9-22-1986

A variant of tale-spin with independent data and rule bases

Laxmi Gupta

Rochester Institute of Technology
School of Computer Science and Technology

A Variant of Tale-Spin with Independent Data and Rule Bases

by
Laxmi N. Gupta

A Thesis, submitted to
The Faculty of the School of Computer Science and Technology,
in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science.

Approved by:

Lawrence A. Coon 9/22/86
Professor Lawrence A. Coon

John A. Biles 9/22/86
Professor John A. Biles

Guy Johnson 9/22/86
Professor Guy Johnson

Title of Thesis: A Variant of Tale-Spin
with
Independent Data and Rule Bases

I, Laxmi N. Gupta, 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.

Laxmi N. Gupta

Date: 9-30-86

Dedicated to the memory of
my mother
who was a spinner of tales, par excellence

Acknowledgements

I would like to thank Professor Larry Coon for approving the project and for agreeing to chair the thesis committee. I acknowledge gratefully his help during the preparation of this thesis. Special thanks are due Professor Al Biles for many suggestions that have improved the quality of this work. I would also like to thank him for allowing me to use parts of an ATN, he wrote. He also let me avail freely of his time for which I am grateful. Professor Guy Johnson's emphasis on a focus for the project helped me keep things in perspective and look at the project critically. For this I thank him. I would also like to express my appreciation to my family for putting up with many inconveniences while I was working on this project.

TABLE OF CONTENTS

CHAPTER 1

Introduction

1.1 Problem Statement

The object of this work is to create a story-writing system that generates stories about a set of characters based on the information provided by the user.

Such a project was carried out in James Meehan's Ph.D. thesis (Meehan, 1976), with a program called Tale-Spin. A version of it is described in (Charniak, Riesbeck, and McDermott, 1980), henceforth called CRM Tale-Spin. This project is loosely based on this version of Tale-Spin, so to honor the original implementor, the program developed here is also called Tale-Spin.

Briefly speaking, the system has three components: an English language front-end to build a data base of facts supplied by the user, a rule base consisting of rules for telling a story, and a driver module for controlling the flow of events. A version of Tale-Spin, called Micro Tale-Spin, is described by Meehan in an article of the same name included in (Schank, and Riesbeck, 1981).

Differences between the system developed here and CRM Tale-Spin are detailed in chapter 4. Here, we only note the major difference between the two systems. In CRM Tale-Spin, as well as the original

Tale-Spin, the domain of the story is at least partially hard-wired. In Micro Tale-Spin the extent of the hard-wiring of the domain is even greater. In the system developed here, the domain is built by the user, using a restricted subset of English. This requires that the rule base and the user-built data base be independent of each other. This feature can be considered an enhancement over CRM or the original Tale-Spin.

1.2 A Brief Summary of the Thesis

Chapter 1 contains the problem statement and a summary of the thesis.

Chapter 2 discusses the background, the central concepts employed, and the basic ideas of story-generation.

Chapter 3 describes the system and its implementation.

Chapter 4 discusses the results, merits, drawbacks, and the directions for future work.

Appendix 1 contains sample sessions with the system and the resulting stories.

Appendix 2 contains the Prolog code for the system.

CHAPTER 2

Background and Conceptual Framework

2.1 Background

Early natural-language processing systems such as Eliza (Weizenbaum, 1966) relied on the recognition of certain key words by the program. These programs did not try to 'understand' or even syntactically analyze the natural language input presented to them.

For a number of years the emphasis was on the 'sentence' as the linguistic unit to be syntactically or semantically analyzed. Only relatively recently, efforts have been made to try to understsnd a sentence in its context.

Schank, Colby, Abelson, and their coworkers first at Stanford and later at Yale have been developing systems that strive to extract semantic information from natural language input and represent it as an appropriate data structure. Schank and Abelson call this semantic representation the 'Conceptual Dependency System', hereafter abbreviated as CD. This system is fully described in (Schank & Abelson, 1977).

An early program that used the CD concept was MARGIE (Meaning Analysis, Response Generation, and Inference on English),

developed by Roger Schank and his students at Stanford. The front-end of the system called the 'conceptual analyzer' accepted an English language input and converted it into an internal CD-representation. The middle phase of the system called the 'inferencer' could deduce a large number of conclusions from the current state of the system's memory. This knowledge was represented in a 'semantic net'. The last part of the system was a text generation module that could convert CD-representations into English-like sentences. MARGIE was capable of running in two modes, 'paraphrase' and 'inference'. In paraphrase mode it would accept a sentence and produce several restatements of it. In inference mode it would try to make inferences from the input sentence. MARGIE is described in (Schank, 1975) and this synopsis is based on the article MARGIE in (Barr & Feigenbaum, 1981).

Schank and Abelson developed the concepts of a 'script' and a 'plan' in understanding stories. A script is an interconnected sequence of events that constitutes an acivity such as a visit to a restaurant or a trip to a supermarket. A plan is a sequence of actions that an individual performs in pursuing a goal. Schank, Abelson, and their students developed the programs SAM (Script Applier Mechanism) and PAM (Plan Applier Mechanism) which utilized these newly developed concepts. Both of these are described in (Schank, 1977). PAM is also thoroughly described in Wilensky's doctoral dissertation (Wilensky, 1978).

Meehan's Tale-Spin is a plan-based story generating program as opposed to a story-understanding program. The domain of these stories is the 'bear-world' which is partially hard-wired. The kind of animals that inhabit the world; the foods they eat and similar information is hard-wired. The user is free to select the characters of the story, the goals for the characters and certain other traits. These facts are converted to CD-form, and a set of story-telling rules are applied to these facts to generate a story in CD-form. The story-telling rules spring from the plans that characters follow to achieve their goals. Thus if a character is hungry, he/she may ask another character for food. The goal is to get food and the character tries to achieve this goal by following the plan of asking another character for food. The CD's constituting the story are then fed to an English language generator which converts them to an English text.

The version of Tale-Spin developed in this thesis is different in several aspects from Meehan's. These differences are detailed in chapter 4, but the most important is that no hard-wired facts are employed here.

Since Tale-Spin, the CD-system has been extensively developed and used in many systems. One such system is FRUMP (Fast Reading Understanding and Memory Program) developed at Yale. FRUMP accepts unrestricted English language input (newswire stories) and produces summaries. This system is described in an article by

DeJong in (Lenhert and Ringle, 1982). A more recent system is CYRUS (Kolodner, 1984). It runs in two modes: Cyrus Vance or Ed Muskie, the two secretaries of state in the Carter administration. It tries to model the memories of these politicians by recording day-to-day events in their lives and answering questions on political topics not encountered before, presumably as these politicians would. CYRUS was designed to learn from experience. To quote Schank, (Schank, 1984), it succeeded in this task "to some degree". Schank's more recent theories are expounded in (Schank, 1982). Popular, more readable accounts of Schank's work are found in (Schank, 1984) and (Schank, and Hunter, 1985).

There are two newer story writing systems based on Schank's theory of remindings and memory structures. AUTHOR, developed by Natalie Dehn (Dehn, 1981), is an author intent based system that attempts to model the creative process of story writing. Tale-Spin, on the other hand, is a character intent based story generator. The main character is given an intent (e.g. to acquire food) which it attempts to fulfill and a story is spun by working out consequences of these attempts. AUTHOR's intent is to create an interesting story, by frequently intervening and making the achievement of a character's goal easier or harder. UNIVERSE (Lebowitz, 1983), has the ambitious goal of writing soap opera stories. The above reference describes the kind of story-world that would be needed for the task. The need for consistency and coherence of the story-telling universe is stressed, and a

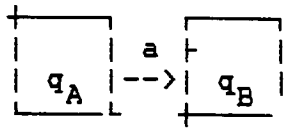
giagantic framework of character stereotypes, interpersonal relationships, and themes is envisaged.

In the present work, an augmented transition network (ATN) is used for the English language front-end. For knowledge representation, Prolog facts and clauses are used, which can be thought of as an implementation of Schank's CDs. The story generation results by following the actions of characters in pursuit of their goals. A character follows a plan of acion in pursuing its goals. Brief accounts of ATNs, CD-theory, and planning are provided in the next three sections.

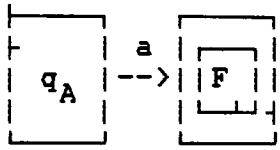
2.2 Augmented Transition Networks

Augmented transition networks were first developed by W. A. Woods (Woods, 1970). An ATN is a very versatile and convenient tool for the representation of natural-language grammars. This account of ATNs is largely based on (Barr & Feigenbaum, 1981).

A finite state transition network (FSTN), as is well known, can be used to represent a regular grammar (with productions of the type A -> aB or A -> a, where A and B are some non-terminals of the grammar and a is a terminal). The non-terminals of the grammar correspond to the states of the FSTN, with the start letter of the grammar corresponding to the initial state of the corresponding FSTN. the production 'A -> aB' corresponds to the state-transition



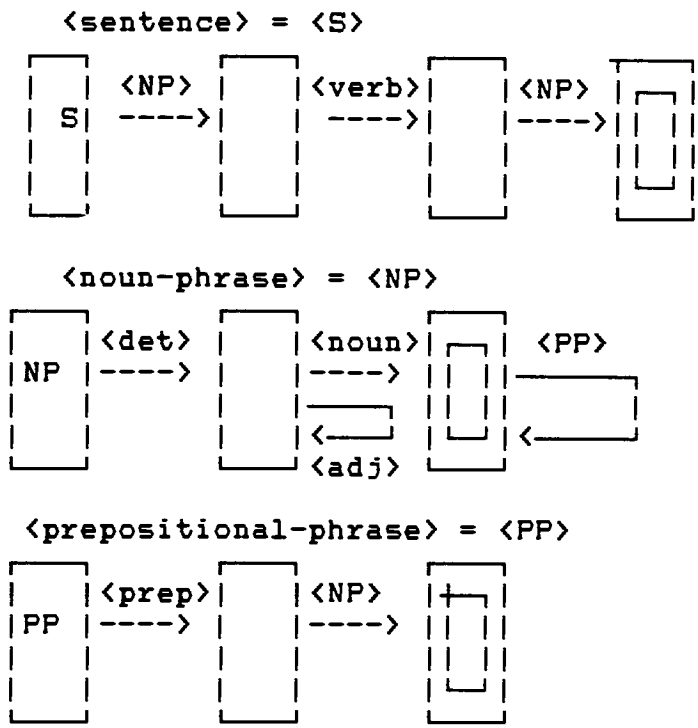
and the production 'A -> a' corresponds to the transition



where F is a final or accepting state of the FSTN.

Regular languages are not powerful enough to model natural languages. The first improvement is to allow not only the terminals to label the arcs of the network but also the syntactic categories such as <noun-phrase>, <prepositional-phrase>. Each of the syntactic categories has a network that represents its structure and these networks can call each other or themselves recursively. This extension of a FSTN is called a Recursive Transition Network (RTN). An example of a RTN is presented next.

Example



The sentence 'The fat lady with the dog hit the jackpot.' will be parsed by the RTN as follows:

<noun-phrase> : The fat lady with the dog

<prep_phrase> : with the dog

<noun-phrase> : the dog

<verb> : hit

<noun-phrase> : the jackpot

<det> : the.

