we can distribute the burden of representing meaning. The "underlying scenario" can best be represented by pragmatic considerations. The case frame analysis, then, only has to represent a consistent set of relations among the linguistic elements of the sentence. We can refer to this representation as the literal meaning of the sentence or utterance. A pragmatic analysis can then be performed on this literal representation, and we can refer to the result as the interpretive meaning of the utterance. We should also point out here that "meaning", at all levels, is just a representation of the relationships between elements. At any one level in the system, a set of relationships can be described as meaningful if they can be processed in a predictable fashion by the next level in the system. Eventually the top level is reached, and a set of consistent relationships is presented to the user as the system's output. The last interpretation belongs to the user, and it is only this interpretation that represents meaning to include a sense of "understanding". A natural language understanding system does not truly understand language; it only represents language in a fashion consistent enough to be understood. Any computer system does not understand its application any more than a chair, for instance, understands the concept of a chair. All that is required in each case is a remarkable consistency in the relationships that it exhibits.

We first encountered the idea that language has two levels of meaning when we examined the idea of an indirect speech act (in section 2.3.4). We speculated that language always has two levels of meaning, a literal meaning and an interpretive meaning, but that these levels only really distinguish themselves from one another through the phenomenon

of semantic indirection. Now that we have extended these earlier considerations to include semantics as well as pragmatics we should always be able to distinguish these two levels of meaning. This is true if we can use a case frame analysis to adequately represent literal meaning. An adequate representation can be achieved if we can derive a case frame analysis without reference to pragmatic considerations, and the resulting semantic representation can be interpreted successfully by purely pragmatic considerations.

We stopped in our earlier analysis of semantic indirection at the point where the example, "Can you pass the salt?", could be represented as two speech acts, Sp2 and Sp1. Sp2 represented the literal interpretation of this utterance as a question. Sp1 represented the recognition of this utterance as a request. Now we would like to be able to discard the notion of a secondary speech act (Sp2) and replace it with a purely semantic representation of meaning. The real strength of speech act theory is that it brings non-linguistic, pragmatic knowledge to bear upon the interpretation of language. On those terms, we would like to say that there is always only one speech act (Sp1) to be derived from an utterance, and that the derivation is always a function of pragmatic considerations.

The literal meaning of "can you pass the salt?" can be represented by the following case frame analysis.

action:	pass
agent:	you
object:	salt
mode:	can (ability)
sentence type:	yes-no question

This representation contains two slots not previously associated with a case frame analysis: mode and sentence type. These slots can be filled by performing a complete syntactic analysis of the sentence. Recognizing the presence and the position of the modal in this particular sentence allows the analysis to fill these slots with the correct values. (For a description of the kind of parsing rules that perform this analysis see chapter 5.) Including these slots and values seems to be a natural extension of a case frame analysis: we would expect that any practical natural language system would want to recognize and preserve these values. Further, a speech act analysis of meaning requires these values; at the same time they can be discovered by a purely syntactic analysis of the sentence.

The above analysis, then, represents an extension of a case frame representation to include all of the information necessary to represent the literal meaning of the sentence. It also fully represents the meaning of Sp2, a yes-no question referencing an agent's ability to 'pass the salt'. It does not, however, represent a true speech act, which must include a representation of the relations that hold between this utterance and the situation that it references. Those relations would

define the place of this particular arrangement of linguistic elements within the overall application or domain of the system. At this level the system has to determine what response truly satisfies these relations (that is, in this case, "the salt" rather than "an answer").

All of this may appear obvious and a little too simple. The example we have examined can easily be handled by our approach, and in the course of designing a system we fully intend to pursue the representation of meaning in this manner (See chapters 5 and 6). Practical considerations are not the real issue here, however. The real issue is the observation that language carries two levels of meaning. On these terms, semantic meaning does not directly reference the world. Rather, it only references another level of meaning, one which has rules of its own for how that reference to the world is accomplished. The reason why semantic issues become "murky" when we approach them from the side of syntax is that syntactic studies seek to describe language via a single formalism. Pragmatic issues, however, also determine meaning, and we know that from this side we must take a pluralist approach. It is not just computer programs that are limited to only being able to describe and formally represent a portion of the English language at any one time: this is the normal state of affairs. A case frame representation of semantic roles is not a failure if it does not achieve a "large-scale satisfactory coverage of English verbs". It is only a failure if it does not serve the particular pragmatic application for which it was designed.

4.1.4. Case Frames as an Interaction between Syntax and Pragmatics

Hopefully we have shown that it is possible to build a representation system that utilizes a syntactic grammar and parser to fill a semantic case frame which can be interpreted by pragmatic rules. However, we still haven't demonstrated why anyone would want to do so. The question here is whether or not the systems of expectation generated by syntax and pragmatics can be made to cooperate. [WINO 83] (p. 319) presents the following example of the difficulties that may be involved. Given the sentence, "The wind opened the door", should "the wind" be identified with the role of actor (as in "Joe opened the door") or should it be identified with the role of instrument (as in "the key opened the door")? If we design a case system to be an extension of syntax, then we should probably want "the wind" to fill the role of actor:

> From one viewpoint, case is an extension of the syntactic mechanisms of transformational grammar. Cases can be used in a systematic way to represent deep structure relationships, using transformation rules to produce the corresponding surface forms. Discussion about the particular cases and their application must be framed within the goal of designing a system of rules that accounts for the data with the greatest overall simplicity. Arguments based on this viewpoint hinge on whether a particular case assignment would lead to valid predictions about other sentences that might occur. Thus "the wind" might be assigned as the actor, since we say "the door was opened by the wind" instead of "the door was opened with the wind", and we would like to have a regular rule relating an actor to "by" and an instrument to "with".

However, if a case system is designed to fulfill pragmatic considerations then we would want the wind to fill the role of instrument:

> Another view of case is based on its ability to express meaningful relationships among the constituents representing participants in an event or a relation. Assigning a constituent of a sentence as the actor or beneficiary is not simply syntactic, but reflects a regularity in how the object described in that constituent is related to the event. There are regularities in the facts that can be inferred from the statement of an action, and these regularities can be organized around the case assignments.
>
> Thus, for example, acts are typically done for reasons and it can be inferred that the actor had a motivation for doing the act, while there is no need to assume that the beneficiary or object did ....
>
> Arguments based on this view hinge on the inferences that could be drawn from an assignment, rather than the precise syntactic forms that could or could not appear. Thus the wind (or more precisely, the act of the wind blowing) could be argued to be an instrument rather than an actor, since the system would have a regular rule connecting an actor but not an instrument to inferences about intentionality and motivation.

Efforts to design a case system that is both an extension of the transformations of generative grammar, and a representation of literal meaning to a pragmatic system, are likely, then, to encounter this kind of contradiction. If we can remove the contradiction, however, it would seem that a system designed to meet pragmatic considerations could still benefit from the power to predict linguistic patterns via a syntactic grammar. At least for the kind of contradiction described above, we can remove the contradiction involved, as follows.

A role (or slot) in a case system must be defined to accept some linguistic elements and to reject others. Linguistic elements, in turn, are typed by their placement in a semantic network defined along an "isa" or class dimension. Thus "the wind" in the above example could be defined as a physical force. A person could be defined as an animate being. Both a physical force and an animate being could be defined to belong to a common type. The actor slot in a case frame could be defined to accept anything belonging to that common type. Thus the system would be able to employ the syntactic-semantic prediction that a passive expression of "the wind opened the door" would employ the preposition "by", as in "the door was opened by the wind". The system would also have access to the pragmatic prediction that a person has a motivation for doing so, but the wind does not. A distinction between the two is recorded in the semantic network, and pragmatic rules could be designed to access that information.

4.2. Alternative Approaches

Most of the experimental natural language systems we have mentioned so far have been developed without incorporating the kind of formal syntactic grammar and parser that generative grammar employs. The work of researchers such as Schank and Wilensky has been oriented heavily towards pragmatic issues, and the parsers that they have

developed depend for the most part on dictionaries to translate linguistic patterns directly into conceptual entities. PHRAN (see [WILE 84] and section 3.1 of this paper), for instance, uses a database of linguistic patterns, and examines a sentence one word at a time, from left to right, building a list of possible patterns that could be filled by the next items in the sentence. The parse succeeds when a terminal pattern has been found. This terminal pattern has a concept associated with it, and that concept is then passed on to be processed by the higher level functions in the system.

Other parsers (ELI and CA for examples) translate English into Conceptual Dependency (CD) notation ([SCHA 75]). CD notation served as input to such story understanding systems as PAM and SAM, and was developed by Schank independently of work being done in generative grammar. This representation uses roles that bear a strong resemblance to those developed in syntactically oriented case grammars, but the orientation of these structures is decidedly pragmatic in nature. The representation employed in CD is intended to be independent of individual languages, and employs conceptual relationships between objects and acts rather than linguistic relationships between verbs and noun phrases and clauses. This representation language has the advantage of being canonical. This means that any two sentences with the same meaning will be represented by the same conceptual structure. This allows for transformations that are semantically based (as opposed to the syntactic transformations of generative grammar). For example, the sentences, "John kissed Mary" and "John gave Mary a kiss" would receive the same representation in CD. The verb "gave" here would be described as a dummy verb having no real meaning. While this kind of transformation does reduce the number and kind of relationships that have to be represented, it also represents a radical departure from the way in which meaning is traditionally represented in the English language (or any other language for that matter). Which is to say that a definition for the verb "to give" as it is used above can readily be found in any dictionary of the English language, and these two supposedly equivalent sentences do not have exactly the same meaning. Another feature of CD notation is that it has a very restricted set of actions that it can express. The verb "gave" in "John gave Mary a book" would be represented as an "atrans" which stands for a social transfer; any action which represents a transfer of ownership or of immediate possession would be represented with an "atrans" as its principle component. Other actions that can be represented are "mtrans" for mental transfer, "ptrans" for a physical transfer, and "do" for no specific action, to name a few. CD does employ nesting and a complex notational scheme that is capable of representing a wide range of English expressions.

While such parsers as PHRAN and ELI were designed to do the kind of work we intend to do here, we have chosen not to imitate the strategies that they employ. For the most part, they contradict the principles and the strategies that have been outlined for this thesis. Most importantly, they collapse semantic and pragmatic meaning into a single

representational scheme, Conceptual Dependency notation, which itself seems to be an effort to define meaning (in a way that only a computer would "understand"). From a practical point of view, these parsers put most of the burden of parsing onto a dictionary, one which must encode every word's place in the language that is Conceptual Dependency. As a result, the parsing strategy employed is a very simple one: each word is examined and replaced by its dictionary entry, as a sentence is examined from left to right.

4.3. Parsing

This system will employ a syntactic parser derived from [MARC 80]. Unlike the more traditional, hypothesis driven ATN style of parsing, Marcus's parser is a bottom-up, data driven syntactic parser that employs look ahead to more efficiently parse an English sentence. Marcus designed his parser to reflect the way in which human beings process a sentence. The goal of his design was to demonstrate what he called the "determinism hypothesis". This hypothesis proposed to prove that a set of syntactic rules could successfully process an English sentence without making mistakes; thus the key feature of his system is that it does not employ backtracking. Also, his major concerns were focused on theories relating to generative grammar, and not upon designing a practical computer system. We will not describe in detail here the functionality of this parser, since all we have really taken from it is the ability to examine more than one word at a time. The functionality adopted from Marcus' work consists of a buffer that holds the next three syntactic elements to be examined, and the use of pattern matching rules that access and manipulate that buffer. (Chapter 5 will present a more detailed description of the parser built here.)

4.4. Application

Chapter 5 will describe how well we have managed to implement the parsing and meaning representation schemes we have outlined above.

5. Implementation: Representing Sentences

This chapter will describe how individual sentences are processed. It will present a picture of how some of the ideas discussed in Chapter 4 have been implemented. The major components of this implementation are a syntactic parser and a case frame interpreter. Input is composed of English language sentences and output is a frame representation of each clause of the input. This frame representation contains all of the syntactic and semantic information necessary for the subsequent components of the system to correctly interpret each clause of the input.

Section 5.1 contains a description of the parsing process. Section 5.2 describes the process of semantic interpretation. Section 5.3 is a representative list of the kinds of sentences that this system can successfully process. Section 5.4 presents a detailed example of how one such sentence is represented within the system.

5.1. The Parsing Process

The parsing process accepts strings in the form of English sentences as input, and produces, as output, frame representations of those sentences. At this level of representation, a sentence frame contains an orderly representation of the elements of an individual sentence, one that is based on a predominantly syntactic analysis. The process of producing such a representation depends primarily upon a parser made up of pattern matching rules that employ look-ahead to efficiently parse the input string into a set of categories ("subject", "object", etc.) containing values. Each sentence (or rather, each clause of a sentence, since compound and complex sentences are handled) belongs to an overall frame based definition of those elements and characteristics of an English sentence that are recognized by this system. Thus, a single top-level "sentence" frame is defined and all the input sentences and clauses are parsed into representations that are defined as instances of this "sentence" class.

This description of the parsing process will begin with the preprocessing stage which describes the minor transformations that a sentence undergoes before it is parsed. This will be followed by a description of how the parser processes an English language sentence to produce a formal, frame based representation of its content. Lastly, the frame representation itself will be described.

5.1.1. Preprocessing

The input English sentence is first processed by a basic string handling routine taken from [CLOC 84]. The output of this routine is a list of atoms. For example:

```
Input:  A test has been scheduled for Tuesday.
Output: [a,test,has,been,scheduled,for,tuesday,.]
```

Each word or atom is then checked to make sure that it is known to the system. While this check is being performed, contractions are replaced in the list by their whole word equivalents.

At the next stage, a transition network handles the parsing of simple noun phrases. This network accepts noun phrases as input and produces a Prolog structure as output. The following are two examples:

```
the important tests:
        np(test,plural,def,[important])
the serious student's book:
        np(book,sing,def,[np(student,sing,def,[serious]])
```

These noun phrases are processed before the main parsing stage. This can be done because the structure of noun phrases does not depend upon their position in a sentence, and because the beginning of a noun phrase is easy to recognize. Also, a simple transition network is used since noun phrase structure is regular, and requires no look-ahead to be parsed efficiently. (This issue will be presented in more detail after the overall operation of the parser has been presented.)

After the noun phrases have been processed, a globally maintained buffer is created and filled with the contents of the current input. This buffer contains three cells holding the next three items from the sentence list currently being processed. An auxiliary list holds the remaining items. Separate routines maintain the buffer, and any cell can be accessed by any rule. When an item has been processed it is removed from the buffer, and the buffer is automatically refreshed. When an item is inserted into a cell, the buffer is automatically adjusted. The following is a sample state of the buffer at the beginning of a parse:

```
Input:  A test has been scheduled for Monday.
Buffer: 1:{np(test,sing,indef,[])}  2:{has}  3:{been}
                remaining: ([scheduled,for,monday,.])
```

An alternative to the preprocessing of noun phrases is described in [MARC 80]. This alternative employs a set of pattern matching rules that are always active and are applied whenever a word that signals the start of a noun phrase shows up in any of the buffer's three cells. Having implemented such an approach in an earlier effort to build a smaller

and simpler look-ahead parser, I can only describe it as a somewhat cumbersome and wasteful attempt to build a one pass parser. The processing of noun phrases derives no benefit and contributes no understanding to the uses of look-ahead in natural language parsing. Any effort to build noun phrases in the midst of applying look-ahead pattern matching rules only leads to a needless complication of the overall parsing mechanism. Marcus found it necessary to add two additional cells to the buffer, and to introduce the notion of a virtual buffer in the sense that any one of the first three cells could be marked (via an offset value) as the first buffer cell. The end result of these complications was logically identical to his earlier use of the method employed here, since the parser still operates independently of all other considerations when it finds and builds a noun phrase node. Although the arrangement employed here (and apparently used in the earliest versions of the Marcus parser (see [WINS 84])) requires the parser to make two passes through each sentence, it is the simplest and most efficient way to accomplish the task. One of the benefits of a look-ahead parser is what Marcus recognized as its similarity to the way in which a human being reads a sentence. This allows us to implement rules which we already know work well, and to explore paths which we expect will not lead to impasses and dead ends. With regard to parsing noun phrases, however, we can profitably depart from the human model without losing any of the logical properties of the method.

5.1.2. A Syntactic Dictionary

A dictionary provides categorical definitions for the words that the system knows. For the most part, this information is syntactic in nature, recognizing the traditional categories for words, such as noun, adjective, verb, and the like. However, some of this information is semantic: "time", for instance is recognized as a category. A third kind of information is recorded: this involves representing syntactic categories in more specific subclasses. For example, "if" is not represented as a conjunction, but is instead represented as a "condition", meaning that the parser will mark any clause in which "if" occurs as a condition, a kind of clause that has a special significance in this application. All of these different kinds of information have one common theme: information is represented in such a way as to allow the parsing routines to have access to sufficient information to properly interpret the lexical items that are encountered in a sentence. Some example entries are:

```
noun_(student,students).
adj_(important,status).
time_(monday).
verb_(study,studies,studied,studied,studying)
```

Verb entries are augmented by intermediary rules which return the infinitive form of any verb form and which return categorical information about the verb form. Thus if a parsing rule queries the dictionary about the word "studying" it learns that this is the present participle form of the verb "study". Another set of intermediary routines that augment the parser allow higher level queries to be written to seek only the most general information necessary to recognize a pattern. The following call, for instance, would only want to know if the current item is an infinitive form of a verb, but would not care what particular verb was involved:

match(1,verb,_,_,[infinitive|_])

These higher level intermediary routines also exist at the parsing level for other lexical items that the parser is likely to encounter.

Throughout the system this principle is maintained: all of the specific information that the system needs is available to it, but at any particular step in the overall process of representation and interpretation, information that is more specific than necessary is never asked for. This principle is fundamental to the overall enterprise of processing informal natural language by means of formal representations in a computer system. Also, this principle is pervasive and natural to this enterprise, so that it was in force before it was recognized, and seems so self evident that it should hardly be mentioned. Still, it does seem to say a lot about the way in which human beings use language some of the time, and it does seem to define those uses of language (which are certainly not all or even most uses of language) which a computer might be expected to process in an intelligent manner.

Currently the dictionary also records another kind of semantic information that might instead be placed in the frame-based semantic network. (See section 5.2.1 for a description of the frame-based semantic dictionary used by the system.) Adjectives are represented by the following type of entry:

adj_(important,status)

The first argument represents the specific word or atom, while the second argument represents the class to which that word belongs. When semantic analysis is finished the class will become a slot name holding the value of the atom or word. This cheaper form of representation is only done for adjectives in the current system. While it would be more consistent to represent this information directly in the frame network, this has not been done for several reasons. First of all, class inclusion is always represented at the frame level and has the possibility to include inheritance. Adjectives are not naturally expressed as frames, but rather as slots attached to a noun frame. While the current frame system,

described in [Hiss 87], has the capability to define slots separately, it does so on the assumption that an independently defined slot belongs to every frame in the system. This however is an implicit feature, which is only implemented at the point of actually calling a particular frame, so that the system is free to proceed as if the slot belonged only to specifically invoked frames. When all is said and done this global representation of an adjective as a slot available to every frame in the system does not represent the truth of the situation, so well as does the current representation (inconsistent as it may be with the rest of the syntactic dictionary). The second reason for this kind of representation is that frames are expensive, having to be asserted into the database in a system that allows for a great deal of information and constraints to be stored in a frame. Adjectives do not play any real role in the higher level processes in this system as it now stands, so it is probably better to assign them this cheaper form of representation for the time being. Their presence in a much simpler form of network serves as a reminder that in a larger, "real world" situation we might employ this cheaper form of representation for nouns as well, only making them into frames when some instance of a noun occurs at runtime.

5.1.3. The Parsing Rules

The major component is a parser that is composed of independent, expert-like rules. These parsing rules process the incoming sentence to produce a frame representation of that sentence. They employ pattern matching and transform elements in the buffer into values in the slots of the current sentence frame. They use look-ahead, and patterns may match one, two, or three cells in the buffer. The look-ahead nature of the rules allows that once a value has been added to a slot it will not be removed or altered during the course of the parse. This last advantage could, however, be achieved in other ways. A simple recursive network could achieve the same effect by waiting until the returns from calls to place elements in the frame. This kind of program is however more difficult to write and to understand after it has been written. (Such a network is currently being used to build system-generated sentences and their accompanying frames.)

Another advantage of using look-ahead is that knowledge about sentence structure is captured in meaningful units. Whatever is necessary to arrive at a determination about a grammatical sentence feature is together in one rule. What is interesting about [MARC 80] is the demonstration that this could be achieved using no more than three buffer cells, so that the use of look-ahead is not only conceptually pleasing but is also computationally effective and cheap.

5.1.3.1. Rule Content

Each rule is composed of two parts: the first part specifies the pattern to be matched, and the second part specifies the actions to be taken if the match is found. The conditions and actions which may appear in a rule are generally limited to the following:

Conditions
- Match a buffer cell item
- Match a slot value in the current frame

Actions
- Add a value to a slot in the active sentence frame
- Manipulate the buffer by:
  - removing a buffer cell item
  - inserting an item into a buffer cell
  - switching one buffer cell item with another
- Activate another rule (limited use)

The second action type (manipulating the buffer) allows the parser to encounter fewer patterns in the course of a parse, and so cuts down on the number of rules that are necessary. If a pattern conveys information, then we are free to alter that pattern once the information that it conveys has been recorded. An example would be the phenomenon of aux inversion, where an auxiliary verb precedes the subject of a sentence. This signals that the sentence is a question seeking a yes or no answer. Once we have recorded that fact in the sentence frame, then we can invert the order of the auxiliary verb and its subject. Since this new pattern conforms to the usual pattern of a simple declarative sentence, one subject rule will be able to handle both types of sentences at the next stage of the parse.

The following are three sample rules:

First example:

```
sent_type(Curr_sent) :-
        match(1,verb,_,_,[infinitive|_]),
        add_slot_val(Curr_sent,sent_type,imper),
        np(NP,[you],_),
        insert_buff(1,NP).
```

An English translation might read:

If the lexical item in the first buffer cell is an infinitive form of a verb, then record the fact that the sentence is imperative, and insert "you" into the first buffer cell.

Example sentence:

Input: Study for the test.

Prior buffer state:

1:{study} 2:{for} 3:{np(test,sing,def,[])}

remaining: ([.])

Resulting buffer state:

1:{np(you,sing_or_pl,2,[])} 2:{study} 3:{for}

remaining: ([np(test,sing,def,[]),.])

Second example:

```
sent_type(Curr_sent) :-
        buff(1,there),
        match(2,be_verb,_,_),
        match(3,np,_,_),
        remove_buff_item(1),
        switch(1,2).
```

An English translation might read:

If the lexical item in the first buffer cell is "there", and the item in the second buffer cell is a form of the verb "to be", and the item in the third buffer cell is a noun phrase, then remove the item in the first buffer cell, and exchange the contents of the first buffer cell with those of the second. (Nothing is done to record the sentence type, since the default value, a declarative, sentence, defined in the frame network, is used here.)

Example sentence:

Input: There is a test scheduled for Monday.

Prior buffer state:

1:{there} 2:{is} 3:{np(test,sing,indef,[])}

remaining: ([scheduled,for,monday,.])

Resulting buffer state:

1:{np(test,sing,indef,[])} 2:{is} 3:{scheduled}

remaining: ([for,monday,.])

Third example:

```
check_act_pass(Curr_sent) :-
        match(1,be_verb,_,_),
        match(2,verb,_,_,[pastpartl_]),
        add_slot_val(Curr_sent,act_pass,passive),
        remove_buff_item(1).
```

In English:

If the item in the first buffer cell is a form of the verb "to be" and the item in the second buffer cell is the past participle form of a verb, then record the fact that the sentence is passive, and remove the item in the first buffer cell.

Example sentence:

Input: A test has been scheduled for Monday.

Prior buffer state:

1:{been} 2:{scheduled} 3:{for}

remaining: ([monday,.])

Resulting buffer state:

1:{scheduled} 2:{for} 3:{monday}
remaining: ([.])

5.1.3.2. Control Structure

The parser is designed to make control simple and intuitive. The rules are divided into sets, and these sets are arranged to conform to the normal grammatical order of an English sentence. In most cases, the control of execution is simply to call these sets in order until the sentence is parsed or the parse fails. Only one rule in a given set will succeed, and each set contains all of the parser's knowledge about its particular place in the overall scheme. This limits the amount of knowledge that each rule must possess, and locates control in an overall parsing scheme, one defined in a separate set of rules or metarules. This arrangement does seem to conform to our basic intuition about the regularity of an English sentence at this level of abstraction. There is however a procedural disadvantage to this scheme: a parse may be finished successfully at many places in the parsing process, and so there is a good deal of redundant checking for the end-of-parse conditions.

Diagram 1 represents the control structure of the parser. The presence of a dependent clause or phrase as either the subject or object of a sentence will cause a temporary departure from this sequence. Such departures will generally employ part of the structure presented here, but will also employ rules not represented here. Once the dependent clause has been processed, then control will return to the structure as it is presented in Diagram 1.

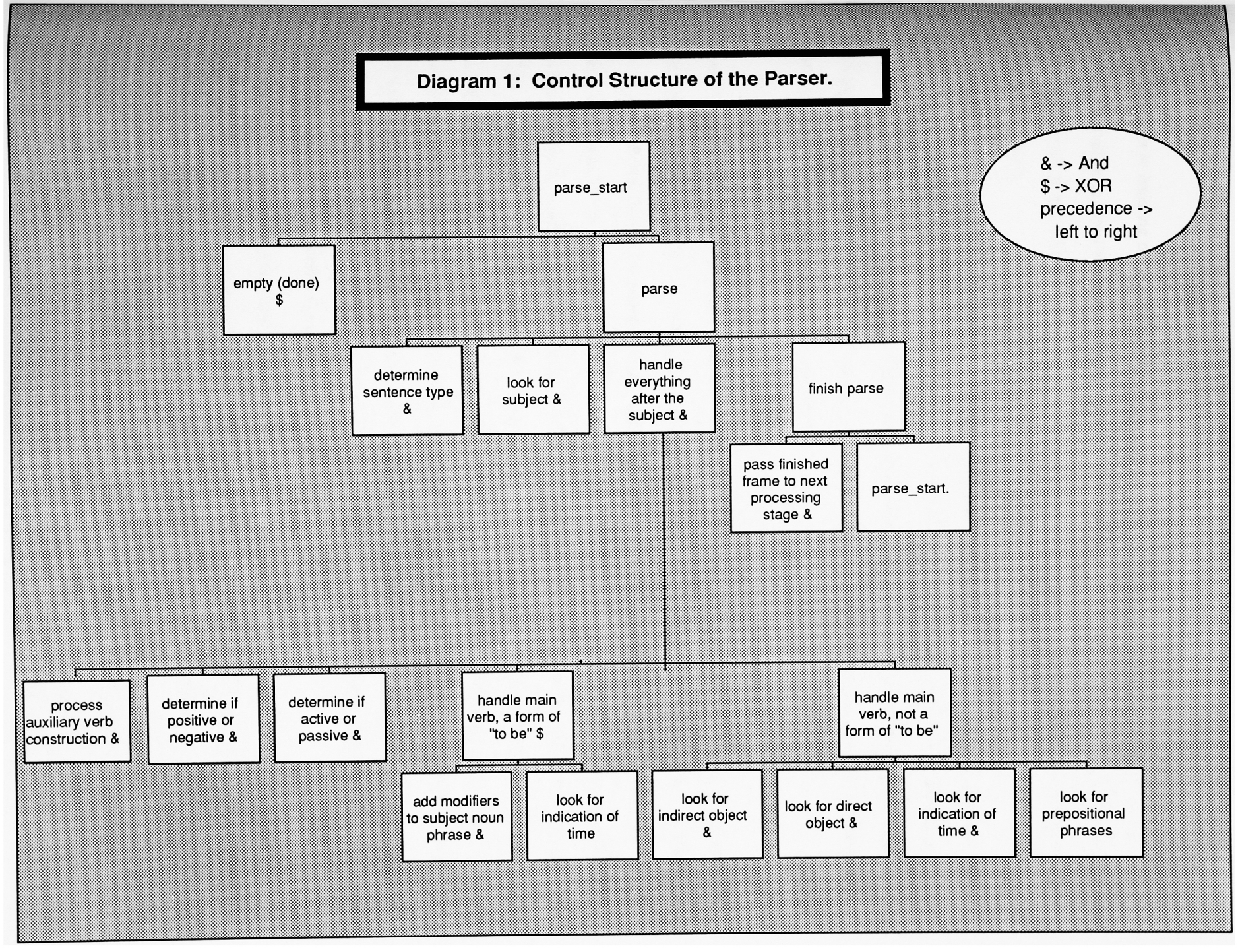
For the most part, these dependent clauses or phrases are verb phrases that play the role of subject or object in the main clause or in another dependent clause. Grammar books usually describe such phrases as part noun and part verb. For example:

I advise you to study for the test.

This sentence contains a verb phrase as its direct object. Such a phrase must be treated much as we treat the main clause of the sentence: it must be parsed into its own frame representation, only in its case, this frame will itself go into the object slot of the main clause. Thus the parser acts upon the infinitive "to study" as a verb, and the only recognition of it as a noun occurs in the placement of the dependent clause frame in the object slot of the major clause frame.

The rules which handle such constructions have been built to handle those cases which are evident from the sample dialogue presented in chapter 1, and the sample sentences listed in section 5.3. These rules are designed to perform three

Diagram 1: Control Structure of the Parser.



major functions in handling these verb phrases.

First of all, a top level rule acts as an independent control structure, calling those normal parsing rules necessary to handle any object or prepositional phrase that is attached to the infinitive (or gerund: "Finishing the lab will take all of the available time").

Secondly, other rules need to consider where to place the verb phrase in the major clause frame. In most cases the normal parsing rule that first recognizes the verb phrase could safely place the finished frame in the same slot as it would place a normal noun frame. However, this is not always the case. The use of "it" as a dummy subject can displace a verb phrase subject so that it appears at a place in the sentence where the general parsing rules expect to find an object. The following serves as an example of this phenomenon:

It will take all of the available time to finish the lab.

Here, "to finish the lab" is actually the subject of the sentence, but the parser employed here waits until it is parsing the object of a sentence to handle this phrase. A special rule recognizes that the object slot of this sentence is already filled by the noun phrase, "all of the available time", and that the subject of the sentence is currently recorded as "it". This rule then discards "it" and puts "to finish the lab" into the subject slot. If the overall control structure described here is not to be subject to major revision, then the only other alternative is to test the pronoun "it" every time it appears as the subject of a sentence: we would have to determine whether or not "it" is a true subject or a dummy subject (as it is in the above example). Such a test would necessitate searching the current sentence for a candidate to replace "it" as the true subject. This would be a complicated and awkward enterprise; instead, the parser simply does nothing at this point, and waits until the normal course of the parse produces a candidate that is the true subject of the sentence. Attempts to place this candidate phrase in its proper slot in the sentence frame produces the recognition that "it" is a dummy subject that simply served as a temporary place holder until the real subject appeared.

We cannot claim that this method will handle the general case, since not enough exploration of the possibilities has gone into the production of this parser. We can however believe that the method employed is sound: first, the method employed here seems to reflect the way in which a human being would read the sentence; secondly, the strategy of "waiting" when look-ahead is not sufficient is the perfect complement to the use of look-ahead in a parser. Indeed, "waiting" is the most important principle in the parser's design (is the 'nothing' which makes the 'something' of look ahead possible and effective). "Waiting" is the real control structure in this or any parser: it becomes more visible to us

in this special case, where what we are waiting for is somewhat unusual and requires an unusual action. (It should be noted, that this is the one case where the parser does remove a slot value from a frame, and so goes against its stated ideal of never having to retract a parsed structure.)

(Look-ahead parsers are often referred to as "wait and see parsers", but this name refers to the look-ahead nature of the rules, and not to the control structure employed. That is, the rules 'wait' by examining more than one lexical item before making a parsing decision.)

The third and last action that these special rules are concerned with is finding the subject of the infinitive phrase. In the first example that we used ("I advise you to study for the test.") "you" is the obvious candidate to fill the role of the subject of "to study". The only doubt that arises is whether or not we should say instead that "you" is the indirect object of "advise". If we change this example slightly and say, "I advised him to study for the test", then it would seem that "him" is the indirect object of "advise" since the pronoun is in the objective case. Traditional English grammar books, however, answer this question differently: "him" is the subject of "to study". To account for the apparent anomaly, a rule is invoked: the subject of an infinitive is always in the objective case! (By the way, the subject of a gerund is usually in the possessive case: "My finishing the lab will take all of the available time.") (See [KRAM 53].) Such a rule seems to indicate that grammarians prefer that a word only play one role in a sentence, and that subjects are commonly believed to be more important than are indirect objects.

This parser treats "you" in the example sentence as both the indirect object of "advise" and as the subject of "to study". This is the most natural solution to the problem and has been employed elsewhere (see [WINO 83]). Once we allow that a word can play more than one role in a sentence, then we really have no problem here. Giving "you" two roles to play also facilitates the parsing process, since the parser can treat "you" as the indirect object in its normal fashion upon the appearance of a pronoun that looks exactly like an indirect object, and then can wait until it is parsing the infinitive phrase to consider "you" as its subject. Again, as discussed above, special rules only have to be created at the point where they are needed, after the fact. The parser simply copies the indirect object of "advise" into the subject of "to study". All of this appears very simple and hardly worth discussing: it represents, however, only the most obvious appearance of a more general and more interesting issue.

The issue involved begins to become more interesting when we consider the passive version of the example sentence: "He has been advised to study for a test." The parser employed here treats "he" as the subject of both "advised"

and "to study", and this appears to be consistent with what has gone before. We should notice, however, that this treatment disagrees with the grammarian's rule: "he" is not in the objective case. More modern and more thorough studies of English grammar, do however believe that "he" is the subject of both verbs. Marcus' parser [MARC 80] (designed to demonstrate the theories of Chomsky) used the idea of a trace linked to the subject of the main clause here, and would insert this trace into the parse, in such cases, as the subject of the infinitive. The trace represents the identical value of "he" without denoting its physical presence in the sentence. (This parser is not given to such fine distinctions, and simply uses the same instance of the pronoun to fill both slots. If such subtleties prove necessary or worthwhile they can be added at a later time.)

Of course, it is easy to believe that the more modern view is the correct one, but that is really to miss the point of what is going on here. The more traditional grammarian felt confident in ignoring this second example case (remember, the subject of an infinitive is always in the objective case), because it did not appear within the scope or range of his vision of syntactic issues; perhaps, when it came time to diagram such a sentence in a left to right analysis, "he" had already been safely tucked away in the subject of the main clause, and so was out of view, in the sense that there was no ambiguity in where it should be placed. Of course, his semantic sense of the sentence would recognize that "he" who had been advised is also "he" who should study.

This new ambiguity between syntactic and semantic considerations is very well recognized by Marcus' approach since the idea of a trace is employed in a purely syntactic parser, but does not seem to stand for a syntactic element. Actually, it is difficult to say just exactly what that "trace" does stand for, even though it does seem to be just exactly the right representation for this situation. This is the really interesting issue: it does seem to be true that at times we cannot tell whether the issue is semantic or syntactic, and the best we can do from a theoretical point of view is to express that ambiguity as it is: as unresolvable. This particular instance or example is slight, but the way in which its "resolution" expands is what holds considerable consequences for the design of this or any parser. Here we are not concerned so much with theoretical issues of grammar, but rather with practical concerns about the use of syntax as a parsing mechanism.

There are further examples which seem to move beyond the scope of even Marcus' method. Consider the following two examples:

> To finish (Finishing) the lab will take me all of the available time.
> To finish (Finishing) the lab will take all of my time.