An ATN is an RTN that has been extended to include registers to store information such as partial derivation trees, and certain tests and actions on the arcs. If an arc is labeled by a test, the

test must be passed before the arc is taken. Similarly if an arc is labeled by an action, this action must be performed when the arc is taken. With these augmentations an ATN becomes as powerful as a Turing machine in its processing power.

It turns out that for our purpose, a simple FSTN is sufficient. The arcs of these FSTNs are labeled by the terminals of the language or by certain semantic categories. This FSTN is described in section 2.4, alongwith its relevance to the project.

2.3 Conceptual Dependency Representation of Sentences

When we hear several news accounts of a major story such as the assasination of a world leader, despite the fact that different words are used, we understand it to be the same story. If called upon to recall the story we will narrate it in our own words. This type of paraphrasing is a mark of our understanding of the story. This strongly suggests that our mental apparatus creates a relatively language-free thought model of the story in our memory. Any understanding system, human or computer, has to have a conceptual representation of events. For computers such a representation takes the form of a data structure.

Schank's Conceptual Dependency Theory is a theory of representation of the meaning of sentences. The theory primarily deals with actions and static situations but does not adequately

deal with nouns or social situations. There are extensions (scripts and themes) that try to deal with these. The two basic tenets of the theory are:

1. For any two sentences that have identical meanings, meaning representations must be the same.

2. Any information that is implicit in a sentence must be made explicit in the meaning representation.

To make the representation economical it is essential to think in terms of primitive actions . Two kinds of primitive actions are considered, namely action primitives and stative primitives.

A conceptualization using an action primitive has the form:

Action	Actor	Object	Direction	Instrument
			From:	(optional)
			To:	

For instance, the action ATRANS refers to the concept of transfer of an abstract relationship with respect to an object such as possession, ownership, or control. Consider the sentence:

'John gave Mary a book.'

Action: ATRANS

Actor: John

Object: Book

From: John

To: Mary

The verb 'take' also involves the primitive ATRANS, but actor is the recipient not the donor. Let us consider the sentence:

'Mary took a book from John.'

Action: ATRANS

Actor: Mary

Object: Book

From: John

To: Mary

The above two examples illustrate the principle that when two conceptualizations involve the same primitive action, their difference is expressed by the way other slots are filled in the representation scheme.

The verb 'buy' is made of two conceptualizations: an ATRANS of money and an ATRANS of the object being bought.

Some other action primitives are:

PTRANS is the transfer of the physical location of an object. It is used to represent verbs such as go, come, and put, etc.

MTRANS is the transfer of mental information. Memory is considered to be partitioned into two pieces: CP(Conscious Processor) and LTM (Long Term Memory). 'Tell' is MTRANS from CP of one person to CP of another. 'See' is MTRANS from eyes to CP. 'Learn' is MTRANS of new information to LTM.

INGEST is the taking of an object by an animal to its inside. This is used to represent eating, drinking, and smoking etc.

Schank and Abelson describe eleven action primitives in (Schank and Abelson, 1977). Besides action primitives, they also postulate stative primitives. A stative conceptualization takes the form:

Object State Value

Some possible states are health, anticipation, mental state, and physical state. Value refers to a measurement of some state attribute. Thus health may have a value-scale of -10 to 10, 10 meaning excellent and -10 dead.

2.4 Planning

Since story-generation is plan-based, an account of the general

notion of planning and its application to the natural-language processing is presented here.

2.4.1 Planning and Problem Solving

This account is based on (Cohen and Feigenbaum, 1982).

A plan is defined as a sequence of actions that need to be performed to achieve a goal, thus planning provides an organized way of solving problems. The seminal work on problem solving is Newell and Simon's "Human Problem Solving" (Newell and Simon, 1972), which advocated an approach called means-ends analysis for solving problems. For this approach, the problem domain is represented as states each of which is essentially a set of assertions. The goal is to get from the current state of the world to the goal state. Problem solving operators try to reduce the difference between the current and the goal states and in the process generate a plan of action. This approach to problem solving was implemented in STRIPS (Fikes and Nilsson, 1971) and ABSTRIPS (Fikes, Hart, and Nilsson, 1972).

Most plans for solving problems have nested subgoal structures and are hierarchical in this sense. However, there is another sense in which the word "hierarchical" is used in the planning literature. In this sense, hierarchy refers to a hierarchy of representations for the plan being developed. A high-level plan is developed at

first and then is successively refined and made more detailed. The difference between STRIPS and ABSTRIPS is that the latter is hierarchical in the sense just explained.

Sacerdoti's NOAH (Sacerdoti, 1977) is another noteworthy system that used means-ends analysis. NOAH is a hierarchical planning system that employs critics to scrutinize the plan being developed for potential problems at each stage of its refinement.

It is pertinent to point out here that Kowalski (Kowalski, 1979) has shown that general problem solving can be carried out entirely in terms of procedural logic, and therefore, in terms of Prolog.

2.4.2 Planning in Natural Language Comprehension and Story Generation

Schank and Abelson (Schank and Abelson, 1977) developed the concept of a script, a stereotyped sequence of events associated with a situation, in order to facilitate the understanding of English text by a computer. It soon became apparent to them that a more general structure dealing with the non-stereotyped situations was needed. A plan is such a structure.

Plans are devised to achieve goals. An understanding system has to be able to detect implicit or explicit goals in the story and expect the associated plans to come into action. Schank and

Abelson note that certain goals called delta-goals or D-goals are often called upon in service of other higher level goals. These are:

D-KNOW (acquire knowledge of something)
D-PROX (get close to something)
D-CONT (get control of something)
D-SOCCONT (gain social control of something)
D-AGENCY (have some-one-else pursue the goal on your behalf).

These are called D-goals because they all involve a change of state and symbol for change in mathematics is often D or delta. Associated with each D-goal are certain plan-boxes. Each plan-box is a somewhat detailed recipe for achieving the goal.

In our story-writing system the first three of these D-goals are used for planning. The associated plan boxes are "asking for", "going there", and "taking the desired object" respectively.

Example: Consider the following story:

"Joe bear was hungry. Joe bear found honey in a bee hive. Joe bear ate the honey."

This story can be understood or generated (which is the object of Tale-Spin) in the context of the following plan which Joe bear

follows to satisfy his hunger.

1. D-Know(Loc(honey)) : Find the location of food.
2. D-Prox(Honey) : Get close to the food.
3. D-Cont(Honey) : Get control of the food.
4. Ingest(Honey) : Eat the food.

Themes are the sources of goals. In simple stories goals are often explicitly stated and an understander is willing to take them at face value. Stories generated by our system are of this kind. In more complex stories goals are not explicitly stated but have to be inferred from the contextual setting. Thus we know that people behave in a certain way when they are in love, when they are business partners, lawyers, crooks, or parents. Any understanding system has to know such themes and know what to expect when such themes are included. A generating system should know what goals and plans should be created for a given theme.

In his more recent works Schank (Schank, 1982) has thoroughly revised the concepts of scripts and plans. Scripts are replaced by MOPs (Memory Organization Packets) and plans by TOPs (Thematic Organization Packets). A MOP is composed of scenes and a scene can be shared by several MOPs. A script, unlike a MOP, is composed of events which are not shared by other scripts. A scene has an existence of its own and if an unexpected failure occurs in the processing of a scene, the understanding system stores this

failure with the scene, thus making it available to all the MOPs sharing the scene. In this way, the system can learn and remember things from its past.

TOPs are goal-driven like Plans and are more general than MOPs. TOPs are also largely domain independent and can represent abstract concepts such as imperialism, evil, or goodness.

2.4.3 Planning in Text Generation

The process of natural-language text generation in this work is based on template-matching and instantiating variables. This approach has been used by several systems with fairly good results. An example is Winograd's SHRDLU (Winograd, 1972).

Another approach has been to use an internal knowledge representation such as CD-system or semantic nets as input to an ATN which acts in an inverse manner from an ATN-recognizer to produce an English text expressing the semantic knowledge content of the input. Goldman's text generator program (Goldman, 1974), used in MARGIE, is an example of this approach using CDs for the knowledge representation. Meehan used a variation of Goldman's system which he called MUMBLE for text generation. An example using semantic nets is Simmons and Slocum's system (Simmons and Slocum, 1972).

A planning-based approach for generating English text has been used recently by Appelt in a system called KAMP (Knowledge and Modalities Planner) described in (Appelt, 1985). The basic premise is that human speech is an act of planning with multiple goals such as informing, ordering, imploring, impressing, or entertaining the listener.

2.4.4 Meta-Planning or How to Plan the Planning

This discussion is based on (Wilensky, 1983).

Most classical artificial intelligence problem solvers operate on the principle of devising a plan to achieve a given goal. Moreover, these systems have no capability to infer their goals from a given situation. On the other hand, understanding systems using the planning mechanism must be able to infer the goals of characters from their actions. Wilensky proposes a planning system that can handle multiple goals, can resolve the goal conflicts, and can infer its own goals from the current state of the world and thereby "understsnd" its own actions.

Such a planning device would have purposes of its own called meta-themes. Wilensky proposes the following meta-themes to guide the planning process (Wilensky, 1983, pp31):

1. Don't waste resources.

2. Achieve as many goals as possible.

3. Maximize the value of goals achieved.

4. Avoid impossible goals.

These meta-themes give rise to meta-goals as a function of the situation at hand.

2.5 Conceptual Framework for the System

The system performs three distinct functions:

1. Extraction of knowledge (meaning) from user-supplied sentences.

2. Storing of above knowledge as Prolog facts. (These facts constitute the data base of facts.)

3. Application of hard-wired rules to the data base to generate a story.

2.5.1 Extraction and Representation of Meaning

The task of meaning extraction is performed by an ATN which is actually just a finite state transition network (FSTN) and is neither recursive nor augmented. Arcs of the network are labeled by the terminals of a grammar to generate a small subset of English or by certain semantic categories, as our primary purpose

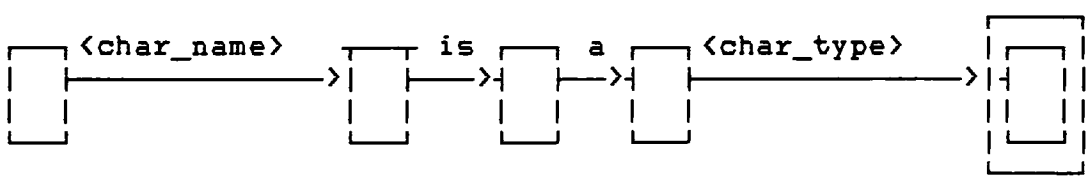
is not to create a parse-tree but to extract the underlying meaning of the sentence. The five semantic categories are:

1. character_name
2. character_type (species of the character)
3. home_type
4. food_type
5. location.

Any word not known to the system must belong to one of these categories. Thus all pronouns, verbs and determiners etc., recognized by the system are hard-wired.

Example 1

The sentence 'Joe is a bear.' can be parsed by the following ATN.

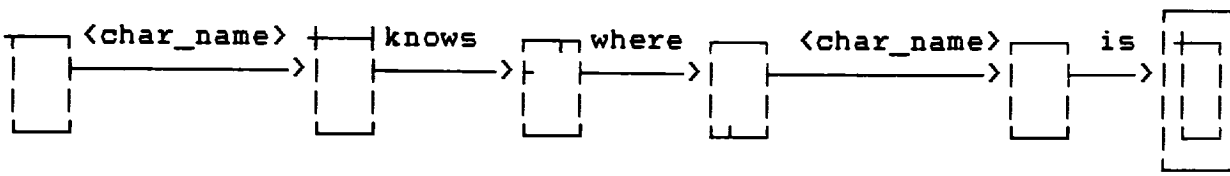


Joe is the name of a character and bear is a character_type. The following prolog facts will be added to the data base:

```
ischar(joe).
species(joe,bear).
```

Example 2

The sentence 'Joe knows where Jill is.' will be parsed by the follwing ATN.

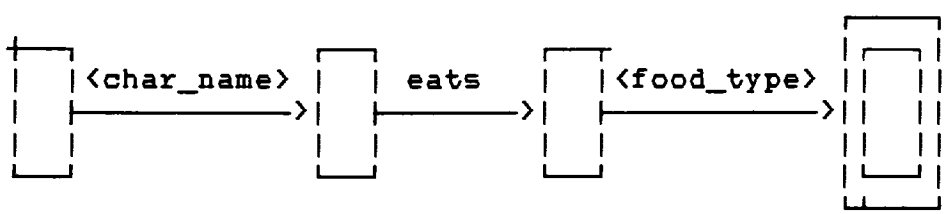


Joe and Jill are character-names. Words 'knows', 'where', and 'is' are hard-wired. The following Prolog facts are added to the data base:

```
ischar(joe).
ischar(jill).
knows_loc(joe,jill).
```

Example 3

The sentence 'Joe eats beans.' will be parsed by the following ATN.



Joe is a character-name and beans are a food-type. The following facts get added to the data base:

```
ischar(joe).
```

```
food_type(beans).
food(joe,beans).
```

2.5.2 Use of Conceptual Dependency Representation

In CRM Tale-Spin, CD representation was used to model primitive actions and states. Actions were PTRANS, ATRANS, MTRANS, and INGEST. Of these we explicitly use PTRANS and ATRANS, and the other two are used indirectly. These actions are modeled as Prolog goals that carry out the required data base changes. States modeled include 'has', 'at', 'home',and 'knows_loc' etc. Some examples are:

```
has(joe,beans).
home(joe,joe_cave).
at(joe,meadow).
knows_loc(joe,jill).
```

2.5.3 Plan-Based Story Generation

To generate a story, a main character is specified and is given a goal to achieve (satisfy desire for food, water or sex). For each of these goals a sequence of plans is available which are tried in succession until one succeeds or they all fail.

A plan is a sequence of actions that need to be performed to

achieve the goal. In this thesis, a plan can be thought of as a sequence of Prolog rules say R1,R2,...,RN. The first (N-1) rules specify the various ways the plan may fail, and the last is the recipe for success. The reason for including explicit recipes for failures is that the story must document failures as well as successes. This makes for more interesting stories.

The right hand side of a Prolog rule RI has three components in the order specified:

1. Preconditions for the rule that do not require any modification of the data base.
2. Data base changes to be made if above preconditions succeed.
3. Instructions for printing the appropriate story-fragment.

The reason for this arrangement is that if a precondition is not met, Prolog will backtrack and try the next rule. If we make data base changes earlier and the present rule does not apply, we are stuck with the undesired changes in the data base, which are not automatically retracted. This may lead to incorrect results.

Example

Suppose the main character X has the goal of making love (sat_desire). A plan for making love is to find a character of the opposite sex and of the same species, who likes X and is liked by X, and is in mood for (wants) sex. Suppose a character Y of the

same species and opposite sex has been found.

Rule 1: Suppose X does not like Y. The plan fails.

Rule 2: Suppose X likes Y but Y does not like X. As before the plan fails.

Rule 3: Suppose X and Y like each other but Y is not in mood. The plan fails.

Rule 4: X and Y like each other and Y is in mood. Presto! Success!

CHAPTER 3

System Description

3.1 An Overall View of the System

Tale-Spin is an interactive system written in Prolog, which generates simple stories based on the information provided by the user.

The user can provide information to the system using a restricted subset of English. This information consists of character names, character types(species), homes, whereabouts, foods, likings, and needs of the characters. It is essential to specify a main character for the story. The main character must be furnished with one of the primal needs: food, water, or sex.

Story generation in Tale-Spin is plan-based. For each of the three primal needs, plans exist which are tried in succession, and the success or failure of each plan is recorded. Eventually either some plan succeeds or they all fail. In either case, a story is printed documenting the various plans tried, failures and eventual success or failure in fulfilling the basic need. Thus the story is generated by tracing the actions of the main character in pursuit of his/her goal.

A story is generated sentence by sentence, and templates for

sentences are embedded directly in the rules for telling the stories.

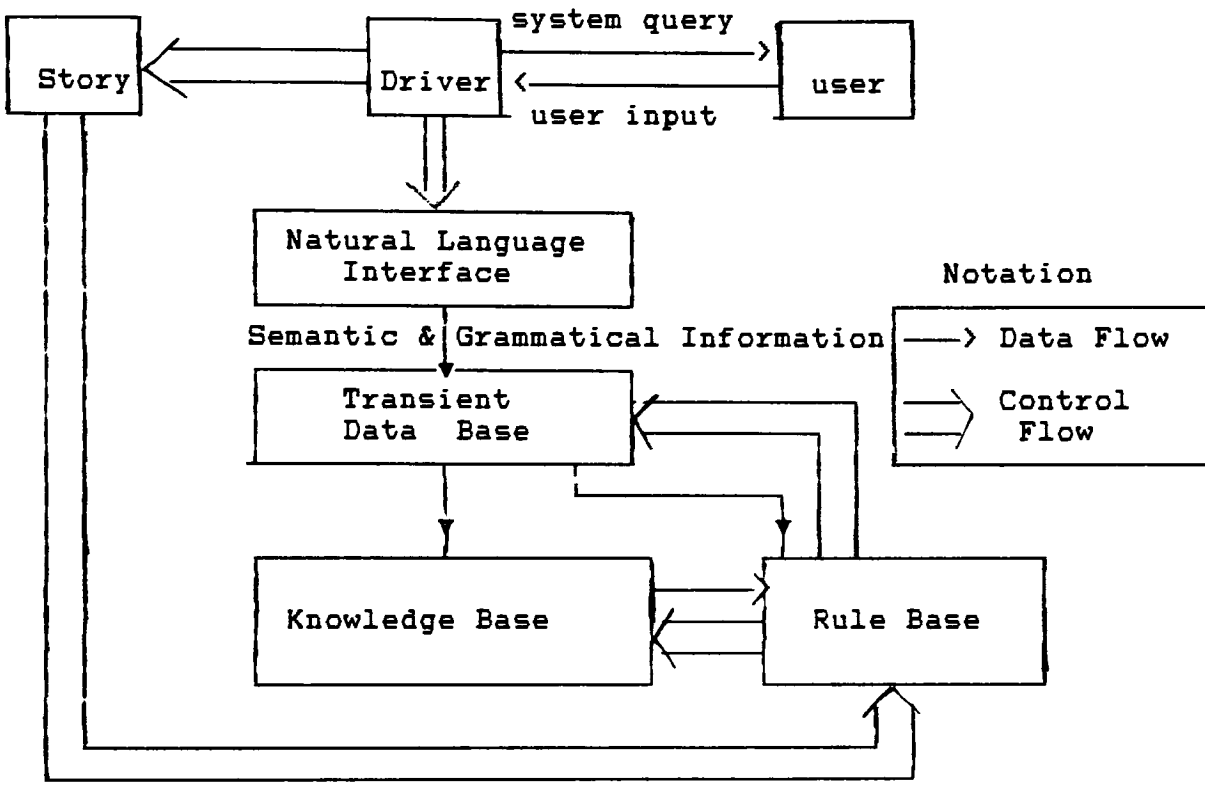
3.2 System Components

The system consists of the following components :

1) A Driver
2) A Natural Language Interface
3) A Knowledge Base
4) A story generating procedure
5) A Rule Base.

It should be pointed out that any breaking up of a Prolog system is arbitrary as all the clauses are accessible to the entire system. The only pair-wise disjoint decomposition would be to list all the distinct clauses. The above decomposition has been chosen only for the ease of description. The knowledge useful for all the three submodules of Rule Base (hunger, thirst, and love) has been isolated and placed in the Knowledge Base. A data and control flow diagram of the system is pictured on the next page:

Data and Control Flow Diagram



3.3 The Driver

The goal that gets the system in action is called "do".

The goal "do" first calls "preamble" which asks the reader to start providing the information from which the story will be generated.

Then "talk" is invoked, which reads one sentence at a time, converts it into a list of words and passes this list to "atn".

The "atn" extracts the pertinent information from the sentence and adds it to the data base. After the user has prompted the system that he/she has told all, "get_more_info", a supplemental ATN, is called.

The goal "get_more_info" checks if all the essential pieces of information for telling the story have been provided. If necessary, the user is queried to provide missing information. The "story" goal is invoked at last.

The story generating procedure "story", generates the opening sentence of a story, describing where the main character is currently and what he/she wants. Then it calls the rule base to generate the rest of the story.

3.4 The Natural Language Interface

The natural language interface consists of the following:

a) an augmented transition network, called simply "atn," to handle the English language input, in form of the sentences.

b) a dictionary of the hard-wired words and a set of rules for adding more words to the dictionary temporarily. This module also contains the grammatical rules for deciding the number and the type of a noun, etc.

c) a supplemental augmented transition network, called "get_more_info" that queries the user for missing essential pieces of information and extracts this information from the user responses. These responses need not be in the form of complete sentences.

d) a submodule to read a sentence and to convert it into a list of words and a "write" routine to convert a list of words into a sentence and print it as a sentence, taking into account capitalization and punctuation. This submodule is called "inout".

Some examples of how the natural-language front-end works are provided at the end of this section.

3.4.1 The Augmented Transition Network

The main goal "s", has the list "L" of words in a sentence as its argument. A word in "L" is either already in the dictionary or falls in one of the following categories or semantic-types :

1)character_name
2)character_type(species)
3)foodtype
4)home_type
5)loc_name.

If a word is not already in the dictionary or already processed to establish its semantic-type and other attributes, the user is queried about the type, number, and gender(if appropriate) of the word being processed.

In passing the control to subgoals of "s", the following device is employed : s_charname(First,Rest) means that the first word of the original list "L" is a character_name and s_charname will handle the rest of processing. Similarly s_props_be(First,Rest) indicates that the first word of the sentence was a prop - a food_type or water - and the second some form of the verb "be".

It would be pertinent to point out that unlike traditional ATNs where a sentence is parsed into its syntactical components and a parse-tree built, we use a simple semantic grammar and the ATN is used to extract semantic information directly from the sentence. This information is then added to the data base, to be used later for telling the story.

Quite often ATNs are used for programming natural language dialogs. We used the ATN primarily for gathering the information. The system does not have any question-answering ability. It responds to the information supplied by the user by asking a question or by saying "ok" if the information is "understood".