This parser makes no effort to find the subject of "To finish" in either case. The only 'principle' involved is that the subject is not readily available at the time of parsing the infinitive phrase. In the second example, the potential subject has found a particularly diabolical hiding place inside the descriptors (adjectives) list attached to a noun phrase that is the object of the main verb clause. The overall system handles these cases in the same way that it would if the subject of the infinitive were truly missing or "understood" in the sentence: it uses semantic inheritance defined in the verb frame definitions to determine that "some_person" is "to finish". (See section 5.2.1.1.) We cannot point to any principle that says that the issue here is now clearly semantic rather than syntactic; we can only say that it is convenient (perhaps "convenient" to the point of becoming reasonable in anyone's view) to do so. From a practical point of view, we can say that the issue is no longer ambiguous, but is now clearly semantic, and will be handled by semantic mechanisms.

In general, we use syntax as a parsing aid not because it offers us a general, exhaustive representation of English, but because it is very abstract and requires a minimum of knowledge in order to translate an English sentence into a formal representation of the sentence. Where rules threaten to proliferate to handle difficult and odd cases, it is not a bad strategy to try and slide over those cases where we can, and to use the semantic component to handle such cases. If the subject is missing, we can use semantic inheritance to fill it in; we are not stretching things too much if we say that being hard to find is the same as "missing". Although theories of syntax try to arrive at a general representation of all of language, it seems likely that issues will always be left over, and will then be considered as semantics, a "murky" area, at best, to most grammarians.

We could handle semantic issues in the same way that we handle syntactic ones: by writing rules that extend to semantic matters. But we can see above that we have to be able to make a distinction between what has enough regularity to be handled by a rule-based parser, and what is difficult to handle that way, and which leads to numerous exceptions, and special rules. The distinction between syntactic and semantic holds up well here, but it is not our basic concern. Our basic concern is to discover patterns of regularity that can be handled in a specific way by a pattern matching mechanism, and also to see where that approach is not efficient or productive. Where it is not appropriate, a frame based definition of general meaning takes over: specific variations are ignored, and general categories of meaning are employed. Only by attempting to use this method in the higher level reasoning functions of the system, will we be able to determine if this method is sufficient to the task.

5.1.3.3. A Grammar

The following is a grammar defining those sentences which the parser will accept. Nonterminals begin in upper case, and terminals begin in lower case. For the sake of brevity, lexical categories have been represented as terminals. In actual practice these categories would still have to be reduced to specific lexical items in the dictionary. Items enclosed in double quotes are specific lexical items. "$" represents an OR connection. Some minor patterns, those handled by transposing sentence elements in the buffer, are not represented here (for instance, "there" as a dummy subject).

Sentence	-->	conjunction Sentence2 $ Sentence2
Sentence2	-->	Pre-subject Subject Verb-part End-sentence
Pre-subject	-->	interrogative $ Aux-construct $ empty
Subject	-->	Noun-phrase
Subject	-->	Infin-phrase
Subject	-->	Gerund-phrase
Subject	-->	empty
Infin-phrase	-->	Subject2 "to" verb(infinitive) After-verb
Gerund-phrase	-->	verb(gerund) After-verb
Subject2	-->	Noun-phrase $ empty
Verb-part	-->	Aux-construct Verb-construct After-verb
Aux-construct	-->	Modal Aux Pos-Neg
Modal	-->	modal $ empty
Aux	-->	auxverb Aux $ empty
Pos-Neg	-->	"not" $ empty
Verb-construct	-->	be-verb verb(past or present participle)
Verb-construct	-->	be-verb "being" verb(past participle)
Verb-construct	-->	be-verb
Verb-construct	-->	verb
Verb-construct	-->	empty
After-verb	-->	Ind-obj Object Prep-phrase Time-construct
Ind-obj	-->	Noun-phrase $ empty
Object	-->	Noun-phrase
Object	-->	Infin-phrase
Object	-->	That-clause
Object	-->	empty
That-clause	-->	"that" Sentence2
Prep-phrase	-->	preposition Noun-phrase $ empty
Time-construct	-->	preposition time-word $ time-word $ empty
Noun-phrase	-->	proper-noun
Noun-phrase	-->	pronoun
Noun-phrase	-->	pronoun "else"
Noun-phrase	-->	noun
Noun-phrase	-->	determiner Noun-phrase2
Noun-phrase2	-->	Descriptors noun
Descriptors	-->	adjective Descriptors
Descriptors	-->	Possessive Noun-phrase2
Possessive	-->	Noun-phrase
End-sentence	-->	"." $ "?" $ "!" $ empty

The above grammar is presented as an attempt to show the kinds of constructions that the parser has been built to recognize. A careful examination of this grammar or of the rules that it represents will show that a great many word patterns that are acceptable to the parser will have no meaning to the rest of the system, and in many cases will not represent a meaningful English sentence. The parser has been constructed to accept meaningful patterns but it has not been written to reject meaningless ones. In this respect the principles that guide its operation are considerably different from those that guide the construction of a parser designed to handle a formal language. Since each token present in a formal language has one and only one meaning, it is only necessary to parse the statements of such a language in order to represent their meaning accurately. At the same time, statements that do not abide by the formalism of such a language are automatically rejected by this process. With natural language, however, parsing is only the first step in the process of representing meaning, and an entirely different principle guides its construction.

This parser is designed to accept an input sentence, and to instantiate a frame with the values that it can recognize in that sentence. As long as the input stream matches a pattern at each successive stage of the parse, the parser will continue to advance through its rule sets until it reaches the end of the input stream and the end of its rules. If the parser fails to match any pattern at a particular stage, then the parse will halt, the frame will contain those values already placed there, and the parser will return to its original state, ready to accept another sentence. On these terms then the parser will accept anything, including an empty sentence. It is not the function of the parser to distinguish meaningful input from meaningless input; rather, it simply recognizes what it can recognize and ignores what it cannot recognize. This is the only viable approach to parsing natural language, allowing a finite machine to attempt to cope with an infinite set of possibilities. It copes by doing nothing with the vast majority of possibilities, and by accepting many constructions which are not meaningful. It is the rest of the system that will be concerned with meaning: for the parser it is only necessary that it be ready to collect constructions which are potentially meaningful to the rest of the system.

As a sidelight to this, we might notice that the parser is designed to be a good deal more general than is the rest of the system: it has been designed to parse many sentences that will have no meaning to the rest of the system. These fully processed frames will be ignored by the higher system processes just as if they were only partially processed. Within the system itself the only way to distinguish among results of the parsing process is by their effects upon the rest of the system. The more general nature of the parser is only meaningful when we consider that the same parser could be used by other systems.

5.1.4. Sentence Frames

The parser described here, as well as the case frame interpreter described in Section 5.2, and the context interpreter described in Chapter 7, all make use of a frame-based system for organizing and representing knowledge. This frame-based system is provided by [HISS 87].

A frame is a named collection of slots, where each slot may have a number of features that are defined and maintained by the frame package. Slot features which are used in this system are those which allow a slot to have a value; to have a type which can constrain what values that slot may have; and to have procedures (called demons) for finding values, and for reacting to the addition or the removal of a value. A special slot called "is_a" or "in_of" allows frames to be connected together in a network that allows inheritance of slots and of their features.

In the system described here, system-defined frames represent the kind of knowledge that can be recognized; input is organized into dynamically created frames which are instances of these frames, inheriting their structure and sometimes inheriting values from these system-defined frames. Not all frames make use of all slot features; how these slots are used will become apparent as we describe the various knowledge representations that are used in this system. Frame representations will only show those slot features which are used, and where the only feature used is the "value" feature, the word "value" will not be shown in the representation.

Each input sentence is preserved and represented in a sentence frame; each such sentence frame is an instance of a system-defined sentence frame. This frame based definition is extremely loose, and does not use typing constraints to define its contents. Instead, it depends upon the parsing rules to apply the actual constraints upon what values may go into what slots in the frame representation. This combination of procedural and declarative representations does, however, represent the overall strategy employed in this system, wherein declarative frame-based representations of knowledge apply a system of general constraints which are then referenced and refined locally by procedure-oriented rule sets. This kind of organization provides maximum flexibility for improving the system at a later time: once enough familiarity is gained with how local patterns apply within an overall pattern, then rules might be replaced with declarative knowledge, and more of the work of the parser might be located in the frame system of representation. At the next stage of representation, when a semantic representation of a sentence is produced, then the frame definitions play a greater role in the processing.

5.1.4.1. Inheritance

Every sentence frame inherits default values from a system defined parent sentence frame. This parent sentence frame defines all the slots that a sentence frame is likely to have (the only missing slot in this definition is the "is_a" slot which lists the frames from which the sentence frame may inherit values). As a part of that definition this frame provides default values for those slots. For instance, a frame that does not possess an object value will inherit the value "nil" from the parent sentence frame. The following represents all the slots and the default values defined in the top level sentence frame.

frame: sentence
subject: nil
verb: nil
object: nil
indirect object: nil
time: nil
by: nil
for: nil
to: nil
interrogative: nil
modal: nil
positive or negative: positive
active or passive: active
sentence type: simple declarative

There are two different types of default values listed here. The first type includes all the linguistic elements recognized in a sentence by this parser. For all these slots the default value is "nil". The second type of slot defines sentence features or classifications that are applied to every sentence. Since every sentence frame possesses each of these three slots (or features), the default value for each has been defined to be the value it is most likely to take. Thus, the default sentence in this system is defined to be a simple declarative sentence that is active in voice and contains no negation.

5.1.4.1.1. Multiple Inheritance in an Earlier Version

While the use of a "nil" default type may be very simple here, it actually came out of a more complex system that preceded this one. In that earlier system, a sentence also inherited values from a verb frame, and the syntactic and semantic representations of a sentence were collapsed together in one frame. As the system stands now, the presence of "nil" as a slot value in a sentence only means that the sentence did not possess a value for that slot - for example, it had no object. In the earlier more complex system, the inheritance mechanism would first look in the verb definition for a

missing value, and only if it did not find a semantic default value would it then inherit the value "nil". In this more complex system, employing multiple inheritance, the meaning of this "nil" value is greatly changed. Its presence there means that the requested value does not belong in this sentence, or is not appropriate in this sentence (actually, "this clause" which means with this verb).

While, we have eliminated this use of multiple inheritance here, its effect on our ability to represent meaning is interesting. It says that by combining sources of inheritance, we can represent meaning that belongs to the combination, but is not to be found in either of the single instances. While the notion of inheritance is in itself simple, a judicious use of multiple inheritance may allow us to express surprisingly complex ideas. In this case, the inheritance of a value from a verb frame, says that a specific value was missing in the sentence frame, but a more general value can be understood to belong to that frame. The inheritance of a nil value says that the value is missing. The order in which the inheritance is tried says that whenever a value is missing (is nil), it is not appropriate for this sentence. It is only the order of inheritance from multiple parents that supplies this last meaning, something which says a good deal more than any thing which has gone before - it says that we have fully represented the meaning of this sentence, and that we are confident of that. We have managed to define the sentence by enclosing it within very real limits, limits which perform a kind of closure on the universe or scope of its meaning.

Given that natural language is able to express a good deal more than is ever said explicitly, this kind of closure is extremely important to achieve if we are ever to represent linguistic meaning in a strictly formal computational setting. In the system that is maintained here, the same kind of complex closure is present but it is not so elegantly expressed. Here the same sentence is represented both as a sentence made up of syntactic elements and as a semantic case frame representing the meaning of the same sentence. (See section 5.2.2 for a description of how a case frame analysis is implemented here.) Since inheritance exists at the case frame level as we have described it above (inheriting from a parent verb definition) we can say that the same idea of closure is expressed if we recognize that the two frames (sentence and case frame) refer to the same linguistic pattern and we know the rules for transforming sentence (syntactic) slot values into case frame (semantic) slot values. In a sense we have sacrificed elegance for simplicity, and as result have to recognize how more disparate elements go together to represent the same idea. The reasons for doing so are that traditionally syntax and semantics are defined along separate dimensions, and the process of exploring an adequate semantic definition is still not very well known, and hence deserves to be considered as a separate problem if only to allay confusion. Perhaps, when those confusions are removed we can find a description of language that includes semantics and

syntax in the sense that we have a third method of description that encompasses all the features of both. Then we might return to a more elegant representation of the way in which a sentence expresses closure, since it does indeed express closure to all of us (or it only expresses confusion, since it was poorly formed by the speaker). Such a method of representation (a map) would be a true guide, while the uses of syntax and semantics now only provide us with partial guides.

5.1.4.1.2. Context Inheritance

Some sentences may inherit values from a preceding sentence in order to supply context. At this stage of its development, the parser only employs this kind of inheritance for the question "Why?". It may be worth exploring to see if this kind of simple inheritance could be made to fill in the missing parts of compound sentences, or even of independent previous sentences. We have held off from exploring that issue because much of this kind of inference is handled by the semantic component of this system. In that component, default values that belong to a class network are defined for verbs, and where those values are missing in a particular sentence, the values are filled in by the semantic component at the most general, but still meaningful level possible. If the reasoning component can be satisfied by these default values then inheriting more specific values in the course of parsing a sentence may not be necessary. An example:

If you study for the test, then you will pass.

Inheriting at the sentence level would say that "you will pass the test", while inheriting at the semantic level will say that "you will pass some_task". While the first interpretation is more specific, the second may be adequate, and may even be the more accurate interpretation, since the speaker may have been too loose and may have meant something more general (such as a course) in his elliptical remark. It may be that the overall context would give this latter interpretation.

The system currently depends upon the more general semantic inheritance where that can be employed, and only uses context inheritance where the verb is missing, and the issues involved are very clear. Thus, context inheritance is employed only to allow the question, "Why?". True context inheritance could prove to be a very difficult and expensive proposition. Thus it seems better to explore whether or not the much simpler and cheaper use of semantic inheritance is not sufficient to supply the needs of the system.

5.1.4.2. Sentence Frame Organization

An instance of a sentence frame may contain any of the slots defined in the parent sentence frame. Since individual rules know what kind of values go into these slots it has not been necessary to define a type of value for each slot. (A "type" is a general classification of frames defined in a hierarchy; a frame is of any type where that type is itself a frame name defined to be an ancestor of the frame in question. An individual slot may be defined to accept as values only those frames which are of a particular type. The use of this kind of constraint will be discussed later in this chapter, in the section on semantic representation.) While typing seems to be very useful in the course of semantic definition, syntactic definition does not seem to derive any real benefit from its use. The only slot in the top level sentence frame that might benefit from having its potential value typed is the indirect object slot. Since this is the only case where semantic information needs to be referenced, this typing has been relegated to the rule itself, which simply looks up the type of a noun phrase before placing it in the indirect object slot of a sentence frame.

The following example represents the kind of values that may be found in the slots of a sentence frame:

```
sentence:  "An important test is scheduled for Monday."
        frame:  sent1
                is_a : sentence
                subject:  test1
                verb:  schedule
                time:  monday
                act_pass:  passive
                case_frame:  case_frame1
```

The first slot in this definition, the "is_a" slot contains a value which is the name of another frame. As described previously, this enables the frame, sent1, to inherit values from the default sentence frame, so that the system knows that sent1 is a simple declarative sentence that is positive, and does not contain a modal, an interrogative, an indirect object, an object, or a prepositional phrase.

The subject slot contains the name of another frame, test1, which is a noun frame defined as:

```
frame:  test1
        in_of:  test
        reference:  indefinite
        number:  singular
        status:  important
```

Noun frames are created from the noun phrase structures described in section 5.1.1. Each noun phrase is represented by a frame whose name represents a unique instance or use of that noun (test1, test2, lab1, etc.). Such instances are

attached to a network that defines each noun known to the system in terms of its place in that network. In this case, test1 is an instance of "test", which is "some_task", which is "a task", which is "an event", which is "an entity". "Entity" is a category that includes all nouns known to the system. The uses of this network will be discussed in the section on representing semantic meaning. At that time we will also discuss why separate instances are defined for each noun, as well as the presence of semantically descriptive slots and values such as that represented by the slot "status" and its value "important". The other two slots in this frame, reference and number, hold syntactic information that record the actual form of usage of this noun in the input sentence.

In addition to noun frames, a sentence frame may also contain embedded sentence frames which belong to the same definition (inherit from the default sentence frame), and which may themselves contain embedded sentence frames. Embedded sentence frames may occur in either the subject or the object slot of a sentence frame or of another embedded sentence frame. The following example shows the recursive nature of these possibilities:

Does the speaker believe that the hearer wishes to pass the test?
frame: sent2
is_a : sentence
sent_type: yes/no question
subject: speaker1
verb: believe
object: embedded_sent1
is_a : sentence
subject: hearer1
verb: wishes
object: embedded_sent2
is_a: sentence
subject: hearer1
verb: pass
object: test2

5.1.4.3. Summary: Benefits of a Frame-Based Syntactic Representation

The syntactic sentence frame contains information that will be useful to the context interpreter: this includes the sentence type, the possible presence of modals, and an indication of whether the sentence is positive or negative. (See chapter 7 for a description of this context interpreter.) This frame also contains the raw material for semantic analysis, including those values which fill semantic roles, as well as the clause structure of the sentence and a record of whether or not the sentence is active or passive. This syntactic representation also contains information necessary to produce a grammatically acceptable reply to the user. Lastly, the use of a frame system at this level of representation also gives the possibility of using inheritance to fill in language context.

The use of frames to represent even the lower level syntactic information contained in a sentence allows the overall system to achieve a uniform, consistent representation: all meaningful content, including user-entered sentences, inferred sentences, system replies, and context representations are represented as system frames.

5.2. Semantic Analysis

Once a clause has been parsed into a frame representation that is predominantly syntactic, a semantic analysis of the syntactic frame elements produces a case frame representation of the clause. This case frame becomes a slot value in the overall sentence frame, and inherits its form, and may inherit some of its values, from a frame that defines the main verb of the clause. (See Diagram 2, at the end of this chapter, for a picture of how the sentence frame, the case frame, and the verb frame, are related to one other.)

In Chapter 4, this form of representation was discussed in general terms. We said that a semantic, or case frame, representation of a sentence could be seen as an intermediary between a syntactic analysis and a pragmatic analysis of a sentence's contents. We also said that a semantic representation of meaning could be seen as a representation of literal meaning, where such a meaning could be said to represent the internal or purely linguistic relations that hold among the elements of the sentence. An actual, or interpreted, meaning could only be arrived at through a third level of analysis based on pragmatic or contextual (what stands next to the text) analysis.

In section 5.2 we will begin by examining how case frames have been implemented in this system as an extension of syntactic analysis. What we describe here should be applicable to any analysis of English language text. Here we are primarily interested in how case frame definitions help us to complete the analysis of individual clauses, so that the entire meaning of a clause, in so far as that meaning can be discovered by examining only that clause (with some previously noted exceptions), can be represented. Here we are primarily concerned with how syntactic elements are placed in semantic roles, and with how elements not explicitly stated in a clause can be filled in by employing semantic default values.

Cases, however, are not defined by their contents (that is, by the "types" they will take), but are defined instead by the uses that we make of them. While cases are filled by syntactically discovered elements, they are defined to serve pragmatic needs. We will end this chapter by briefly examining this other side of semantic analysis: cases are roles chosen and created to meet the needs of a pragmatic analysis. At that time we will look at what roles the cases

employed here are intended to represent, and we will examine the issues of how to represent those roles.

5.2.1. Semantic Dictionaries

5.2.1.1. Verb Definitions

Each case frame is an instance of a parent verb frame. The following are the case frame definitions of the verbs "to schedule" and "to advise":

```
schedule:
        is_a:  action_verb
        cases:  actor, object, recipient, time
        actor:
                type:  person
                value:  some_person
        object:
                type:  event
                value:  some_event
        recipient:
                type:  person
                value:  some_person
        time:
                type:  time
                value:  future_time


advise:
        is_a:  belief_verb
        cases:  believer, object, recipient
        believer:
                type:  person
                value:  some_person
        recipient:
                type:  person,
                value:  some_person
        object:
                type:  sentence
                value:  nil_sent
```

Each verb frame contains a slot that lists the cases that it will take. Each of those cases is then defined in terms of the type of value that it will take, and the default value that will be inherited by a case frame attached to this verb if that value has not been supplied by the input. For example, in the clause, "a test has been scheduled for Monday", the input will provide values for the object slot ("test1") and for the time slot ("Monday") of the case frame. The parent verb frame will provide default values for the actor slot ("some_person") and the recipient slot ("some_person").

These default values say nothing about individuals (they are not individuals, but are only classes), so the same default value may fill more than one slot of a definition as it does in the above example. The presence of these values supplies the higher level reasoning components of the system with a regular representation of meaning. A case frame attached to the verb "to schedule" will always have a recipient who is assumed to be a person, even if it is not known who that recipient is. In the same manner there will always be an actor who does the scheduling, an event that is scheduled, and a time when the scheduled event will take place. These roles will always be present whenever the verb is present; the input attached to that verb will determine how much specific detail informs this pattern.

The two examples have been chosen to show that there are two different kinds of verbs that can naturally be distinguished in the expression of practical arguments: there are those verbs that belong to actions that commonly make up antecedents, consequents, and events; and there are those verbs that are performative in nature and point to the logical structures (practical arguments) that we deal with here. This second category of verb may be explicitly used or (what is more likely) they may be implicitly understood in the course of giving advice or a warning.

Since we have two kinds of verbs we could define more kinds of verbs, and introduce distinctions and a hierarchy that simply served to make our definitions more economical in expression. From our actual distinctions, we could, for example, define the "believer" case at the level of the category or class of "belief_verbs", and let all verbs that belong to that class inherit that case. This has not been done in this application. Currently there are not enough verbs in the system's working vocabulary to make this worthwhile. Beyond this, there are enough subtleties involved that any application, however large, should begin by defining cases at the level of individual verbs. Only later would enough experience and confidence be gained to begin to employ a hierarchy to make those definitions more economical.

Our second example ("advise" classified as a "belief" verb) also shows a situation we have not considered previously. In a sentence such as "I advise you to study for the test", "I" fills the role of believer, "you" fills the role of recipient, and "you to study for the test" is the object. The object role is filled by another sentence frame, one which will have its own semantic roles defined in its own case frame. (It would probably be more logically consistent to define this case to take another case frame rather than a sentence frame, but it is not entirely clear that this would be the best choice, and so the more inclusive frame has been made the value of that slot.) This recursive definition of cases parallels the syntactic structure of the sentence, and seems to be necessary to handle some kinds of verbs. In order to retain some uniformity in representation and processing, these recursive definitions have been handled here by using case slots, but

we should notice that a case that takes a sentence or clause is significantly different from a case that takes a single semantic entity or noun. A case itself is a role, but a sentence or clause cannot fill a role, as becomes obvious when we try to supply it with a default value. The only default value we can give it is a marker to indicate that there is a value that must be supplied by the context. Probably the best way to understand the difference is to see that a slot that takes a sentence for its value is a slot that belongs to the processing of linguistic input, and does not belong to the representation of meaning which is the end product of that process.

5.2.1.2. Noun Definitions

Nouns which may fill the case slots defined in the verb definitions are defined by their placement in a network. The following represents the definition of the noun, "test":

```
test:
        is_a:  some_task
some_task:
        is_a:  task
task:
        is_a:  event
event:
        is_a:  entity
entity:
        description:  any_noun
```

This particular location (or definition) of a "test" is, of course, specific to the application that might be imagined for the sample dialogue presented in Chapter 1. Here we are concerned with students, and books, and the like. Within that domain or application a "test" is, in its most literal sense, a task; it is not a scientific experiment and it is not an ordeal (as in 'trial by ordeal').

5.2.2. Case Frame Analysis

The verb is the key element in a semantic analysis of the sentence. Every verb is defined to take a set of cases. A case analysis cannot be performed until the verb of a clause has been successfully parsed and placed in the sentence frame. It then remains a question as to whether or not to begin the case frame analysis immediately and to allow it to proceed in parallel with the syntactic analysis, or to wait until the syntactic analysis is complete before beginning the case analysis. In this system the case frame analysis waits until the sentence frame analysis is complete before it is

begun. This is done purely for the sake of simplicity: it helps to keep the two processes separate while exploring the domain of each. There is no reason however why the first alternative could not be done, and, of course, it would be the better choice if a parallel processing system were to be employed here.

The verb definition drives the process of producing a case frame analysis whose input is a frame that represents the syntactic elements and relations of an individual clause. The following verb definition controls this process for a sentence such as "you should study for the test".

```
study :
        is_a  action_verb.
        cases : actor,object,objective,time.
        actor : type: person.
                value : some_person.
        object : type : knowledge_source.
                value : knowledge_source.
        objective: type : task.
                value : some_task.
        time:  type: time.
        value:  future_time
```

Each verb definition contains a list of cases to be filled. For each case that is to be filled, there is a set of rules that examines the current sentence frame, looking for the proper value to fill that case slot. Where more than one syntactic slot may contain the value, and there are no syntactic clues (such as voice) to aid in placement, the frame-based typing mechanism will cause an incorrect candidate to be rejected, and the rules will eventually find the correct candidate, or will stop searching, and then the parent verb frame's default values will be used for that case. This process will be repeated for each case in the list of cases. Thus, each verb has an individual definition of the cases that always go with it; for each of those cases, there is a list of syntactic conditions and slots that may provide the value to fill that case for any verb that takes it; each verb defines the type of values that may go into a particular case; and lastly each verb contains a default value that will be used to fill a case if its value is not found in the input clause or sentence.

The following are two Prolog rules that might be fired in an attempt to fill the "actor" slot in a verb that takes an "actor" case.

```
continue_case_analysis(Curr_sent,Case_frame,[actor|Remaining_cases]) :-
        get_slot_val(Curr_sent,act_pass,active),
        get_slot_val(Curr_sent,subject,Actor),
        Actor \== nil,
        add_slot_val(Case_frame,actor,Actor),
        continue_case_analysis(Curr_sent,Case_frame,Remaining_cases).
```

```
continue_case_analysis(Curr_sent,Case_frame,[actor|Remaining_cases]) :-
        get_slot_val(Curr_sent,act_pass,passive),
        get_slot_val(Curr_sent,by,Actor),
        Actor \== nil,
        add_slot_val(Case_frame,actor,Actor),
        continue_case_analysis(Curr_sent,Case_frame,Remaining_cases).
```

The first rule says that if the sentence frame being examined is "active" then look in the subject slot for the value to fill the role of "actor". The second rule says that if the sentence is passive, then look in the "by" slot for the sought after value. In either situation it is possible that there is a value to be found in the designated syntactic slot, but it is not the value that should fill the "actor" slot. In all attempts to fill a case slot, a further check on the "type" of the candidate is performed, and if the candidate value is not of the right type, then that value will be rejected, and the search will continue until either the proper value is found, or the possibilities are exhausted. If no acceptable value is found, the verb frame's default value will be inherited by the case frame.

Such a system combines general syntactic patterns with the need to very carefully explore and understand the domain of the application in which these particular verbs will be used. The typing mechanism requires that only the most general category that could possibly be attached to a particular verb be used. For instance, we know that only human beings "study" and only human beings "schedule", and so we are safe in defining the actor (or agent) slot of both verbs to be "some_person".

Actually, we are only safe with this definition if we are assured that the verbs will be constrained to a literal usage within the domain we have described (tests, school, books, etc.). In order to employ default values and typing mechanisms that carry some meaning, it is necessary to make a series of "common sense" assumptions about the limits of usage of the verbs to be defined. Upon closer examination, however, we are likely to find that all of those "common sense" assumptions have their source in the nature of the application, and the restrictions that it applies to a use of the English language. From this experience, we can see that the use of a case frame representation of the English language necessarily takes us toward a domain-oriented approach to the English language.

While there is no guarantee that this coupling of a search pattern and a semantic constraint will always find the correct value for a case, it has enough flexibility to be expanded when situations arise that have not previously been covered. For the examples used in this thesis, it has proven sufficient to handle all the situations so far encountered. Beyond this it remains an interesting question as to how we might go about filling a case frame without having to be so rigid as to extend the syntactic grammar to include all the semantic possibilities. This possibility would entail defining

the search pattern for each verb individually. This is not an entirely pleasant prospect since it seems to indicate that there really is no difference between syntax and semantics. We would also like to think that there is an element of uniformity that belongs to each case, so that it is not defined individually for each verb. The other side of this issue is how we handle the "typing" of values that may go into a particular case. Here we have defined that individually at the verb level, and while there is consistency across those definitions, that consistency is not absolute: for some cases, a different type is employed between verbs taking that case.

The method we have employed tries to follow a middle road, one that says, on one side, that semantics is not syntax and should not involve the extension of a syntactic grammar to include semantic issues with the same rigid formality. On the other side, semantic meaning is not merely a typing mechanism, although it does employ such a mechanism to achieve the necessary effect. A case is not a type; rather a "case" is a role, and roles can only be understood in the relation between semantics and pragmatics.

5.2.3. Case Frames as Roles

If we look beyond the typing mechanisms used to define the contents of a case, and examine the use to which cases are put, then we will see that cases are nothing more than predefined roles in a set of predefined circumstances. At that level of representation, a case is not a linguistic category, but rather is an attempt to define and lend regularity to pragmatic situations. And this appeal to pragmatics is the only really adequate definition of a what a case is. The case cannot be defined by the type of value it will take, and it certainly cannot be defined by the associated default value that goes with that type. We could expect those types and defaults to change within a single case from one verb to the next. (If we do not allow the use of different types in the same case, then we are going to need a lot of cases, so many in fact, that the whole system is not likely to prove useful.) All of this is not to say that we must take a domain-oriented approach to the English language in order to process it and understand it; it only says that if we attempt to use a case-based system to do so then we will find ourselves drawn into a domain-oriented approach whether we wish it or not.

In the system used here, the knowledge base is composed of plans and goals, and the cases employed carry the elements of linguistic content that can help to express those plans and goals. For example, a plan may be expressed as a Prolog fact:

```
(study,person,test,planner(person)).
```

The elements here represent, in order, an action, an agent to carry out that action, an objective, and lastly an expression of the constraint that the one who plans and the agent must be the same person. This is one possible way to express a plan in a knowledge base. (Another possibility might be to include the instrument as an element of the plan - here, that might be a knowledge-source.) Whatever method we choose for representing a plan, we would expect that method to be consistent throughout the knowledge base, so that all plans would have the same number of elements, and the same general expectations with regard to each element (for instance, 'the second element of the expression is the agent'). The English language, of course, does not employ any such consistency when it comes to expressing plans, or any other general category of knowledge or organization of knowledge. The purpose of a case frame system is to serve as an intermediary between English (which does have some consistency at the level of syntax) and a consistent expression of knowledge. The cases are roles that can be filled with syntactically recognized content. Definitions of verbs are written to provide all the necessary roles for an application or knowledge base, and general default values are provided to insure that all the roles that need to be recognized will have content when it comes time to process them in the terms that are preserved and recognized in the knowledge base.

5.3. Sample Sentences

The following is a representative list of the sentences that the parser and case frame interpreter can successfully process. This list is not intended to be exhaustive, but is intended to present a fair representation of the kinds of sentences and variations which can be handled. The sentences listed here are variations on those to be found in the sample dialogue in chapter 1.

A test is scheduled for Monday.
A test has been scheduled for Monday.
There is a test scheduled for Monday.
The teacher scheduled a test for tomorrow.

If you study for the test, you will pass the test, so study for the test.
Study for the test.
If you don't study for the test, then you won't pass the test, so study.
Study.
You should study for the test.
I advise you to study for the test.
I advise that you study for the test.
You are advised to study for the test.
I am warning you to study.

Why?
Why study for the test?

Why advise the hearer to study for a test?

Does the speaker believe that the hearer wishes to pass the test?

It will take all of the available time to finish the lab.
To finish the lab will take all the available time.
Finishing the lab will take all of the time.

Did you forget about the test?
Did you forget the test?

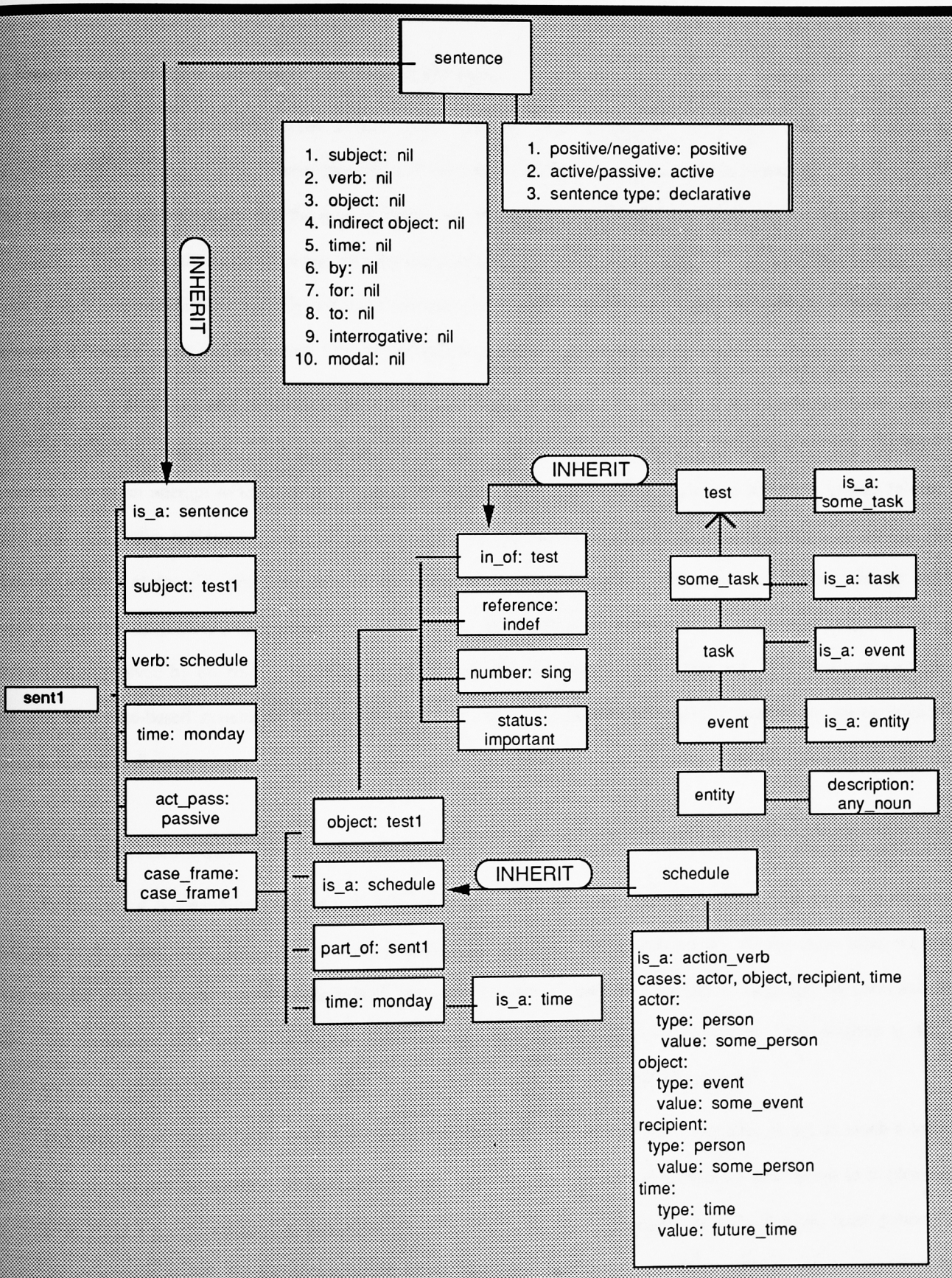
You could cheat and someone else could finish the lab.

You may not be caught.

5.4. Complete Example

See Diagram 2.

Diagram 2: Sentence Representation. "An important test is scheduled for Monday."