The subset of English accepted by the ATN is represented by the

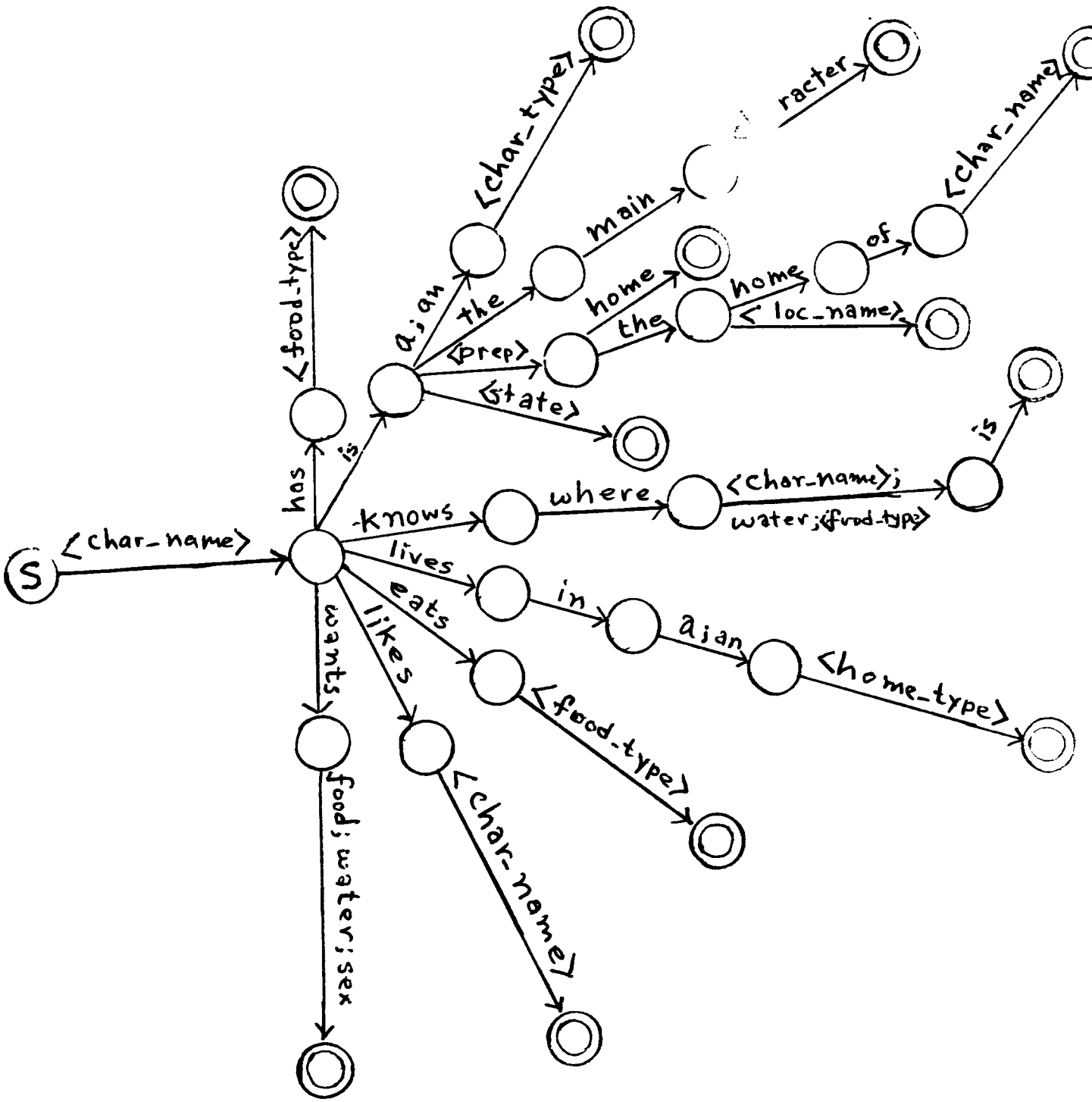
finite state diagrams on the next two pages.

3.4.2 The Dictionary

The Dictionary contains the list of hard-wired words such as : "is", "are", "food", "sex", "water" etc.

Any time the system encounters a new word, the user is queried about the semantic-type of word. the goal bad_word(W) succeeds if "W" is a word unknown to the system. The goal put_in_dic(W) collects the information about W from the user and places it in the dictionary.

FINITE STATE DIAGRAMS FOR THE ATN



<prep> = at; on; in; by

<state> = hungry; horny; thirsty

3.4.3 The Supplemental Augmented Transition Network

The purpose of this module called "get_more_info" is to gather additional information from the user and process it. After the user has given all the information about the story, the system checks whether any essential piece of information is missing. For instance if there is more than one character in the story, it is essential to establish a main character. Similarly if the goal of the main character is to obtain food, the system must be told what foods the main character eats. The responses to these queries need not be complete sentences. For example : "Where does John live?" might elicit response: "in a cave". The supplemental ATN will process this response and add:

home(john,john_cave).

to the data base.

The subgoals of get_more_info are:

- specify_main_char
- specify_theme
- specify_want_others
- specify_home
- specify_food
- specify_likes
- specify_char_loc
- specify_props_loc

- specify_knows_loc_props
- specify_knows_loc_char

These goals are invoked in order. Often the user is given another chance to supply more information than what was asked for if he/she wishes to do so. For instance specify_char_loc first tries to establish the current location of the main character and then asks the reader to provide information about the location of other characters if the user wishes to do so.

3.4.4 The Input-Output Module

The module "inout" handles the input and output.

The goal "read_sent(S)", reads a sentence from the standard input and converts it into a list of words. This part of the code is borrowed from Clocksin & Mellish (pp. 87-88).

Write_sent(S), writes out a sentence nicely, capitalizing the first word and known proper nouns. This code is borrowed from an atn written by Professor Al Biles.

Example 1

Let us examine how the sentence:

"Joe is at the meadow."

is processed by the front-end of the system.

The predicate read_sent(S) reads the sentence from the standard input and converts it into the list:

S=[joe,is,at,the,meadow,.].

Note that any upper case letter is converted to the corresponding lower case letter. Let us assume that the words "joe" and "meadow" have not been encountered by the system previously. The predicate check_words(S), checks if the words in the list S are in the dictionary. Since "joe" is not in the dictionary, bad_word(joe) is invoked which queries the user about the semantic type of the word "joe". Recall the five semantic types: character, character_type, hometype, location, and foodtype. Suppose the user responds with the answer "character". The predicate put_in_dic(joe,character) is invoked and as a result the fact:

ischar(joe).

is added to the data base. The user is queried about Joe's sex. Assuming the user response to be "male", the fact:

male(joe).

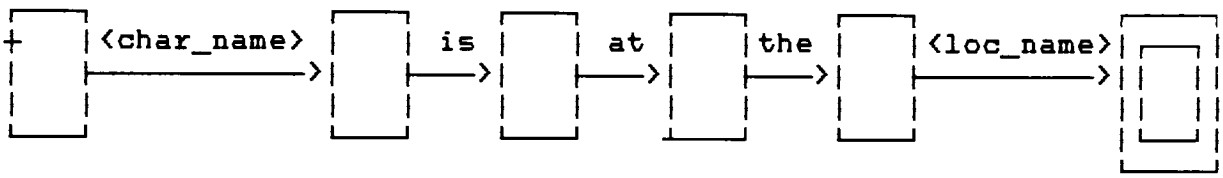
is added to the data base.

The words "is", "at", and "the" are hard-wired, so these are accepted by check_words but "meadow" is not. As a result bad_word(meadow) is called, which in turn calls put_in_dic(meadow). The user is queried as before. Assuming that

semantic category for "meadow" is "location", the following fact is added to the data base:

```
loc_name(meadow).
```

After all the words have been placed in the dictionary, the ATN is called with the predicate s(S). The path followed by the sentence under consideration through the net is pictured below:



The following clause of the network predicate s applies:

```
s([First|Rest]) :- ischar(First),
                   s_charname(First,Rest),
                   write_sent(['ok']).
```

The predicate ischar(joe) succeeds and s_charname(joe,[is,at,the,meadow,.]) is called. The following clause of this predicate applies:

```
s_charname(Subj,[First|Rest]) :- (First = 'is'),
                                 s_charname_is(Subj,Rest).
```

Thus s_charname_is(joe,[at,the,meadow]) is invoked. The appropriate clause of this predicate is:

```
s_charname_is(Subj,[First|Rest]) :-
                    First = at,
                    s_charname_is_prep(Subj,[First|Rest]).
```

This results in s_charname_is_prep(joe,[the,meadow,.]) being called. The code for the applicable clause is:

```
s_charname_is_prep(Subj,[First|Rest]) :-
                    First = the,
                    s_charname_is_prep_the(Subj,Rest).
```

As a result s_charname_is_prep_the(joe,[meadow,.]) gets called. The relevant code is:

```
s_charname_is_prep_the(Subj,[First|Rest]) :-
                    loc_name(First),
                    assert(at(Subj,First)).
```

As a result the fact:

```
                    at(joe,meadow).
```

is added to the data base, the predicate

```
                    s_charname(joe,[is,at,the,meadow,.])
```

succeeds, and control is returned to s(S) which prints "ok" and accepts the input sentence.

Example 2

Since we have already seen how the syntax-driven ATN accepts a sentence, in this example we will concentrate on the meaning extraction process. Suppose the user types the sentence:

"Dogs live in doghouses."

Assume that it has already been established that Spot and Jackie are dogs. The system will query the user about the semantic type of "doghouses" and then its number (singular or plural). As a result of appropriate user responses, the following facts will be added to the data base:

```
home_type(doghouse,doghouses).
live(dog,doghouse).
```

As a result of earlier processing, we already have the following clauses in the data base:

```
species(spot,dog).
species(jackie,dog).
```

The predicate buildhome(Character,Home,Inst_home) will build a separate home for each dog in the data base. As a result the following clauses will be added to the data base:

```
home(spot,spot_doghouse).
home(jackie,jackie_doghouse).
```

Example 3

Continuing in the context of the Example 2, suppose that the user further types the sentence:

"Dogs eat biscuits."

System will query the user to establish that "biscuits" is a foodtype and plural. The following clauses will be inserted in the data base:

```
foodtype(biscuits).
eat(dog,biscuits).
```

The knowledge base has a clause:

```
food(X,Y) :- species(X,Z),eat(Z,Y).
```

As a consequence, the following facts can be deduced:

```
food(spot,biscuits).
food(jackie,biscuits).
```

3.5 The Knowledge Base

The name of the module is abbreviated to "kbase". It houses the hard-wired facts and relations used by any of the three rule bases: hunger_rules, thirst_rules, love_rules.

3.5.1 Facts

Following are some examples of the relational facts stored in the

knowledge base:

i) If character X belongs to species Z and animals of species Z eat food Y, then X eats Y.

ii) If X is male and Y is female or vice-versa then they are of opposite sex.

iii) If X and Y are of opposite sex and belong to same species then it is possible to have sex between them.

The Prolog code for these facts can be found on pp 17 of Appendix 2.

3.5.2 Grammar Rules

Hard-wired grammar rules are used for the text generation only. Some examples are given below:

i) All character names are proper nouns (and therefore the first letter should be capitalized by the output routine).

ii) Clauses to determine correct pronouns for a character (nominative, objective or possessive) of given gender.

iii) Clauses to determine appropriate forms of the auxiliary verb "be" to go with the number and gender of the subject.

3.5.3 Primitive Actions

A character can perform the following primitive actions:

- ptrans(W,W,X,Y) : Transport character W from location X to

location Y.

- atrans(X,Y,Z): Transfer possession of object Z from character X to character Y.

The goals described next use above primitive actions to generate the story.

3.5.4 Story-Fragments

The following goals generate pieces of English text for the story.

desire ptrans(X,Y,P,Q)

Character X is at the location P and character Y is at location Q. X wants to go where Y is. If both X and Y are at the same location, an appropriate statement indicating this is printed. Otherwise, if X knows where Y is, he/she goes there. If X does not know where Y is, he/she goes to the home of Y. In each case an appropriate sentence is printed.

take(X,Z)

Character X takes possession of object Z. Appropriate changes are made in the data base and an appropriate sentence is printed to indicate that X took Z from the place P.

give(X,Y,Z)

Character X gives character Y the object Z. Appropriate changes are made to the data base and a suitable English sentence is generated, as part of the story. Actually "give" is more complicated than this. If X has Z, then he just gives Z to X. If Z is at the home of X, X goes home to get Z, brings it and gives it to Y.

These are not the only story-fragments used. The three theme-based rule bases have a number of print statements (print_hunger1 etc.), each of which prints a fragment of the story. The ones described here are used by all the three rule bases.

3.6 The Story Generating Procedure

This module is called "story". The goal "story" prints the opening sentence of the tale being told :
" Once upon a time X was by the P and (was hungry) / (was thirsty) / (wanted to make love)", depending upon the need the main character wants to fulfill.

Then it invokes one of the subgoals : sat_hunger, sat_thirst or sat_desire, according to the theme of the story. Each of these subgoals incorporates all the plans that are available for achieving the respective goal. Thus sat_hunger(X) calls all the

plans available to X for obtaining food. These plans are invoked in order until one succeeds or all of them fail.

Each of sat_hunger, sat_thirst or sat_desire return the control to story which prints "The end.", at the end of the story.

3.7 The Rule Base

This module has three submodules :

i) hunger_rules

ii) thirst_rules

iii) love_rules.

The clauses utilized by all three are generally speaking in kbase.

We will now describe each of the submodules.

3.7.1 Hunger rules

The main goal is sat_hunger(X) (i.e. X has the goal of satisfying his/her hunger). X tries the following plans in succesession until one succeeds or they all fail. The story is generated by tracing these failures and eventual success or failure.

Plan 1

If X has on its person a food-item Z, the goal sat_hunger(X) succeeds. The story-fragment modules print_hunger1(X,Z) and print_hunger2(X,Z) generate and print the appropriate story.

Plan 2

If plan 1 fails (i.e. X has no food on its person), he/she goes home to look for food. If there is food at home, sat_hunger(X) succeeds. The story fragments record the success or failure. In case of failure plan 3 is called.

Plan 3

If the food Y is at a location P which is not anybody's home and the main character X knows this, then X goes to P and takes food Y. Appropriate print statements record the story. In case of failure plan 4 is called.

Plan 4 (Trade)

The general idea is to trade something X has with another character Z for food. A filter is used to check that X has something he/she can trade and there exists a prospective trade-partner Z who may want what X has. Several variations result due to following possibilities:

i) X either has object Y to be traded on his/her person or the object is at X's home.

ii) X is at home or at another location.

iii) X knows or does not know where Z is.

iv) Z, the character with whom the trade might take place is at his/her home or at another location.

Story fragments record the success or failure of the trade plan.

Plan 5

This is invoked only when all else fails, simply to record that X was not able to satisfy his/her hunger.

3.7.2 Thirst rules

The main goal is sat_thirst(X) i.e. X has the goal of satisfying its thirst. X tries the following plans until one succeeds or all attempts to quench thirst lead to failure.

Plan 1

If X is at a place where water is, X drinks the water and sat_thirst succeeds.

Plan 2

If X knows where water is, X goes to the location of water (if he/she is not already there), drinks the water and sat_thirst succeeds.

Plan 3

If X has something he/she can trade for the knowledge of the location of water, he/she tries to do so. A precondition for X asking a character Y for this trade is that X likes Y. Several stories result because of the different possible arrangements of facts. Some of these possibilities are listed below :

i) X is at the same location as Y. If not, X may or may not know where Y is. If X does not know where Y is, he/she goes to the home of Y. Of course Y may or may not be home.

ii) X may have the object he/she wants to trade on his/her person or this object may be at his/her home. In the latter case X has to go home to fetch the object.

Plan 4

If X does not know where water is and has not been able to trade something for the knowledge of the location of water, he/she finds a character Y, he/she likes and asks for the location of water is. If Y likes X and knows where water is, he/she tells X the location

of water. Clearly several stories are possible here, depending upon what facts happen to hold at the time.

Plan 5

This is tried if all else fails. The system simply narrates the fact that X failed to quench its thirst.

3.7.3 Love rules

The main goal is sat_desire(X) i.e. X tries to satisfy his/her desire for sex. Sex is possible only between the animals of opposite sex and of the same species. Also both parties must like each other and both must want sex. The following plans are tried in succession.

Plan 1

There is a character Y, a potential love-partner at the same location as X. If X does not like Y, he/she will follow another plan, and the failure of this plan will be duly recorded in the story. On the other hand, if X likes Y, he/she will ask Y. If Y likes X and wants sex (is in mood), he/she will respond favorably to X's advances. Otherwise the plan will fail.

Plan 2

If the potential love-partner Y is not at the same location as X, X may or may not know where Y is. In case X knows Y's whereabouts, he/she goes there otherwise he/she goes to the home of Y. In later case Y may or may not be home. As is clear, different stories will result depending upon what combination of facts happens to be true.

Plan 3

If all other plans fail, this plan is tried. It does nothing but record the fact that X failed in his/her attempts to make love.

Remarks

It should be noted that if X's attempts to make love with one potential partner fail and another potential partner exists, he/she will repeat these attempts with the new potential partner. This can be done easily, due to backtracking facilities of Prolog. Similar remarks apply to sat_hunger and sat_thirst.

CHAPTER 4

Results and Conclusions

Here we assess the strengths and the drawbacks of the system, directions for future work and the success of Prolog in implementing this application. We also compare the system as developed to the system as described in (Charniak, Riesbeck, & McDermott) as the latter system was the starting point for this project.

4.1 The Strengths of the System

The primary goal was to produce a story-writing system equipped with a rule base applicable to a data base of facts supplied by the user. This rule base was to operate on the data base to produce a story. The idea was that the same rule base could be applied to different data bases.

A secondary goal was to allow the user to impart the knowledge of the world to the system in natural language.

The system was to be implemented in Prolog with an eye towards assessing suitability of Prolog for this application.

All of these goals have been met. An augmented transition network handles natural language input, although a restricted subset of

English is recognized. As a rule, only relations have been hard-wired and not facts. This allows us to achieve some independence of the rule base from the user-supplied data base of facts. The system has been implemented in Prolog which turned out to be tailor-made for this application.

4.2 Comparison with CRM Tale-Spin

CRM refers to the reference (Charniak, Riesbeck, and McDermott). The system described in this reference does not have the property of the rule base being independent of the data base of facts, and the fact base is at least partially hard-wired. The kind of facts that are hard-wired are the kind of animals, what they eat, where they live, their peculiar habits and so on. The user chooses some characters, their present whereabouts and needs. In our system there are no hard-wired facts, only hard-wired relations. Therefore, with the exception of the three types of story plans, the domain is completely established by the user and not partially established as in the original Tale-Spin.

Using a hard-wired domain, it is possible to tell more complex stories because of the rich relationships and details that can be built into the domain. In our approach, the user has to build whatever complexities he/she desires, subject of course to the limitations imposed by the rule base.

On the other hand, in some sense, our approach is better as it allows the user much more freedom in selecting the characters, their habitat, current location, and their likes and dislikes. Of course, we cannot utilize the rich structure of a carefully built hard-wired domain since it does not exist.

Another difference with CRM Tale-Spin is that it required that input to the system be in conceptual dependency form whereas we allow input to be in a restricted subset of English.

CRM Tale-Spin has a goal monitor that records the successes and failures of the subgoals of the top goal in a tree and passes this tree to an English-generator to write a story. Our implementation differs from this principally in that we use a flatter structure than a tree and our English-generator is embedded in the story-writing rules. Suppose there are two plans A and B to accomplish the top goal. Suppose further that the plan A can fail in (n-1) ways. We have n separate clauses for plan A, recording all potential failures and the success. The templates for the sentences of the potential story are embedded in these clauses. If plan A fails, we will generate sentences telling of this failure and move to plan B for further generation of the story. Thus our English-generator is not a separate entity but an integral part of the story-writing rules.

For the last difference, we point out that we can utilize the

backtracking mechanism of Prolog, whereas a Lisp programmer (the language recommended in CRM) has to write his/her own backtracking procedure.

4.3 An Evaluation of Prolog as an implementation language

We think that the choice of Prolog for this application was a good one. After an initial period of getting used to the way of thinking that Prolog imposes, it became relatively easy to think and program in Prolog.

The backtracking facility and the ability to control it by means of the cut was tremendously useful. We illustrate this by means of an example.

Suppose the main character X has the goal of satisfying its hunger. The top goal is sat_hunger(X). The first plan is to check if X has some food on his/her person. If these conditions are met, we use a cut to tell the system that the right plan has been selected. If the conditions are not met, the next plan is tried, which consists of checking if X has some food at home. We check if there is a food Z at the home P of X, and if so, we again use a cut to indicate that the correct plan has been chosen. If plan 2 fails, we actually generate this failure to print a part of the story i.e. X went home to look for food and did not find any there. We make sat_hunger(X) fail by including an explicit 'fail'

statement. Prolog will now backtrack and try the next plan. This scheme continues until the eventual success or failure is produced. All this can be done in lisp but the resulting program would be longer and more complex. One would have to worry about the proper binding of the variables and provide one's own backtracking procedure whenever necessary.

On the negative side, the use of cuts can lead to unpredictable results, especially in long programs. Therefore, cuts have to be used with great care.

The success of Prolog springs from the fact that all the tools used for natural language analysis such as grammars, augmented transition networks, semantic nets, expert systems or production rules can be expressed in the form of if-then rules and Prolog facilitates a simple implementation of these rules.

4.4 The Drawbacks and Possible Extensions of the System

1. At present only three top-level goals (satisfy hunger, thirst, or sexual desire) are possible for the characters. For more complex stories, more goals could be pursued. Some possible examples of these goals are activities such as playing, winning a fight, finding a place to rest, or finding a job.

2. The number and complexity of plans available for achieving

goals of obtaining food, water or sex can be enhanced.

3. The system, as it stands, is structured essentially linearly. For example the following type of situation is not possible in the system: "Kate, the bird, lives in a tree which is in a forest." The system only allows the situation that: "Kate, the bird lives in a tree." This was done to keep things relatively simple. For more complex stories a deeper nested and potentially recursive structure should be allowed.

4. The only abstract concept that has been modeled is liking or loving. For richer stories, one could model deceit, guile, treachery or whatever.

5. The only verb used for describing physical transportation is "to go". One could try differentiating between walking, flying, or riding in an automobile, for example.

6. At present, the only way to communicate with a character is to go where the character is and talk. It would be interesting to model processes like sending a message through a messsenger, mail, or phone.

7. It should be possible to increase the sophistication of the atn, so as to cut down on the excessive questioning that the user has to put up with. Some examples are given below.

Example 1

User : Joe is the main character.