6. Understanding Natural Language: Syntax and Semantics

In chapter 4, we considered some of the issues involved when attempting to represent meaning in a Natural Language Understanding system. Three categories of representation were discussed: syntax, semantics, and pragmatics. The center of this consideration was the area of semantics, which sometimes resembles syntax, and sometimes resembles pragmatics. The semantic mode of representation discussed was a case frame system. While the major problems confronting such a representation could be resolved theoretically, it still remained true, when we were done, that semantics remained a "murky" area, and that cases were an ambiguous mode of representation, at best.

Chapter 5 described several cooperating mechanisms that can be used to implement the representation of meaning that was outlined in chapter 4. Descriptions in Chapter 5 focused on details of implementation, but we can also think of a mechanism as an attempt to describe a portion of the world, since a mechanism, once it is complete, should be able to cooperate with reality without outside intervention. Thus the mechanisms described in Chapter 5 serve as another, more detailed, attempt to describe and embroider on the issues discussed in chapter 4. Now in Chapter 6, an attempt will be made to use the mechanisms of a parser and a case frame interpreter as a description that may shed some light on the issues left unresolved by the more theoretical considerations that were presented in Chapter 4. The center of attention is the idea of a case-based system for representing meaning: is it an extension of syntactic analysis, or an expression of pragmatic analysis?

6.1. Syntax is not Semantics

In Chapter 4 we examined the idea that English uses word order to convey meaning, and that it is somewhat different in this from those languages that use a case-based system. (See section 4.1.2.) At the same time, we never confused the semantic case system implemented here with the use of cases by such natural languages as Latin and Russian. We recognized that cases used in a natural language are a purely syntactic phenomenon. The problem is that we never clearly understood what constitutes (and what does not constitute) "syntactic phenomena".

The notion that English uses word order to convey meaning clouds the issue. This idea is not so much a mistake as it is simply not precise enough: at least, it lacks precision when it comes time to create a mechanism to implement it. In order to build a parser we need to recognize the patterns that a natural language uses. In English, those patterns are made up, to a great extent, by the sequences of words. In case-based languages, those patterns are composed of case

endings. To say that English is dependent on word order to do what was previously done by cases (word endings), is to say, more precisely, that English uses the sequencing of words as a syntactic notation rather than using word endings. In all natural languages there is the same kind of syntactic ordering; in languages that use case endings, the word sequence may not be important but the case ending that a noun has gives it a position within the syntactic ordering of a sentence. Regardless of the method, a natural language has a basic order to its expressions. That order, however, is too abstract and too general to convey meaning: it is simply that words commonly go together in somewhat expected ways, and this does simplify efforts to understand one another. English does not use word order to convey meaning; it does use sequential pattems of words to represent order. Perhaps this use of sequencing is an improvement on the more formal use of case endings to represent order (and perhaps not).

The confusion of order and meaning is a natural one since, in our use of language, we never find them apart. However, when we attempt to build a mechanism to do what we do every day, we find that the two can be separated, and indeed, they have to be separated in order to proceed with that enterprise. It is quite possible to build an ordered representation of an English sentence without considering that representation as one which also represents meaning. The frame that is produced by the predominantly syntactic parser is just such a representation. (See Diagram 2: in that example, the frame called "sent1" represents the syntactic order of the example sentence.) The parser uses word sequence, together with a dictionary that knows the parts of speech of words, and it is able to produce an ordered representation of an English sentence.

The major point to be made here is that the construction of the syntactic parser, and the semantic case frame interpreter, as separate mechanisms shows us that it is possible to represent the syntactic order of a sentence prior to any attempt to interpret its semantic "meaning". The next step is to attempt to discern the relationship that does exist between syntactic order and semantics. If semantics is simply a more discerning examination of more details in the pattem of expression, then semantics is an extension of syntax, and the difference we have described here will evaporate into a difference that is merely one of degree. In such a relationship, a syntactic parse of a sentence represents a kind of quick overview of its structure, and semantic interpretation is a more careful and detailed examination of that structure with all of its subtleties and refinements. If such a relationship can be demonstrated, then we know enough about language to be able to represent its literal meaning without reference to pragmatic issues. This is not the opinion arrived at here.

6.2. Semantics Is Not an Extension of Syntax

English can be said to have very little in the way of syntactic regularity when we come to consider the ideas that it conveys. We saw examples of this phenomenon briefly in chapter 4, where [WINO 83] provided us with two sets of examples to highlight this phenomenon.

> We baked every Wednesday evening.
> The pecan pie baked to a golden brown.
> This oven bakes evenly.
>
> Cyril liked spaghetti.
> Cyril ate a meatball.
> Cyril got a bellyache.

In the first set of sentences, the subject of the verb 'to bake' plays in turn the semantic roles of actor, affected, and instrument. In the second set of sentences, the subject, Cyril, plays entirely different roles in each. First he is in a certain state, then he performs an action, then he undergoes something. In all six sentences, merely recording the subject of the sentence does little or nothing to indicate the role that it actually plays in the structure and meaning of the sentence.

As we said in Chapter 4, the mapping of semantic roles to syntactic structures is a very loose affair in English; that is, the syntactic role of a noun phrase in English tells us very little about its semantic role. Now that we have built mechanisms that attempt to represent this loose mapping of syntactic structures to semantic cases, we can go one step further and try to explain why English works in this way. This "looseness" is there (between semantic roles and syntactic structures) because something else intervenes between the two. At least in the mechanisms constructed here that intervention takes place in the form of two cooperating representations of world knowledge that participate in the transformation from syntactic structures to semantic cases.

In the mechanisms created here, the process of filling a semantic representation from a syntactic one follows a process of examination that is similar in appearance to the one employed by the parser. The most discernible immediate difference is that the parser had to deal with a sequence of words; the case frame interpreter has a much simpler task in that it only has to deal with a limited number of categories that may or may not have content following the syntactic analysis of a sentence. The syntactic structures are examined by a set of pattern matching rules and this does give the appearance, at first, that semantics is an extension of syntax. These rules, however, are very simple, representing little more than an ordered sequence of places to look for an element to fill a semantic role. For example, to find the object of an action the rules will examine the subject slot of a syntactic frame if the sentence is passive, then will examine the

syntactic object slot if the subject slot did not supply the semantic object. In our example of 'the boy who threw the dog a bone', the "bone" is the object and it may appear as the subject of a passive sentence ("A bone was thrown to the dog by the boy."), or it may appear as the object of the verb in a passive sentence ("The dog was thrown a bone by the boy.").

The pattern matching rules however only represent the steps to be followed in attempting to arrive at a semantic representation of a sentence. Those steps follow a path that is marked out by a dictionary of verb definitions that indicates, for each verb, what roles should be filled, and which also limits or constrains the content that is appropriate for each role. In the above example, the role (or case) of object attached to the verb "to throw" would be defined to accept a "bone" but reject a "dog". The verb definitions then define what we look for (the cases attached to the verb) and what can fill each of those cases (the type that limits its content). This dictionary is not an extension of syntactic (or language based) knowledge. The roles are derived from world knowledge. The typing constraints are also the product of world knowledge, a knowledge expressed as a classification of nouns. It is important to realize that this world knowledge must be present before we can successfully create and fill a semantic structure from a syntactic one. This means that semantics is not an extension of syntax, and that even the literal meaning of a sentence is arrived at by the application of prior knowledge. Semantics, it appears, is an extension of pragmatics only; albeit the pragmatic issues involved are a part of common sense, and tend to be conventional.

Now that we have attempted to deal with this loose coupling between syntax and meaning, we can see that this is probably the only way that language can work. Syntax, by itself, has always been too thin a structure to carry meaning. In order to introduce meaning, we must add world knowledge to the process of interpretation and representation. This is true even before we begin to represent the sentence as one belonging to a specific context. Just to represent the sentence in an ordered and meaningful way to the process of context interpretation, we must first make use of two sources of world knowledge (or prior context). These two are a system of general knowledge (a classification system), and a system of knowledge specific to the domain we cover (a knowledge of roles). A case based system of representation cannot be arrived at without these two kinds of world knowledge. Prior to introducing those two areas of knowledge to the process of interpretation, we can represent the elements of a sentence in an ordered way, but not in a meaningful way.

We can make our argument stronger here if we examine more closely what cases, whose contents are defined by types, are supposed to accomplish. Suppose, for the sake of argument, that there is simply something lacking in the

mechanism we have described here. Suppose that a mechanism could be built which did not use typing as a constraint to fill the various roles or cases. Such a mechanism would use a much more detailed analysis of the syntactic order in the attempt to determine which syntactically discovered element belongs to which case. Possibly the mechanism described here could have been built along those lines, using pattern matching rules to fill the cases. This was not attempted only because the use of typing constraints seemed to make it unnecessary to go any further along this route. Although using typing constraints does seem to be the cheaper and the more efficient method, its use does not demonstrate that it is necessary, or that an extension of the parser to include semantic roles is impossible. However, when we look beyond the production of a case frame to its actual use, then we can see that the typing of a case's contents is vital to any representation of meaning. It is the type that a case takes that supplies us with a default value for a case, and it is the availability of default values (that is types or classes) that makes it possible to infer and fill out the meaning of a sentence. Many sentences can be represented to the next level of interpretation and representation only because of the inferences we supply. Further, the ability to make inferences is vital to any system for understanding natural language as it is really used.

At every level of interpretation, the ability to supply inferences is necessary in order to achieve a meaningful representation of natural language. And at the level of mapping a case structure to a syntactic structure, the vehicle for supplying inferences is the system of classes or types which constrains what content may fill any given role. Without the system of classification and of typing constraints there is no ability to make inferences, and without the ability to make inferences, there is no possibility of representing meaning. That we use this typing mechanism as a shortcut to filling the cases is to recognize the pervasive role they play in determining and representing meaning in a case-based system.

Chapter 2 briefly presented the idea of an indirect speech act, and chapter 4 took that discussion further, attempting to allow that language has two levels of meaning, one a literal meaning, and the other an interpreted meaning. The interpreted meaning could be derived by examining the literal meaning in the context of a specific situation with a set of rules that describe that situation. The literal meaning described in chapter 4 corresponds to the case frame representation (together with syntactic clues such as sentence type and the presence of a modal) presented in chapter 5. We attempted to show in chapter 4 that there is only one speech act and that it is to be found at the interpreted level of meaning. The literal meaning would not be considered at the level of a speech act because it could be derived entirely from an examination of language. Now that we have built the mechanisms to implement a literal representation of meaning, we know that we went too far when we said that a literal meaning could be derived without reference to pragmatic issues. The crucial issue here is that we cannot represent the literal meaning without reference to a non-language source of

knowledge. Even the literal meaning is dependent on a prior context, albeit, one that is (and should be) utterly conventional and without controversy. Whatever we call these two levels of meaning, we should recognize that the difference between them is one of degree and not of kind. It would be better to return to the idea of two speech acts, Sp2 and Sp1, and to attempt to describe the difference in terms of access to experience. Neither Sp2 nor Sp1 can be derived without the presence of world knowledge; the difference is that Sp2 is derived from a store of conventional knowledge, built up over time, which does not directly examine the current state of the world. Sp1 is then derived from that conventional meaning in conjunction with a direct examination of the current state of the situation to which the speech act refers.

6.3. Syntax and Semantics as Parallel Systems

Our description of semantics depends on using a case frame structure as the form of representation. Other forms of representation may not give the same view as the one presented here. However, it is difficult to imagine other adequate methods of representation, and in Chapter 4 we learned that systems with wholly different perspectives on language still tend to use some form of case frame representation. Meaning seems to be well represented in a form that recognizes what roles (cases) go with a particular verb. However, an early attempt to build the mechanisms described here did not recognize the distinction between syntax and semantics, and the resulting mechanism was not suitable to the task of representing meaning. It might be worthwhile to examine the relationship between syntax and semantics by beginning at that point.

6.3.1. Syntax and Semantics as Separate Systems

An initial attempt to build the parser described in Chapter 5 attempted to use the syntactic categories as the categories of semantic representation. In this system, a sentence would always be represented by its active form and the parser would directly make the necessary transformations. The following example was used in Chapter 4 to help explain the use of cases, and it is simple enough to be handled by the method originally employed here.

The boy threw the dog a bone.
A bone was thrown to the dog by the boy.
The dog was thrown a bone by the boy.

Since all three sentences represent the same event and set of relations, we should represent the meaning portrayed in all three sentences by a single structure, one which can be derived by a set of rules or transformations form the different

surface structures of the sentences. In the original parser and form of representation, the subject would always be "the boy", the indirect object would always be "the dog", and the direct object would always be "the bone".

There were three major faults with this initial effort. First, the parser was unnecessarily complicated since it tried to do two things at once: it attempted to analyze the syntactic order of a sentence, while at the same time attempting to transform its elements in the record that it produced. Secondly, the limited number of syntactic categories available were not sufficient for semantic representation, so some further mode of semantic representation would still be needed; that is, the transformations employed did not really accomplish anything worthwhile. Third, the surface structure of the sentence (what was actually written) was lost. This initial, naive effort was abandoned in favor of cases as the categories of semantic representation.

The implementation described in chapter 5 began with a fundamental misconception, attempting to collapse the entire representation of syntax and semantics into one level. Eventually that misconception began to become apparent and the system was redesigned to have both a syntactic parser and a semantic interpreter and representation. In the course of doing so, however, confusions of the two remained, and are reflected in some of the descriptions of Chapter 5. Now, an effort will be made to completely dissolve this confusion.

In Section 6.1, syntax was described as order, and was considered in terms of the roles that words play in that order. The categories, or roles, are such as subject, object, indirect object, and the like. In section 6.2, we considered semantics as a collection of roles together with a network of classes that define the content of those roles. Now we can also consider that those roles that are syntactic also take content which can be described in terms of classes. Here the classes are categories such as noun, verb, adjective and the like. Such classifications are broad, and are usually not broken up into subclasses. Thus the method for representing the syntactic classes is different from that employed for the semantic classes (a network), but the basic relationship between roles and classes is essentially the same. In both representations, the classes define what content can fill what roles. A syntax-based rule, for example, that looks for the subject of a sentence always looks for a noun phrase, a structure made up of nouns, and words that modify nouns, but not of verbs or adverbs.

In the syntactic scheme, the elements are words "as words", or, perhaps better, words as "tokens". These words have existence in their own right, apart from any meaning we might give to them. We know this because we can define a word (or token) as a member of a class: as a noun, or as a verb, etc. The semantic scheme is made up of concepts

only. The classes that are represented here are such elements as "task" and "person"; along with these concepts there are roles attached to actions, and these determine how those classes may interact with one another. If words, as words, could be thought of as having weight, then this semantic realm could be described as weightless.

The distinction now is so clear that we might refer to it as the "classic" view of language: the view it presents is clear and difficult to find fault with. The distinction is so clear, that we have to ask how it could have been avoided for so long here: perhaps it is just that the first mistake (building a parser that tried to be as simple as possible) takes the longest to dissolve itself. Perhaps the answer is that having explored parsing by syntax, it took time for the less well known semantic issues to separate themselves and make themselves clear. The key to making the separation was to realize that the introduction of world knowledge in the form of the typing system and the system of roles, gave the semantic realm a clear and distinct identity. This realization, in turn, freed the syntactic scheme from having to carry any semantic burden.

Now that we have managed to separate syntax and semantics, we can make a brief attempt to describe language as the parallel application of these two systems. Instead of trying to collapse the two into one as we did at first, and instead of applying first one and then the other (as the system described in Chapter 5 does), we entertain the possibility that the two can be applied on an equal, parallel basis all along the way. This seems the most likely way that human beings understand language. In order to do this, we will first look at a simple case of cooperation between the two systems. Next we will briefly examine what each looks like when it stands entirely alone. Following that, we will try to imagine how the two can be combined to make what we know as a natural language.

6.3.2. Using Semantic Information in Syntactic Analysis

Syntax is composed of such roles as subject, object, and the like; those roles are filled by contents defined as belonging to such classes as noun, adjective, and the like. Semantics uses such roles as agent, recipient, and the like; such classes as person, event, and the like contain elements which may fill those roles. We imagine here that the two systems are entirely distinct and are employed in parallel as tools for interpreting and understanding an English sentence or statement. This is not a particularly easy situation to imagine or describe. There is only one activity going on, and if we use the metaphor of a path for the activity of understanding language, then we can describe this activity as made up of steps along this path, where each step is simultaneously syntactic and semantic, and no step is one without the other.

Even so, it seems that the two systems must work together so well as to be unobtrusive, and if that is so then we can expect a certain amount of redundancy: if a syntactic clue were missing, then perhaps a semantic one could serve in its place. At least that is a possible explanation of how we recognize the indirect object (a syntactic category, whose semantic equivalent is the recipient of an action). Take our previous example and alter it to consider how we distinguish the indirect object.

(1) The boy threw the dog a bone.
(2) The boy threw a bone to the dog.
(3) The boy threw to the dog a bone.
(4) The boy threw a bone the dog.
(5) The boy threw the dog to the ground.

The first two variations are the norm; the third sounds a little strange as one's expectations momentarily stumble, and one's reading slows (and so one compensates by thinking that it sounds a little bit "affected"). The fourth variation is not acceptable, and is so bad that it even sounds like the dog is being thrown to the bone. In the last variation one stumbles just slightly in the understanding, as one has to go back and contradict the expectation that the dog is the recipient when the dog is definitely now the thing being thrown. From the first two examples, we know that order is important: the indirect object may just be a way of eliminating the use of the preposition "to", provided that the indirect object comes immediately after the verb. To support this view, we seldom say or hear anything like the third example, and we simply do not accept the fourth. So we know that order is important, and even the fifth example cannot convince us that it is not sufficient: where there is some confusion, it is simply necessary to get more information and then adjust one's interpretation in the light of that information. We do read from left to right, but we also go back and reread difficult passages, or we do just when attention momentarily flags.

But it also seems that there is more than order involved here. The fourth example is terrible, because one just doesn't throw animate beings to inanimate objects. We are using the type of the noun to distinguish the indirect object from the direct object. It is a strong clue, and it is not one that we can ignore. We even try to mentally switch the types of the nouns in the fourth example before we give up on it entirely. The parser described in Chapter 5 does use both order and semantic type when it looks for the indirect object: it looks for the indirect object before it looks for the direct object, and it rejects those candidate noun phrases whose principle noun's type is inanimate.

6.3.3. Syntax and Semantics as Standalone Systems

While the combination of psychological observation and mechanistic construction does make a case for the kind of cooperation between parallel systems that we imagine, it certainly does not prove anything. Another approach to examining the possibility of parallel systems is to separate them in the course of describing them: when each stands alone, what does it look like? Inherent in that attempt is the question: can each stand alone (or are they like "form" and "matter" in Aristotle's philosophy: inseparable)? If we want to have two parallel systems, then it seems that each should be able to operate independently of the other. We have built cooperating mechanisms and we have described syntax and semantics as separate systems, using the mechanisms as our evidence. Now the effort is to determine if what stands behind each of those mechanisms could be used without the aid of the other, and if so, then what kind of a system, in each case, would we have?

6.3.3.1. Syntax by Itself

Previously we said that syntax governs words "as words": such words are tokens that can be divided into such classes as "noun", "verb", etc. These words do not have meaning so long as we only consider them within this syntactic description. If we are limited to a purely syntactic system we can see however that these tokens do have a possible use: they can be used as labels. If the concepts supplied by semantics are missing, then the tokens that normally reference those concepts can instead be directly attached to things. Absolute consistency is required of such a system: once a given label has been associated with a thing, then it must always be associated with that same thing.

Once we accept that such a system can be constructed, then we have to ask whether it can be constructed towards any worthwhile use. Of course labels are a legitimate use of words, and so we name everything we know, and where those things are products we expect those names to be absolutely consistent, and to insure this, producers of those products are allowed to claim the exclusive use of some of these labels as trademarks. Such a system works because it stops at the syntactic level, and because all of the tokens are just tokens, in a system whose only principles are consistency and exclusivity. Meaning does get attached to such a system, but there are no principles and no test for truth in the way that meaning is attached. Advertising exists because each advertiser is perfectly free to attach whatever meaning he wants to his use of these labels. Principles (a Code of some kind) that are applied work only after a fashion (once in awhile, inconsistently) because those principles belong to some other (ethical) system, and are always applied from out-

side the system. The persistence of meaning is also governed by a single principle that lies outside the system: the success of the product.

From our point of view, such a system is not very interesting: it is syntactically poor, allowing only the use of consistent concrete nouns as labels, and the use of exclusive adjectives as trademarks. Another more interesting use of this kind of labeling system does however lie close at hand: every computer user interface is such a labeling system. While other signs are often used in such systems (icons for instance), words are still the most popular form of label used by computer systems. Such a system has a much richer syntax: actions are also involved in this system and those actions are labeled. The user who chooses an item on a menu rightly expects the same action to be attached to that choice every time he makes it. (It is an interesting sidelight that the legal system of this country is now trying to determine whether the labels attached to these actions are nouns, and available to every one, or adjectives that can be claimed for exclusive use by their inventors. Of course, they are neither, but a legal system that is heavily based on precedent will probably take a long time to recognize this.)

A computer interface is close enough to a complete use of language to serve as an example of a syntactic system that can stand on its own. The principles involved in the successful creation of such systems recognize this: engineers tend to object if a different expression is used to represent the same thing in different places of a system. While this kind of consistency is poor practice in the art of English composition, it is just exactly the way a good labeling system should work. Meaning of course is a consideration in such systems, but it is not a full partner in the use of language. Words and icons are chosen to evoke meanings which are mnemonic, helping the user to better remember which labels go with which actions and displays. Often times this looks like meaning as we commonly experience it, but the degree of inflexibility in such associations signals that such a system is constructed to use what we are very familiar with in order to aid in the construction of what we are not so familiar with. The attachment of meaning to these labels is stronger than the previous example of advertising, but it is not a full partnership of a natural language. Here we look for more than the common usage of words in order to express meaning: we look for the kind of usage that approaches and sometimes reaches the level of cliche. The important thing is not that the user of such a system understand the system; the important thing is that he remembers the system.

The system we have described should convince us that a purely syntactic system can be constructed and used effectively. Still the syntax of such a system is not really comparable to the richness of a natural language. In the

interests of trying to achieve that kind of complexity we might go one step further: we might introduce a higher level of 'token' and call that level of organization a pattern. A pattern is made up of a recognizable sequence of tokens. Many such sequences can be defined. When we are done we might want to call such a system an Augmented Transition Network (ATN), or perhaps we will use look-ahead to define these patterns and we call what we produce a "look-ahead" (or a Wait and See) parser. Such a system looks very familiar: of course it is represented by the parser described in chapter 5. Such systems represent the first efforts made to write computer programs that recognize and manipulate natural language. Such a system is an imitation of a natural language, but is in reality composed of an underlying database query language (a simple set of patterns), and all of the patterns recognized by the surface parsing mechanism must be translatable into the database query patterns to have any usefulness. Both sets of patterns are entirely syntactic (in some systems there are intermediate patterns of translation: they are also entirely syntactic). For all the work that goes into them, such systems are not an improvement on the underlying database query language. The query language is the sole depository of meaning, and an effective interface would work at building a syntax that makes use of the kind of mnemonic meaning we described previously. If nothing else, however, the building of these early natural language processing systems did demonstrate that it is possible to build a purely syntactic system that does look like a natural language.

On the side of syntax, then, the answer to our original question is that it is possible to build systems that are entirely syntactic in nature. Such systems do however make use of semantics as an aid in constructing and maintaining a system: we, after all, are already very familiar with a system that employs both syntax and semantics in an effective way. The first system we described really abuses this familiarity and in the form of advertisement attaches semantics to a syntactic system in such a way that we can never rely on it to be truthful, or even to have any underlying recognition of what it means to be truthful. The second system, a computer interface, is a use of syntax that adopts semantics in a purely mnemonic form. There are many examples that show that this can be an effective and even an aesthetically pleasing way to communicate. There were, of course, other examples that were not so pleasing to anyone, and perhaps it was these that led to efforts to tack a simulation of natural language (in the form of pattern recognition) onto them in the hopes of making them more familiar to their users.

6.3.3.2. Semantics by Itself

A system that is syntactical uses words as labels, attaching these tokens directly to what they represent; meaning is added only as an afterthought, as an effort to make the labels 'stick'. A system that is entirely semantic should not have words available to it as tokens, but at least in the forms of representation we have used here, words are what the system is made of. As we indicated earlier, perhaps we can think of a semantic use of words as a more ghostly apparition of words, or of words as "weightless". The important thing is that we do not have to be concerned with how such words operate as words; all we have now is a representation of meaning. Such a system has no way to introduce new meanings: if we wish to do so, then we must operate on the system from outside. In the example of the mechanism described in chapter 5, we can introduce new classes, and roles, or we can redefine those that are currently employed, but to the system itself, such changes will appear to be entirely capricious and unpredictable.

A purely semantic system must then be a purely dogmatic system. It can have no internal explanation of how it came to be, and it can have no means of accepting change. Even if the system were to be changed on a frequent and a regular basis, it still would have no way of recognizing that such changes have taken place, and are likely to go on taking place. From inside, the assertion must always be that things are as they have always been, and always will be. If we were to experience such a system from the inside, then we would probably object to any changes that were drastic enough to be recognized as changes. Merely to recognize that such changes are possible would be to jeopardize the system.

Certainly a notorious example of such a closed system was the world view confronted by Galileo when he looked through his telescope and confirmed, and sought to promulgate, a new, Copernican view of the universe. His enemies did not try to contradict the accuracy of his observations or the likelihood of his theory, but rather sought to maintain what they viewed as a complete and harmonious system of belief founded upon the philosophy of Aristotle. Maintaining a system of belief was considered more important than any new fact about the world. Of even more consequence, those who felt it was their duty to maintain this system of belief also understood that the introduction of any new fact not accounted for by the system would completely overthrow the system. This was the source of their fear, and of their authoritarian reaction: it seemed to them that the right thing to do was to suppress such a threat by whatever means were available.

Today we are a lot less likely to make the same kind of mistake as did the persecutors of Galileo. We have seen many such systems come and go, and we have a better idea of why we create these systems, and of what they are good for. Calling them models, rather than religions, we are less likely to stake our sense of ourselves on them, and are more likely to recognize that they are useful explanations of a portion of experience. As such, their life expectancy is limited to their usefulness, and their replacement is surely to come in the form of some new model deemed to be more useful as a description of the world and of our experience of it. On a personal basis we may each suffer some of the fears that characterized the religious leaders of Galileo's time, but where we gather together and discuss shared theories and models of the world, we are much less likely to countenance anything but a reasoned and a considered acceptance or rejection of the latest effort to understand our common experience. (This should probably be described as a tendency; there are plenty of examples, in the modern world, of the opposite and self-defeating tendency described in the previous paragraph.)

What all of these purely semantic systems have in common is that they are closed to new ideas because they do not have access to the stuff from which they were created: they do not have access to words as words. The compensation for this is that very sophisticated representations of meaning are possible: such systems are tested against the world over a prolonged period of time, and the kind of simple-minded attaching of labels to things that a syntactic system can achieve will not suffice here. No matter how impressive the achievement, however, any purely semantic system or model will eventually have to be replaced. Over widely differing situations the experience of replacement is essentially the same: from the sublime to the relatively trivial, the only real difference in the process is the amount of pain that is experienced by those who must give up one well known model for another that is not so well known. The same process governs the replacement of religions and scientific theories as governs the release of new versions of computer software; however much we might object (we have so much belief at stake in the former, and only a little memory at stake in the latter), that they are not at all the same, still they are the same. (When we consider language, then everything that is human falls within the view of that consideration, and everything rests on the same plane.)

Natural language understanding systems that favor the semantic side of language have been previously described in chapter 2. For the most part these are story understanding systems whose principle area of concern is the need to make inferences. Their avowed attempt is to understand natural language as it is used by human beings. In most respects they are radically different from the syntax-oriented systems that serve as front ends to database query languages. Not surprisingly, the semantic systems are unconcerned with syntax, usually adopting their own methods of pattern

recognition and translation into formal representations of meaning that serve the semantic limits of the system or model. This kind of system is characterized by its heavy dependence on a model (what we have called 'world knowledge' in chapter 2 and elsewhere).

Parallels of advertising and dogmatism, mnemonic systems and models, and lastly of syntactically oriented and semantically oriented natural language understanding systems have helped us to highlight these two aspects of language: syntax and semantics. Emphasizing each in turn we have found that there are different ways that the two can go together and there are arrangements that are most likely derived from natural language and our familiarity with it as a system for carrying meaning. Hopefully we have been able to demonstrate that each (syntax and semantics) has an independent character of its own and that there are actual systems of representation and meaning which are built out of a strong emphasis on one or the other. None of these systems however represents the kind of partnership between syntax and semantics that is achieved by a natural language. Furthermore it is our experience of natural language that shores up the more artificial supports of the systems we have attempted to describe.

Now that we have achieved some sense of syntax and semantics as separate systems, we would like to make some effort to describe how they work together in a natural language. However, a theory of language is, to the say the least, beyond the scope of this study. What we seek here is to understand the partnership between syntax and semantics in such a way as to help us to determine if it is possible to build a true natural language understanding system or system of mechanisms. If the system described in Chapter 5 is not adequate to the task, then what kind of a system might be adequate? More importantly, what is our motivation for attempting to build a mechanism for understanding natural language? The approach we take to answering these questions must be very general, only using the abstractions we have managed to put forward so far. A complete description of a syntactic system and a complete description of a semantic system are again considerably beyond the scope of this study. However, we can begin to get a glimpse of what is involved if we stay with the abstractions we have managed to put forward, and we ask ourselves how it is that these models (semantics) might make themselves available to a syntactic system, and how a syntactic system might make itself available to a collection of models? What we are interested in is the orientation of one to the other. That relationship must be within the scope of our study. However limited our representation of syntax and semantics, we must arrive at the correct relationship between the two or we have nothing.

6.4. Syntax and Semantics as Cooperating Systems

6.4.1. Joining Syntax and Semantics by Instantiation

The system described in Chapter 5 attempts to join syntax (where names are words) and semantics (where names are categories or classes) by the process of instantiation. Syntactically discovered and represented elements (words) are recognized as instances of the classes that are defined in the semantic network. Any particular noun has two definitions. It is defined as a syntactic element by its inclusion in a syntactic class (it is a noun), and it is defined semantically by its inclusion in the network of classes that is one of the components of the semantic mechanism. Words as words are tied to meanings rather than being used as labels. The system of meaning itself can be as sophisticated and as complex as we need it to be. Superficially it appears as though the kind of parallel cooperation between syntax and semantics that we seek has been represented. Certainly the use of instantiation is a time honored way to represent knowledge, one used in classic systems of logic, and one heavily used in computer implemented knowledge-based systems.

If we look closer at this partnership, however, we soon discover that it is not much of a partnership. Almost all of the system belongs to the semantic side, where all of the recognizable meanings have been defined ahead of any statement. What is said or written is important only for its quality of being said at a particular moment. If it is recognized as a meaningful statement then it is one of those combinations already present in the semantic network. Further, the system has the ability to infer any meaning that it can recognize (by using the network of classes as default values for the roles that it recognizes). Such a system so strongly favors the semantic side of the partnership that we must wonder why we would bother to say anything at all when everything we can say can just as easily be inferred by the system.

Such a system is only useful to a naive user, one who does not already know the knowledge represented in that system; yet such a user will have the most trouble formulating statements (which must always be instantiations of classes contained in the knowledge base) which have meaning within the system. The better the user learns the system, the more likely his statements or questions will have a meaning to the system, and the less useful he will find it to be. It is this inherent contradiction that undermines this kind of a semantically based natural language understanding system. The problem we confront here is not one of completeness, because no use of natural language, by humans or by mechanisms, ever achieves anything approaching completeness. Perhaps the inclusion of more and more world knowledge in bigger and bigger systems or models can hide this inherent contradiction for awhile, but the basic contradiction will remain, and the usefulness of such systems will not justify the kind of effort it will take to put a natural language interface between

them and a human user. While instantiation expresses a well-ordered, "classical" approach to the world, it does not represent the full partnership between syntax and semantics that characterizes our day to day use of natural language. The relationships represented by this kind of a system are confined to a static and closed system. Natural language is instead a dynamic and an open system.

6.4.2. Variety of Expression Is Infinite

Natural language is not the best interface between a naive user and a knowledge base that requires instantiations. If a knowledge base models experience or knowledge in terms of classes, then the only statements that can be interpreted by the system are ones which provide it with instantiations of those classes. Yet the person who seeks to access that knowledge (because he does not possess it himself) is the least likely person to supply the system with such instantiations. The best interface to such a system is to provide the user with instantiations among which he must choose. Some kind of a formal, restricted language would best serve an instantiation system: the kind of dogmatic system that we described for a purely semantic system would seem to be the best and most effective means of communication between a naive user and a knowledge base or model of experience. Natural language is not a good interface to such a system because natural language allows each user to create his own expressions. Natural language is for experienced users, and it fits an activity that is more sophisticated than merely accessing knowledge.

This kind of argument is a little too specific to make our case here, but it is presented because the kind of system described in Chapter 5 is just such an instantiation based system. Even though the system described in Chapter 5 started off attempting to deal with natural language in terms of general patterns that are predominantly syntactic, still when it had to represent those patterns as meaningful, then the system immediately became what we have described above, and the contradiction we have described became apparent. Still, the attempt to provide a natural language understanding system is also present in the implementation described in chapter 5, and we can use that attempt to make a more general argument against this kind of a system as an effective use of natural language. From that description we may be able to derive a better understanding of what natural language is.

One of the major problems that any natural language understanding system must deal with is that a natural language provides many ways to say what amounts to the same thing. This problem must be addressed by some kind of parsing mechanism that is able to recognize these patterns and to produce ordered representations where what amounts to

the same thing is always represented in the same way. One way to implement this strategy is to do what we described in Chapter 5; a general syntactic parser handles various English patterns and produces a formal representation which can then be interpreted by a case frame system that can compare these ordered representations to the meanings it contains. At the point of comparison a conversion takes place, and a case frame is produced: one case frame may result from a number of different syntactic patterns as we saw in some of the examples used in this and in the preceding two chapters. The other general approach to this problem is to build a pattern recognition system that makes conversions based on recognizable semantic patterns. The systems that were described in Chapter 2 tend to employ that kind of method. We will work from the former method here simply because we have more experience with it. The argument we make can however be applied to either method.

A syntactic parser attempts to deal with variety in forms of expression by recognizing that there are thematic patterns throughout English which are used over and over again. The most common of these is found in the example that we have used elsewhere where the variety is based on the purely syntactic variation on the voice of the verb, and the arrangement of subject, object, and indirect object. Transformations based on voice are entirely syntactic, and if that were all we had to deal with, then we could believe that there is some kind of "deep structure" to English. However, any attempt to build a parser soon finds that the kind of predictable variety provided by the voice of the verb is only the beginning of variety when it comes to using the English language. The truth is that there are many, many ways to say the same thing in English, and that this variety does not have any discernible regularity to it. Voice probably only appears to us first because it presents us with a syntactic variety that is generally present throughout English.

If we look back at some of the phenomena we encountered in our description of the parser we built here, then we can see that there are other kinds of syntactic variety that we encountered and we sometimes had to invent methods to deal with that variety. The phenomenon of finding the subject of an infinitive phrase was one such experience; another was the need to deal with a dummy subject. If we took more time to look, we could find many other syntactic patterns not currently represented by the parser.

Once we arrive at the level of representing meaning, then we can see that variety of expression can have many sources. The syntactic variety based on voice is only one of many sources. The sad (or happy) truth is that if I take any particular expression (say "I advise you to study for the test.") I can find an infinite number of different ways of saying the same thing. At least I can be pretty sure that when I tire of the enterprise, there will still be other varieties that I

have not mentioned. Below are a few quickly constructed examples:

(1) Study for the test.
(2) You should study for the test.
(3) My advice is that you study for the test.
(4) I'm advising you to study for the test.
(5) There's a test coming up: you should start studying now.
(6) If you want my advice: you should study for that test.
(7) This time, why not try studying for the test?