System : I do not know the word Joe.
Is Joe a character, charactertype, hometype, location or foodtype?

A smarter atn might realize that Joe being the main character must be a character. This of course means more semantics included in the parsing.

Example 2

User : Bears eat honey.

System : I do not know the word honey.
Is honey a character, charactertype, hometype, location or foodtype?

It is assumed here that the word "bears" has been previously processed. A smarter atn would guess that the word following the verb "eat" must be a foodtype.

8. The atn can be extended to accept a larger subset of English. It can also be made less finicky. An attempt was made to ignore some grammatical errors on the part of the user but, the program could be made to ignore more grammatical errors.

9. The way the system is currently structured, there is one main character from whose point of view the story is narrated. Professor Coon has suggested that it may be possible to extend the system so that the different characters alternate as the main character with a novelist or story-writer (not necessarily a character in the story) to coordinate multiple plots. This would require a major revision of the system in order to handle the considerable amount of added complexity that would be necessary.

10. Story-writing failure may result in the following manner. For the system to tell a story, there must be a main character whose whereabouts have been established and who has a need (food, water, or sex). In the absence of such a framework, no story will result.

11. The understanding capability of the system is somewhere between that of ELIZA and CYRUS. It understands more than ELIZA as it does not simply respond to certain key words. On the other hand, it does not learn anything from the stories told before as CYRUS does. It would be an interesting and worthwhile project to develop a story-writing system that extracted and stored knowledge from a story it told.

12. There is no facility in the system to detect inconsistencies. A story-writing failure or an infinite loop may result in such a case.

4.5 Conclusion

What we have here is an experimental system that uses procedural logic to create simple, mildly intersting stories based on user-supplied information and problem solving methods available in the rule base. Prolog allows us to do this relatively neatly. The feature that sets this system apart from the original Tale-Spin is the independence of the data domain (user-supplied) from the rule base (hard-wired) that operates on this domain to create the stories.

There are many ways in which this work can be extended. The complexity of the stories produced depends upon the richness of the user input and the sophistication of the problem solving rules. Both of these can be enhanced in the existing framework. However, in order to tell more interesting stories, this framework would need revamping by perhaps including a meta-level processor controlled by a creator or an arbiter. One could conceive of the creator as a planner which is motivated by the following meta-themes:

1. Create an intersting story.
2. Make sure the story is coherent.

It would be interesting to explore the possibility of creating a system with a meta-panning facility that could understand as well

as create stories.

Schank's theory of remindings and memory structures might be useful in creating a planning system that would learn from its experience. For a story-writing-understanding system, it might mean a system capable of telling better stories as a result of its story-understanding experience. Suppose, while in understanding mode, the system finds a scene that differs from the stored prototypical scene. The difference is recorded and stored with the scene. All the MOPs and TOPs that share this scene are now cognizant of this knowledge, and the next time the system tells a story that utilizes any of those MOPs or TOPs, the story might have a new twist not encountered in the earlier stories that the system told.

BIBLIOGRAPHY

Appelt, D. E. 1985, Planning English Sentences, Cambridge University Press, Cambridge.

Barr, A., and Feigenbaum, E. A. 1981, The Handbook of Artificial Intelligence, Vol.1, William Kaufman, Los Altos, Callif.

Charniak, E., Riesbeck, C. K., and McDermott, D. V. 1980, Artificial Intelligence Programming, Lawrence Erlbaum Associates, Hillsdale, N.J.

Clocksin, W. F., and Mellish, C. S. 1981, Programming in Prolog, Springer Verlag, New York.

Cohen, P. R., and Feigenbaum, E. A., 1982, The Handbook of Artificial Intelligence, Vol3, William Kaufman, Los Altos, Calif.

Dehn, N. 1981, "Story Generation after Tale-Spin", Proc. IJCAI 7.

Fikes, R. E., Hart, P. E., and Nilsson, N. J., 1972, "Learning and Executing Generalized Plans", Artificial Intelligence 3:251-258.

Fikes, R. E., and Nilsson, N. J. 1971, "STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving", Artificial Intelligence 2:189-208.

Goldman, N. 1974, Computer Generation of Natural Language from a Deep Conceptual Base, Stanford Artificial Intelligence Laboratory Technical Report.

Kolodner, J. L. 1984, Retrieval and Organizational Strategies in Conceptual Memory: A Computer Model, Lawrence Erlbaum Associates, Hillsdale, N. J.

Kowalski, R. 1979, Logic for Problem Solving, American Elsevier, New York.

Lebowitz, M. 1985, "Creating a Story-Telling Universe", Proc. IJCAI 8.

Lenhert, W. G., and Ringle, M. H. 1977. Strategies for Natural Language Processing, Lawrence Erlbaum Associates, Hillsdale, N.J.

Meehan, J. R. 1976, The Metanovel: Writing Stories by Computer, Yale Computer Science Research Report 74.

Newell, A., and Simon, H. A., Human Problem Solving, Prentice Hall, Englewood Cliffs, N. J.

Sacerdoti, E. 1977, A Structure for Plan and Behavior, North Holland, Amsterdam.

Schank, R. C. 1975, Conceptual Information Processing, American Elsevier, New York.

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

Schank, R. C., and Riesbeck, C. K. 1981, Inside Computer Understanding, Lawrence Erlbaum Associates, Hillsdale, N. J.

Schank, R. C. 1982, Dynamic Memory: A Theory of Learning in Computers and People, Cambridge University Press, Cambridge.

Schank, R. C. 1984, The Cognitive Computer, Addison-Wesley, Reading, Mass.

Schank, R. C., and Hunter, L. 1985, "The Quest to Understand Thinking", Byte, Vol. 10, No. 4.

Simon, R. F., and Slocum, J. 1972, "Generating English Discourse from Semantic Nets", Communications of the ACM, Vol.15.

Weizenbaum, J. 1966, "ELIZA - A Computer Program for the Study of Natural Language Communication between Man and Machine", CACM 9: 36-45.

Wilensky, R. 1978, Understanding Goal-Based Stories, Doctoral

Thesis, Yale University.

Wilensky, R. 1983, Planning and Understanding, A Computational Approach to Human Understanding, Addison-Wesley, Reading, Mass.

Winograd, T. 1972, Understanding Natural Language, Academic Press, New York.

Woods, W. A. 1970, "Transition Network Grammars for Natural Language Analysis", CACM Vol. 13, No. 10: 591-606.