The first two varieties above are based on syntax and are handled by the context interpreter described in chapter 7. The third and fourth varieties are based on one kind of speech act that this system can recognize: they simply make the performative verb ("to advise") explicit. The fifth version makes use of what could be considered to be inferable details about the time frame involved. Versions six and seven are what might be considered as indirect speech acts, requiring some kind of context interpretation and correction in order to recognize them as equivalent to the others.

Many of the patterns we find above go beyond the normal syntactic categories and include details that would be difficult to describe in any general way. Perhaps that is why the kind of systems described in Chapter 2 have never had much use for the recognition of general syntactic patterns. Those systems try to handle this problem by building pattern recognition systems that recognize various ways of saying what belongs to their formal language (or model) and converting those patterns into the formal language of the model. Because those systems begin by establishing a domain or model of the world, they tend to recognize the scope of the problem very early and so they are less likely to try to take any shortcuts into general syntactic pattern recognition. At the same time, such systems carry no illusion of generality, and are immediately recognizable in the terms of the kind of contradiction we described previously, where a naive user attempts to address an instantiation based model of the world.

When we leave the investigation of known general syntactic patterns and begin to enter the area where we must convert patterns of expression into patterns of meaning then we find that the variety of expression increases and becomes less amenable to general rules: exceptions begin to proliferate. This phenomenon really should not surprise us if we understand how it is that more than one expression can be said to have the same meaning: that is, even to say that this is possible does not begin to make any real sense until we we try to deal with both the syntactic ordering of words in expressions, and the semantic representation of meaning in a model. When we say that two expressions have the same meaning, we actually can only mean that the model we are employing cannot distinguish between these statements. At the same time we should realize that no two expressions that are formed differently truly have exactly the same meaning.

Meaning is the same for each only in the interpretation provided by a particular model. Such a model saves the common part of each statement: that is what we mean when we say that two statements have the same meaning.

The other very necessary side to this relationship is that different models will provide a different meaning for one and the same statement or expression. We commonly refer to this as ambiguity. In our everyday experience, we seldom ever deal with only one model of the world at a time. Ambiguity is an important part of the way in which language allows people to communicate. Ambiguity is pervasive to a real use of natural language, where many models may be employed at once. Often times the best productions of a natural language are the most ambiguous.

Variety of expression and ambiguity are not inconvenient features of natural language: rather, they are the way that language works. Knowledge based systems that attempt to employ natural language attempt to control these aspects of language, and attempt to reduce them to something which can be handled by a formal model. Instead, both of these aspects of language should be pursued and encouraged to proliferate: expansion gives the opportunity to discover new meanings, which is ultimately the only chance to have meanings at all. Natural language is best used among relatively equal understandings. Variety of expression and ambiguity allow us to build models together and to learn about the world in the process. Natural language is a tool for building models of the world. When syntax (ordered expressions) and semantics (models) truly go together as equal partners in a natural language, then the only thing that arrangement is good for is for building models. For other activities, a different arrangement between syntax and semantics (such as we have described previously) will take over and some other kind of language will come to be used. These languages are poor imitations of the real thing: just because words appear together in patterns, and just because meaning gets attached in some fashion, it does not follow that there is a natural language. Natural language is used only for building models.

6.4.3. Language Builds Models

Instantiation is an attempt to tie the here and now to a previously existing model; instantiation however does not allow for the creation of new models, and the real function of natural language is to operate as a creator or builder of models. Instantiation is a way of delivering knowledge to a naive client, but natural language is too powerful a tool to do the same job: users of natural language must be relatively proficient in the task of building models in order to use natural language effectively. For our previous look at syntax and semantics as separate systems we have seen that it is possible to construct artificial languages which can better serve specialized purposes. Delivering a model to a naive user

is better done by presenting that user with a formal language as a description of the model.

Anyone who tries to make use of a model of the world or of a portion of the world must achieve a great deal of familiarity with the pattern of organization and the limitations of that particular model. This is the traditional way of making knowledge available to those who need it or who want it. Today however there are so many specialized models that we would like to make them available to anyone who needs them without incurring the traditional learning time associated with such activity. Thus we try to compile the knowledge of persons who are expert in the use of such models, and we try to make that compiled knowledge available to anyone who might need to access a portion or all of that knowledge. Creating an effective means of accessing an 'expert' system is a difficult task; there is the temptation to believe that natural language, since it is a familiar part of the life of all concerned, would represent the perfect interface for such systems. Here however we are saying that such a belief is just exactly the opposite of the truth. Natural language is the tool that allows us to build such systems of belief (models) and by building such systems enables us to learn about the world. In order to do this, natural language must be as dynamic and changing as is the unmodeled world. As such it supplies us with an infinite variety of expression; such a system is suitable for those who want to learn a system of belief (and who are willing to incur the loss of time that this takes), but natural language only compounds the problems of making an already constructed model available to a naive user. That paradigm is far better served by a severely restricted set of acceptable expressions. The most naive user will then be forced to choose among expressions which have meaning to the system.

This situation or paradigm would not be of interest to us except for the belief put forward here that all instantiation based systems are knowledge based systems, and all knowledge based systems arrive eventually at this need to interface with a naive user. All of the natural language systems we have described are knowledge-based systems. The contradiction involved in building such systems cannot be overcome by any method of expansion of either the knowledge represented, or the variety of expression that is recognized. The history of research in this area has one and only one scenario (repeated again and again). This scenario presents a relatively successful effort to deal with a limited domain (a toy system) with absolutely no chance of ever extending the methods employed to any real world use of language as we use it. If we understand the contradiction involved then we should have no trouble understanding why this scenario holds and will always hold for this kind of research effort. Natural language is not a tool for discussing or describing a model of the world. Each such model comes with its own formal language that is the only way to describe it. Natural language is the one tool that we have for building these models. All of us who use natural language use it to build such

models of the world. We communicate with one another by engaging in this model building activity.

6.4.4. Natural Language is the Only Model of Natural Language

Natural language is a means of creating systems of belief. It may even be that this is our single evolutionary advantage: a language that allows us to generate models of experience. The motto "everything has a name" perfectly describes a human, language-using, model-building, experience of the world: if we are human, then by the time we are aware of something we have already given it a name. (In a model, a "name" indicates inclusion in a class.) First comes the hypothesis, then the experience or the test. We can only get away with this by doing it together, communally. For such activity to be successful, the motto that describes the generation of labels must be balanced by a caveat that governs their use: just because we have a name for something does not mean that it necessarily exists. Between these two we manage to eke out a shared experience of the world: it is characteristic of language users that they must share their experience in order to have it at all.

For our purposes here, the germane question is the one that begins with the observation that it is language that creates models. That being so, can we expect to create any model that describes language? The twentieth century has conditioned us to immediately grab the logical answer and say 'no, it is not possible'. After all, we know that there are no complete systems, and no progression through metalevels can help us to create one. However, logic belongs to the caveat we described above: it is the business of logic to weed out and to discard the names that do not correspond to things. This is not the whole activity: logic is a feature of language but it is not the whole thing. We also have a way of generating the names: this method cannot itself be logical, but must precede logic, and therefore cannot follow the rules of logic. Everything has a name because we generate the names first. This means that we can have all the names that we like, and that is the only rule that governs this side of the activity that is language.

We can use language to describe language. There is nothing in the nature of language to prevent this. This should encourage us to try; we are however immediately confronted by a second question: we would like to know whether or not our description of language is correct. We still need logic to discard the bogus names! What shall we do? There is only one possibility for proceeding here: we must create natural language all over again, and then we can examine what we have created and we can logically and scientifically determine if what we have created is the same thing as what we already have. The only model of language is language itself. When we attempt to create a mechanism that understands

natural language, we are attempting to do just that, but we are so far away from the actual accomplishment, that there is a strong psychological injunction against saying just what it is we are really trying to do. Nothing in the systems we use can tell us that we even have the right tools yet. We only know that the prospect is an interesting one. It is interesting because there is a third question that waits for us to successfully build again what we already have: will we gain any benefit from doing so? If we accept that there is an evolutionary advantage to using language, will there be an even greater advantage in knowing what it is we are using?

At the point of posing this last question, we can speculate only in the terms provided by our current understanding of evolution. As we currently understand evolution, changes occur by happenstance and then are preserved by the lucky circumstance that they confer upon their possessors some advantage in the struggle for subsistence. If language can be said to confer an evolutionary advantage, then we must abide by the terms by which we currently understand the term 'evolutionary advantage'. On those terms we do not know the effect of any change until we have experienced the change, and usually this means 'until we have experienced the change for a very long time'. The model building, logic-based, activity that language provides us with attempts to overcome this situation by providing us with methods for testing reality ahead of time. Language, however, is not encompassed by any such activity; rather it is language that makes that kind of activity possible. In order to describe this greater system we must find ourselves outside the circle of logic and science. Whatever language is, it exists in the world under the same conditions as any other evolved behavior. If logic is involved in the discarding of bogus names, then we must move outside of that inner activity, and we must include the capacity for generating names in any description of language. It is the capacity itself which must be included.

6.4.5. Natural Language Is a Method for Learning about the World

We all begin our educations by learning a natural language. Later we learn to understand and to create specialized models. The same order of education produces scientists as well as poets: all find a use for this one tool, a natural language, that allows the building of models. None of its users ever know, ahead of time, what that use will be.

Most of us never do much more than participate in the common usage of words, meanings, and models which is usually referred to as "common sense" reasoning. But in truth that is almost as much as anyone does. Language as a model building activity looks like the ground we walk on: we wonder about it as often as we wonder about gravity.

It would be useful to build a natural language understanding system in order to show ourselves what it is we do when we use language. This means creating a system that uses language as we do. Such a system must be a system for learning about the world, in partnership with other such users of language. Any user of natural language participates in the building of models of the world. In that process new words become old words; this is not the same as instantiation where all new words have already been established as old words. Any knowledge base that goes with such a system will be created through the use of language.

As we look around we see that all those who use natural language, first had to learn natural language. Is it possible to build a learning-based system? All our abstractions will be different.

7. Context Building: Representing Practical Arguments

Chapter 5 described the processing of individual sentences. Chapter 7 will now describe how those individual sentences are collected and organized into a representation of the overall context built through the course of a dialogue. The resulting representation of context is the literal meaning of the input text. The kind of inference that we are concerned with here is the simple filling in of missing elements of text which can be readily inferred from an understanding of the nature and form of practical arguments.

In this chapter we are primarily concerned with a practical argument frame as our basic data structure; we are also concerned with the process by which we can fill those practical argument frames, and other auxiliary frames, with both explicitly entered and inferred values. Throughout this chapter we will use the following excerpt from the dialogue presented in Chapter 1 to illustrate these forms and methods:

Input: A test is scheduled for Monday. (If you study for the test, then you will pass the test.) You should study for the test.
Output: The speaker has advised the hearer to study for a test.
Input: Why?
Output: The speaker believes that the hearer will pass the test if the hearer studies for the test.

Note: The above section of dialogue has been simplified from the presentation in Chapter 1; this has been done in order to simplify parsing and language generation issues somewhat. These issues have been limited to what needs to be present in order to fill a practical argument frame with its basic propositional content. In the first input section the parenthetical material may be included or omitted. The system will infer this material if it is omitted.

7.1. The General Processing Method

The Context Builder begins in a bottom-up or data driven fashion. Each sentence frame (representing an independent clause) is examined immediately after the parser has finished it. This examination involves determining the kind of role that this sentence is likely to play in the overall context. Such an examination is limited to a recognition of the literal meaning of the sentence, and only examines one or more key elements in the identification of that literal meaning. Input consists of the sentence frame, which includes a case frame interpretation, as represented in Diagram 2, appearing at the end of chapter 5.

In most cases, it is enough to determine the sentence type to complete this aspect of the interpretation. In our example, the clause "if you study for the test", is recognized as a "conditional" (or the "if" part of a hypothetical declarative sentence). In a similar manner, the clause, "then you will pass the test". is recognized as a consequent (the "then" part of a hypothetical declarative sentence). Other sentence types recognized by the interpreter are a simple declarative sentence, such as "a test has been scheduled for Monday", and an imperative sentence, such as "study for the test". Conversions may also take place, so that a simple declarative sentence, such as "you should study for the test", is treated as an imperative when that is appropriate. For the most part, the parser has given each independent clause the designation that it would receive as a part of a normal syntactic analysis of the clause or sentence. In most cases, this is enough for the initial phase of the interpretation process. Where this indication is not sufficient, some very simple further syntactic and semantic checks are added.

The sentence type, then, serves as the bridge between the parsing and semantic interpretation of individual sentences on the one side, and the process of building a context that is composed of such individual elements, on the other side. Normal syntactic categories, such as "simple declarative sentence", are for the most part sufficient for our needs here; where they are not sufficient, simple additions and transformations are added to the analysis. Thus the type of a sentence serves both as a well recognized syntactic category and as an initial designation of the sentence's role in the specialized system we are creating here. Since we are attempting to manipulate whole elements we can expect that the names we give to those elements are going to give us the first indications of how we should handle those elements in relation to the rest of the system. It is, however, encouraging that we can use very common names taken from the much broader system of syntactic recognition of the language as the names that are appropriate for our specialized system. This means that we build on a solid foundation, one which we can expect to find in any area of natural language understanding where this system might be applied or incorporated.

The type of a sentence corresponds to the role it plays in the context, and the context is primarily described by the structure of a practical argument frame. (That structure is described in the next section.) After the context interpreter has determined the type of sentence, and its role or slot in the practical argument frame, it must then determine the location of a particular sentence frame. This means that it has to know what argument frame to put the sentence frame into. This requires some semantic interpretation. This checking is done by rules and procedures that lie outside the practical argument structure itself. In the next section we will see that the relations between slots in a practical argument frame are defined by procedures (called demons) that are attached to its individual slots. These demons handle the internal

relations of the frame, and fire in reaction to the placing of a value in the slot that contains them. We do not, however, want anything to happen at that level until we are sure that a particular value (a frame name) does indeed belong to a particular frame. This type of check is done, then, outside of the frame structure itself.

In summary, the process of interpretation at the context level is made up of three stages. The first stage determines the role of a sentence in the overall context. Here a sentence is determined to go into an antecedent or a consequent, or some other slot in the context that is defined by a practical argument frame. The second stage involves finding out which practical argument frame should take which antecedent, consequent, etc. Here we want to know, for instance, that the plan derived from an antecedent is one which can indeed achieve the goal put forward by the consequent. The final stage involves filling out the information that a practical argument frame can carry once it has sufficient input. For example, the process of determining the kind of speech act that a particular practical argument frame represents is handled by procedures defined within the frame. These procedures are designed to fire when the frame has enough information to determine its speech act.

The next section will describe a practical argument frame in detail, and will describe this three stage process for each slot in that frame's structure.

7.2. Practical Argument Frames

A practical argument, as described in section 2.1, is the principal structure we will use to represent meaning in the course of a dialogue. A practical argument specifies an action to be taken, and provides reasons for taking that action. These reasons may be either explicitly provided in the text of the dialogue, or they may be implicit in that dialogue so that an inferencing mechanism will be needed in order to supply them. Also, the practical arguments that we will use are all able to be labeled as specific speech acts. This ability to deduce the speech act involved for each such argument will allow the system to represent the relationship between the speaker's intent and the hearer's interests.

A frame structure will be used to facilitate all three of these activities (collecting, inferencing, and naming). This practical argument frame is composed of the following slots:

event: knowledge about an event which the practical argument references
antecedent: the conditional part of the argument (the action)
consequent: the result of the condition (also, the reason for the taking the action)
imperative: the action suggested by the speaker
speech_act: the kind of speech act that this frame represents (advice or warning)

status: internal information - is the frame complete?

Given the following input:

> A test is scheduled for Monday. If you study for the test, then you will pass the test. You should study for the test.

the following collection will be made:

```
argument1:
        instance_of:
                value: argument
        event:
                value: event1 {A test is scheduled for Monday}
        antecedent:
                value: sent2 {If you study for the test}
        consequent:
                value: sent3 {then you will pass the test}
        imperative:
                value: sent4 {You should study for the test}
        speech_act:
                value: advice
        status:
                value: complete
```

If we look back at diagram 2 (appearing at the end of chapter 5), then we can see that a single sentence or clause requires a complex set of representations. The major slots (event, antecedent, consequent, and imperative) in the practical argument frame that represents the above collection actually hold the names of frames that represent the sentences and clauses listed above. We can see then that the practical argument frame is a collection of a wealth of representational data that has been built up through successive processing levels. While we are able to refer to these slots in terms of their original English language input, (and will do so), we should remember that what the slot actually holds is a name that serves as a reference by which we can access a very complex representation of that meaning. The advantage of referring directly to the English language input is that we are able to push the representational details into the background in favor of the language itself, with which we are all readily familiar. The disadvantage is that we take for granted a complex formal representation, one which may or may not conform to the unconscious interpretations and representations that we supply to English language text.

The following subsections will describe how each slot is filled and represented in the system.

7.2.1. Events

In a practical argument frame, an event slot holds the name of a frame which represents an event occurring in the real world. In our example, the event is 'the test that has been scheduled for Monday'. A fully processed event frame for this example contains the following information:

```
event1:
        instance_of:
                value: event
        event_description:
                value: sent1 {A test is scheduled for Monday.}
                if_added: infer_goal
        status:
                value: has_goal
        goal:
                value: [pass, some_person, test1]
```

Event frames are processed independently of practical argument frames, and are only placed in the practical argument frame if they contain a goal which matches the goal expressed in the consequent of a practical argument (see the section below that describes the processing of a consequent).

Creation of an event frame occurs when the context interpreter recognizes that a sentence describes an event occurring in the outside world. Placing the sentence frame which describes an event into the event frame causes a demon to fire; this demon looks for a knowledge-base goal whose object matches the object referenced by the event. If a knowledge-base goal is found, a goal is instantiated and placed in the event frame. The instantiated goal is composed of the action contained in the knowledge base goal, together with an agent (the recipient of the event) and an object contained in the sentence describing the event. This inference of a goal causes the goal to become active and a part of the current context.

A goal activated in this way is different from a knowledge based goal in that it contains a specific object and a specific agent. The degree of specificity depends upon the sentence in the event frame. In this example, the object is a test, referenced as "test1" (a frame) and the agent is a default value, "some_person". The inferred goal says that an unknown person has the goal of passing a real test. The knowledge base goal only says that the person who takes a test has the goal of passing that test.

Goals are expressed in the knowledge base as a Prolog fact with four arguments. The first argument represents an action; the second represents an agent; the third argument represents the object of the action; the fourth argument

expresses the relationship between the agent and the entire expression of a goal, saying that the agent is the person who has this goal.

goal(pass,person,test,has_goal(person)).

This last argument distinguishes a goal from a belief; in a belief, the person having the goal and the agent are necessarily different. (One could argue that this is not always the case if one wishes to express the condition of the person having a goal also having a belief referencing the same goal. Such expressions, however, seem to be redundant since it would be difficult to imagine a case where a person who has a goal does not also possess a belief that he has that goal.)

An event frame does not belong to the logical relations expressed by a practical argument. The system for making inferences does not draw any of its expectations from the presence of events or from the language describing those events. For example, if the antecedent and consequent are missing from a practical argument, then they are supplied inferentially based on information that is present in the imperative. The example we are using might be given in its short form:

A test is scheduled for Monday. You should study for the test.

In this case the antecedent ('if you study for the test'), and the consequent ('then you will pass the test') would be drawn from the imperative ('you should study for the test') and from the representation of plans and goals that is present in the knowledge base. (See the section describing imperatives, below, for a more detailed description of how this is done.)

When an event has been fully processed, it is put onto a stack of events. Each practical argument frame that is formed will examine that stack to see if any event refers to its propositional content. If there is such an event, then it will be placed in the event slot of the practical argument frame and its information can be used in the course of determining the soundness of an argument (where the issues involved are not of a logical nature). A practical argument frame does not need to contain an event to be considered complete.

Even though an event does not belong to the logical relations expressed by a practical argument, it still has enough significance to that argument to justify putting it in the practical argument frame. The event contains information which will be needed when attempting to determine the soundness of the argument. For instance, the plan of action put forward by a particular argument may not be sound for other than logical reasons. The plan may require time that is not available to the person expected to carry out that plan. We can expect the event to carry information that will help the system to detect such impediments to the successful execution of a plan.

7.2.2. Antecedents

An antecedent is derived from the conditional part of a hypothetical declarative sentence or it is inferred from the presence of an imperative. In our example, the antecedent is the clause, "if you study for the test". An antecedent can also be said to represent a plan for achieving a goal. The goal, in turn, is expressed by the consequent. The plan put forward by the antecedent must be recognized as a legitimate method for achieving the goal represented by the consequent. From these conditions, then, the processing method for determining that a given clause is the antecedent in a given practical argument is to ascertain that the clause is the conditional part of a hypothetical declarative sentence and that its propositional content can be interpreted as a plan for achieving a goal defined in the consequent of the same practical argument. The plan put forward by the antecedent must also be semantically equivalent with the plan represented by the imperative in that practical argument frame.

It should be noted that in the course of processing input, we have to begin somewhere, and in the interests of beginning somewhere, a plan matches a goal if the goal is unknown, and likewise it is semantically equivalent with any other plan that is unknown. This means that an empty slot matches anything that has content. If the antecedent slot is to be filled first (because the input happens to be arbitrarily ordered in that way) then it is filled, and matches both the consequent slot which is empty and the imperative slot which is also empty. When there is input that matches the requirements for a consequent, then that input is matched to the already present antecedent. If they are found to match, then the consequent will be placed in that frame. If they do not match, then another frame will be tried, and if none is found then a new frame will be created, and the consequent will be placed there. The processing method is to always handle the immediate case as fully as possible where that case is a part of a logical argument (and hence necessary for completeness). Only the event frame, which is not a part of the logical argument, is deferred until later. We will see that this process is carried one step further when we process the imperative (see below).

There are no demons attached to the antecedent slot of a practical argument frame. Once it is determined that an antecedent belongs to a given practical argument frame, processing is complete. (In the section describing how the imperative slot is processed, we will describe how an antecedent may be inferred and placed in a practical argument frame if it is not present explicitly in the dialogue.)

7.2.3. Consequents

In a practical argument frame, the consequent slot holds the name of a frame that represents the consequent or "then" part of a hypothetical declarative sentence. If the antecedent represents an action, the consequent represents the result of that action. If we can also say that an antecedent represents a plan, then we can also extend the idea of a consequent to represent a goal. In our example, the consequent is the clause, "then you will pass the test". The consequent is the partner of the antecedent, and as such, its processing method is the complement of the antecedent's. That is, in a particular practical argument frame, the consequent must represent a goal that can be achieved by the plan present in that practical argument frame's antecedent. The steps involved in filling the consequent slot mirror those involved in filling the antecedent slot. The first step is to recognize that a given clause is the consequent part of a hypothetical declarative sentence, and the second step is to determine that the propositional content of that clause does represent a goal that can be achieved by the plan represented by the antecedent in the same practical argument frame. Since the consequent may be stated before the antecedent, any consequent will match an empty (as yet to be determined) antecedent.

While there is symmetry in the process of filling an antecedent and a consequent slot, there is asymmetry in what happens immediately after the slot is filled. The antecedent slot has no demons attached to it, but its processing method does ask for a match beyond that with the consequent; it should represent a plan that is semantically consistent with the plan represented by the imperative. As we shall see in more detail in the next section, the imperative is the focus for determining the speech act involved, and is the center of the inferencing process that defines a practical argument frame. The imperative then represents the relation between the practical argument frame and the language side of representing the context of a dialogue; the antecedent is tied very closely to the imperative. This is one way of saying that we use language to formulate plans. The consequent does not require a match with the imperative (this is already taken care of by the relation between the imperative and the antecedent). It does, however, have a demon attached to it. Whenever a consequent slot is filled, its internal demon looks for an event that can be used to fill the event slot of a practical argument frame. If an event is found that has a goal attached to it, and that goal is semantically equivalent to the goal represented by the consequent, then that event frame fills the event slot in the practical argument frame. Thus the consequent has its direct tie to the real world of events. This is another way of saying that the world gives us our goals.

Once a practical argument frame contains a consequent, then a goal can be associated with that frame, and the system can look for an event to go along with that argument frame. In order for an event frame to become a part of a prac-

tical argument frame, the goal that has been created for the event must match the goal that has been created from the consequent of the practical argument. In our example, the event is described by the sentence, "a test is scheduled for Monday"; the consequent is expressed as, "you will pass the test". The goal generated for this event is:

[pass,some_person,test1,has_goal(some_person)).

The goal generated for the consequent is:

(pass,you2,test3,has_goal(you2)).

The goals match if their predicates ("pass" in this case) are the same, and if the agents ("some_person" and "you2" here) can be unified, and if the objects or objectives ("test1" and "test3") can also be unified. These noun instances can be unified if one is a more general type of the other. Since the pronoun "you" is of the type "some_person" the unification takes place. Here, the token "you2" is an instance of "some_person"; that is, this use of the pronoun refers to a person, even though that person may only be hypothetical (it does not matter to the validity of the argument whether or not a real person is referenced by this pronoun).

The situation is more complicated with "test1" and "test3". Here we are dealing with two instances of the same type. The potential for unification is there, but determining that this unification should take place is a more difficult issue. In the course of parsing and processing the original input, we have to treat the sentences purely as language; this means that we need to create a separate token for each word. Where words are the same, we need to generate a unique representation for each noun and pronoun. We do this by incrementing a counter for each noun and pronoun that shows up in the course of the dialogue. That number is appended to the noun in question and this concatenation becomes the token representing just that one occurrence of that noun. Now that it is time to make sense of what we have processed, we need to determine whether or not these unique tokens reference the same entity.

We will not pursue this further beyond noting the general principles that are involved. Tests have to be constructed for each area of knowledge that the system possesses. Everyone of those tests has to be checked to some extent. Each such test must go far enough to determine if there is a difference. If all of the tests fail to demonstrate a difference, then the tokens are the same. We can never prove that the tokens refer to the same concept or thing. We can only assume an identical reference if we fail to demonstrate that they are different.

7.2.4. Imperatives

The imperative slot is the most important of the slots in a practical argument frame. In the examples that we use here, it represents the focus of the relationship between the speaker and the hearer. It puts forward an action which the speaker believes the hearer should perform (or should avoid performing). The imperative is the focus of the inference mechanisms present at the context level of the dialogue. The rest of the practical argument frame is largely an effort to represent the full meaning of the imperative. Filling the imperative slot involves the same three step process as described above for the other slots, but more is involved in the last two steps of the process, and the final result of processing an imperative is a complete practical argument frame. As we shall see, the one word sentence, "Study", is enough to create and fill a practical argument frame.

In the first stage of the process, a sentence is recognized as an imperative if its sentence type is that of a simple imperative, or if its import is consistent with that of an imperative sentence, as is the case with our example, a simple declarative sentence containing the modal, "should": "You should study for the test." In the second stage of the process, the system attempts to place the imperative in a practical argument frame, matching it to the antecedent and the consequent as described above. (The matching is based on the plan that can be derived from the imperative.) As described for the antecedent and the consequent, an empty slot is always a successful match; with the imperative, however, more happens when it is placed in a practical argument frame which has either an empty antecedent or an empty consequent (or both are empty). The system does not wait to see if further input will supply these values. Instead it generates inferences and places those inferences, represented as sentence frames, in the appropriate slots. This inferencing process is triggered when a value is placed in the imperative slot. The slot contains demons which examine the antecedent slot and the consequent slot to see if those slots have values. If those slots are found to be empty, the appropriate inferences are made and are placed into those slots.

For the most part, the system uses the same rules for making inferences as it does for matching. Prolog's use of unification allows the same rules to generate a plan or a goal if none are to be found. The plan and the goal are then passed to special routines that incorporate each into a sentence structure representing their meaning. If the hypothetical declarative present in our example ("If you study for the test, then you will pass the test.") were missing, then the system would generate it based on the information present in the imperative, and the knowledge about plans and goals that is contained in the knowledge-base.

When a sentence frame fills the imperative slot, and the above mentioned demons have insured that both the antecedent and the consequent slots have values, then a third demon attached to the imperative slot attempts to determine the type of speech act that this practical argument frame represents. This is done by comparing the antecedent and the imperative. If both are positive or both are negative, then the type of speech act is "advice". If one is positive and the other is negative, then the speech act is marked as a "warning". In our example, both the antecedent ("If you study for the test") and the imperative ("you should study for the test") are positive, and the speech act is one of giving advice. From this we know that the speaker considers 'passing a test' to be a goal which the speaker should pursue. If the speech act were determined to be a warning then we would know that the consequent represents a state of affairs which the speaker believes the hearer should avoid.

A practical argument frame is essentially an amplifier of the imperative. The event slot allows the meaning of the imperative to be extended into the world of events, and attempts to tie it to such measurable quantities as the time by which a goal must be accomplished. The antecedent and consequent slots allow the extension of the imperative's meaning into both a logical and a psychological dimension. The antecedent and the consequent are logical entities as the premises of an argument. They can also be interpreted within a psychological dimension when we consider their propositional contents to represent a plan (the antecedent) and a goal (the consequent). Lastly, the speech act slot allows the imperative to be interpreted along a dimension which expresses the relationship between the speaker and the hearer in terms of the goal put forward by the consequent.

With these extensions in mind we can see that we use the practical argument frame as the first step in building a sense of context, and that this 'first step' is less concerned with the relationships between sentences and is more concerned with placing a single sentence (the imperative) in that context. The syntactic parser allows for the interpretation of the internal construct of an utterance in terms of a common grammar. The semantic (or case frame) interpreter also focuses on the internal relations of a single sentence in terms of what we know about common usage and meaning (especially the usage and meaning of verbs). The practical argument interpreter goes one very important step further: it takes that single sentence as its focus, but it turns attention outward towards both the language and the world that surrounds it. At the point of doing so, we have attempted to represent the sentence as it truly is: as a unit of meaning in a world composed of such units of meaning. While it does not seem we have gone very far beyond the inward looking syntactic and semantic interpretations of a single sentence, the truth is that we have probably gone as far as it will profit us to treat context in terms of language and what we know about language. The relationship between different speech acts as units

of meaning are relationships that are more likely to be well expressed by one or more knowledge bases which are constructed purely in terms of the pragmatic needs of an application. Here the constructs of language are likely only to seem serendipitous, and of no help in constructing formal representations of knowledge.

7.2.5. Making Inferences and Deductions

One advantage of this form of representation is that it now allows the system to infer information not explicitly present in the text. This gives us another level of inference, one which is a good deal cheaper to implement than is the semantic inference supplied by case frame definitions, and one which supplies inferences which are logical entities that can be manipulated as carriers of meaning.

Besides filling in the text with logically derived inferences, we can also use this representation of a practical argument to deduce the kind of speech act that a given argument represents: here we limit ourselves to distinguishing between advice and warnings, but other kinds of speech acts can be discerned from the same practical argument form. (See [DONA 86].) Without referencing any domain or world knowledge, we can deduce whether or not the consequent of the argument is believed by the speaker to be a benefit to the hearer or is believed to be harmful to the hearer. (See Chapter 1, and section 2.1 for a description of how this can be done.)

The third advantage of this form of representation is that it allows us to store information in a highly structured and uniform format. We can then compare these stored structures and their contents to detect conflicts among sequentially related arguments. We can then apply rules of logic to these arguments to see how these conflicts might be resolved, or to detect in some cases that internally valid arguments are no longer valid when considered in the context of other equally valid arguments. Of course, it is this last advantage which justifies the work that has gone into defining the lower structures and layers of meaning that allow us to reach this higher level of attempting to automate one side of a discourse, or more simply put, to process context and meaning, and not just individual sentences.

We should consider that it is the power to make inferences that is the range of a system's ability to represent meaning. In a fully realized system, the inferencing power should increase as the context is built. That is, the more information that a system has, the more it should be able to infer from that information, and the more it should need to supply such inferences in order to be able to adequately represent the context. Where this ability to make inferences does not keep pace with the increasing complexity of the context, then we can say that the complexity beyond the range

of inference does not carry meaning as far as the system is concerned. While the system may be able to arrive at a minimal organization of this information, and may be able to parrot or paraphrase it for the user, we cannot say that it can represent that information as meaning.

Most Natural Language Understanding systems treat the need to supply inferences as a basic necessity, one derived from our ability to leave out a great deal of what we mean when we speak or write. The system must be able to supply what is readily understood just as any human participant in a conversation would. What we are saying here, goes further than this necessity. We are saying that our ability to supply inferences is our ability to represent meaning. This extra step is important because we can measure our ability to supply inferences, and we can use those measurements to determine how well we represent meaning. This is probably true for more than automated natural language understanding systems.

7.3. Other Context Frames

In the dialogue excerpt used as an example in this chapter, some of the material would not be represented in a practical argument frame. Secondary, less complex, context frames are used to handle dialogue elements which respond to the user (here in the form of a paraphrase),

Output: The speaker has advised the hearer to study for a test.

allow the user to ask questions,

Input: Why?

and allow the system to respond to those questions,

Output: The speaker believes that the hearer will pass the test if the hearer studies for the test.

While the contents of these additional dialogue activities do refer to the information contained in the practical argument frame, the form which this information takes requires additional frame structures and additional processing methods. By far the most important of these additional capabilities is the need to generate English language text. Here we will only touch on some of the problems and some of the features of this other side to processing Natural Language.

The dialogue example continues beyond the practical argument to respond to the user by paraphrasing the argument, and to further respond to the user's request for justification of the argument:

Output: The speaker has advised the hearer to study for a test.
Input: Why?
Output: The speaker believes that the hearer will pass the test if the hearer studies for the test.

The first "output" represents a paraphrase of the prior input, reporting the contents of the imperative, and identifying the speech act it represents as "advice". When the status slot of a practical argument frame changes from "incomplete" to "complete", a demon is triggered to do this. At this point, the system recognizes that it has just finished constructing a unit of the context, and it looks for something to do with that unit. Thus the unit (a practical argument frame) defines the ways in which it can fit into a context. Those "ways" are represented as actions (or procedures) which are likely to create more records or frames to represent the overall context of the dialogue. In this case, there is nothing in particular to do, so the system performs its basic operation of recognizing a unit of meaning, and for want of anything better to do, it reports that recognition to the user. Since that recognition is reported in the form of a sentence, it is recorded as a sentence frame similar in form to those sentence frames that preserve the input. The following frame preserves the paraphrase as a part of the dialogue:

sentence5:
subject: speaker1,
verb: advise,
indirect_object: hearer1,
object: sentence4,
refers: argument1,
sent_type: system_reply,
is_a: output_sentence

This frame exists to record the system's output in case some further reference is made to it in the course of the dialogue; it also exists as an expression of the way in which an automated system should generate English language text. That generation is best done in two stages: at the first stage, the system generates a representation of the concepts to be expressed; at the second stage, it forms that concept representation into English. The system described here is primarily concerned with studying forms for language, and does not adequately deal with the knowledge representation side of the process; hence we have allowed the first stage to be expressed in the form of a sentence frame: in a more practical system, the formation of this sentence frame would probably be the last in a series of steps that belong to this concept formation stage. In fact, the sentence frame is so close to the actual English language statement, that it can be built in parallel with that statement, without making any significant difference to the architecture of the overall system. We experiment with both arrangements here: the frame and the sentence are constructed in parallel for the paraphrase represented above, and the frame is created first, and the sentence is derived from it, when the system forms an answer to

the user's question, "Why?".

7.4. Conclusion

The implementation ends here. A great deal more work would be necessary to handle the dialogue presented in Chapter 1. The work that has been presented here began with the intention of evaluating practical arguments within the context of a dialogue with a human user; however, that intention shifted its focus to the more tractable area of representing individual sentences so that they could become input to the pragmatic and context level of a dialogue system. That material has been presented in Chapters 4, 5, and 6, and most of the conclusions in this study can be found in Chapter 6 (which, for the most part, was written after the previous sections of Chapter 7).

While the prospects for building a Natural Language Understanding system are not good, still the effort to explore some of the issues involved has been interesting and offers benefits to those who would continue in that exploration. While there are no immediate prospects for building any kind of a practical system, still the effort to build a mechanism has merit as an area of research. Most importantly, the effort to build a mechanism allows the opportunity to think about natural language in a way that is different from introspection and observation. For much of the effort and time involved in building such a mechanism, the details dominate attention and the general picture is temporarily lost. Eventually enough work is done, and a general view begins to reestablish itself. That general view or understanding may not be the same after that work as it had been before that work was begun. In the case of this study, a different view of semantic representation came into focus: the study began with a belief that semantic representation was an aspect of knowledge about language, and ended by seeing it as representing knowledge about the world. At the same time, this change of opinion led to an opportunity to think about relationships between syntax and semantics in natural language.

The problem that we have in discovering how natural language operates is that we are too familiar with it in a way that does not require us to have that kind of an understanding; another way of stating this problem is to say that Shakespeare probably did not understand how language works any better than do the rest of us. Neither familiarity nor skill takes us any closer to understanding how natural language operates. This may mean that it is not worthwhile to understand how natural language works. If it is worthwhile to understand, however, then we need some way to challenge our own familiarity. Building an automated mechanism offers us the opportunity to do that. Since this is research, and we do not begin by knowing, the outcome is uncertain.

BIBLIOGRAPHY

Allen, James F., Alan M. Frisch, and Diane J. Litman. ARGOT: The Rochester Dialogue System, *Proceedings of the Second Annual National Conference on Artificial Intelligence*, Pittsburg, P.A., August 1982, pp. 66-70.

Allen, James F. Recognizing Intentions from Natural Language Utterances, in [BRAD 83].

Allen, James F., and Diane J. Litman. Plans, Goals, and Language, *Proceedings of the IEEE*, 74, 7 (July, 1986), pp. 939-947.

Berwick, Robert C. *The Acquisition of Syntactic Knowledge*. Cambridge, Massachusetts: The MIT Press, 1985.

Brady, Michael, and Robert C. Berwick, eds. *Computational Models of Discourse*. Cambridge, Massachusetts: The MIT Press, 1983.

Cabonell, J. G., et al. The XCALIBUR Project, IJCAI 83: *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*, pp. 653-656, 1983.

Chamiak, Eugene. On the Use of Framed Knowledge in Language Comprehension, *Artificial Intelligence*, 11, 3 (December, 1978), pp. 225-265.

Clocksin, W. F., and C. S. Mellish. *Programming in Prolog*, second edition. Berlin, Germany: Springer-Verlag, 1984.

Donaghy, Kevin. Recognizing Promises, Advice, Threats and Wamings in Natural Language Discourse, Masters Thesis, Rochester Institute of Technology, School of Computer Science and Technology, 1986.

Dyer, Michael George. *In-depth Understanding: a Computer Model of Integrated Processing for Narrative Comprehension*. Cambridge, Massachusetts: The MIT Press, 1983.

Faletti, Joseph. PANDORA A Program for Doing Commonsense Planning in Complex Situations, *Proceedings of the Second Annual National Conference on Artificial Intelligence*, Pittsburg, P.A., August 1982, pp. 185-188.

Fiks, D. E. and N. J. Nilsson. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving, *Artificial Intelligence*, vol. 2, 1971.

Fillmore, Charles. The Case for Case, in *Universals in Linguistic Theory*. Emmon Bach and Robert T. Harms, eds. Chicago: Holt, Rinehart and Winston, 1968.

Garfield, Jay L., ed. *Modularity in Knowledge Representation and Natural-Language Understanding*. Cambridge, Massachusetts: The MIT Press, 1987.

Grosz, Barbara J., Karen Sparck Jones, and Bonnie Lynn Webber, eds. *Readings in Natural Language Processing*. Los Altos, California: Morgan Kaufmann Publishers, Inc., 1986.

Hiss, LeMora S. A Frame Virtual Machine in C-Prolog, Masters Thesis, Rochester Institute of Technology, School of Computer Science and Technology, 1987.

Marcus, Mitchell P. *A Theory of Syntactic Recognition for Natural Language*. Cambridge, Massachusetts, 1980.

McKeown, Kathleen R. Language Generation: Applications, Issues, and Approaches, *Proceedings of the IEEE*, 74, 7 (July, 1986), pp. 961-968.

Michalski, Ryszard S., Jaime G. Carbonell, and Tom M. Mitchell, eds. *Machine Learning: An Artificial Intelligence Approach*. Palo Alto, California: Tioga Publishing Company, 1983.

Perrault, C. R., James Allen, and P. R. Cohen. Speech Acts as a Basis for Understanding Dialogue Coherence, *Proceedings of the Second Conference on Theoretical Issues in Natural Language Processing*. Champaigne-Urbana, Illinois, 1978.

Schank, Roger C., ed. *Conceptual Information Processing*. North-Holland, Amsterdam, 1975.

Schank, Roger C. and R. P. Abelson. *Scripts, Plans, Goals, and Understanding*. Hillsdale, N. J.: Lawrence Erlbaum Associates, Publishers, 1977.

Schank, Roger C., C. K. Riesbeck, and K. Christopher. *Inside Computer Understanding: Five Programs Plus Miniatures*. Hillsdale, N. J.: Lawrence Erlbaum Associates, Publishers, 1981.

Searle, John R. *Speech Acts: An Essay in the Philosophy of Language.* Cambridge, England: Cambridge University Press, 1969.

Searle, John R. *Expression and Meaning.* Cambridge: Cambridge University Press, 1975.

Searle, John R., and Daniel Vanderveken. *Foundations of Illocutionary Logic.* Cambridge: Cambridge University Press, 1985.

Vilain, Marc B. A System for Reasoning about Time, *Proceedings of the Second Annual National Conference on Artificial Intelligence*, Pittsburgh, P. A., August 1982, pp. 197-201.

Wilensky, Robert. *Planning and Understanding: A Computational Approach to Human Reasoning.* Reading, Massachusetts: Addison-Wesley Publishing Company, Advanced Book Program, 1983.

Wilensky, Robert, Yigal Arens, and David Chin. Talking to UNIX in English: An Overview of UC, *Communications of the ACM*, 27, 6 (June, 1984), pp. 574-593.

Wilensky, Robert. Understanding Goal-based Stories, Ph.D. Thesis, Yale (Y140), September, 1978.

Winograd, Terry. *Language as a Cognitive Process, Volume 1: Syntax.* Reading, Massachusetts: Addison-Wesley Publishing Company, 1984.