APPENDIX 1

Sample Stories Generated by the System

```
isis!lng9330[8] pr example.thirst


Jun 15 15:05 1986  example.thirst Page 1


Script started on Sun Jun 15 12:58:05 1986
kisis-lng9330[1] prolog
C-Prolog version 1.4
| ?- [talespin].
driver consulted 772 bytes 0.216667 sec.
more_info consulted 8176 bytes 2.033333 sec.
atn consulted 7040 bytes 1.800001 sec.
dict consulted 4764 bytes 1.350002 sec.
kbase consulted 4048 bytes 0.883334 sec.
inout consulted 2772 bytes 0.86667 sec.
tools consulted 120 bytes 0.050001 sec.
story consulted 768 bytes 0.150002 sec.
love_rules consulted 4788 bytes 1.066668 sec.
thirst_rules consulted 8144 bytes 1.816672 sec.
hunger_rules consulted 6640 bytes 1.383336 sec.
talespin consulted 48032 bytes 11.866661 sec.

yes
| ?- do.
Tell me something about the story.
Say "This is it." when done.
|: joe is the main character.
I do not know the word  joe.
Is  joe  a character,charactertype,hometype,location or foodtype?
|: character.
Is Joe  male or female?
|: m.
Ok.
|: joe is a bear.
I do not know the word  bear.
Is  bear  a character,charactertype,hometype,location or foodtype?
|: charactert.
Is  bear  singular or plural?
|: s.
What is the plural of  bear?
|: bears.
Ok.
|: jill is a bear.
I do not know the word  jill.
Is  jill  a character,charactertype,hometype,location or foodtype?
|: character.
Is Jill  male or female?
|: f.
Ok.
|: joe is thirsty.
Ok.
|: jill is hungry.
Ok.
|: bears eat honey.
I do not know the word  honey.
Is  honey  a character,charactertype,hometype,location or foodtype?
|: f.
Is  honey  singular or plural?
|: s.
Ok.
|: joe likes jill.
```

```
Ok.
|: bears live in caves.
I do not know the word  caves.
Is  caves  a character,charactertype,hometype,location or foodtype?
|: h.
Is  caves  singular or plural?
|: p.
What is the singular of  caves?
|: cave.
Ok.
|: joe is at the rock.
I do not know the word  rock.
Is  rock  a character,charactertype,hometype,location or foodtype?
|: l.
Ok.
|: jill is at home.
Ok.
|: this is it.
Let me see if you have told me, all I need.
Where is water?
|: in the river.
I do not know the word  river.
Is  river  a character,charactertype,hometype,location or foodtype?
|: l.
Tell me the location of other props, if you wish.
Say "This is it.", when finished.
|: honey is at the home of joe.
Ok.
|: this is it.
Let me see if you have told me, all I need.
Does Joe  know where water is?
|: n.
Who, if anybody, knows where  water  is?
|: jill.
Once upon a time Joe  was by the  rock  and was thirsty.
Joe  thought  he  would give  honey  to Jill  if  she  told  him  where water wa
s.
He  went to the home of Jill.
Joe  told Jill  that  he  would give  her  the  honey  if Jill  would tell  him
 the location of  water.
Joe  went home to get the  honey.
Joe  took the  honey  from the  joe_cave.
Joe  returned to the  jill_cave  and gave the  honey  to Jill.
Jill  knew where  water  was.
So  she  told Joe,  water  was at the  river.
Joe  went to the  river.
Joe  drank water from the  river.
He  was no longer thirsty.
The end.

Yes
| ?- ^D
[ Prolog execution halted ]
```

```
.sis!lng9330[10] pr example.hunger

Jul 10 21:17 1986  example.hunger Page 1


Script started on Sun Jun 15 15:47:34 1986
(isis-lng9330[1] prolog
)-Prolog version 1.4
  ?- [talespin].
driver consulted 772 bytes 0.2 sec.
more_info consulted 8176 bytes 2.066666 sec.
atn consulted 7040 bytes 1.750001 sec.
dict consulted 4764 bytes 1.366668 sec.
base consulted 4048 bytes 0.86667 sec.
mout consulted 2772 bytes 0.833335 sec.
tools consulted 120 bytes 0.050004 sec.
story consulted 768 bytes 0.15 sec.
love_rules consulted 4788 bytes 1.050003 sec.
thirst_rules consulted 8144 bytes 1.81667 sec.
hunger_rules consulted 6640 bytes 1.466672 sec.
talespin consulted 48032 bytes 11.883331 sec.

yes
| ?- do.
Tell me something about the story.
Say "This is it." when done.
|: mary is a bird.
I do not know the word  mary.
Is  mary  a character,charactertype,hometype,location or foodtype?
|: character.
Is Mary  male or female?
|: f.
I do not know the word  bird.
Is  bird  a character,charactertype,hometype,location or foodtype?
 : charactert.
Is  bird  singular or plural?
 : s.
What is the plural of  bird?
|: birds.
Ok.
|: john is a bear.
I do not know the word  john.
Is  john  a character,charactertype,hometype,location or foodtype?
|: character.
Is John  male or female?
|: m.
I do not know the word  bear.
Is  bear  a character,charactertype,hometype,location or foodtype?
|: charactert.
Is  bear  singular or plural?
|: s.
What is the plural of  bear?
|: bears.
Ok.
|: bears live in caves.
I do not know the word  caves.
Is  caves  a character,charactertype,hometype,location or foodtype?
|: h.
Is  caves  singular or plural?
 : p.
What is the singular of  caves?
```

```
|: cave.
)k.
|: mary is hungry.
)k.
|: john wants food.
)k.
|: bears eat honey.
: do not know the word  honey.
[s  honey  a character,charactertype,hometype,location or foodtype?
 : f.
[s  honey  singular or plural?
 : s.
)k.
|: birds eat worms.
[ do not know the word  worms.
[s  worms  a character,charactertype,hometype,location or foodtype?
|: f.
[s  worms  singular or plural?
|: p.
)k.
|: mary has honey.
)k.
|: worms are at the home of john.
)k.
|: this is it.
-et me see if you have told me, all I need.
Who is the main character?
|: mary.
Where does Mary  live?
|: in a tree.
[ do not know the word  tree.
[s  tree  a character,charactertype,hometype,location or foodtype?
|: h.
[s  tree  singular or plural?
|: s.
What is the plural of  tree?
|: trees.
Please specify which characters like which, if you wish.
Say "This is it." when done.
|: this is it.
-et me see if you have told me, all I need.
Where is Mary  currently?
|: at home.
Tell me where other characters are, if you wish.
Say "This is it." , when done.
|: joe is at home.
: do not know the word  joe.
[s  joe  a character,charactertype,hometype,location or foodtype?
|: character.
[s Joe  male or female?
: m.
)k.
|: john is at home.
)k.
|: this is it.
-et me see if you have told me, all I need.
```

```
Tell me the location of other props, if you wish.
Say "This is it.", when finished.
|: this is it.
Let me see if you have told me, all I need.
Does Mary  know where  worms  is?
|: n.
Once upon a time Mary  was by the  mary_tree  and was hungry.
She  had no food at home.
Mary  thought  she  would give the  honey  to John  if  he  would give  her  som
e food.
She  went to the home of John.
Mary  told John  that  she  would give  him  the  honey  if John  would give  he
r  some food.
Mary  gave the  honey  to John.
John  took the  worms  from the  john_cave.
John  gave the  worms  to Mary.
Mary  ate the  worms.
There  were  no more  worms.
Mary  was not hungry any more.
The end.

yes
| ?- ^D
[ Prolog execution halted ]
kisis-lng9330[2] exit
kisis-lng9330[3]
script done on Sun Jun 15 15:55:49 1986
```

```
isis!lng9330[6] pr example.love

Jul 10 21:09 1986  example.love Page 1


Script started on Sun Jun 15 15:16:41 1986
kisis-lng9330[1] prolog
C-Prolog version 1.4
| ?- [talespin].
driver consulted 772 bytes 0.216667 sec.
more_info consulted 8176 bytes 1.966666 sec.
atn consulted 7040 bytes 1.766667 sec.
dict consulted 4764 bytes 1.36667 sec.
dbase consulted 4048 bytes 0.966668 sec.
inout consulted 2772 bytes 0.866668 sec.
tools consulted 120 bytes 0.033337 sec.
story consulted 768 bytes 0.15 sec.
love_rules consulted 4788 bytes 1.066669 sec.
thirst_rules consulted 8144 bytes 1.8 sec.
hunger_rules consulted 6640 bytes 1.433335 sec.
talespin consulted 48032 bytes 11.933327 sec.

yes
| ?- do.
Tell me something about the story.
Say "This is it." when done.
|: john is a chimp.
I do not know the word  john.
Is  john  a character,charactertype,hometype,location or foodtype?
|: character.
Is John  male or female?
|: m.
I do not know the word  chimp.
Is  chimp  a character,charactertype,hometype,location or foodtype?
|: charactert.
Is  chimp  singular or plural?
|: s.
What is the plural of  chimp?
|: chimps.
Ok.
|: john is horny.
Ok.
|: sue is a chimp.
I do not know the word  sue.
Is  sue  a character,charactertype,hometype,location or foodtype?
|: character.
Is Sue  male or female?
|: f.
Ok.
|: kate is a chimp.
I do not know the word  kate.
Is  kate  a character,charactertype,hometype,location or foodtype?
|: character.
Is Kate  male or female?
|: f.
Ok.
|: sue wants sex.
Ok.
|: kate wants sex.
Ok.
|: chimps live in trees.
```

```
I do not know the word  trees.
Is  trees  a character,charactertype,hometype,location or foodtype?
|: h.
Is  trees  singular or plural?
|: p.
What is the singular of  trees?
|: tree.
Ok.
|: john likes sue.
Ok.
|: kate likes john.
Ok.
|: john is at home.
Ok.
|: sue is by the river.
I do not know the word  river.
Is  river  a character,charactertype,hometype,location or foodtype?
|: l.
Ok.
|: john knows where sue is.
Ok.
|: kate is at home.
Ok.
|: john likes kate.
Ok.
|: this is it.
Let me see if you have told me, all I need.
Who is the main character?
|: john.
Once upon a time John  was by the  john_tree  and wanted to make love.
John  liked Sue.
John  knew Sue  was at the  river  so  he  went there.
John  asked Sue  if  she  would like to make love.
Sue  refused because  she  did not like John.
John  liked Kate.
He  went to the home of Kate.
She  was at home.
John  asked Kate  if  she  would like to make love.
Kate  agreed because  she  liked John  and was in mood.
The end.

yes
| ?- ^D
[ Prolog execution halted ]
kisis-lng9330[2] exit
kisis-lng9330[3]
script done on Sun Jun 15 15:22:56 1986
```