APPENDIX: CODE

The following pages contain the Prolog implementation discussed in this thesis. It should be noted that the frame representation included in this implementation makes use of additional Prolog code presented in [HISS 87]. The listing for that code is not included here.

```
1
2     start :-        /* continue reading until user says "bye."   */
3             repeat,
4             do_one(Input),
5             Input == [bye,'.'].
6
7     do_one(Input) :-
8             clean_up,                       /* abolish items - last sent  */
9             read_sent(Input), !,            /* read input sentence into a list */
10            ( Input == [bye,.]);            /* halt on "bye."  */
11            prepare[Input],                 /* prepare input list for parsing */
12            parse_start ), !.                       /* parse input list  */
13
14
15    clean_up :-             /* remove data structures used for previous parse */
16            abolish(buff,2),
17            abolish[remaining,1),
18            abolish[process_list,1).
19
20    prepare(Input) :-
21            check_words(Input,Checked),     /* make sure all are in dictionary */
22            preprocess(Checked,Curr_sent1),  /* parse noun phrases  */
23            fill_buff(1,Curr_sent1,Leftover),/* create and fill buffer (3 cells) */
24            assert(remaining(Leftover)), !. /* save remainder of input list  */
25
26
27    /*
28            "Preprocess" receives the input sentences represented as a list.
29            It returns the list with all of the noun phrases replaced by
30            noun phrase structures.  See the file "np" for the routines that
31            process noun phrases.
32    */
33
34    preprocess([Last],[Last]) :-
35            end_sent(Last), !.                      /* done parsing np's */
36
37    preprocess([First|Original],[NP|Processed]) :-  /* look for noun phrases */
38            leading_edge(First),                    /* found initial word of np */
39            np(NP,[First|Original],Leftover), !,    /* so parse the np  */
40            preprocess(Leftover,Processed).         /* do rest of the list  */
41
42    preprocess([First|Original],[First|Processed]) :- /* not part of np    */
43            preprocess(Original,Processed).         /* so check next word  */
44
45    leading_edge(First) :-                          /* a np may begin with a      */
46            ( det(First,_);                         /*  a determiner              */
47             adj_(First,_);                         /*  an adjective              */
48             noun(First,_,_);                       /*  a noun (common or proper) */
49             pronoun(First,_,_) ).                  /*  or a pronoun              */
50
51    end_sent('.').
52    end_sent('!').
53    end_sent('?').
```

```
1
2     /*  "sentence" defines the valid slots that may appear in a sentence frame.
3           It also defines a default value for each slot.
4     */
5
6     :- def_frame(sentence:[
7                             subject:[value:[nil]],
8                             verb:[value:[nil]],
9                             object: [value:[nil]],
10                            ind_obj:[value:[nil]],
11                            by:[value:[nil]],
12                            for:[value:[nil]],
13                            to:[value:[nil]],
14                            time:[value:[nil]],
15                            interrog:[value:[nil]],
16                            modal:[value:[nil]],
17                            pos_neg:[value:[positive]],
18                            act_pass:[value:[active]],
19                            sent_type:[value:[decl]]
20                           ]).
21
22    /*  There are four kinds of sentences:  */
23
24    :- def_frame(input_sent:[
25                            is_a:[value:[sentence]],
26                            description:[value:[user_entered]]
27                          ]).
28
29    :- def_frame(embedded_sent:[
30                            is_a:[value:[input_sent]],
31                            description:[value:[dependent_clause]]
32                          ]).
33
34    :- def_frame(context:[
35                            is_a:[value:[sentence]],
36                            description:[value:[system_generated]]
37                          ]).
38
39    :- def_frame(nil_sent:[
40                            is_a:[value:[sentence]],
41                            description:[value:[absent_clause]]
42                          ]).
43
44    /*  Create a new sentence frame          */
45    set_curr_sent_frame(Curr_sent) :-
46            get_name(sent,Curr_sent),        /* generate a unique name */
47            def_frame(Curr_sent:[
48                            is_a:
49                                  [value:[input_sent],
50                                  max:2]
51                           ]).
52
53    /* Create a frame, making sure it has a unique name.   */
54    make_frame(Type,Instance) :-
55            get_name(Type,Instance),
56            def_frame(Instance:[is_a:[value:[Type]]]).
```

```
/*-------------------------------------------------------------------------*/
/*
        This file contains the parser.  The input list is maintained in a
    globally maintained buffer, referenced as buff(Cell,Item), where the
    first argument is the index of the cell, and may take the values, "1",
    "2", "3", and the second argument is the item contained in the buffer.

    A set of intermediate calls are usually used in the pattern matching rules.
    These intermediate calls - "match(parameter list)" are contained in the
    file, "parse_aux".  Usually it is these intermediate rules that reference
    the buffer.
*/
/*-------------------------------------------------------------------------*/
/* if the buffer is empty then the parse is finished  */
parse_start :-
        buff(1,empty).

/* Create a uniquely named frame to hold the analyzed contents of the buffer */
parse_start :-
        set_curr_sent_frame(Curr_sent), !,
        parse(Curr_sent).

/*-------------------------------------------------------------------------*/

/* The parsing rules are divided into four sets.  If the sentence is simple
        (has no embedded sentences), the sets are activated in sequence, and
        no rule in one set calls a rule in any of the other sets.  Each set
        is guarded with a cut, so that no backtracking will occur if a rule
        in that set succeeds.
*/
parse(Curr_sent) :-
        sent_type(Curr_sent), !,        /* declarative, interogative, etc. */
        parse_subj(Curr_sent), !,       /* process the subject             */
        parse_verb_part(Curr_sent), !,  /* process everything else         */
        finish_parse(Curr_sent), !.     /* check that we're done           */

/*-------------------------------------------------------------------------*/

/*
        sent_type(Curr_sent) -- examines the initial elements of the sentence
        and determines the category of sentence type to which it belongs.
        In some cases it may delete, add, or move sentence elements
        in the buffer.  The kind of sentences that are recognized are:
                Declarative
                Imperative
                Interrogative ("wh" questions)
                Yes-no question
                Condition -- "if ..."
                Consequent  -- "[then]...
                Answer(yes or no)
*/

/*-------------------------------------------------------------------------*/

/* A simple declarative sentence -- the default sentence type */

sent_type(Curr_sent) :-
        match(1,np,_,_),                    /* a noun phrase */
        match(2,verb_type,_).               /* any modal, auxverb, or verb */
        /* use default value -- (Curr_sent,sent_type,decl)  */

sent_type(Curr_sent) :-
        buff(1,to),                         /* "to"                        */
        match(2,verb,_,_,[infinitive|_]).  /* infinitive form of a verb */
        /* use default value -- (Curr_sent,sent_type,decl)  */

sent_type(CurrSent) :-
        buff(1,not),
        buff(2,to),
        match(3,verb,_,_,[infinitive|_]).  /* infinitive form of a verb */
        /* use default value -- (Curr_sent,sent_type,decl)  */

sent_type(CurrSent) :-
        match(1,verb,_,_,[prespart|_]).        /* "ing" form of verb */
        /* use default value -- (Curr_sent,sent_type,decl)  */
```



```

sent_type(CurrSent) :-
        buff(1,not),                        /* "not"                   */
        match(2,verb,_,_,[prespart|_]).     /* "ing" form of verb      */
        /* use default value -- (Curr_sent,sent_type,decl)  */

sent_type(Curr_sent) :-
        buff(1,there),                      /* dummy -- "There is", etc.   */
        match(2,be_verb,_,_),               /* any form of "to be"         */
        match(3,np,_,_),                    /* a noun phrase               */
        remove_buff_item(1),                /* discard "there" & adjust buff */
        switch(1,2).                        /* reverse order of buff cells 1 & 2 */
        /* use default value -- (Curr_sent,sent_type,decl)  */

sent_type(Curr_sent) :-
        buff(1,there),                      /* dummy "there" --    */
        match(2,modal,_,_),                 /*    "there should be" or  */
        ( match(3,be_verb,_,_);             /* "There should have been" */
          match(3,auxverb,_,_,_) ),
        remove_buff_item(1),                /* remove "there" from first cell */
        find_first_np(NP),                  /* subject may not be in buffer yet */
        insert_buff(1,NP).                  /* Put real subject in 1st cell   */
        /* use default value -- (Curr_sent,sent_type,decl)  */


/*-------------------------------------------------------------------------*/

/* A question that is not to be answered by a simple yes or no */

sent_type(Curr_sent) :-
        match(1,interrog,Word),         /* why, how, etc.  */
        add_slot_val(Curr_sent,sent_type,question),
        add_slot_val(Curr_sent,interrogative,Word),
        remove_buff_item(1),
        add_slot_val(Curr_sent,status,needs_context),
        make_context_link(Curr_sent).
/*       continue_interrog(Curr_sent,Word).     */
/* only allowing "Why" as a question not seeking a "yes" or "no" answer. */
/*
continue_interrog(_,_) :-
        buff(1,Item),
        ( end_sent(Item);
            Item == empty ).
*/

/*-------------------------------------------------------------------------*/

/* A question that expects a "yes" or "no" answer.  */

sent_type(Curr_sent) :-
        (match(1,auxverb,_,_,_);
          match(1,modal,_,_) ),
        match(2,np,_,_),
        switch(1,2),
        add_slot_val(Curr_sent,sent_type,yes_no_q).

sent_type(Curr_sent) :-
        (match(1,auxverb,_,_,_);
          match(1,modal,_,_) ),
        match(2,negative,_),
        match(3,np,_,_),
        switch(1,3),
        switch(2,3),
        add_slot_val(Curr_sent,sent_type,yes_no_q).

sent_type(Curr_sent) :-
        match(1,be_verb,_,_),
        match(2,np,_,_),
        add_slot_val(Curr_sent,sent_type,yes_no_q),
        switch(1,2).

sent_type(Curr_sent) :-
        match(1,be_verb,_,_),
        match(2,negative,_),
        match(3,np,_,_),
        add_slot_val(Curr_sent,sent_type,yes_no_q),
```

```
153         switch(1,3), switch(2,3).
154
155
156     /*-------------------------------------------------------------------------*/
157
158     /* An imperative sentence.  */
159
160     sent_type(Curr_sent) :-
161             match(1,verb,_,_,[infinitive|_]),       /* infinitive form only */
162             add_slot_val(Curr_sent,sent_type,imper), /* sent is imperative */
163             np(NP,[you],_),                         /* create a np: "you"   */
164             insert_buff(1,NP).                      /* put it in 1st buffer cell */
165
166     sent_type(Curr_sent) :-
167             match(1,auxverb,do,_,_),                /* "do" only    */
168             match(2,verb,_,_,[infinitive|_]),
169             add_slot_val(Curr_sent,sent_type,imper),
170             np(NP,[you],_),
171             insert_buff(1,NP).
172
173     sent_type(Curr_sent) :-
174             match(1,auxverb,do,_,_),                /* "Do"         */
175             match(2,negative,_),                    /* "not"        */
176             match(3,verb,_,_,[infinitive|_]),
177             add_slot_val(Curr_sent,sent_type,imper),
178             np(NP,[you],_),
179             insert_buff(1,NP).
180
181
182     /*-------------------------------------------------------------------------*/
183
184     /* A hypotetical declarative -- if ... then ...
185             Is treated as two sentences
186     */
187
188     sent_type(Curr_sent) :-                         /* The conditional part */
189             buff(1,if),
190             add_slot_val(Curr_sent,sent_type,cond),
191             remove_buff_item(1).
192
193     sent_type(Curr_sent) :-                 /* The consequence of the conditional */
194             buff(1,then),
195             add_slot_val(Curr_sent,sent_type,conseq),
196             remove_buff_item(1).
197
198     /*-------------------------------------------------------------------------*/
199
200     /* An answer to a yes-no question   */
201
202     sent_type(Curr_sent) :-
203             buff(1,Word),
204             Word == yes,
205             remove_buff_item(1),
206             add_slot_val(Curr_sent,sent_type,answer).
207
208     sent_type(Curr_sent) :-
209             buff(1,Word),
210             Word == no,
211             remove_buff_item(1),
212             add_slot_val(Curr_sent,pos_neg,negative),
213             add_slot_val(Curr_sent,sent_type,answer).
214
215     /*-------------------------------------------------------------------------*/
216
217     /* Remove a leading conjunction; then determine the sentence type  */
218     sent_type(Curr_sent) :-
219             buff(1,Word),
220             conj_(Word),
221             remove_buff_item(1),
222             sent_type(Curr_sent).
223
224
225
226     /*-------------------------------------------------------------------------*/
227     /*-------------------------------------------------------------------------*/
228     /*      Parse the subject of the sentence.  Due to the buffer manipulations
```



```
229          in the previous rule set, the subject should always be the first
230          element or group of elements in the buffer.
231     */
232     /*-------------------------------------------------------------------------*/
233     /*-------------------------------------------------------------------------*/
234
235
236     /* The most common case: a simple noun phrase.          */
237     parse_subj(Curr_sent) :-
238             match(1,np,NP,_),
239             add_noun_inst(Curr_sent,subject,NP), !,
240             remove_buff_item(1).
241
242     /* Have arrived at the end of the sentence.  Make sure it can inherit from the
243        the preceding sentence. */
244     parse_subj(Curr_sent) :-
245             buff(1,Item),
246             ( end_sent(Item);
247               Item == empty ),
248             ( get_slot_val(Curr_sent,status,needs_context);
249                (add_slot_val(Curr_sent,status,needs_context),
250                 make_context_link(Curr_sent)) ).
251
252     /* Subject is an infinitive phrase.  Ex.: "To finish the lab will take time."
253             Control is passed to a separate set of routines that handle this kind
254             of dependent clause.
255     */
256     parse_subj(CurrSent) :-
257             buff(1,to),
258             match(2,verb,_,Inf,[infinitive|_]),
259             remove_buff_item(1),
260             remove_buff_item(1),
261             cont_inf_phrase(CurrSent,positive,Inf).
262
263     parse_subj(CurrSent) :-
264             buff(1,not),
265             buff(2,to),
266             match(3,verb,_,Inf,[infinitive|_]),
267             remove_buff_item(1),
268             remove_buff_item(1),
269             remove_buff_item(1),
270             cont_inf_phrase(CurrSent,negative,Inf).
271
272     /* The subject is a gerund phrase.  Ex.: "Finishing the lab ..."
273             Handle in the same way as an infinitive phrase.          */
274     parse_subj(CurrSent) :-
275             match(1,verb,_,Inf,[prespart|_]),
276             remove_buff_item(1),
277             cont_inf_phrase(CurrSent,positive,Inf).
278
279     parse_subj(CurrSent) :-
280             buff(1,not),
281             match(2,verb,_,Inf,[prespart|_]),
282             remove_buff_item(1),
283             remove_buff_item(1),
284             cont_inf_phrase(CurrSent,negative,Inf).
285
286     /* subject is missing, so make sure it can be inherited from previous
287             sentence. Ex.: "Why study for the test?"
288     */
289     parse_subj(CurrSent) :-
290             match(1,verb_type,_),           /* any modal, auxverb, or verb */
291             ( get_slot_val(CurrSent,status,needs_context);
292                (add_slot_val(Curr_sent,status,needs_context),
293                 make_context_link(Curr_sent)) ).
294
295     /*-------------------------------------------------------------------------*/
296     /* parse everything after the subject  */
297     /*-------------------------------------------------------------------------*/
298
299
300     /* Have arrived at the end of the sentence.  Make sure it can inherit from the
301        the preceding sentence. */
302     parse_verb_part(Curr_sent) :-
303             buff(1,Item),
304             ( end_sent(Item);
```

```
305             Item == empty ),
306           ( get_slot_val(Curr_sent,status,needs_context);
307              (add_slot_val(Curr_sent,status,needs_context),
308               make_context_link(Curr_sent)) ).
309
310   /* At this second level of control, first call those rule sets that process
311           the auxiliary structure of the verb, then process the main verb and
312           its dependencies.
313   */
314   parse_verb_part(Curr_sent) :-
315           process_aux(Curr_sent), !,
316           check_pos_neg(Curr_sent), !,
317           check_act_pass(Curr_sent), !,
318           parse_verb(Curr_sent), !.
319
320   /*-------------------------------------------------------------------------*/
321   /*  process_aux(Curr_sent)
322           This set of rules examines the current sentence for auxiliary
323           verbs and removes them.  It does not currently save any of the in-
324           formation that they convey.  These rules could be altered to
325           do so, however.  The dictionary is able to return information on such
326           grammatical considerations as tense and number, and the rules could be
327           written to examine the auxiliary structure in more detail. At present
328           this is not done because such information does not tend to be very
329           helpful.  For example, 'The teacher scheduled a test for Monday" and
330           "The teacher has scheduled a test for Monday" use the past and present
331           perfect forms of "to schedule", yet the fact which is of concern is
332           actually a future event.
333                   There are hidden complexities involved in actually making
334           use of tense to aid in determining meaning, and
335           it does not seem appropriate at this time to add this kind of com-
336           plexity to the parser.  To continue the above example consider the
337           meanings of the following three sentences:
338                   A test is scheduled for Monday.
339                   A test was scheduled for Monday.
340                   A test had been scheduled for Monday.
341   */
342
343   /* Have arrived at the end of the sentence.  Make sure it can inherit from the
344      the preceding sentence. */
345   process_aux(Curr_sent) :-
346           buff(1,Item),
347           ( end_sent(Item);
348             Item == empty ),
349           ( get_slot_val(Curr_sent,status,needs_context);
350              (add_slot_val(Curr_sent,status,needs_context),
351               make_context_link(Curr_sent)) ).
352
353   /* If the first buffer item is a main verb that is not "have" or "do" we are
354           done. */
355   process_aux(Curr_sent) :-
356           match(1,verb,_,Inf,_),
357           Inf \== have,
358           Inf \== do.
359
360   /* "have" or "do" are the main verb, so we're done.  */
361   process_aux(Curr_sent) :-
362           ( match(1,verb,_,have,_);
363                   match(1,verb,_,do,_) ),
364           not match(2,verb,_,_,_),
365           not match(2,be_verb,_,_),
366           not match(2,negative,_).
367
368   /* the first buffer item is "not" so we're done.   */
369   process_aux(Curr_sent) :-
370           buff(1,Item),
371           negative_(Item).
372
373   /*the first item is a form of "to be" so we're done.   */
374   process_aux(Curr_sent) :-
375           match(1,be_verb,_,_).
376
377   /*  The first buffer item is a modal, so save it in the sentence frame, remove
378           it from the buffer, and continue looking for auxiliary verbs.  */
379   process_aux(Curr_sent) :-
380           match(1,modal,Word,_),
```



```
381           add_slot_val(Curr_sent,modal,Word),
382           remove_buff_item(1),
383           process_aux(Curr_sent).
384
385   /*  The first buffer item is an auxiliary verb, so remove it and continue
386           looking for auxiliary verbs.  */
387   process_aux(Curr_sent) :-
388           match(1,auxverb,_,_,_),
389           remove_buff_item(1),
390           process_aux(Curr_sent).
391
392   /*-------------------------------------------------------------------------*/
393
394   /* If there are no more lexical items, we're done. */
395   check_pos_neg(Curr_sent) :-
396           buff(1,Item),
397           ( end_sent(Item);
398             Item == empty ).
399
400   /* First buffer item is "not". */
401   check_pos_neg(Curr_sent) :-
402           buff(1,Item),
403           negative_(Item),
404           add_slot_val(Curr_sent,pos_neg,negative),
405           remove_buff_item(1).
406
407   check_pos_neg(Curr_sent) :-
408           match(1,be_verb,_,_),           /* any form of "to be"          */
409           buff(2,Item),                   /* "not"                        */
410           negative_(Item),
411           add_slot_val(Curr_sent,pos_neg,negative),
412           remove_buff_item(2).
413
414   check_pos_neg(Curr_sent).       /* use default value :  positive   */
415
416
417   /*-------------------------------------------------------------------------*/
418
419   /* If there are no more lexical items, we're done. */
420   check_act_pass(Curr_sent) :-
421           buff(1,Item),
422           ( end_sent(Item);
423             Item == empty ).
424           /* use default value -- (Curr_sent,act_pass,active), */
425
426   /* a passive construction is always indicated by a form of the verb "to be"
427           followed by the past participle form of the main verb.  */
428   check_act_pass(Curr_sent) :-
429           match(1,be_verb,_,_),
430           match(2,verb,_,_,[pastpart|_]),
431           add_slot_val(Curr_sent,act_pass,passive),
432           remove_buff_item(1).
433
434   /* The form of the main verb is not the past participle.  To be grammatically
435           correct it must be the present particple ("-ing"), but we use the
436           looser restriction here since we have not been enforcing grammatical
437           correctness beyond the scope of determining meaning.    */
438   check_act_pass(Curr_sent) :-
439           match(1,be_verb,_,_),
440           match(2,verb,_,_,[Verb_form|_]),
441           Verb_form \== pastpart,
442           remove_buff_item(1).
443           /* use default value -- (Curr_sent,act_pass,active), */
444
445   /* If a form of "to be" is present, it is the main verb.        */
446   check_act_pass(Curr_sent) :-
447   (       match(1,verb,_,_,_); match(1,be_verb,_,_) ).
448           /* use default value -- (Curr_sent,act_pass,active), */
449
450   /*-------------------------------------------------------------------------*/
451
452   /*  parse_verb handles the main verb, and controls the next stage of the
453           parse by distinguishing between a form of the verb "to be" and
454           all other verbs.
455   */
456
```

```
/* Have arrived at the end of the sentence.     */
parse_verb(Curr_sent) :-
        buff(1,Item),
        ( end_sent(Item);
          Item == empty ).

/* First buffer item is any form of a main verb.        */
parse_verb(Curr_sent) :-
        match(1,verb,Word,Inf,_),       /* return the infinitive form of verb */
        add_slot_val(Curr_sent,verb,Inf),/* save in the current clause frame */
        remove_buff_item(1),  !,        /* remove verb form from buffer      */
        parse_objects(Curr_sent), !.    /* everything after main verb        */

/* First buffer item is any form of "to be".    */
parse_verb(Curr_sent) :-
        match(1,be_verb,_,_),
        add_slot_val(Curr_sent,verb,be),
        remove_buff_item(1), !,
        continue_be_verb(Curr_sent).


/*-------------------------------------------------------------------------*/

continue_be_verb(Curr_sent) :-
        buff(1,Item),
        ( end_sent(Item);
           Item == then;
                conj_(Item) ).


continue_be_verb(Curr_sent) :-
        buff(1,Item),
        adj(Item,Category),
        get_slot_val(Curr_sent,subject,Subject),
        add_value(Subject,Category,Item),
        remove_buff_item(1), !,
        continue_be_verb(Curr_sent).

continue_be_verb(Curr_sent) :-
        ( match(1,prep,for) ;
          match(1,prep,on) ),
        match(2,time,Word),
        get_slot_val(Curr_sent,subject,Subject),
        add_slot_val(Curr_sent,time,Word),
        add_slot_val(Subject,time,Word),
        remove_buff_item(1),
        remove_buff_item(1), !,
        continue_be_verb(Curr_sent).

/*-------------------------------------------------------------------------*/

parse_objects(CurrSent) :-
        buff(1,Item),
        ( end_sent(Item);
           Item == then;
                conj_(Item) ).

parse_objects(CurrSent) :-
        match(1,verb_type,_).
/*
parse_objects(CurrSent) :-
        match(1,np,_,_),
        match(2,verb_type,_).
*/

parse_objects(CurrSent) :-
        parse_ind_obj(CurrSent),
        parse_objects(CurrSent).

parse_objects(CurrSent) :-
        parse_obj(CurrSent),
        parse_objects(CurrSent).

parse_objects(CurrSent) :-
        parse_prep_phrase(CurrSent),
        parse_objects(CurrSent).
```



```

parse_objects(CurrSent) :-
        parse_inf_phrase(CurrSent);
          parse_that_clause(CurrSent).

parse_objects(CurrSent) :-
        parse_time_constr(CurrSent),
        parse_objects(CurrSent).

/*-------------------------------------------------------------------------*/

parse_ind_obj(CurrSent) :-
        match(1,np,NP,SingForm),
        frame_subsumes(animate,SingForm),
        add_noun_inst(CurrSent,ind_obj,NP),
        remove_buff_item(1), !.

parse_obj(CurrSent) :-
        match(1,np,NP,_),
        add_noun_inst(CurrSent,object,NP),
        remove_buff_item(1), !.

parse_prep_phrase(Curr_sent) :-
        match(1,prep,Prep),
        match(2,np,NP,_),
        add_noun_inst(Curr_sent,Prep,NP),
        remove_buff_item(1),
        remove_buff_item(1), !.

parse_inf_phrase(CurrSent) :-
        buff(1,to),
        match(2,verb,_,Inf,_),
        remove_buff_item(1),
        remove_buff_item(1),
        cont_inf_phrase(CurrSent,positive,Inf).

parse_inf_phrase(CurrSent) :-
        buff(1,not),
        buff(2,to),
        match(3,verb,_,Inf,_),
        remove_buff_item(1),
        remove_buff_item(1),
        remove_buff_item(1),
        cont_inf_phrase(CurrSent,negative,Inf).

parse_time_constr(Curr_sent) :-
        ( match(1,prep,for) ;
          match(1,prep,on) ),
        match(2,time,Word),
        add_slot_val(Curr_sent,time,Word),
        remove_buff_item(1),
        remove_buff_item(1).

parse_that_clause(Curr_sent) :-
        buff(1,that),
        make_frame(embedded_sent,Name),
        add_slot_val(Curr_sent,object,Name),
        remove_buff_item(1),
        sent_type(Name), !,
        parse_subj(Name), !,
        parse_verb_part(Name), !,
                ( case_analysis(Name) ;
                        write('Case analysis failed') ).


/*-------------------------------------------------------------------------*/

finish_parse(Curr_sent) :-
        buff(1,Item),
        end_sent(Item),
        remove_buff_item(1),
                ( case_analysis(Curr_sent) ;
                        write('Case analysis failed') ),
        examine(Curr_sent).

finish_parse(Curr_sent) :-
```

```
609         get_slot_val(Curr_sent,sent_type,Type),
610         Type == cond,
611         ( match(l,conseq,then);
612             insert_buff(l,then) ),
613                 ( case_analysis(Curr_sent) ;
614                         write('Case analysis failed') ),
615         !, examine(Curr_sent),
616         parse_start.
617
618 finish_parse(Curr_sent) :-
619         buff(l,Item),
620         conj_(Item),
621         remove_buff_item(l),
622                 ( case_analysis(Curr_sent) ;
623                         write('Case analysis failed') ),
624          !, examine(Curr_sent),
625         parse_start.
626
627 /*-------------------------------------------------------------------------*/
628 /*
629         Special rule that handles infinitive (and gerund) phrases that
630 are either the subject or the direct object in the main clause of a sentence.
631 */
632 cont_inf_phrase(Curr_sent,Pos_neg,Verb) :-
633         make_frame(embedded_sent,Name),
634         add_slot_val(Name,verb,Verb),
635         add_slot_val(Name,pos_neg,Pos_neg),
636         place_inf_phrase(Curr_sent,Name),
637         parse_objects(Name),
638                 ( case_analysis(Name) ;
639                         write('Case analysis failed') ).
640
641 /*-------------------------------------------------------------------------*/
642 /*
643         Put the embedded sentence frame in the proper slot in the current
644 sentence (the 'major' clause) frame.  These rules were created in order
645 to address the situation described in the third rule, where 'it' represents
646 a dummy subject of the sentence.  The following is an example of such a
647 construction:
648         "It will take all of the available time to finish the lab.'
649 Here, "to finish the lab" is the subject of the sentence, but because of
650 its placement in the sentence, it must be recognized as an infinitive
651 phrase by those rules that normally handle the object of a sentence.
652         It should be noted that these rules will not be adequate if the
653 major sentence employs this dummy 'it' construction but does not have an
654 object.  This might be the case if the verb of the main clause is a form of
655 "to be" (perhaps, 'It is good for you to study for the test').  Since the
656 parser does not handle such constructions in general, and only addresses the
657 verb "to be" in very limited ways, it would not be appropriate to recognize
658 that possibility in these rules.
659 */
660
661
662 /*  The parser always fills the subject slot before moving on to the verb
663 construction, so if that slot is empty then we know that the embedded sentence
664 belongs in the subject slot.
665 */
666 place_inf_phrase(CurrSent,EmbeddedSent) :-
667         get_slot_val(CurrSent,subject,nil),
668         add_slot_val(CurrSent,subject,EmbeddedSent).
669
670 /*  If the object slot is empty, and the subject slot is not empty, then put
671 the embedded sentence in the object slot.
672 */
673 place_inf_phrase(CurrSent,EmbeddedSent) :-
674         get_slot_val(CurrSent,object,nil),
675         find_subj_inf_object(CurrSent,Subj_of_inf),     /* only for objects */
676         add_slot_val(EmbeddedSent,subject,Subj_of_inf),
677         add_slot_val(CurrSent,object,EmbeddedSent).
678
679 /*  If both the subject and the object slots are filled, then check to see if
680 'it' is in the subject slot.  If so, remove 'it' from the subject slot and
681 replace it with the embedded sentence.
682 */
683 place_inf_phrase(CurrSent,EmbeddedSent) :-
684         get_slot_val(CurrSent,subject,Subject),
```



```
685         frame_subsumes(it,Subject),
686         remove_value_from_slot(CurrSent,subject,_),
687         add_slot_val(CurrSent,subject,EmbeddedSent).
688
689 /*-------------------------------------------------------------------------*/
690
691 /*  An infinitive phrase may take a subject, but will do so only if it is the
692 object in the main clause.  Examples:
693         "I advise you to study for the test."
694         " You are advised to study for the test."
695 In both cases, 'you' is the subject of 'to study'. Here, we assume that if
696 there is an indirect object it is the subject of the infinitive, otherwise
697 the subject of the main verb is also the subject of the infinitive phrase.
698 */
699
700 find_subj_inf_object(CurrSent,Subj_of_inf) :-
701         get_slot_val(CurrSent,ind_obj,Subj_of_inf),
702         Subj_of_Inf \== nil.
703
704 find_subj_inf_object(CurrSent,Subj_of_inf) :-
705         get_slot_val(CurrSent,subject,Subj_of_inf).
706
```

```
/*-------------------------------------------------------------------------*/
/*
        Calls used by the parsing rules to access information in the buffer
        and in the dictionary:  first, get the word or structure from the
        buffer, and then look it up in the dictionary.
*/
/* match on a verb form */
match(Cell,verb,Word,Inf,Features) :-
        buff(Cell,Word),
        verb(Inf,Word,Features).

/* match on an auxiliary verb form */
match(Cell,auxverb,Word,Inf,Features) :-
        buff(Cell,Word),
        auxverb(Inf,Word,Features).

/* match on a form of "to be" */
match(Cell,be_verb,Word,Features) :-
        buff(Cell,Word),
        be_verb_(Word,Features).

/* match on a modal - "should", "would", etc. */
match(Cell,modal,Word,Features) :-
        buff(Cell,Word),
        modal(Word,Features).

/* match on any verb form or modal */
match(Cell,verb_type,Word) :-
        buff(Cell,Word),
        ( verb(_,Word,_);
           auxverb(_,Word,_);
            be_verb_(Word,_);
             modal(Word,_) ).

/* match on a noun phrase structure */
match(Cell,np,NP,Word) :-
        buff(Cell,np(Word,_,_,_)),
        buff(Cell,NP).

/* match on a preposition*/
match(Cell,prep,Word) :-
        buff(Cell,Word),
        prep_(Word).

match(Cell,cond,Word) :-
        buff(Cell,Word),
        cond_(Word).

match(Cell,conseq,Word) :-
        buff(Cell,Word),
        conseq_(Word).

match(Cell,negative,Word) :-
        buff(Cell,Word),
        negative_(Word).

match(Cell,time,Word) :-
        buff(Cell,Word),
        time_(Word).

match(Cell,interrog,Word) :-
        buff(1,Word),
        interrog_(Word).

/*-------------------------------------------------------------------------*/
/*      Special rules for one case where a looked for noun phrase may be
        either in the 3rd buffer cell, or may be the first item of the
        remainder list.  This is the only case where the parse rules have
        to examine more than 3 sentence elements. (See sent_type rules in
        the parser.
*/
find_first_np(NP) :-
        match(3,np,NP,_),
        remove_buff_item(3).

find_first_np(NP) :-
```



```
        remaining([np(_,_,_,_)|_]),
        retract(remaining([NP|Rest])),
        assert(remaining(Rest)).

/*-------------------------------------------------------------------------*/
/* the following routines attach a frame to the previously generated frame
        as its potential context (one from which it can inherit values).
*/

make_context_link(Frame) :-
        get_slot_val(Frame,status,needs_context),
        get_prev_frame(Frame,Previous),
        add_slot_val(Frame,is_a,Previous).

get_prev_frame(Frame,Previous) :-
        return_num(Frame,Num,Frame_type),
        Prev_num is Num -1,
        attach(Frame_type,Prev_num,Previous).

return_num(Frame,Num,Frame_type) :-
        name(Frame,F_list),
        divide_list(F_list,Num_list,Frame_type_list),
        name(Num,Num_list),
        name(Frame_type,Frame_type_list), !.

divide_list([],[],[]).
divide_list([First|Rest],[First|Num],Frame_type) :-
        First >= "0", First =< "9",
        divide_list(Rest,Num,Frame_type).

divide_list([First|Rest],Num,[First|Frame_type]) :-
        divide_list(Rest,Num,Frame_type).

attach(Frame_type,Num,Frame) :-
        name(Frame_type,Name_list),
        name(Num,Num_list),
        append(Name_list,Num_list,Frame_list),
        name(Frame,Frame_list).

/*-------------------------------------------------------------------------*/


/* match on a slot value */
get_slot_val(Frame,Slot,Value) :-
        not var(Value),
        return_value_from_slot(Frame,Slot,[ReturnedVal]), !,
        ReturnedVal == Value.

/* return a slot value - it is often convenient to return it as an atom rather
than as a list (only when it is known that there is only one item
in the list).
*/
get_slot_val(Frame,Slot,Value) :-
        var(Value),
        return_value_from_slot(Frame,Slot,[Value]), !.

/* Given a frame name and a slot value, return the slot name.              */
return_slot_name(Frame,Slot,Value) :-
        return_value_from_slot(Frame,Slot,[Value]), !.

/*-------------------------------------------------------------------------*/
/* add a value to a slot only if it does not already contain that value  */
add_value(Frame,Slot,Value) :-
        (get_slot_val(Frame,Slot,Value);
         add_slot_val(Frame,Slot,Value) ).


/* Make only one effort to add a value to a slot. */
add_slot_val(Frame,Slot,Value) :-
        add_value_to_slot(Frame,Slot,Value), !.

/*-------------------------------------------------------------------------*/
/* routines that manipulate atom elements to create unique  atoms (names)  */

get_name(Oldname,Newname) :-
        increment_reg(Oldname,Value),
```

```
153           name(Oldname,X),
154           name(Value,Y),
155           append(X,Y,Z),
156           name(Newname,Z).
157
158   return_name(Reg,Name) :-
159           R =.. [Reg,Num],
160           call(R),
161           name(Reg,X),
162           name(Num,Y),
163           append(X,Y,Z),
164           name(Name,Z).
```

```
1
2       /* Noun phrase net -
3
4               Returns NP as the structure
5
6                       NP = np(Noun,Num,DefOrIn,Adj).
7
8               where Noun is the singular form of the noun from the sentence;
9               Num is 'sing', 'plu', or 'prop' (proper);
10              DefOrIn is 'def', 'indef' for nouns, or
11              'first', 'second' or 'third' person for pronouns;
12              and Adj is a list of qualifying adjectives and noun phrases
13              where the noun is possessive.
14                      Ex.:
15                              the important test:
16                                      np(test,sing,def,[important]).
17                              the student's important test:
18                                      np(test,sing,def,[np(student,sing,def,[])]).
19      */
20
21      /*-------------------------------------------------------------------------*/
22
23      np(NP,[First|Rest],Leftover) :-
24              det(First,DefOrIn),
25              np2(NP,DefOrIn,[],Rest,Leftover).       /* continue processing np */
26
27      np(NP,[First|Rest],New_rest) :-
28              pronoun(First,Person,Num),              /* any pronoun          */
29              after_pro(Rest,Adj,New_rest),           /* "someone else"       */
30              build_np(First,Num,Person,Adj,NP).      /* done processing np   */
31
32      np(NP,[First|Rest],Rest) :-
33              noun(SingForm,First,Num),
34              guess_def_or_in(First,Num,DefOrIn),
35              build_np(SingForm,Num,DefOrIn,[],NP).   /* done processing np   */
36
37      np(NP,[First|Rest],Leftover) :-
38              adj(First,_),
39              np2(NP,indef,[First],Rest,Leftover).    /* continue processing np */
40
41      /*-------------------------------------------------------------------------*/
42      /*
43              Handle cases where a quantifier precedes a determiner or "of":
44              for example, "all the available time" or "all of the available time".
45      */
46
47      np(NP,[First,Second|Rest],Leftover) :-
48              adj(First,quantity),                    /* adjective is a quantifier */
49              det(Second,_),                          /* followed by a determiner  */
50              np(NP,[Second,First|Rest],Leftover).    /* invert order & continue   */
51
52      np(NP,[First,Second|Rest],Leftover) :-
53              adj(First,quantity),                    /* adjective is a quantifier */
54              prep_(of),                              /* followed by "of" */
55              np(NP,[First|Rest],Leftover).           /* discard "of" & continue   */
56
57      /*-------------------------------------------------------------------------*/
58      /* handle possessive case for proper nouns and plural nouns.               */
59
60      np(NP,[First|Rest],Leftover) :-
61              poss(First,Base),
62              noun(SingForm,Base,Num),
63              guess_def_or_in(Base,Num,DefOrIn),
64              build_np(SingForm,Num,DefOrIn,[],NP1),
65              np2(NP,DefOrIn,[NP1],Rest,Leftover).
66
67      /*-------------------------------------------------------------------------*/
68
69      np2(NP,DefOrIn,Adj,[First|Rest],Leftover) :-
70              adj(First,_),
71              np2(NP,DefOrIn,[First|Adj],Rest,Leftover). /* continue with np  */
72
73      np2(NP,DefOrIn,Adj,[First|Rest],Rest) :-
74              noun(SingForm,First,Num),
75              build_np(SingForm,Num,DefOrIn,Adj,NP).  /* done processing np   */
76
```



```
77      np2(NP,DefOrIn,Adj,[First|Rest],Leftover) :-
78              poss(First,Base),                       /*  possessive noun     */
79              noun(SingForm,Base,Num),
80              build_np(SingForm,Num,DefOrIn,Adj,NP1), /* build its np         */
81              np2(NP,DefOrIn,[NP1],Rest,Leftover).    /* process main np      */
82
83      /*-------------------------------------------------------------------------*/
84
85      /* a pronoun may be followed by "else" (an adjective)                      */
86      after_pro([else|Rest],[else],Rest] :- !.
87
88      after_pro(Rest,Adj,Rest).               /* but usually is not.          */
89
90
91      /*-------------------------------------------------------------------------*/
92      /* For a possessive form, strip the apostrophe and return the root form
93              (returns singular or plural form)
94      */
95
96      poss(First,Base) :-
97              name(First,Name_list),
98              strip_apostrophe(Name_list,Base_list),
99              name(Base,Base_list).
100
101     strip_apostrophe([First|Rest],[]) :-
102             [First] == "'".
103
104     strip_apostrophe([First|Rest],[First|Name_rest]) :-
105             strip_apostrophe(Rest,Name_rest).
106
107     /*-------------------------------------------------------------------------*/
108
109     noun(Word,Word,sing) :- noun_(Word,_).          /* Parts of speech */
110     noun(SingForm,Word,plu)  :- noun_(SingForm,Word).
111     noun(Word,Word,prop) :- proper_(Word).
112
113     pronoun(Word,Person,Num) :- pro_(Word,Person,Num).
114
115     adj(Word,Cat) :- adj_(Word,Cat).
116
117     det(Word,DefOrIn) :- det_(Word,DefOrIn).
118
119
120     guess_def_or_in(Noun,prop,def) :- !.            /* Proper nouns are def   */
121     guess_def_or_in(Noun,_,Def_or_In) :-            /* Instances are definite */
122             Def_or_In == def, !.
123     guess_def_or_in(Noun,plu,indef) :- !.           /* Plurals are indefinite */
124     guess_def_or_in(_,_,indef).                     /* All else is indefinite */
125
126     /*-------------------------------------------------------------------------*/
127
128     build_np(Noun,Num,DefOrIn,Adj,NP) :-
129             NP = np(Noun,Num,DefOrIn,Adj).
```

```
1
2       /* routines that help in handling noun phrases and translating
3          them into frames and slot values.
4       */
5
6       /*-------------------------------------------------------------------------*/
7       /*      1st argument:  the name of a sentence frame
8               2nd argument:  the name of a syntactic slot in that frame
9               3rd argument:  a noun phrase structure.
10          Convert the noun phrase structure into a noun phrase frame and store
11          its name in this slot in this sentence frame.
12      */
13      add_noun_inst(Curr_sent,Slot,NP) :-
14              np(Word,Num,Def_or_Indef,Adj) = NP,
15              make_inst(Word,Instance),
16              add_slot_val(Curr_sent,Slot,Instance),
17              add_slot_val(Instance,reference,Def_or_Indef),
18              add_slot_val(Instance,number,Num),
19              add_descriptors(Instance,Adj), !.
20
21      /* A noun frame may be rejected by a slot due to a type inconsistency (typing
22              is used to help control semantic interpretation).  If this happens then
23              this rule will cause the frame and instance created in the above rule
24              to be abolished.
25      */
26      add_noun_inst(Curr_sent,Slot,NP) :-
27              np(Word,_,_,_) = NP,
28              return_name(Word,Instance),
29              retract(frame(Instance:_)),
30              abolish_instance(Word), !,
31              fail.
32
33      /*-------------------------------------------------------------------------*/
34      /*  create a frame where the first argument is a frame name, and the second
35      argument is a unique instance of that name. ex: 1st arg: student; 2nd arg:
36      student1.
37      */
38      make_inst(Frame,Instance) :-
39              get_name(Frame,Instance),       /* gen a unique version of "Frame" */
40              def_frame(Instance: [
41                                   in_of : [value : [Frame]]
42                                  ]).
43
44      /*-------------------------------------------------------------------------*/
45      add_descriptors(Noun_frame,[]) :- !.
46
47      add_descriptors(Noun_frame,[Adj|Rest]) :-
48              adj_(Adj,Type),
49              add_slot_val(Noun_frame,Type,Adj),
50              add_descriptors(Noun_frame,Rest).
51
52      add_descriptors(Noun_frame,[NP|Rest]) :-
53              np(_,_,_,_) = NP,
54              add_noun_inst(Noun_frame,possessor,NP),
55              add_descriptors(Noun_frame,Rest).
56
57      /*-------------------------------------------------------------------------*/
58      abolish_instance(Word) :-
59              pop_reg(Word,Val),
60              NewVal is Val - 1,
61              push_reg(Word,NewVal).
```

```
153  interrog_(where).
154
155
156  /* special forms -- contractions        */
157
158  contr_['can''t',can,not).
159  contr_('didn''t',did,not).
160  contr_('don''t',do,not).
161  contr_('couldn''t',could,not).
162  contr_('wouldn''t',would,not).
163  contr_('shouldn''t',should,not).
164  contr_('won''t',will,not).
165  contr_('it''s',it,is).
166
167
168  /*
169          Verb entries:
170            Arguments:
171                  First:  the infinitive form;
172                  Second: third person singular, tenseless;
173                  Third:  past tense;
174                  Fourth: past participle;
175                  Fifth:  present participle;
176  */
177
178  verb_(believe,believes,believed,believed,believing).
179  verb_(study,studies,studied,studied,studying).
180  verb_(pass,passes,passed,passed,passing).
181  verb_(schedule,schedules,scheduled,scheduled,scheduling).
182  verb_(take,takes,took,taken,taking).
183  verb_(give,gives,gave,given,giving).
184  verb_(have,has,had,had,having).
185  verb_(advise,advises,advised,advised,advising).
186  verb_(warn,warns,warned,warned,warning).
187  verb_(touch,touches,touched,touched,touching).
188  verb_(burn,burns,burned,burned,burning).
189  verb_(afford,affords,afforded,afforded,affording).
190  verb_(buy,buys,bought,bought,buying).
191  verb_(go,goes,went,gone,going).
192  verb_(rob,robs,robbed,robbed,robbing).
193  verb_(finish,finishes,finished,finished,finishing).
194  verb_(improve,improves,improved,improved,improving).
195  verb_(wish,wishes,wished,wished,wishing).
196  verb_(complete,completes,completed,completed,completing).
197  verb_(cheat,cheats,cheated,cheated,cheating).
198  verb_(do,does,did,done,doing).
199  verb_(risk,risks,risked,risked,risking).
200  verb_(expel,expels,expelled,expelled,expelling).
201  verb_(enroll,enrolls,enrolled,enrolled,enrolling).
202  verb_(entail,entails,entailed,entailed,entailing).
203  verb_(catch,catches,caught,caught,catching).
204  verb_(contradict,contradicts,contradicted,contradicted,contradicting).
205  verb_(pursue,pursues,pursued,pursued,pursuing).
206  verb_(learn,learns,learned,learned,learning).
207  verb_(fail,fails,failed,failed,failing).
208  verb_(know,knows,knew,known,knowing).
209  verb_(get,gets,got,gotten,getting).
210  verb_(forget,forgets,forgot,forgotten,forgetting).
211
212
213  auxverb_(have,has,had,had,having).
214  auxverb_(do,does,did,done,doing).
215
216  modal(shall,[future,s1_pl]).
217  modal(will,[future,gen]).
218  modal(would,[subjunctive,gen]).
219  modal(should,[subjunctive,gen]).
220  modal(can,[subjunctive,gen]).
221  modal(cannot,[subjunctive,gen]).
222  modal(could,[subjunctive,gen]).
223  modal(may,[subjunctive,gen]).
224
225  be_verb_(be,[infinitive,gen]).
226  be_verb_[am,[present,s1]).
227  be_verb_(is,[present,s3]).
228  be_verb_(are,[present,s2_p]).
```



```
229  be_verb_(was,[past,s1_3]).
230  be_verb_(were,[past,s2_p]).
231  be_verb_(been,[pastpart,gen]).
232  be_verb_(being,[prespart,gen]).
233
234
235  /*-------------------------------------------------------------------------*/
236  /*
237          Intermediate routines that are able to return the infinitive form
238          and the characteristics of the particular verb form used.
239                  first argument:  the infinitive form;
240                  second argument:  the actual form used;
241                  third argument: a list:
242                                          first element:  name of the form;
243                                          second element:  number & person.
244  */
245
246  verb(Form,Form,[infinitive,gen]) :-
247          verb_(Form,_,_,_,_).
248
249  verb(Inf,Form,[present,s3]) :-
250          verb_(Inf,Form,_,_,_).
251
252  verb(Inf,Form,[past_or_pastpart,gen]) :-
253          verb_(Inf,_,Form,Form,_), !.
254
255  verb(Inf,Form,[past,gen]) :-
256          verb_(Inf,_,Form,_,_).
257
258  verb(Inf,Form,[pastpart,gen]) :-
259          verb_(Inf,_,_,Form,_).
260
261  verb(Inf,Form,[prespart,gen]) :-
262          verb_(Inf,_,_,_,Form).
263
264  auxverb(Inf,Inf,[infinitive,gen]) :-
265          auxverb_(Inf,_,_,_,_).
266
267  auxverb(Inf,Form,[present,s3]) :-
268          auxverb_(Inf,Form,_,_,_).
269
270  auxverb(Inf,Form,[past_or_pastpart,gen]) :-
271          auxverb_(Inf,_,Form,Form,_], !.
272
273  auxverb(Inf,Form,[past,gen]) :-
274          auxverb_(Inf,_,Form,_,_).
275
276  auxverb(Inf,Form,[pastpart,gen]) :-
277          auxverb_(Inf,_,_,Form,_).
278
279  auxverb(Inf,Form,[prespart,gen]) :-
280          auxverb_(Inf,_,_,_,Form).
```

```
interrog_(where).


/* special forms -- contractions        */

contr_('can''t',can,not).
contr_('didn''t',did,not).
contr_('don''t',do,not).
contr_('couldn''t',could,not).
contr_('wouldn''t',would,not).
contr_('shouldn''t',should,not].
contr_('won''t',will,not).
contr_('it''s',it,is].


/*
        Verb entries:
          Arguments:
                First:  the infinitive form;
                Second: third person singular, tenseless;
                Third:  past tense;
                Fourth: past participle;
                Fifth:  present participle;
*/

verb_(believe,believes,believed,believed,believing).
verb_(study,studies,studied,studied,studying).
verb_(pass,passes,passed,passed,passing).
verb_(schedule,schedules,scheduled,scheduled,scheduling).
verb_(take,takes,took,taken,taking).
verb_(give,gives,gave,given,giving).
verb_(have,has,had,had,having).
verb_(advise,advises,advised,advised,advising).
verb_(warn,warns,warned,warned,warning).
verb_(touch,touches,touched,touched,touching).
verb_(burn,burns,burned,burned,burning).
verb_(afford,affords,afforded,afforded,affording).
verb_(buy,buys,bought,bought,buying).
verb_(go,goes,went,gone,going).
verb_(rob,robs,robbed,robbed,robbing).
verb_(finish,finishes,finished,finished,finishing).
verb_(improve,improves,improved,improved,improving).
verb_(wish,wishes,wished,wished,wishing).
verb_(complete,completes,completed,completed,completing).
verb_(cheat,cheats,cheated,cheated,cheating).
verb_(do,does,did,done,doing).
verb_(risk,risks,risked,risked,risking).
verb_(expel,expels,expelled,expelled,expelling).
verb_(enroll,enrolls,enrolled,enrolled,enrolling).
verb_(entail,entails,entailed,entailed,entailing).
verb_(catch,catches,caught,caught,catching).
verb_(contradict,contradicts,contradicted,contradicted,contradicting).
verb_(pursue,pursues,pursued,pursued,pursuing).
verb_(learn,learns,learned,learned,learning).
verb_(fail,fails,failed,failed,failing).
verb_(know,knows,knew,known,knowing).
verb_(get,gets,got,gotten,getting).
verb_(forget,forgets,forgot,forgotten,forgetting).


auxverb_(have,has,had,had,having).
auxverb_(do,does,did,done,doing).

modal(shall,[future,s1_pl]).
modal(will,[future,gen]).
modal(would,[subjunctive,gen]).
modal(should,[subjunctive,gen]).
modal(can,[subjunctive,gen]).
modal(cannot,[subjunctive,gen]).
modal(could,[subjunctive,gen]].
modal(may,[subjunctive,gen]).

be_verb_(be,[infinitive,gen]).
be_verb_(am,[present,s1]).
be_verb_(is,[present,s3]).
be_verb_(are,[present,s2_p]).
```



```
be_verb_(was,[past,s1_3]).
be_verb_(were,[past,s2_p]).
be_verb_(been,[pastpart,gen]).
be_verb_(being,[prespart,gen]].


/*-------------------------------------------------------------------------*/
/*
        Intermediate routines that are able to return the infinitive form
        and the characteristics of the particular verb form used.
                first argument:  the infinitive form;
                second argument:  the actual form used;
                third argument: a list:
                                        first element:  name of the form;
                                        second element:  number & person.
*/

verb(Form,Form,[infinitive,gen]) :-
        verb_(Form,_,_,_,_).

verb(Inf,Form,[present,s3]) :-
        verb_(Inf,Form,_,_,_).

verb(Inf,Form,[past_or_pastpart,gen]] :-
        verb_(Inf,_,Form,Form,_), !.

verb(Inf,Form,[past,gen]) :-
        verb_(Inf,_,Form,_,_).

verb(Inf,Form,[pastpart,gen]) :-
        verb_(Inf,_,_,Form,_).

verb(Inf,Form,[prespart,gen]) :-
        verb_(Inf,_,_,_,Form).

auxverb(Inf,Inf,[infinitive,gen]) :-
        auxverb_(Inf,_,_,_,_).

auxverb(Inf,Form,[present,s3]) :-
        auxverb_(Inf,Form,_,_,_).

auxverb(Inf,Form,[past_or_pastpart,gen]) :-
        auxverb_(Inf,_,Form,Form,_), !.

auxverb(Inf,Form,[past,gen]) :-
        auxverb_(Inf,_,Form,_,_).

auxverb(Inf,Form,[pastpart,gen]) :-
        auxverb_(Inf,_,_,Form,_).

auxverb(Inf,Form,[prespart,gen]) :-
        auxverb_(Inf,_,_,_,Form).
```

```
1
2     /* indic -- checks to see if the words in a sentence are known
3             to the system.  The "check_words" rule makes sure all the
4             words in the sentence are in the dictionary somewhere.
5             For each word that is not in the dictionary, the user is
6             queried for a word to replace that word.  The original word is
7             discarded, and the new word is added to the list if it is found to
8             be in the dictionary.
9
10            While this check is being performed, contractions are replaced with
11            their full word equivalents.
12    */
13
14    /*  check_words:
15            first parameter is the input list;
16            second parameter is the processed list.
17    */
18
19    check_words([W|Rest],[W|NRest]) :-
20            in_dic(W), !,
21            check_words(Rest,NRest).
22
23    check_words([W|Rest],[W1,W2|NRest]) :-
24            contr(W,W1,W2), !,
25            check_words(Rest,NRest).
26
27    check_words([W|Rest],[NewW|NRest]) :-
28            repl_word(W,NewW), !,
29            check_words([NewW|Rest],[NewW|NRest]).
30
31    check_words([],[]).
32
33
34    in_dic(W) :-
35            pro_(W,_,_);
36            noun_(W,_);
37            noun_(_,W);
38            prop_(W);
39            adj_(W,_);
40            adverb_(W);
41            time_(W);
42            det_(W,_);
43            prep_(W);
44            conj_(W);
45            negative_(W);
46            cond_(W);
47            conseq_(W);
48            verb(_,W,_);
49            be_verb_(W,_);
50            modal(W,_);
51            auxverb(_,W,_);
52            interrog_(W);
53            W == there;
54            W == that;
55            punctuation(W).
56
57    in_dic(W) :-
58            poss(W,Y),      /* the possessive form of a noun is not explicitly */
59            ( noun_(Y,_);   /*  listed in the dictionary, so check its root form. */
60              noun_(_,Y);
61               prop_(Y);
62                pro_(Y,_,_) ).
63
64    contr(Word,W1,W2) :-            /* use dic entry to replace contraction
65                                           with full form */
66            contr_(Word,W1,W2).
67
68
69    repl_word(W,NewW) :-
70                            /* allow the user the opportunity to
71                                    replace an unknown (usually a mis-
72                                    spelled) word.  */
73            write_sent([i,do,not,know,W,'.  ',replace,':']),
74            read_sent(Ans),
75            get_first_word(Ans,NewW).
76
```



```
77    get_first_word([First|_],First).
```

```
1     /*-------------------------------------------------------------------------*/
2     /*                                                                         */
3     /*                                                                         */
4     /*      The routines contained in this file are responsible for maintaining
5             a 3 cell buffer and a list of remaining items.  Input is a list of
6             items (either atoms, or Prolog structures); Output is a globally
7             defined structure consisting of three cells and a simple list of
8             those items that do not currently fit in the buffer.  A buffer
9             cell is defined as:
10                        buff(Cell,Item)
11            where "Cell" is an integer between 1 and 3,
12            and Item is a Prolog atom or structure.
13            The state of this structure may be altered either by:
14                        removing an item from a cell,
15                        inserting an item into a cell,
16                        or switching one cell item with another.
17    */
18
19
20    /*-------------------------------------------------------------------------*/
21    /* create and fill the buffer with items.                                  */
22
23    fill_buff(4,Rest,Rest) :- !.            /* buff is full  */
24
25    fill_buff(Slot,[Next|Rest],Rest) :-             /* input is done */
26            punctuation(Next),      /* because we've reached final punctuation */
27            assert(buff(Slot,Next)),
28            NewSlot is Slot + 1,
29            empty_slots(NewSlot), !.
30
31    fill_buff(Slot,[Next|Rest],Leftover) :- /* init buffer with words read  */
32            assert(buff(Slot,Next)),
33            NewSlot is Slot + 1,
34            fill_buff(NewSlot,Rest,Leftover).
35
36    empty_slots(4) :- !.                    /* 3 slots made */
37
38    empty_slots(Slot) :-                    /* if original input < 3 items */
39            assert(buff(Slot,Empty)),       /* make empty cells            */
40            NewSlot is Slot + 1,
41            empty_slots(NewSlot).
42
43    /*-------------------------------------------------------------------------*/
44    /* The following routines insert an item into the buffer. */
45
46    insert_buff(3,Item) :-
47            slide_right(3,Item), !.
48    insert_buff(Cell,Item) :-
49            retract(buff(Cell,Save_item)),  /* save current item in that cell */
50            assert(buff(Cell,Item)),        /* put new item in                */
51            NewSlot is Cell + 1,            /* pass saved item to next cell   */
52            slide_right(NewSlot,Save_item), !.
53
54    slide_right(3,Item) :-
55            retract(buff(3,Save_item)),     /* done moving right - put last    */
56            assert(buff(3,Item)),           /* item back on front of input list */
57            put_back(Save_item).
58
59    slide_right(Slot,Item) :-               /* move buff items right to make   */
60            retract(buff(Slot,Save_item)),  /*  room for an item dropped  into */
61            assert(buff(Slot,Item)),        /*   the buffer                    */
62            NextSlot is Slot + 1,
63            slide_right(NextSlot,Save_item).
64
65    /* put an item pushed out of the buffer back onto the input list */
66
67    put_back(Item) :-
68            remaining(Leftover),
69            retract(remaining(Leftover)),
70            assert(remaining([Item|Leftover])), !.
71
72    get_next(Item) :-
73            retract(remaining([Item|Rest])),
74            assert(remaining(Rest)).
75
76    get_next(Item) :-
```



```
77            remaining(List),
78            List == [],
79            Item = empty.
80
81    /*-------------------------------------------------------------------------*/
82
83    /* The following routines delete an item from the buffer. */
84
85    remove_buff_item(Cell) :-               /* remove an item from the buffer */
86            retract(buff(Cell,Item)),
87            NextSlot is Cell + 1,           /* and fill the empty cell by     */
88            slide_left(NextSlot), !.        /* moving everything to the left  */
89
90    slide_left(4) :-                /* all items have been moved over 1 slot */
91            get_next(Next),         /* get next input item  */
92            assert(buff(3,Next)), !.  /* and put it in third slot - can be empty*/
93
94    slide_left(Slot) :-                     /* move buff items to the left to  */
95            buff(Slot,Item),                /* fill an empty cell              */
96            retract(buff(Slot,Item)),
97            NewSlot is Slot - 1,
98            assert(buff(NewSlot,Item)),
99            NextSlot is Slot + 1,
100           slide_left(NextSlot).
101
102   /*-------------------------------------------------------------------------*/
103   /*      Switch buffer cell items.                                          */
104
105   switch(Cell_X,Cell_Y) :-
106           retract(buff(Cell_X,Item1)),
107           retract(buff(Cell_Y,Item2)),
108           assert(buff(Cell_Y,Item1)),
109           assert(buff(Cell_X,Item2)).
110
```

```
/* case_analysis:
        first parameter is a processed sentence frame.

        Create a case frame to represent the semantic meaning of this
        sentence.
*/

case_analysis(Curr_sent) :-
        get_slot_val(Curr_sent,verb,Verb),
        create_case_frame(Case_frame,Verb),
        add_slot_val(Curr_sent,case_frame,Case_frame),
        add_slot_val(Case_frame,part_of,Curr_sent),
        return_value_from_slot(Case_frame,cases,Cases),
        continue_case_analysis(Curr_sent,Case_frame,Cases).

create_case_frame(Case_frame,Verb) :-
        get_name(case_frame,Case_frame),        /* make a unique name */
        def_frame(Case_frame:[
                        is_a:
                                [value:[Verb],
                                max:1]
                        ]).

/*-------------------------------------------------------------------------
        continue_case_analysis:
                1st parameter:  a sentence frame;
                2nd parameter:  its case frame representation;
                3rd parameter:  a list of cases to be filled.

        These rules look in the sentence frame for the proper value
        to fill a case.  Where more than one syntactic slot may
        contain the value, and there are no syntactic clues (such
        as voice) to aid in placement, the frame based typing
        mechanism will cause an incorrect candidate to be rejected,
        and the rules will eventually find the correct candidate, or
        will stop searching, and then the parent verb frame's
        default values will be used for that case.
-------------------------------------------------------------------------*/

continue_case_analysis(Curr_sent,Case_frame,[]) :- !.

/*-------------------------------------------------------------------------*/

continue_case_analysis(Curr_sent,Case_frame,[actor|Remaining_cases]) :-
        get_slot_val(Curr_sent,act_pass,active),
        get_slot_val(Curr_sent,subject,Actor),
        Actor \== nil,
        add_slot_val(Case_frame,actor,Actor),
        continue_case_analysis(Curr_sent,Case_frame,Remaining_cases).

continue_case_analysis(Curr_sent,Case_frame,[actor|Remaining_cases]) :-
        get_slot_val(Curr_sent,act_pass,passive),
        get_slot_val(Curr_sent,by,Actor),
        Actor \== nil,
        add_slot_val(Case_frame,actor,Actor),
        continue_case_analysis(Curr_sent,Case_frame,Remaining_cases).

continue_case_analysis(Curr_sent,Case_frame,[actor|Remaining_cases]) :-
        get_slot_val(Curr_sent,act_pass,active),
        get_slot_val(Curr_sent,subject,nil),
        continue_case_analysis(Curr_sent,Case_frame,Remaining_cases).

continue_case_analysis(Curr_sent,Case_frame,[actor|Remaining_cases]) :-
        get_slot_val(Curr_sent,act_pass,passive),
        get_slot_val(Curr_sent,by,nil),
        continue_case_analysis(Curr_sent,Case_frame,Remaining_cases).

/*-------------------------------------------------------------------------*/

continue_case_analysis(Curr_sent,Case_frame,[believer|Remaining_cases]) :-
        get_slot_val(Curr_sent,act_pass,active),
        get_slot_val(Curr_sent,subject,Believer),
        Believer \== nil,
        add_slot_val(Case_frame,believer,Believer),
        continue_case_analysis(Curr_sent,Case_frame,Remaining_cases).
```



```
continue_case_analysis(Curr_sent,Case_frame,[believer|Remaining_cases]) :-
        get_slot_val(Curr_sent,act_pass,passive),
        get_slot_val(Curr_sent,by,Believer),
        Believer\== nil,
        add_slot_val(Case_frame,believer,Believer),
        continue_case_analysis(Curr_sent,Case_frame,Remaining_cases).

continue_case_analysis(Curr_sent,Case_frame,[believer|Remaining_cases]) :-
        get_slot_val(Curr_sent,act_pass,active),
        get_slot_val(Curr_sent,subject,nil),
        continue_case_analysis(Curr_sent,Case_frame,Remaining_cases).

continue_case_analysis(Curr_sent,Case_frame,[believer|Remaining_cases]) :-
        get_slot_val(Curr_sent,act_pass,passive),
        get_slot_val(Curr_sent,by,nil),
        continue_case_analysis(Curr_sent,Case_frame,Remaining_cases).

/*-------------------------------------------------------------------------*/

continue_case_analysis(Curr_sent,Case_frame,[object|Remaining_cases]) :-
        get_slot_val(Curr_sent,act_pass,active),
        get_slot_val(Curr_sent,object,Object),
        Object \== nil,
        add_slot_val(Case_frame,object,Object),
        continue_case_analysis(Curr_sent,Case_frame,Remaining_cases).

continue_case_analysis(Curr_sent,Case_frame,[object|Remaining_cases]) :-
        get_slot_val(Curr_sent,act_pass,active),
        get_slot_val(Curr_sent,about,Object),
        Object \== nil,
        add_slot_val(Case_frame,object,Object),
        continue_case_analysis(Curr_sent,Case_frame,Remaining_cases).

continue_case_analysis(Curr_sent,Case_frame,[object|Remaining_cases]) :-
        get_slot_val(Curr_sent,act_pass,passive),
        get_slot_val(Curr_sent,subject,Object),
        Object \== nil,
        add_slot_val(Case_frame,object,Object),
        continue_case_analysis(Curr_sent,Case_frame,Remaining_cases).

continue_case_analysis(Curr_sent,Case_frame,[object|Remaining_cases]) :-
        get_slot_val(Curr_sent,act_pass,passive),
        get_slot_val(Curr_sent,object,Object),
        Object \== nil,
        add_slot_val(Case_frame,object,Object),
        continue_case_analysis(Curr_sent,Case_frame,Remaining_cases).

continue_case_analysis(Curr_sent,Case_frame,[object|Remaining_cases]) :-
        get_slot_val(Curr_sent,act_pass,active),
        get_slot_val(Curr_sent,object,nil),
        continue_case_analysis(Curr_sent,Case_frame,Remaining_cases).

continue_case_analysis(Curr_sent,Case_frame,[object|Remaining_cases]) :-
        get_slot_val(Curr_sent,act_pass,passive),
        get_slot_val(Curr_sent,subject,nil),
        continue_case_analysis(Curr_sent,Case_frame,Remaining_cases).

/*-------------------------------------------------------------------------*/

continue_case_analysis(Curr_sent,Case_frame,[recipient|Remaining_cases]) :-
        get_slot_val(Curr_sent,act_pass,active),
        get_slot_val(Curr_sent,ind_obj,Recipient),
        Recipient \== nil,
        add_slot_val(Case_frame,recipient,Recipient),
        continue_case_analysis(Curr_sent,Case_frame,Remaining_cases).

continue_case_analysis(Curr_sent,Case_frame,[recipient|Remaining_cases]) :-
        get_slot_val(Curr_sent,act_pass,active),
        get_slot_val(Curr_sent,for,Recipient),
        Recipient \== nil,
        add_slot_val(Case_frame,recipient,Recipient),
        continue_case_analysis(Curr_sent,Case_frame,Remaining_cases).

continue_case_analysis(Curr_sent,Case_frame,[recipient|Remaining_cases]) :-
        get_slot_val(Curr_sent,act_pass,passive),
        get_slot_val(Curr_sent,subject,Recipient),
```

```
153           Recipient \== nil,
154           add_slot_val(Case_frame,recipient,Recipient),
155           continue_case_analysis(Curr_sent,Case_frame,Remaining_cases).
156
157   continue_case_analysis(Curr_sent,Case_frame,[recipient|Remaining_cases]) :-
158           get_slot_val(Curr_sent,act_pass,passive),
159           get_slot_val(Curr_sent,ind_obj,Recipient),
160           Recipient \== nil,
161           add_slot_val(Case_frame,recipient,Recipient),
162           continue_case_analysis(Curr_sent,Case_frame,Remaining_cases).
163
164   continue_case_analysis(Curr_sent,Case_frame,[recipient|Remaining_cases]) :-
165           get_slot_val(Curr_sent,act_pass,passive),
166           get_slot_val(Curr_sent,for,Recipient),
167           Recipient \== nil,
168           add_slot_val(Case_frame,recipient,Recipient),
169           continue_case_analysis(Curr_sent,Case_frame,Remaining_cases).
170
171   continue_case_analysis(Curr_sent,Case_frame,[recipient|Remaining_cases]) :-
172           get_slot_val(Curr_sent,act_pass,active),
173           get_slot_val(Curr_sent,ind_obj,nil),
174           continue_case_analysis(Curr_sent,Case_frame,Remaining_cases).
175
176   continue_case_analysis(Curr_sent,Case_frame,[recipient|Remaining_cases]) :-
177           get_slot_val(Curr_sent,act_pass,active),
178           get_slot_val(Curr_sent,for,nil),
179           continue_case_analysis(Curr_sent,Case_frame,Remaining_cases).
180
181   continue_case_analysis(Curr_sent,Case_frame,[recipient|Remaining_cases]) :-
182           get_slot_val(Curr_sent,act_pass,passive),
183           get_slot_val(Curr_sent,subject,nil),
184           continue_case_analysis(Curr_sent,Case_frame,Remaining_cases).
185
186   /*-------------------------------------------------------------------------*/
187
188   continue_case_analysis(Curr_sent,Case_frame,[objective|Remaining_cases]) :-
189           get_slot_val(Curr_sent,for,Objective),
190           Objective\== nil,
191           add_slot_val(Case_frame,objective,Objective),
192           continue_case_analysis|Curr_sent,Case_frame,Remaining_cases).
193
194   continue_case_analysis(Curr_sent,Case_frame,[objective|Remaining_cases]) :-
195           get_slot_val(Curr_sent,object,Objective),
196           Objective\== nil,
197           add_slot_val(Case_frame,objective,Objective),
198           continue_case_analysis(Curr_sent,Case_frame,Remaining_cases).
199
200   continue_case_analysis(Curr_sent,Case_frame,[objective|Remaining_cases]) :-
201           get_slot_val(Curr_sent,for,nil),
202           continue_case_analysis(Curr_sent,Case_frame,Remaining_cases).
203
204   /*-------------------------------------------------------------------------*/
205
206   continue_case_analysis(Curr_sent,Case_frame,[time|Remaining_cases]) :-
207           get_slot_val(Curr_sent,time,Time),
208           Time \== nil,
209           add_slot_val(Case_frame,time,Time),
210           continue_case_analysis(Curr_sent,Case_frame,Remaining_cases).
211
212   continue_case_analysis(Curr_sent,Case_frame,[time|Remaining_cases]) :-
213           get_slot_val(Curr_sent,act_pass,active),
214           get_slot_val(Curr_sent,time,nil),
215           continue_case_analysis(Curr_sent,Case_frame,Remaining_cases).
216
217   /*-------------------------------------------------------------------------*/
218
219   continue_case_analysis(Curr_sent,Case_frame,[action|Remaining_cases]) :-
220           get_slot_val(Curr_sent,act_pass,active),
221           get_slot_val(Curr_sent,subject,Action),
222           Action \== nil,
223           add_slot_val(Case_frame,action,Action),
224           continue_case_analysis(Curr_sent,Case_frame,Remaining_cases).
225
226   continue_case_analysis(Curr_sent,Case_frame,[action|Remaining_cases]) :-
227           get_slot_val(Curr_sent,act_pass,passive),
228           get_slot_val(Curr_sent,object,Action),
```



```
229           Action \== nil,
230           add_slot_val(Case_frame,action,Action),
231           continue_case_analysis(Curr_sent,Case_frame,Remaining_cases).
232
233   continue_case_analysis(Curr_sent,Case_frame,[action|Remaining_cases]) :-
234           get_slot_val(Curr_sent,act_pass,active),
235           get_slot_val(Curr_sent,subject,nil),
236           continue_case_analysis(Curr_sent,Case_frame,Remaining_cases).
237
238   continue_case_analysis(Curr_sent,Case_frame,[action|Remaining_cases]) :-
239           get_slot_val(Curr_sent,act_pass,passive),
240           get_slot_val(Curr_sent,object,nil),
241           continue_case_analysis(Curr_sent,Case_frame,Remaining_cases).
242
243   /*-------------------------------------------------------------------------*/
244
245   continue_case_analysis(Curr_sent,Case_frame,[Failed|Remaining_cases]) :-
246           continue_case_analysis(Curr_sent,Case_frame,Remaining_cases).
```

```
/* verb definitions  */

:- def_frame(verb : [
                        description : [value : [language_input]]]).

/*-------------------------------------------------------------------------*/
:- def_frame(action_verb : [
                        is_a: [value : [verb]]]).

:- def_frame(belief_verb : [
                        is_a: [value : [verb]]]).

:- def_frame(condition_verb : [
                        is_a: [value : [verb]]]).

:- def_frame(be_verb : [
                        is_a: [value : [verb]]]).

/*-------------------------------------------------------------------------*/

:- def_frame(study : [
                            is_a : [value : [action_verb]],
                            cases : [max : 6,
                                     value : [actor,object,objective,time]],
                            actor : [type: person,
                                     value : [some_person]],
                            object : [type : knowledge_source,
                                      value : [knowledge_source]],
                            objective: [type : task,
                                  value : [some_task]],
                            time:[type: time,
                                      value:[future_time]]
                            ]).

 :- def_frame(pass : [
                            is_a : [value : [action_verb]],
                            cases : [max : 6,
                                     value : [actor,object,objective,time]],
                            actor:[type : person,
                                     value : [some_person]],
                            object : [type : task,
                                      value: [some_task]],
                            objective: [type : task,
                                  value : [some_task]],
                            time:[type: time,
                                      value:[future_time]]
                        ]).

 :- def_frame(schedule : [
                            is_a : [value : [action_verb]],
                            cases : [max : 6,
                                     value : [actor,object,recipient,time]],
                            actor:[type : person,
                                     value: [some_person]],
                            object:[type: event,
                                     value: [some_event]],
                            recipient:[type: person,
                                     value: [some_person]],
                            time:[type: time,
                                     value:[future_time]]
                       ]).

 :- def_frame(finish: [
                            is_a : [value : [action_verb]],
                            cases : [max : 6,
                                     value : [actor,objective,time]],
                            actor:[type : person,
                                     value : [some_person]],
                            objective: [type : task,
                                  value : [some_task]],
                            time:[type: time,
                                      value:[future_time]]
                        ]).

 :- def_frame(cheat: [
                            is_a : [value : [action_verb]],
```



```
                            cases : [max : 6,
                                     value : [actor,object]],
                            actor:[type : person,
                                     value : [some_person]],
                            objective: [type : task,
                                  value : [some_task]]
                       ]).

:- def_frame(catch: [
                            is_a : [value : [action_verb]],
                            cases : [max : 6,
                                     value : [actor,object]],
                            actor:[type : person,
                                     value : [some_person]],
                            object: [type : person,
                                  value : [some_person]]
                       ]).

/*-------------------------------------------------------------------------*/

:- def_frame(advise : [
                          is_a : [value : [belief_verb]],
                            cases : [max : 6,
                                     value : [believer,object,recipient]],
                          believer:[type :  person,
                                   value : [some_person]],
                          recipient: [type : person,
                                    value : [some_person]],
                          object : [type : sentence,
                                    value : [nil_sent]]
                          ]).

:- def_frame(warn : [
                          is_a : [value : [belief_verb]],
                            cases : [max : 6,
                                     value : [believer,object,recipient]],
                          believer:[type :  person,
                                   value : [some_person]],
                          recipient: [type : person,
                                    value : [some_person]],
                          object : [type : sentence,
                                    value : [nil_sent]]
                       ]).

                            :- def_frame(believe: [
                          is_a : [value : [belief_verb]],
                            cases : [max : 6,
                                     value : [believer,object,recipient]],
                          believer:[type :  person,
                                   value : [some_person]],
                          recipient: [type : person,
                                    value : [some_person]],
                          object : [type : sentence $$ knowledge_source,
                                    value : [nil_sent]]
                          ]).

:- def_frame(forget: [
                          is_a : [value : [belief_verb]],
                            cases : [max : 6,
                                     value : [believer,object]],
                          believer:[type :  person,
                                   value : [some_person]],
                          object : [type : sentence $$ entity,
                                    value : [nil_sent]]
                          ]).

:- def_frame(wish: [
                          is_a : [value : [belief_verb]],
                            cases : [max : 6,
                                     value : [believer,object]],
                          believer:[type :  person,
                                   value : [some_person]],
                          object : [type : sentence,
                                    value : [nil_sent]]
                          ]).
```

```
153  /*-------------------------------------------------------------------------*/
154
155  :- def_frame(take: [
156                          is_a : [value : [condition_verb]],
157                             cases : [max : 6,
158                                      value : [action,object]],
159                          action :[type :  embedded_sent,
160                                   value : [some_action]],
161                          object : [type : time,
162                                    value : [some_time]]
163                          ]).
164
165  /*-------------------------------------------------------------------------*/
166
167  :- def_frame(be : [
168                  is_a : [value : [be_verb]]
169                   ]).
170
```

```
1
2     :- def_frame(action :[is_a : [value: [embedded_sent]]]).
3     :- def_frame(entity :[description: [value: [any_noun]]]).
4
5     :- def_frame(some_action :[is_a: [value: [action]]]).
6
7     :- def_frame[animate : [is_a : [value : [entity]]]].
8     :- def_frame[inanimate:[is_a : [value : [entity]]]].
9     :- def_frame(social_entity:[is_a:[value:[entity]]]).
10    :- def_frame(physical_force: [is_a : [value : [entity]]]).
11    :- def_frame(event : [is_a : [value : [entity]]]).
12    :- def_frame(time : [is_a : [value : [entity]]]).
13    :- def_frame(place: [is_a : [value : [entity]]]).
14
15    :- def_frame(nil : [is_a : [value : [entity]]]).
16
17    :- def_frame[they : [is_a : [value : [animate $$ inanimate $$ social_entity]]]].
18    :- def_frame[them : [is_a : [value : [animate $$ inanimate $$ social_entity]]]].
19
20    :- def_frame(person : [is_a : [value : [animate]]]).
21    :- def_frame(animal: [is_a : [value : [animate]]]).
22    :- def_frame(vegetable: [is_a : [value : [animate]]]).
23
24    :- def_frame(thing : [is_a : [value : [inanimate]]]).
25    :- def_frame(it : [is_a : [value : [inanimate]]]).
26
27    :- def_frame(knowledge_source : [is_a : [value : [social_entity]]]).
28    :- def_frame(machine : [is_a : [value : [social_entity]]]).
29    :- def_frame(institution : [is_a : [value : [social_entity]]]).
30    :- def_frame(goal: [is_a : [value : [social_entity]]]).
31    :- def_frame(plan: [is_a : [value : [social_entity]]]).
32    :- def_frame(advice: [is_a : [value : [social_entity]]]).
33    :- def_frame(warning: [is_a : [value : [social_entity]]]).
34    :- def_frame(degree:[is_a : [value : [social_entity]]]).
35    :- def_frame(work:[is_a : [value : [social_entity]]]).
36    :- def_frame(motive:[is_a : [value : [social_entity]]]).
37
38    :- def_frame(some_event: [is_a : [value : [event]]]).
39    :- def_frame(task : [is_a : [value : [event]]]).
40
41    :- def_frame(some_time: [is_a : [value : [time]]]).
42    :- def_frame(future_time: [is_a : [value : [time]]]).
43    :- def_frame(day: [is_a : [value : [time]]]).
44    :- def_frame(today: [is_a : [value : [time]]]).
45    :- def_frame(tomorrow: [is_a : [value : [time]]]).
46    :- def_frame(yesterday: [is_a : [value : [time]]]).
47    :- def_frame(monday: [is_a : [value : [time]]]).
48    :- def_frame(tuesday: [is_a : [value : [time]]]).
49    :- def_frame(wednesday: [is_a : [value : [time]]]).
50    :- def_frame(thursday: [is_a : [value : [time]]]).
51    :- def_frame(friday: [is_a : [value : [time]]]).
52    :- def_frame(saturday: [is_a : [value : [time]]]).
53    :- def_frame(sunday: [is_a : [value : [time]]]).
54
55    :- def_frame(somewhere : [is_a : [value : [place]]]).
56
57    :- def_frame(some_person: [is_a : [value : [person]]]).
58    :- def_frame(student : [is_a : [value : [some_person]]]).
59    :- def_frame(speaker: [is_a : [value : [some_person]]]).
60    :- def_frame(hearer: [is_a : [value : [some_person]]]).
61    :- def_frame(teacher : [is_a : [value : [some_person]]]).
62    :- def_frame(he : [is_a : [value : [some_person]]]).
63    :- def_frame(she : [is_a : [value : [some_person]]]).
64    :- def_frame(i : [is_a : [value : [some_person]]]).
65    :- def_frame(me:[is_a : [value : [some_person]]]).
66    :- def_frame(you : [is_a : [value : [some_person]]]).
67    :- def_frame[we : [is_a : [value : [some_person]]]].
68    :- def_frame[myself:[is_a : [value : [some_person]]]].
69
70    :- def_frame(some_task: [is_a : [value : [task]]]).
71    :- def_frame[test: [is_a : [value : [task]]]].
72    :- def_frame(lab : [is_a : [value : [task]]]).
73    :- def_frame(course:[is_a : [value : [task]]]).
74    :- def_frame(program:[is_a : [value : [task]]]).
75
76    :- def_frame(book: [is_a : [value : [knowledge_source]]]).
```



```
77    :- def_frame(chance: [is_a : [value : [entity]]]).
78    :- def_frame(condition: [is_a : [value : [entity]]]).
79    :- def_frame(herself:[is_a : [value : [some_person]]]).
80    :- def_frame(yourself:[is_a : [value : [some_person]]]).
81    :- def_frame(himself:[is_a : [value : [some_person]]]).
82    :- def_frame(themselves:[is_a : [value : [some_person]]]).
83    :- def_frame(someone : [is_a : [value : [some_person]]]).
```

```
/*-------------------------------------------------------------------------------

        After an individual sentence has been parsed, and a case frame
        analysis has been completed, then it is time to attempt to find a
        place for that sentence in the current context.  This process
        begins by examining the type of the current sentence.

        Functor:   examine
        Argument:  Sentence -- the frame that represents the current
                sentence.
        Description:  For a given sentence frame, find the value of
                its sentence type, and pass that frame, and its
                type to the next routines.

-------------------------------------------------------------------------------*/

examine(Sentence) :-
        get_slot_val(Sentence,sent_type,Type),
        exam_continue(Sentence,Type).


/*-------------------------------------------------------------------------------

        A sentence frame's type may be:
                cond:  "if" part of a hypothetical declarative
                        (ex: "if you study for the test,")
                conseq: "then" part of a hypothetical declarative
                        (ex: "then you will pass the test.")
                imper: simple imperative
                        (ex: "Study for the test.")
                decl:  simple declarative
                        (ex: "A test has been scheduled for Monday.")
                question:  a simple question
                        (ex: Why study for the test?)
                yes_no_q:  a question seeking a 'yes' or 'no' answer
                        (ex; "Does the speaker believe that the hearer
                                wishes to pass the test?")


        Functor:   exam_continue
        1st Argument:  Sentence -- the frame that represents the current
                                sentence.
        2nd Argument:  The value of that frame's sentence type.
        Description:  For each recognized sentence type, try to find
                a place for it in a context frame.  Some conversions
                may take place (for example, a simple declarative that
                contains the modal "should" will be treated as an
                imperative).

-------------------------------------------------------------------------------*/

/*-------------------------------------------------------------------------------

        If the sentence type is a "cond" (for "conditional"), then get
        the next available Practical Argument frame whose antecedent slot
        is "undetermined", and if this conditional semantically matches
        other elements already in the frame, then place the name of this
        sentence frame in the antecedent slot of the Practical Argument
        frame.

-------------------------------------------------------------------------------*/

exam_continue(Sentence,cond) :-
        check_arg_frame(Arg_inst),
        get_slot_val(Arg_inst,antecedent,undetermined),
        match_antecedent(Arg_inst,Sentence),
        add_slot_val(Arg_inst,antecedent,Sentence).


/*-------------------------------------------------------------------------------

        If the sentence type is a "conseq" (for "consequent"), then get
        the next available Practical Argument frame whose consequent slot
        is "undetermined", and if this consequent semantically matches
        other elements already in the frame, then place the name of this
```



```
        sentence frame in the consequent slot of the Practical Argument
        frame.

-------------------------------------------------------------------------------*/
exam_continue(Sentence,conseq) :-
        check_arg_frame(Arg_inst),
        get_slot_val(Arg_inst,consequent,undetermined),
        match_consequent(Arg_inst,Sentence),
        add_slot_val(Arg_inst,consequent,Sentence).

/*-------------------------------------------------------------------------------

        If the sentence type is an "imper" (for "imperative") then get
        the next available Practical Argument frame whose imperative slot
        is "undetermined", and if this imperative semantically matches
        other elements already in the frame, then place the name of this
        sentence frame in the imperative slot of the Practical Argument
        frame.

-------------------------------------------------------------------------------*/
exam_continue(Sentence,imper) :-
        check_arg_frame(Arg_inst),
        get_slot_val(Arg_inst,imperative,undetermined),
        match_imperative(Arg_inst,Sentence),
        add_slot_val(Arg_inst,imperative,Sentence).

/*-------------------------------------------------------------------------------

        If the main verb of a simple declarative sentence is "advise" or
        "warn", and the object of that sentence is itself an independent
        clause (or "embedded sentence") then use the object clause as the
        sentence to examine, and treat it as an imperative sentence.
        An example:  "I advise you to study for the test", where "you to
        study for the test" is to be the imperative.  Note that no
        assumption is made that this example is advice rather than a
        warning.  Which it is (advice or warning) will be determined
        without regard to the explicit verb.

-------------------------------------------------------------------------------*/
exam_continue(Sentence,decl) :-
        ( get_slot_val(Sentence,verb,advise);
         get_slot_val(Sentence,verb,warn) ), !,
        get_slot_val(Sentence,object,Embedded_sent),
        frame_subsumes(sentence,Embedded_sent),
        exam_continue(Embedded_sent,imper).

/*-------------------------------------------------------------------------------

        If a simple declarative sentence contains the modal "should", and
        has a person for a subject, then treat that sentence as an imperative.

-------------------------------------------------------------------------------*/
exam_continue(Sentence,decl) :-
        get_slot_val(Sentence,modal,Modal),
        Modal == should,
        get_slot_val(Sentence,subject,Person),
        frame_subsumes(person,Person), !,
        exam_continue(Sentence,imper).

/*-------------------------------------------------------------------------------

        If a simple declarative sentence does not satisfy either of the
        special cases described above, then process it as an event, and
        place the resulting event frame on a stack of events.  When a
        practical argument is complete then these event frames will be
        checked to see if any one references an event that is also
        referenced by the practical argument.

-------------------------------------------------------------------------------*/
exam_continue(Sentence,decl) :-
        make_event_frame(Name),
        add_slot_val(Name,event,Sentence),
        event_still_to_process(Name).

/*-------------------------------------------------------------------------------
```

```
153         Attempt to answer a simple question.  Currently the system will only
154         handle the quesetion "Why?".
155
156   ------------------------------------------------------------------------------*/
157   exam_continue(Sentence,question) :-
158         answer_quest(Sentence).
159
160   exam_continue(Sentence,yes_no_q) :-
161         write_sent([last,sentence,was,a,yes_no,question,'.']).
162
163
164   /*------------------------------------------------------------------------------
165
166         Functor:  match_antecedent
167         1st Argument:  Arg_inst -- The name of the candidate Practical
168                Argument frame.
169         2nd Argument:  Antecedent -- The Sentence frame that is a potential
170                antecedent in this argument frame.
171         Description:  Match the semantic contents of the potential
172                antecedent with the semantic contents of the Arguement
173                frame's Consequent.  Do the same with the Argument
174                frame's Imperative.
175
176   ------------------------------------------------------------------------------*/
177   match_antecedent(Arg_inst,Antecedent) :-
178         match_antecedent_consequent(Arg_inst,Antecedent),
179         match_antecedent_imperative(Arg_inst,Antecedent).
180
181   /*------------------------------------------------------------------------------
182
183         Functor:  match_antecedent_consequent
184         1st Argument:  Arg_inst -- The name of the candidate Practical
185                Argument frame.
186         2nd Argument:  Antecedent -- The Sentence frame that is a potential
187                antecedent in this argument frame.
188         Description:  Match the semantic contents of the potential
189                antecedent with the semantic contents of the Arguement
190                frame's Consequent.  A match is successful if the consequent
191                slot is empty, or if a "plan" derived from the antecedent can
192                achieve a "goal" derived from the consequent.
193
194   ------------------------------------------------------------------------------*/
195   match_antecedent_consequent(Arg_inst,Antecedent) :-
196         get_slot_val(Arg_inst,consequent,undetermined).
197
198   match_antecedent_consequent(Arg_inst,Antecedent) :-
199         not get_slot_val(Arg_inst,consequent,undetermined),
200         get_plan(Antecedent,Plan),
201         get_slot_val(Arg_inst,consequent,Consequent),
202         get_goal(Consequent,Goal),
203         match_plan_goal(Plan,Goal,_,_).
204
205   /*------------------------------------------------------------------------------
206
207         Functor:  match_antecedent_imperative
208         1st Argument:  Arg_inst -- The name of the candidate Practical
209                Argument frame.
210         2nd Argument:  Antecedent -- The Sentence frame that is a potential
211                antecedent in this argument frame.
212         Description:  Match the semantic contents of the potential
213                antecedent with the semantic contents of the Arguement
214                frame's Imperative.  A match is successful if the Imperative
215                slot is empty.  When an Imperative is added to a Practical
216                Argument frame that does not have an Antecedent, the
217                Antecedent is created by inference based on the contents of
218                the Imperative.  Therefore, the system will never add an
219                explicitly stated Antecedent to a frame that already has
220                an Imperative and no matching between these two slots is
221                necessary at this time.  If this were to change then the
222                matching process would be the same as the one that occurrs
223                when an Imperative is added to a Practical Argument frame
224                that already has an Antecedent.  Strictly speaking, the
225                match that is tried here on an empty Imperative slot is
226                redundant, since the slot will always be empty in a frame
227                whose Antecedent slot is empty.
228
```



```
229   ------------------------------------------------------------------------------*/
230   match_antecedent_imperative(Arg_inst,Antecedent) :-
231         get_slot_val(Arg_inst,imperative,undetermined).
232
233
234   /*------------------------------------------------------------------------------
235
236         Functor:  match_consequent
237         1st Argument:  Arg_inst -- The name of the candidate Practical
238                Argument frame.
239         2nd Argument:  Consequent -- The Sentence frame that is a potential
240                Consequent in this argument frame.
241         Description:  Match the semantic contents of the potential
242                Consequent with the semantic contents of the Arguement
243                frame's Antecedent.  Do the same with the Argument
244                frame's Imperative.
245
246   ------------------------------------------------------------------------------*/
247
248   match_consequent(Arg_inst,Consequent) :-
249         match_consequent_antecedent(Arg_inst,Consequent),
250         match_consequent_imperative(Arg_inst,Consequent).
251
252
253   /*------------------------------------------------------------------------------
254
255         Functor:  match_consequent_antecedent
256         1st Argument:  Arg_inst -- The name of the candidate Practical
257                Argument frame.
258         2nd Argument:  Consequent -- The Sentence frame that is a potential
259                Consequent in this argument frame.
260         Description:  Match the semantic contents of the potential
261                Consequent with the semantic contents of the Arguement
262                frame's Antecedent.  A match is successful if the Antecedent
263                slot is empty, or if a "plan" derived from the Antecedent can
264                achieve a "goal" derived from the Consequent.
265
266   ------------------------------------------------------------------------------*/
267
268   match_consequent_antecedent(Arg_inst,Consequent) :-
269         get_slot_val(Arg_inst,antecedent,undetermined).
270
271   match_consequent_antecedent(Arg_inst,Consequent) :-
272         not get_slot_val(Arg_inst,antecedent,undetermined),
273         get_slot_val(Arg_inst,antecedent,Antecedent),
274         get_plan(Antecedent,Plan),
275         get_goal(Consequent,Goal),
276         match_plan_goal(Plan,Goal,_,_).
277
278
279   /*------------------------------------------------------------------------------
280
281         Functor:  match_consequent_imperative
282         1st Argument:  Arg_inst -- The name of the candidate Practical
283                Argument frame.
284         2nd Argument:  Consequent-- The Sentence frame that is a potential
285                Consequent in this argument frame.
286         Description:  Match the semantic contents of the potential
287                Consequent with the semantic contents of the Arguement
288                frame's Imperative.  A match is successful if the Imperative
289                slot is empty.  When an Imperative is added to a Practical
290                Argument frame that does not have a Consequent, the
291                Consequent is created by inference based on the contents of
292                the Imperative.  Therefore, the system will never add an
293                explicitly stated Consequent to a frame that already has
294                an Imperative and no matching between these two slots is
295                necessary at this time.  If this were to change then the
296                matching process would be the same as the one that occurrs
297                when an Imperative is added to a Practical Argument frame
298                that already has a Consequent.  Strictly speaking, the
299                match that is tried here on an empty Imperative slot is
300                redundant, since the slot will always be empty in a frame
301                whose Consequent slot is empty.
302
303   ------------------------------------------------------------------------------*/
304
```

```
match_consequent_imperative(Arg_inst,Consequent) :-
        get_slot_val(Arg_inst,imperative,undetermined).


/*-------------------------------------------------------------------------

        Functor:  match_imperative
        1st Argument:  Arg_inst -- The name of the candidate Practical
                Argument frame.
        2nd Argument:  Imperative -- The Sentence frame that is a potential
                imperative in this argument frame.
        Description:  Match the semantic contents of the potential
                imperative with the semantic contents of the Arguement
                frame's Antecedent.  Do the same with the Argument
                frame's Consequent.

-------------------------------------------------------------------------*/
match_imperative(Arg_inst,Imperative) :-
        match_imperative_antecedent(Arg_inst,Imperative),
        match_imperative_consequent(Arg_inst,Imperative).

/*-------------------------------------------------------------------------

        Functor:  match_imperative_antecedent
        1st Argument:  Arg_inst -- The name of the candidate Practical
                Argument frame.
        2nd Argument:  Imperative -- The Sentence frame that is a potential
                imperative in this argument frame.
        Description:  Match the semantic contents of the potential
                imperative with the semantic contents of the Arguement
                frame's Antecedent.  A match is successful if the Antecedent
                slot is empty, or if a "plan" derived from the Antecedent can
                match a "plan" derived from the Imperative.

-------------------------------------------------------------------------*/

match_imperative_antecedent(Arg_inst,Imperative) :-
        get_slot_val(Arg_inst,antecedent,undetermined).

match_imperative_antecedent(Arg_inst,Imperative) :-
        not get_slot_val(Arg_inst,antecedent,undetermined),
        get_slot_val(Arg_inst,antecedent,Antecedent),
        get_plan(Antecedent,Plan1),
        get_plan(Imperative,Plan2),
        match_plan_plan(Plan1,Plan2,_,_,_).


/*-------------------------------------------------------------------------

        Functor:  match_imperative_consequent
        1st Argument:  Arg_inst -- The name of the candidate Practical
                Argument frame.
        2nd Argument:  Imperative -- The Sentence frame that is a potential
                imperative in this argument frame.
        Description:  Match the semantic contents of the potential
                imperative with the semantic contents of the Arguement
                frame's Consequent.  A match is successful if the consequent
                slot is empty, or if a "plan" derived from the imperative can
                achieve a "goal" derived from the consequent.

-------------------------------------------------------------------------*/
match_imperative_consequent(Arg_inst,Imperative) :-
        get_slot_val(Arg_inst,consequent,undetermined).

match_imperative_consequent(Arg_inst,Imperative) :-
        not get_slot_val(Arg_inst,consequent,undetermined),
        get_plan(Imperative,Plan),
        get_slot_val(Arg_inst,consequent,Consequent),
        get_goal(Consequent,Goal),
        match_plan_goal(Plan,Goal,_,_).

/*-------------------------------------------------------------------------

        Functor:  get_plan
        1st Argument:  Sentence -- the name of a sentence frame (input).
        2nd Argument:  Plan -- a list of three elements representing a plan
```



```
                (output).
        Description:  If a sentence frame has a case frame that contains or
                can inherit values for its "is_a", "actor", and "objective"
                slots, then return the values of those slots as a plan
                contained in this sentece frame.  If any of these conditions
                are not met, fail.

-------------------------------------------------------------------------*/

get_plan(Sentence,Plan) :-
        get_slot_val(Sentence,case_frame,Case_frame),
        get_slot_val(Case_frame,is_a,Action),
        get_slot_val(Case_frame,actor,Actor),
        get_slot_val(Case_frame,objective,Objective),
        Plan = [Action,Actor,Objective].


/*-------------------------------------------------------------------------

        Functor:  get_goal
        1st Argument:  Sentence -- the name of a sentence frame (input).
        2nd Argument:  Goal -- a list of three elements representing a goal
                (output).
        Description:  If a sentence frame has a case frame that contains or
                can inherit values for its "is_a", "actor", and "object"
                slots, then return the values of those slots as a goal
                contained in this sentece frame.  If any of these conditions
                are not met, fail.

-------------------------------------------------------------------------*/

get_goal(Sentence,Goal) :-
        get_slot_val(Sentence,case_frame,Case_frame),
        get_slot_val(Case_frame,actor,Actor),
        get_slot_val(Case_frame,is_a,Action),
        get_slot_val(Case_frame,object,Object),
        Goal = [Action,Actor,Object].


/*-------------------------------------------------------------------------

        Functor:  match_plan_plan
        1st Argument: A list of three elements representing a plan.
        2nd Argument: A list of three elements representing a plan.
        3rd Argument: An atom representing an action common to those plans.
        4th Argument:  An actor common to those plans.
        5th Argument:  An objective common to those plans.
        Description:  Checks to see if the two plans are semantically
                equivalent.  First we must know that each is a plan, so
                they are looked up in a knowledge base of plans.  Next, they
                are compared to one another, and if each slot is semantically
                equivalent, a common value for that slot is returned.
                For example:
                        1st plan:  [study,some_person,test1]
                        2nd plan:  [study,you1,test2]
                        Action:  study
                        Actor:  you1
                        Objective:  test1.

-------------------------------------------------------------------------*/
match_plan_plan([P_action1,P_Actor1,P_Objective1],
                [P_action2,P_Actor2,P_Objective2],Action,Actor,Objective) :-
        activate_plan(_,P_action1,P_Actor1,P_Objective1,P_Actor1),
        activate_plan(_,P_action2,P_Actor2,P_Objective2,P_Actor2),
        unify(P_action1,P_action2,Action),
        unify(P_Actor1,P_Actor2,Actor),
        unify(P_Objective1,P_Objective2,Objective).

/*-------------------------------------------------------------------------

        Functor:  match_plan_goal
        1st Argument: A list of three elements representing a plan.
        2nd Argument: A list of three elements representing a goal.
        3rd Argument: A list of three elements representing a plan whose
                "actor" and "objective" elements are common to the plan and
                the goal.
```

```
457         4th Argument: A list of three elements representing a goal whose
458                 "actor" and "objective" elements are common to the plan and
459                 the goal.
460         Description:  Checks to see if the plan can achive the goal.
461                 First each must be known to the system, so
462                 they are looked up in a knowledge base.  Next, a check is
463                 performed to see if the plan is known as a way of achieving
464                 the goal.  Lastly, the Actor and the Objective values are
465                 compared to one another, and if each is semantically equivalent,
466                 a common value is returned for each.
467                 For example:
468                         plan:  [study,some_person,test1]
469                         goal:  [pass,you1,test2]
470                         new plan:  [study,you1,test1]
471                         new goal:  [pass,you1,test1]
472
473   ------------------------------------------------------------------------*/
474   match_plan_goal([P_action,P_Actor,P_Objective],
475                           [G_action,G_Actor,G_Objective],
476                           [P_action,Actor,Objective],
477                           [G_action,Actor,Objective]) :-
478           activate_plan(P_index,P_action,P_Actor,P_Objective,P_Actor), /* known */
479           activate_goal(G_index,G_action,G_Actor,G_Objective,G_Actor), /* known */
480           valid_plan(P_index,G_index),  /* this plan will achieve this goal */
481           unify(P_Actor,G_Actor,Actor),
482           unify(P_Objective,G_Objective,Objective).
483
484   /*------------------------------------------------------------------------
485
486           Functor:  activate_plan
487           1st Argument:  An Integer used to match plans with appropriate goals.
488           2nd Argument:  A predicate that is the plan's 'action'.
489           3rd Argument:  A noun which represents the actor or agent of the plan.
490           4th Argument:  A noun which represents the plan's objective.
491           5th Argument:  A noun representing the planner (identical to the actor).
492           Description:  Looks up a proposed plan in a knowledge base of general
493                   plans, and succeeds if it finds such a plan.  Could also be
494                   used to generate plans of a general nature.
495
496   ------------------------------------------------------------------------*/
497   activate_plan(Index,Action,Actor,Objective,Actor) :-
498           plan(Index,Action,Actor_type,Objective_type,planner(Actor_type)),
499           unify(Objective_type,Objective,_),
500           unify(Actor_type,Actor,_).
501
502   /*------------------------------------------------------------------------
503
504           Functor:  activate_goal
505           1st Argument:  An Integer used to match goals with appropriate plans.
506           2nd Argument:  A predicate that is the goal's 'action'.
507           3rd Argument:  A noun which represents the actor or agent of the goal.
508           4th Argument:  A noun which represents the goal's object.
509           5th Argument:  A noun representing the person who has the goal
510                   (identical to the actor).
511           Description:  Looks up a proposed goal in a knowledge base of general
512                   goals, and succeeds if it finds such a goal.  Is also be
513                   used to generate goals of a general nature.
514
515   ------------------------------------------------------------------------*/
516
517   activate_goal(Index,Action,Actor,Object,Actor) :-
518           goal(Index,Action,Actor_type,Object_type,has_goal(Actor_type)),
519           unify(Object_type,Object,_),
520           unify(Actor_type,Actor,_).
521
522   /*------------------------------------------------------------------------
523
524           Functor:  unify
525           1st Argument:   an atom that is the name of a frame.
526           2nd Argument:   an atom that is the name of a frame.
527           3rd Argument:   the name of a frame that represents the
528                   more specific instance or the lowest subclass that can
529                   represent the first and second arguments.
530           Description:  "Unify" checks to see if its first and second
531                   arguments are semantically equivalent, and if they are
532                   it returns the one which carries the most information
```



```
533                 in its third argument.  They are semantically equivalent
534                 if both are identical, or if either is a type of the other,
535                 or if both are instances of the same noun.  Note:  they
536                 are also semantically equivalent if one or the other, or
537                 both are not specified.
538
539   ------------------------------------------------------------------------*/
540   unify(Name,Name,Name).          /* The first and second args are identical. */
541
542   unify(Name1,Name2,Name) :-
543           frame_subsumes(Name1,Name2),  /* The 2nd arg is a type of the 1st.  */
544           Name = Name2.                 /* Return the more specific 2nd arg.  */
545
546   unify(Name1,Name2,Name) :-
547           frame_subsumes(Name2,Name1),  /* The 1st arg is a type of the 2nd.  */
548           Name = Name1.                 /* Return the more specific 1st arg.  */
549
550   unify(Name1,Name2,Name) :-
551           get_slot_val(Name1,in_of,Noun), /* The 1st and 2nd args are both   */
552           get_slot_val(Name2,in_of,Noun), /*  instances of the same noun.    */
553           Name = Name1.                   /* Arbitrarily return the 1st arg. */
554
555           /* Note that the "in_of" link will not follow the network beyond
556             the first link, so the two candidates must match on the first try.
557             No matches will succeed on any common higher type.  */
558
```

```
/*-------------------------------------------------------------------------

 Definitions of the top level context frames.

-------------------------------------------------------------------------*/

:- def_frame(has_goal:{}).
:- def_frame(no_goal:{}).
:- def_frame(complete:{}).
:- def_frame(incomplete:{}).
:- def_frame(undetermined:{}).


:- def_frame(argument:
                {
                description:  {value : {practical_argument}},
                event       : { min: 0,
                                max: 1,
                                type: event_frame $$ undetermined,
                                value : {undetermined}},
                antecedent  : { min: 0,
                                max: 1,
                                type: sentence $$ undetermined,
                                value : {undetermined}},
                consequent  : { min: 0,
                                max: 1,
                                type: sentence $$ undetermined,
                                value : {undetermined}},
                imperative : { min: 0,
                                max: 1,
                                type: sentence $$ undetermined,
                                value : {undetermined}},
                speech_act  : { min: 0,
                                max: 1,
                                type: advice $$ warning $$ undetermined,
                                value : {undetermined}},
                arg_status  : { min: 0,
                                max: 1,
                                type: complete $$ incomplete,
                                value : {incomplete}}
                        }).


/*-------------------------------------------------------------------------

        Functor:  init_arg_frame
        Argument:  Name -- returns a frame name (ex. "argument1")
        Description:  Create an instance of a Practical Argument frame.
                This frame will have a unique name composed of "argument"
                as the root, and an incremented counter.

-------------------------------------------------------------------------*/

init_arg_frame(Name) :-
        get_name(argument,Name),  /* generate a unique name for the frame  */
        def_frame(Name:{
                        in_of : {value : {argument}},
                     /* event : {}, */
                     /* antecedent : {}, */
                        consequent : {if_added: {look_for_event_goal}},
                        imperative: {if_added : {check_antecedent,
                                                 check_consequent,
                                                 determine_speech_act
                                                }},
                        speech_act : {if_added: {check_argument_complete}},
                        arg_status :   {if_added : report}
                        }).


/*-------------------------------------------------------------------------

        Functor: look_for_event_goal
        1st Argument:  Arg_frame -- the name of the frame containing the
                slot that possesses this demon.
        3rd Argument:  Consequent -- the name of the sentence frame that
                is the Consequent in this Argument frame.
```



```
        Description:  When a sentence frame fills the Consequent slot,
                this demon is activated.  It forms a goal from the Consequent,
                and attempts to match it with a goal belonging to any already
                existing event frame.  If a successful match is made, this
                demon then adds the event frame's name to the event slot
                of the Practical Argument frame containing this Consequent.

-------------------------------------------------------------------------*/
look_for_event_goal(Arg_frame,_,Consequent) :-
        get_goal(Consequent,Goal1),
        find_active_goal(Event,Goal2),
        match_goal_goal(Goal1,Goal2),
        add_slot_val(Arg_frame,event,Event).

look_for_event_goal(_,_,_).

/*-------------------------------------------------------------------------

        Functor:  match_goal_goal
        1st Argument:  A list of three elements representing a goal.
        2nd Argument:  A list of three elements representing a goal.
        Description:  The two goals are equivalent if the actions are
                identical, and if the subjects and objects can be unified.
                (Unification requires either they are identical, or they
                are instances of the same noun, or one is a type of the
                other.)

-------------------------------------------------------------------------*/
match_goal_goal([Action1,$subject1,Object1],[Action2,$subject2,Object2]) :-
        Action1 == Action2,
        unify(Object1,Object2,Object),
        unify($subject1,$subject2,$subject).

/*-------------------------------------------------------------------------

        Functor:  find_active_goal
        1st Argument:  An event frame.
        2nd Argument:  A goal (a three element list).
        Description:  Get the next event from the event list, and if it
                contains a goal, return both the event frame's name and
                the goal to the calling function.

-------------------------------------------------------------------------*/

find_active_goal(Event,Goal) :-
        event_process_list(List),
        member(Event,List),
        get_slot_val(Event,status,has_goal),
        get_slot_val(Event,goal,Goal).


/*-------------------------------------------------------------------------

        Functor:  check_antecedent
        1st Argument:  Arg_frame -- the name of the frame containing the
                slot that possesses this demon.
        3rd Argument:  Imperative -- the name of the sentence frame that
                is the Imperative in this Argument frame.
        Description:  When a value is added to the Imperative slot, this
                demon will check to see if this frame has an Antecedent.
                If it is missing, an attempt will be made to infer the
                Antecedent.

-------------------------------------------------------------------------*/

check_antecedent(Arg_frame,_,_) :-
        not get_slot_val(Arg_frame,antecedent,undetermined), !.

check_antecedent(Arg_frame,_,Imperative) :-
        get_slot_val(Arg_frame,antecedent,undetermined), !,
        infer_antecedent(Arg_frame,Imperative),
        get_slot_val(Arg_frame,antecedent,Antecedent),
        match_antecedent_consequent(Arg_frame,Antecedent).

infer_antecedent(Arg_frame,Imperative) :-
        get_slot_val(Arg_frame,antecedent,undetermined),
```

```
153         get_plan(Imperative,Plan),
154         [_,_,Objective] = Plan,
155         return_slot_name(Imperative,Objective_slot,Objective),
156         make_inference(Plan,Objective_slot,Inference),
157                                         /* make it into a sentence  */
158         add_slot_val(Arg_frame,antecedent,Inference).
159
160   /*-------------------------------------------------------------------------
161
162         Functor:  check_consequent
163         1st Argument:  Arg_frame -- the name of the frame containing the
164                 slot that possesses this demon.
165         3rd Argument:  Imperative -- the name of the sentence frame that
166                 is the Imperative in this Argument frame.
167         Description:  When a value is added to the Imperative slot, this
168                 demon will check to see if this frame has a Consequent.
169                 If it is missing, an attempt will be made to infer the
170                 Consequent.
171
172   -------------------------------------------------------------------------*/
173
174   check_consequent(Arg_frame,_,_) :-
175         not get_slot_val(Arg_frame,consequent,undetermined), !.
176
177   check_consequent(Arg_frame,_,Imperative) :-
178         get_slot_val(Arg_frame,consequent,undetermined), !,
179         infer_consequent(Arg_frame,Imperative),
180         get_slot_val(Arg_frame,consequent,Consequent),
181         match_consequent_antecedent(Arg_frame,Consequent).
182
183   infer_consequent(Arg_frame,Imperative) :-
184         get_slot_val(Arg_frame,consequent,undetermined),
185         get_plan(Imperative,Plan),
186         match_plan_goal(Plan,_,_,Specific_goal), /* generate a goal */
187         make_inference(Specific_goal,object,Inference),
188                                         /* make it into a sentence  */
189         add_slot_val(Arg_frame,consequent,Inference).
190
191   /*-------------------------------------------------------------------------
192
193         Functor:  make_inference
194         1st Argument:  A list of three atoms representing a verb, a noun
195                 intended to be the subject, and a second noun intended to
196                 be the object of the verb or of a preposition.  (The list
197                 represents either a plan or a goal.)
198         2nd Argument:  The name of the slot that will be created for the object.
199         3rd Argument:  A unique name for the sentence frame that will be
200                 created.
201         Description:  Generate a unique name for a sentence frame to fill
202                 the third argument, and pass that name together with the
203                 values of the first two arguemnts, to a routine that
204                 will create a sentence frame instantiated with that
205                 information.  Also indicate that this sentence frame
206                 is to be an instance of a "context" frame (a
207                 type of sentence) and its sentence type is to be recorded
208                 as an "inference".  After this sentence frame has been
209                 created, do a case analysis of its contents (which creates
210                 a case frame linked to the sentence frame).
211
212   -------------------------------------------------------------------------*/
213
214   make_inference([Action,Subject,Object],Objective_slot,Name) :-
215         get_name(sent,Name),
216         make_context_frame(Name,context,inference,Subject,
217                         Action,Object,Objective_slot),
218         (case_analysis(Name);
219                 write('case analysis failed')).
220
221   /*-------------------------------------------------------------------------
222
223         Functor:  make_context_frame
224         Arguments:  The name of a frame together with those values and slot
225                 names that may vary with each call to this routine.
226         Description:  Create a frame, using the values in the argument list.
227
228   -------------------------------------------------------------------------*/
```



```
229
230   make_context_frame(Name,Parent,Type,Subject,
231                         Action,Object,Object_slot) :-
232         def_frame(Name : [
233                         in_of : [value : [Parent]],
234                         sent_type:[value:[Type]],
235                         subject:[value : [Subject]],
236                         verb:[value : [Action]],
237                         Object_slot :[value : [Object]]
238                         ]).
239
240   /*-------------------------------------------------------------------------
241
242         Functor:  determine_speech_act
243         1st Argument:  Arg_frame -- the name of the frame containing the
244                 slot that possesses this demon.
245         3rd Argument:  Imperative -- the name of the sentence frame that
246                 is the Imperative in this Argument frame.
247         Description:  When a sentence frame fills the imperative slot,
248                 this demon attempts to determine the type of speech act
249                 that this Practical Argument frame represents.  This
250                 is done by comparing the antecedent and the imperative.
251                 If both are positive or both are negative, then the type
252                 of speech act is "advice".  If one is positive and the
253                 other is negative, then the speech act is marked as a
254                 "warning".  If anything causes this demon to fail, then
255                 leave the speech act marked as "undetermined".
256
257   -------------------------------------------------------------------------*/
258
259   determine_speech_act(Arg_frame,_,Imperative) :-
260         not get_slot_val(Arg_frame,antecedent,undetermined),
261         get_slot_val(Arg_frame,antecedent,Antecedent),
262         match_pos_neg(Imperative,Antecedent,Speech_Act),
263         add_slot_val(Arg_frame,speech_act,Speech_Act).
264
265   /*-------------------------------------------------------------------------
266
267         Functor:  match_pos_neg
268         1st Argument:  A sentence frame.
269         2nd Argument:  A sentence frame.
270         3rd Argument:  The result returned from the attempt to match them
271                 based on their status as positive or negative sentences.
272         Description:  From each of two sentences, retrieve the values of their
273                 "pos_neg" slots and compare them, returning the result of
274                 that match.
275
276   -------------------------------------------------------------------------*/
277
278   match_pos_neg(Imperative,Antecedent,advice) :-
279         get_slot_val(Imperative,pos_neg,Pos_Neg),
280         get_slot_val(Antecedent,pos_neg,Pos_Neg).
281
282   match_pos_neg(Imperative,Antecedent,warning) :-
283         get_slot_val(Imperative,pos_neg,Pos_Neg1),
284         get_slot_val(Antecedent,pos_neg,Pos_Neg2),
285         Pos_Neg1 \== Pos_Neg2.
286
287
288   /*-------------------------------------------------------------------------
289
290         Functor:  check_argument_complete
291         1st Argument:  Arg_frame -- the name of the Practical Argument frame
292                 that contains this demon.
293         3rd Argument:  Speech_act -- the value of the slot containing this
294                 demon.
295         Description:  Once the type of speech act has been determined, then
296                 check to make sure that all of the slots listed have values.
297                 If they do all have values, then mark the frame as complete.
298                 (Note:  the "event" slot is not a pat of the logical
299                 argument, and does not have to contain a value for the
300                 frame to be considered as complete.)
301
302   -------------------------------------------------------------------------*/
303   check_argument_complete(Arg_frame,_,Speech_act) :-
304   not get_slot_val(Arg_frame,arg_status,complete),
```

```
(Speech_act == advice ;
        Speech_act == warning),
not get_slot_val(Arg_frame,antecedent,undetermined),
not get_slot_val(Arg_frame,consequent,undetermined),
not get_slot_val(Arg_frame,imperative,undetermined),
add_slot_val(Arg_frame,arg_status,complete).

check_argument_complete(_,_,_).

/*-------------------------------------------------------------------------

        Functor:  report
        Argument:  Frame -- the name of the frame containing the slot that
                possesses this demon.
        Description:  When all the pertinent slots in  a Practical Argument
                frame have values, then the frame's status slot will be
                set to indicate that the frame is complete, and this demon
                will start the process of finding a place for this frame
                in the overall context.  If there are no problems to report
                then the current Practical Argument frame will be paraphrased
                and output to the user.

-------------------------------------------------------------------------*/

report(Frame,_,_) :-
        reply(Frame).


/*-------------------------------------------------------------------------

        Functor:  check_arg_frame
        Argument:  Instance -- the name of a Practical Argument frame.
        Description:  Returns the name of a Practical Argument frame whose
                status is incomplete, and can therefore accept more in-
                formation.

-------------------------------------------------------------------------*/

check_arg_frame(Instance) :-
        get_arg_frame(Instance),
        get_slot_val(Instance,arg_status,incomplete).


/*-------------------------------------------------------------------------

        Functor:  get_arg_frame
        Argument:  Instance -- the name of a Practical Argument frame.
        Description:  Returns the next Practical Argument frame from a LIFO
                list of such frames.  If the list is empty or exhausted then it
                creates a new argument frame, returns it, and adds it to
                the list.

-------------------------------------------------------------------------*/

get_arg_frame(Instance) :-
        arg_process_list(Arg_list),
        member(Instance,Arg_list).

get_arg_frame(Instance) :-
        init_arg_frame(Instance),
        still_to_process(Instance).


/*-------------------------------------------------------------------------

        Functor:  still_to_process
        Argument:  Frame -- the name of a Practical Argument frame.
        Description:  Retrieves the list of Practical Argument frames, and
                adds a new frame to the front of that list.  If that list does
                not exist, it creates the list.

-------------------------------------------------------------------------*/

still_to_process(Frame) :-
        retract(arg_process_list(OldList)),
        append([Frame],OldList,NewList),
```



```
        assert(arg_process_list(NewList)), !.

still_to_process(Frame) :-
        assert(arg_process_list([Frame])), !.


/*-------------------------------------------------------------------------
-------------------------------------------------------------------------*/
:- def_frame(event_frame:
                        [
                        event  : [ min: 0,
                                   max: 1,
                                   type: sentence],
                        goal   : [ min: 0,
                                   max: 1],
                        status : [ min: 0,
                                   max: 1,
                                   type: has_goal $$ no_goal]
                        ]).

/*-------------------------------------------------------------------------

        A simple declarative sentence does not belong to a practical
        argument, but does reference an event in the "real" world
        to which a practical argument may refer.

        Functor:  make_event_frame
        Argument:  Name -- a unique name generated for this event.
        Description:  Create an instance of an Event frame.
                This frame will have a unique name composed of "event_frame"
                as the root, and an incremented counter.

-------------------------------------------------------------------------*/

make_event_frame(Name) :-
        get_name(event_frame,Name),
        def_frame(Name:[
                        in_of : [value: [event_frame]],
                        event: [if_added : infer_goal]
                        ]).


/*-------------------------------------------------------------------------

        Demon:  Whenever an event frame is created, an attempt is
                made to find and attach a goal to it.  For example:
                if a test is scheduled, we can assume that whoever takes
                the test will have the goal of passing the test.  A goal,
                as it appears here, is more specific than the general know-
                ledge base goal since it references a specific test, rather
                than any test or task.

        Functor:  infer_goal
        1st Argument:  Event_frame -- the name of the frame containing
                the slot that possesses this demon.
        3rd Argument:  Sentence -- the name of the  sentence frame that
                is the value of the slot that possesses this demon.
        Description:  From the Case frame attached to this sentence, get
                the values of the Object and the Recipient cases, and look
                for a knowledge base goal that can accept the Recipient value
                as its Agent, and the Object value as its Object.  If this is
                successful, then create a list of three elements, where the
                first element is the predicate belonging to the knowledge
                base goal, the second element is the value of the Recipient
                case of the current sentence, and the third element is the
                value of the Object case of the current sentence.  Put this
                list in the "goal" slot of the current Event frame, and let
                its status slot indicate that this event has a goal attached
                to it.  If this rule fails for any reason, then let the
                Event frame's status slot indicate that there is no goal
                associated with this event.

-------------------------------------------------------------------------*/

infer_goal(Event_frame,_,Sentence) :-
        get_slot_val(Sentence,case_frame,Case_analysis),
```

```
457           get_slot_val(Case_analysis,object,Object),
458           get_slot_val(Case_analysis,recipient,Person),
459           activate_goal(_,Goal_action,Person,Object,Person),
460           Goal = [Goal_action,Person,Object],
461           add_slot_val(Event_frame,goal,Goal),
462           add_slot_val(Event_frame,status,has_goal).
463
464   infer_goal(Frame,Slot,Value) :-
465           add_slot_val(Frame,status,no_goal).
466
467
468   /*-------------------------------------------------------------------------
469
470           Functor:  event_still_to_process
471           Argument:  Frame -- the name of an Event frame.
472           Description:  Retrieves the list of Event frames, and
473                   adds a new frame to the end of that list.  If that list does
474                   not exist, it creates the list.
475
476   -------------------------------------------------------------------------*/
477
478   event_still_to_process(Frame) :-
479           retract(event_process_list(OldList)),
480           append([Frame],OldList,NewList),
481           assert(event_process_list(NewList)), !.
482
483   event_still_to_process(Frame) :-
484           assert(event_process_list([Frame])), !.
485
```

```
1
2       plan(1,study,person,test,planner(person)).
3       goal(1,pass,person,test,has_goal(person)).
4
5       plan(2,finish,person,lab,planner(person)).
6       goal(2,pass,person,lab,has_goal(person)).
7
8       valid_plan(I,I) :-
9       plan(I,P_action,Actor,Objective,planner(Actor)),
10      goal(I,G_action,Actor,Objective,has_goal(Actor)).
```

```
1
2       /* If "Frame" is a Practical Argument frame, then paraphrase it and write
3               its paraphrase to the terminal.  */
4       reply(Frame) :-
5               frame_subsumes(argument,Frame), !,
6               paraphrase(argument,Frame,Sent),
7               append(Sent,['.'],Reply),
8               write_sent(Reply).
9
10      paraphrase(argument,Arg_Frame,Sent) :-
11              set_curr_sent_frame(System_reply),
12              add_slot_val(System_reply,sent_type,sys_reply),
13              add_slot_val(System_reply,refers,Arg_Frame),
14              get_slot_val(Arg_Frame,imperative,Sent_Reference),
15              get_slot_val(Sent_Reference,case_frame,Case_frame),
16              subject(argument,Arg_Frame,System_reply,Sent,Case_frame).
17
18      subject(argument,Arg,Curr_sent,Sent,Case_frame) :-
19              get_slot_val(Case_frame,actor,Hearer),
20              frame_subsumes(you,Hearer),
21              Noun = speaker,
22              make_np(Noun,def,sing,[],Subject),
23              verb_form(Arg,Curr_sent,Subject,present_perfect,Sent_fragment),
24              append(Subject,Sent_fragment,Sent).
25
26      verb_form(Arg,Curr_sent,Subject,present_perfect,Sent) :-
27              get_slot_val(Arg,speech_act,Speech_act),
28              get_root(Speech_act,Root),
29              Aux = have,
30              verb(Root,Verb,[pastpart,_]),
31              VP1 = [Aux,Verb],
32              agree(Subject,VP1,VP),
33              ind_obj(Arg,Curr_sent,Sent_fragment),
34              add_slot_val(Curr_sent,verb,Root),
35              np(NP,Subject,_),
36              add_noun_inst(Curr_sent,subject,NP),
37              append(VP,Sent_fragment,Sent).
38
39      get_root(advice,advise).
40      get_root(warning,warn).
41
42      agree(Subject,Verb_form,VP) :-
43              get_noun(Subject,Noun),
44              num(Noun,sing),
45              person(Noun,p3),
46              [Aux,Verb] = Verb_form,
47              auxverb(Aux,Aux_form,[present,s3]),
48              VP = [Aux_form,Verb].
49
50
51      ind_obj(Arg,Curr_sent,Sent) :-
52              get_slot_val(Arg,imperative,Sent1),
53              get_slot_val(Sent1,case_frame,Case_frame),
54              get_slot_val(Case_frame,actor,Val),
55              make_generic(Val,Hearer),
56              Hearer == you,
57              Noun = hearer,
58              make_np(Noun,def,sing,[],For_to),
59              object(Arg,Curr_sent,Sent_fragment),
60              np(NP,For_to,_),
61              add_noun_inst(Curr_sent,ind_obj,NP),
62              append(For_to,Sent_fragment,Sent).
63
64      object(Arg,Curr_sent,Sent) :-
65              get_slot_val(Arg,imperative,Sent1),
66              get_slot_val(Sent1,case_frame,Case_frame),
67              get_slot_val(Case_frame,is_a,Action),
68              get_slot_val(Case_frame,objective,Objective),
69              return_slot_name(Sent1,Slot_name,Objective),
70              make_generic(Objective,Noun),
71              make_np(Noun,indef,sing,[],NP),
72              add_slot_val(Curr_sent,object,Sent1),
73              form_obj_phrase(Slot_name,Action,Obj_phrase),
74              append(Obj_phrase,NP,Object),
75              append([],Object,Sent).
76
```



```
77      form_obj_phrase(object,Action,[to,Action]).
78
79      form_obj_phrase(Slot_name,Action,[to,Action,Slot_name]).
```

```
1
2       make_np(Noun,Ref,Num,Modifiers,NP) :-
3               det_(Det,Ref),
4               noun(Noun,Form,Num),
5               NP = [Det,Form].
6
7       get_noun(NP,Noun) :-
8               last(Noun,NP).
9
10      num(Noun,Num) :-
11              noun(_,Noun,Num).
12
13      person(Noun,p3) :-
14              not pronoun(Noun,_,_).
15
16
17      /*-------------------------------------------------------------------------
18      -------------------------------------------------------------------------*/
19      make_generic(Instance,Class) :-
20              name(Instance,Name_list),
21              strip_num(Name_list,Class_list),
22              name(Class,Class_list), !.
23
24      strip_num([],[]).
25
26      strip_num([First|Rest],[]) :-
27              First >= "0", First =< "9".
28
29      strip_num([First|Rest],[First|Name_rest]) :-
30              strip_num(Rest,Name_rest).
31
```

```
1
2     answer_quest(Question) :-
3             get_slot_val(Question,interrogative,why),
4             get_slot_val(Question,refers,Reference),
5             frame_subsumes(argument,Reference),
6             derive(Question,Reference,Belief),
7             paraphrase_belief(Belief,Sentence),
8             append(Sentence,['.'],Answer),
9             write_sent(Answer).
10
11    derive(Question,Reference,Belief) :-
12            return_goal(Reference,Goal),
13            return_plan(Reference,Plan),
14            return_believer(Question,Believer),
15            return_believe_subj(Question,Subj),
16            make_belief_frame(Believer,Subj,Plan,Goal,Belief).
17
18    return_goal(Frame,Goal) :-
19            get_slot_val(Frame,consequent,Consequent),
20            get_slot_val(Consequent,case_frame,Case_frame),
21            get_slot_val(Case_frame,actor,Actor),
22            get_slot_val(Case_frame,is_a,Action),
23            get_slot_val(Case_frame,objective,Objective),
24            Goal = (Action,Actor,Objective].
25
26    return_plan(Frame,Plan) :-
27            get_slot_val(Frame,antecedent,Antecedent),
28            get_slot_val(Antecedent,case_frame,Case_frame),
29            get_slot_val(Case_frame,actor,Actor),
30            get_slot_val(Case_frame,is_a,Action),
31            get_slot_val(Case_frame,objective,Objective),
32            Plan = (Action,Actor,Objective).
33
34    return_believer(Question,Believer) :-
35            get_slot_val(Question,case_frame,Case_frame),
36            get_slot_val(Case_frame,believer,Believer).
37
38    return_believe_subj(Question,Subj) :-
39            get_slot_val(Question,case_frame,Case_frame),
40            get_slot_val(Case_frame,recipient,Subj).
41
42    make_belief_frame(Believer,Subj,Plan,Goal,Belief) :-
43            get_name(belief,Belief),
44            def_frame(Belief:{
45                              is_a : {value: {belief}},
46                              believer:{value:{Believer}},
47                              subject:{value:{Subj}},
48                              plan:{value:{Plan}},
49                              goal:{value:[Goal]}
50                             }).
51
52
53    paraphrase_belief(Belief,Sent) :-
54            make_generic(Belief,Type),
55            Type == belief,
56            subject_2(Belief,Sent).
57
58    subject_2(Belief,Sent) :-
59            get_slot_val(Belief,believer,Believer),
60            make_generic(Believer,Noun),
61            make_np(Noun,def,sing,{},NP),
62            verb_form_2(Belief,NP,present,Sent_fragment),
63            append(NP,Sent_fragment,Sent).
64
65    verb_form_2(Belief,NP,Tense,Sentence) :-
66            make_generic(Belief,Type),
67            Type == belief,
68            Verb = believe,
69            agree_2(NP,Verb,Tense,VP),
70            object_subj(Belief,Sent_fragment),
71            append([VP],Sent_fragment,Sentence).
72
73    agree_2(Subject,Verb,present,VP) :-
74            get_noun(Subject,Noun),
75            num(Noun,sing),
76            person(Noun,p3),
```



```
77            verb(Verb,Form,[present,s3]),
78            VP = Form.
79
80    agree_2(Subject,Verb,future,VP) :-
81            get_noun(Subject,Noun),
82            not pronoun(Noun,_,_),
83            Aux = will,
84            VP = [Aux,Verb].
85
86    object_subj(Belief,Sentence) :-
87            get_slot_val(Belief,subject,Subject),
88            make_generic(Subject,Noun),
89            make_np(Noun,def,sing,{},NP),
90            object_goal(Belief,NP,future,Sent_fragment),
91            append(NP,Sent_fragment,Sentence).
92
93    object_goal(Belief,Subject,future,Sent_fragment) :-
94            get_slot_val(Belief,goal,Goal),
95            (Action,Subject2,Object1) = Goal,
96            agree_2(Subject,Action,future,VP),
97            make_generic(Object1,Object),
98            make_np(Object,def,sing,{},Obj_NP),
99            object_plan(Belief,Subject,Obj_NP,Sent1,Sent),
100           append(Obj_NP,Sent1,Sent2),
101           append(VP,Sent2,Sent_fragment).
102
103   object_plan(Belief,Subject,Obj_NP,Sent_fragment,Sent) :-
104           get_slot_val(Belief,plan,Plan),
105           (Action,_,Object) = Plan,
106           get_noun(Obj_NP,Obj_goal),
107           agree_2(Subject,Action,present,VP),
108           append([for],Obj_NP,For_to),
109           append([VP],For_to,VP_For_to),
110           append(Subject,VP_For_to,S_VP_For_to),
111           append([if],S_VP_For_to,Sent_fragment).
```

```
1
2     /* Read a sentence (from pp. 87-88)
3             Returns as its argument a list of atoms built from strings of
4             non-blank characters read from standard input.  Upper case letters
5             are mapped to lower case and "words" may contain "-", "_", "'" or
6             digits.  Punctuation to separate words includes ";", ":" and ",".
7             Punctuation to end sentences includes ".", "!" and "?".
8     */
9
10    read_sent([First|Rest]) :-
11            get0(C),                        /* Priming read                 */
12            read_word(C,First,NextC),       /* Read first word              */
13            rest_sent(First,NextC,Rest).    /* Read rest of words           */
14
15                                            /* Given a word and the character after
16                                               it, read the rest of the sentence */
17    rest_sent(First,_,[]) :-
18            last_word(First), !.            /* Handle end of sentence       */
19    rest_sent(First,C,[NextW|Rest]) :-
20            read_word(C,NextW,C1),          /* Read next word               */
21            rest_sent(NextW,C1,Rest).       /* Read rest of words           */
22
23
24            /* Given an initial character "C", read a single word "Word"
25               and remember what character "NextC" came after the word.     */
26
27    read_word(C,Word,NextC) :-              /* Handle punctuation           */
28            single_char(C), !,
29            name(Word,[C]),
30            get0(NextC).
31    read_word(C,Word,NextC) :-              /* Handle words                 */
32            in_word(C,MapC), !,
33            get0(C1),
34            rest_word(C1,Rest,NextC),
35            name(Word,[MapC|Rest]).
36    read_word(C,Word,NextC) :-              /* Eat white space              */
37            get0(C1),
38            read_word(C1,Word,NextC).
39
40    rest_word(C,[MapC|Rest],NextC) :-       /* Handle rest of chars         */
41            in_word(C,MapC), !,
42            get0(C1),
43            rest_word(C1,Rest,NextC).
44    rest_word(C,[],C).                      /* Handle last char in word     */
45
46
47                            /* These characters form words on their own     */
48    /* single_char(44). */            /* , */
49    single_char(59).                  /* ; */
50    single_char(58).                  /* : */
51    single_char(63).                  /* ? */
52    single_char(33).                  /* ! */
53    single_char(46).                  /* . */
54
55
56    in_word(C,C) :-             /* These characters can appear within a word    */
57            C>="a", C=<"z".
58    in_word(C,L) :-
59            C>="A", C=<"Z",
60            L is C-"A"+"a".           /* Map upper to lower case              */
61    in_word(C,C) :-
62            C>="0", C=<"9".
63    in_word(39,39).                   /* ' */
64    in_word(45,45).                   /* - */
65    in_word(95,95).                   /* _ */
66
67
68    last_word('.').                   /* These words terminate a sentence     */
69    last_word('!').
70    last_word('').
71
72    punctuation(',').                 /* Punctuation atoms                    */
73    punctuation(';').
74    punctuation(':').
75    punctuation('!').
76    punctuation('.').
```



```
77
78    /*    Writes a sentence nicely.
79            The first word in the list is capitalized and a newline is put
80            at the end of the sentence.
81            arguments : list - represents words in a sentence
82    */
83
84
85    write_sent([]) :- nl.                   /* Handle null sentence         */
86    write_sent([First|Rest]) :-
87            f_cap(First,CapFirst),          /* Capitalize first word        */
88            write(CapFirst),
89            f_write_rest(Rest).
90
91    f_write_rest([]) :- nl.                 /* Handle end of sentence       */
92    f_write_rest([Next|Rest]) :-
93            f_punctuation(Next), !,         /* No space before punctuation  */
94            write(Next),
95            f_write_rest(Rest).
96    f_write_rest([Next|Rest]) :-
97            tab(1),                         /* Space before a word          */
98            write(Next),
99            f_write_rest(Rest).
100
101   f_cap(Word,CapWord) :-
102           name(Word,[First|Rest]),        /* Explode word to be Cap'ed    */
103           f_caplet(First,NewFirst),       /* Capitalize first letter      */
104           name(CapWord,[NewFirst|Rest]).  /* Put cap'ed word back together*/
105
106   f_caplet(C,MapC) :-                     /* Capitalize lower case letter */
107           C>="a", C=<"z", !,
108           MapC is C-"a"+"A".
109   f_caplet(C,C).
```

```
1
2     member(X,[X|_]).                  /* Succeeds if 1st argument is found   */
3     member(X,[_|Rest]) :-             /*  in 2nd argument, which is a list.  */
4             member(X,Rest).
5
6
7     append([],L,L).                   /* append 1st arg to the 2nd arg       */
8     append([X|L1],L2,[X|L3]) :-       /*  and return in 3rd arg              */
9             append(L1,L2,L3).
10
11
12    last(Last,[Last]).                /* Get last element of a list, (p. 139) */
13    last(Last,[_|Rest]) :-
14            last(Last,Rest).
15
16
17    rev(L1,L2) :-                     /* Reverse a list (p. 141)             */
18            revzap(L1,[],L2).
19    revzap([X|L],L2,L3) :-
20            revzap(L,[X|L2],L3).
21    revzap([],L,L).
22
23
24    /* Register manipulators for ATN's and the like
25    */
26
27    push_reg(Reg,Val) :-                    /* Push new copy of Reg          */
28            NewReg =.. [Reg,Val],           /* Create new Reg                */
29            asserta(NewReg), !.             /* Put it at the top             */
30
31    pop_reg(Reg,Val) :-                     /* Pop top level of Reg          */
32            functor(OldReg,Reg,1),
33            call(OldReg),
34            arg(1,OldReg,Val),              /* Get value to return           */
35            retract(OldReg), !.             /* Blow just that one away       */
36
37    set_reg(Reg,Val) :-                     /* Set Reg to contain Val        */
38            abolish(Reg,1),                 /* Get rid of any old values     */
39            push_reg(Reg,Val), !.           /* Replace old value with new    */
40
41    increment_reg(Reg,NewVal) :-            /* Increment value in Reg        */
42            pop_reg(Reg,OldVal),            /* Get old value from Reg        */
43            NewVal is OldVal + 1,           /* Increment it                  */
44            push_reg(Reg,NewVal), !.        /* Replace old value with new    */
45    increment_reg(Reg,1) :-
46            push_reg(Reg,1).                /* Create new one if none exists */
47
48    cons_reg(Reg,Elem) :-                   /* "Cons" Elem to list in Reg    */
49            pop_reg(Reg,OldList),           /* Get old value from Reg        */
50            NewList = [Elem|OldList],       /* Cons new Elem to it           */
51            push_reg(Reg,NewList), !.       /* Replace old value with new    */
52    cons_reg(Reg,Elem) :-
53            push_reg(Reg,[Elem]).           /* Create new one if none exists */
54
```