Appendix 2

Prolog Code

```
sisling9830[2] cat code.the

eb  5 10:34 1986  driver Page 1


 * This is the driver module. */

o :-
   preamble,
   talk,
   get_more_info,
   story.


 reamble :-
   write_sent(['Tell me something about the story. ]),
   write_sent(['Say "This is it." when done.']).

talk :-
   repeat,          /* Keep processing until the user says "This is it."   */
   do_one(S),       /* Process one sentence.                               */
   S == [this,is,it,.].

do_one(S) :-
   repeat,           /* Keep reading until get a legal sentence.           */
   read_sent(S),     /* Turn the sentence read into a list of words in it. */
   check_words(S),   /* All words must be in dictionary.                   */
   s(S),             /* Invoke atn to extract the pertinent info.          */
   !.                /* Prevent backtracking if succeeds.                  */

istkb :-             /* For debugging.                                     */
   listing([ischar,char_type,foodtype,home_type,loc_name,main_char,at,food,
            male,female,home,singular,plural,species,wants,likes,knows_loc]).
```

```
jsis!lng3330[2] cat atn.ths

Jun  2 06:17 1986  atn Page 1


/* This file is named  atn .                                              */
/* This files contains the augmented transition network that analyzes the
   natural language sentence  handed it by the driver module, extracts the
   information from the sentence and adds it to the database. It also
   asks questions of the reader about the incoming information that the
   system needs to know and handles the incoming responses.
*/

s([First|Rest]) :-
        ischar(First),s_charname(First,Rest),write_sent(['ok.']).
s(L) :- s_chartype(L),write_sent(['ok.']).
s([First|Rest]) :-
        (foodtype(First) ; First == 'water'),
        s_props(First,Rest),
        write_sent(['ok.']).
s([this,is,it,.]) :-
        write_sent(['Let me see if you have told me, all I need. ]).
/* The following is tried when all else fails. */
s(L) :- write_sent(['I cannot understand you at all.']).

s_props(Subj,[First|Rest]) :-
        (First = 'is' ; First = 'are'),
        s_props_be(Subj,Rest).

s_props_be(Subj,[First|Rest]) :-
        (First = 'at' ; First = 'in' ; First = 'on' ; First = 'by'),
        s_props_be_prep(Subj,Rest).

s_props_be_prep(Subj,[First|Rest]) :-
        (First = 'the'),
        s_props_be_prep_the(Subj,Rest).

s_props_be_prep_the(Subj,[First|Rest]) :-
        loc_name(First),
        Rest = [.], !,
        assert(at(Subj,First)).
s_props_be_prep_the(Subj,[First|Rest]) :-
        (First = 'home'),
        Rest = [of,Z,.],
        ischar(Z), !,
        home(Z,P),
        assert(at(Subj,P)).


s_chartype([First|Rest]) :-
        char_type(X,First),  /* First is the plural form of a chartype */
        s_chartype_v(X,Rest).

s_chartype_v(Z,[First|Rest]) :-
        (First = 'eat'),
        s_chartype_eat(Z,Rest).

s_chartype_v(X,[First|Rest]) :-
        (First = 'live'),
        s_chartype_live(X,Rest).
```

```
_chartype_live(X,[First|Rest]) :-
        (First = 'in'),
        s_chartype_live_in(X,Rest).
_chartype_live_in(X,[First|Rest]) :-
        home_type(Y,First), /* First is the plural form of a hometype */
        assert(live(X,Y)).

s_chartype_eat(Z,[First|Rest]) :-
        foodtype(First),
        (Rest = [.]),
        assert(eat(Z,First)).


_charname(Subj,[First|Rest]):-
        (First = 'is'),s_charname_is(Subj,Rest).
s_charname(Subj,[First|Rest]) :-
        (First = 'knows'),
        s_charname_knows(Subj,Rest).
s_charname(Subj,[First|Rest]) :-
        (First = 'lives'),
        s_charname_lives(Subj,Rest).
s_charname(Subj,[First|Rest]) :-
        (First = 'eats'),
        s_charname_eats(Subj,Rest).
s_charname(Subj,[First|Rest]):-
        (First = 'likes'),
        s_charname_likes(Subj,Rest).
s_charname(Subj,[First|Rest]) :-
        (First = 'wants'),
        s_charname_wants(Subj,Rest).
s_charname(Subj,[First|Rest]) :-
        (First = 'has'),
        s_charname_has(Subj,Rest).

s_charname_has(Subj,[First|Rest]) :-
        foodtype(First),
        assert(has(Subj,First)).

s_charname_wants(Subj,[First|Rest]) :-
        (First = sex), !,
        assert(wants(Subj,sex)).
s_charname_wants(Subj,[First|Rest]) :-
        (First = food), !,
        assert(wants(Subj,food)).
s_charname_wants(Subj,[First|Rest]) :-
        (First = water), !,
        assert(wants(Subj,water)).


s_charname_likes(Subj,[First|Rest]) :-
        ischar(First),
        (Rest = [.]),
        assert(likes(Subj,First)).


s_charname_lives(Subj,[First|Rest]) :-
```

```
        (First = 'in'),
        s_charname_lives_in(Subj,Rest).
s_charname_lives_in(Subj,[First|Rest]) :-
        ((First = 'a');(First = 'an')),
        s_charname_lives_in_a(Subj,Rest).
s_charname_lives_in_a(Subj,[First|Rest]) :-
        hometype(First),
        (Rest = [.]),
        buildhome(Subj,First,Inst_home).
/* [john,lives,in,a,cave,.] leads to the following clause added to the database:
home(john,john_cave). */
buildhome(Subj,First,Inst_home) :-
        name(Subj,List1),
        name(First,List2),
        append(List1,"_",List3),
        append(List3,List2,List4),
        name(Inst_home,List4),
        assert(home(Subj,Inst_home)).


s_charname_knows(Subj,[First|Rest]) :-
        (First = 'where'),
        s_charname_knows_where(Subj,Rest).
s_charname_knows_where(Subj,[First|Rest]) :-
        ischar(First),
        (Rest = [is,.]),
        assert(knows_loc(Subj,First)).
s_charname_knows_where(Subj,[First|Rest]) :-
        (foodtype(First); First = 'water'),
        ((Rest = [is,.]);(Rest = [are,.])),
        assert(knows_loc(Subj,First)).

s_charname_eats(Subj,[First|Rest]) :-
        foodtype(First),
        (Rest = [.]),
        assert(food(Subj,First)).



s_charname_is(Subj,[First|Rest]) :-
        ((First = 'a');(First = 'an')),!,s_charname_is_a(Subj,Rest)
s_charname_is(Subj,[First|Rest]) :-
        (First = 'the'),s_charname_is_the(Subj,Rest).
s_charname_is(Subj,[First|Rest]):-
        (First = 'at'; First = 'on'; First = 'in'; First = 'by'),
        s_charname_is_prep(Subj,Rest).
s_charname_is(Subj,[First|Rest]) :-
        statename(First),
        (Rest = [.]),
        s_charname_is_state(Subj,First).

statename(W) :-
        ((W = 'hungry');(W = 'thirsty');(W = 'horny')).
s_charname_is_state(Subj,State) :-
        (State = 'hungry'),!,
        assert(wants(Subj,food)).
s_charname_is_state(Subj,State) :-
```

```
        (State = 'thirsty'),!,
        assert(wants(Subj,water)).
s_charname_is_state(Subj,State) :-
        (State = 'horny'),!,
        assert(wants(Subj,sex)).



s_charname_is_prep(Subj,[First|Rest]) :-
        (First = 'home'),
        (Rest = [.]),!,
        home(Subj,Z),
        assert(at(Subj,Z)).
s_charname_is_prep(Subj,[First|Rest]) :-
        (First = 'the'),
        s_charname_is_prep_the(Subj,Rest).

s_charname_is_prep_the(Subj,[First|Rest]) :-
        (First = 'home'),
        s_charname_is_prep_the_home(Subj,Rest).
s_charname_is_prep_the(Subj,[First|Rest]) :-
        loc_name(First),
        assert(at(Subj,First)).

s_charname_is_prep_the_home(Subj,[First|Rest]) :-
        (First = 'of'),
        s_charname_is_prep_the_home_of(Subj,Rest).

s_charname_is_prep_the_home_of(Subj,[First|Rest]) :-
        ischar(First),
        (Rest = [.]),!,
        home(First,Z),
        assert(at(Subj,Z)).



s_charname_is_a(Subj,[First|Rest]) :-
        chartype(First),(Rest=[.]),assert(species(Subj,First)).
s_charname_is_the(Subj,[First|Rest]) :-
        (First = 'main'),
        (Rest == [character,.]),
        assert(main_char(Subj)).
```

```
:sis'lng8330[4] cat dict.ths

Feb 19 15:55 1985  dict Page 1


:hartype(X) :- char_type(X,_).

:ometype(X) :- home_type(X,_).

:heck_words([W|Rest]) :- in_dic(W),!,check_words(Rest).
:heck_words([W|Rest]) :- bad_word(W),!,check_words(Rest).
:heck_words([]).

bad_word(W) :-
      write_sent(['I do not know the word',W,.]),
      write_sent(['Is',W,'a character,charactertype,hometype,location or foodtyp
e?']),
      read_sent(Ans),
      get_first_word(Ans,PST),    /* PST = possible semantic-type */
      get_st(PST,ST),             /* ST = legal semantic-type     */
      put_in_dic(W,ST).

get_first_word([First|_],First).

get_st(character,ischar).
get_st(charactertype,char_type).
get_st(hometype,home_type).
get_st(location,loc_name).
get_st(l,loc_name).
get_st(loc,loc_name).
get_st(h,home_type).
get_st(home,home_type).
get_st(charactert,char_type).
get_st(foodtype.foodtype).
get_st(food,foodtype).
get_st(f,foodtype).


in_dic(W) :-
      dic_entry(W);
      char_type(W,_);
      char_type(_,W);
      home_type(W,_);
      home_type(_,W);
      foodtype(W);
      ischar(W);
      loc_name(W);
      punctuation(W).


put_in_dic(W,ischar) :-
      assert(ischar(W)),
      write_sent(['Is',W,'male or female?']),
      read_sent(Ans),
      get_first_word(Ans,Sex),
      det_sex(W,Sex).

det_sex(W,Sex) :-
      ((Sex == 'male');(Sex == 'm')), !,
      assert(male(W)).
det_sex(W,Sex) :-
```

```
      ((Sex == 'female');(Sex == 'f')), !,
      assert(female(W)).


put_in_dic(W,char_type) :-
      write_sent([ Is',W, singular or plural?']),
      read_sent(Ans),
      get_sing_and_pl(W,Ans,S,P), /* S and P are sing. and plural forms of W*/
      assert(char_type(S,P)).

get_sing_and_pl(W,Ans,S,P) :-
     get_first_word(Ans,PNum),
     s_or_p(PNum,W,S,P).

s_or_p(PNum,W,W,P) :-
     (PNum == s ; PNum == sing ; PNum == singular), !,
     write_sent(['What is the plural of',W,'?']),
     read_sent(Ans),
     get_first_word(Ans,P).
s_or_p(PNum,W,S,W) :-
     (PNum == p ; PNum == plu ; PNum == plural), !,
     write_sent(['What is the singular of',W,'?']),
     read_sent(Ans),
     get_first_word(Ans,S).

put_in_dic(W,home_type) :-
        write_sent(['Is',W,'singular or plural?']),
        read_sent(Ans),
        get_sing_and_pl(W,Ans,S,P),
        assert(home_type(S,P)).


put_in_dic(W,loc_name) :- assert(loc_name(W)).


put_in_dic(W,foodtype) :-
        assert(foodtype(W)),
        write_sent(['Is',W,'singular or plural?']),
        read_sent(Ans),
        get_first_word(Ans,PNum),
        get_s_or_p(PNum,W).

get_s_or_p(PNum,W) :-
        (PNum == s ; PNum == sing ; PNum == singular),
        !, assert(singular(W)).
get_s_or_p(PNum,W) :-
        (PNum == p ; PNum == pl ; PNum == plural), !,
        assert(plural(W)).


plural(W) :- char_type(_,W) ; home_type(_,W).

singular(W) :- char_type(W,_);home_type(W,_),loc_name(W);ischar(W).


/* System needs to know the following words. They are hard-wired. */
```

```
dic_entry( is').
dic_entry('are').
dic_entry( a').
dic_entry( an').
dic_entry('the').
dic_entry('this').
dic_entry('it').
dic_entry('he').
dic_entry('his').
dic_entry('her').
dic_entry('she').
dic_entry('home').
dic_entry('water').
dic_entry('food').
dic_entry('sex').
dic_entry('of').
dic_entry('at').
dic_entry('in').
dic_entry('on').
dic_entry('by').
dic_entry('has').
dic_entry('wants').
dic_entry('knows').
dic_entry('lives').
dic_entry('eats').
dic_entry('likes').
dic_entry('live').
dic_entry('eat').
dic_entry('where').
dic_entry('main').
dic_entry('character').
dic_entry('hungry').
dic_entry('horny').
dic_entry('thirsty').
```

```
* This is more_info file. get_more_info, checks if the system has enough
   information to tell the story and queries the user to get additional
   information, if necessary.
*/

get_more_info :-
   specify_main_char,
   specify_theme,
   specify_want_others,
   specify_home,
   specify_food,
   specify_likes,
   specify_char_loc,
   specify_props_loc,
   specify_knows_loc_props,  /* Props are food and water.                */
   specify_knows_loc_char.


specify_main_char :- main_char(X), !.
specify_main_char :-
   ischar(Z),                  /* If there are more than one characters   */
   ischar(Y),                  /* query the user about the main character */
   (Z \== Y), !,
   write_sent(['Who is the main character?']),
   repeat,                    /* Try until a legal main character is given */
   read_sent(Ans),
   get_first_word(Ans,X),
   ischar(X),!,
   assert(main_char(X)),
   specify_main_char.
specify_main_char :-
   ischar(X),
   assert(main_char(X)).


specify_theme :-
   main_char(X),
   wants(X,_),!.
specify_theme :-
   main_char(X),
   write_sent(['What does',X,'want: food,water or sex?']),
   repeat,                     /* Try until a legal theme is specified. */
   read_sent(Ans),
   get_first_word(Ans,Y),
   (Y == food ; Y == water ; Y == sex), !,
   assert(wants(X,Y)),
   specify_theme.

specify_want_others :-
   ischar(X),
   not(main_char(X)),
   not(wants(X,_)), !,
   write_sent(['Specify what other characters want, if you wish.']),
   write_sent(['Say "This is it.", when done.']),
   talk.
specify_want_others.
```

```
specify_home :-
   specify_home_main_char,
   specify_home_others.

specify_home_main_char :-
   main_char(X),
   home(X,Y), !.
specify_home_main_char :-
   main_char(X),
   write_sent(['Where does',X,'live?']),
   repeat,                  /* Try until a valid home is established. */
   read_sent(S),
   check_words(S),
   extract_home_facts(X,S),
   !,specify_home_main_char.

extract_home_facts(X,S) :-
   (S = [in,a,Y,.]),
   hometype(Y), !,
   buildhome(X,Y,Z).
extract_home_facts(X,S) :- s(S).


specify_home_others :-
   ischar(X),
   not(main_char(X)),
   not(home(X,_)),!,       /* There is a character whose home is not known */
   write_sent(['Tell me where other characters live,if you wish.']),
   write_sent(['say "This is it." when done.']),
   talk.
specify_home_others.       /* Catch all clause                              */

specify_char_loc :-
   specify_main_char_loc,
   specify_other_char_loc.

specify_main_char_loc :-
   main_char(X),
   at(X,P), !.
specify_main_char_loc :-
   main_char(X),
   write_sent(['where is',X,'currently?']),
   repeat,    /* Get a legal location. */
   read_sent(Ans),
   check_words(Ans),
   extract_loc_facts(X,Ans), !,
   specify_main_char_loc.

extract_loc_facts(X,Ans) :-
   Ans = [First,the,Y,.],
   (First = at ; First = on ; First = by ; First = in),
   loc_name(Y), !,
   assert(at(X,Y)).
extract_loc_facts(X,Ans) :-
```

```
   name(X,Ans) ; Ans = [at,his,_,.] ; Ans = [at,her,_,.]),
   assert(at(X,P)).
extract_loc_facts(X,Ans) :-
   Ans = [at,the,home,of,Y,.],
   ischar(Y),
   home(Y,P), !,
   assert(at(X,P)).
extract_loc_facts(X,Ans) :- s(Ans).

specify_other_char_loc :-
   ischar(X),
   not(main_char(X)),
   not(at(X,_)), !,
   write_sent(['Tell me where other characters are. if you wish.']),
   write_sent(['Say "This is it." , when done.']),
   talk.
specify_other_char_loc.

specify_props_loc :-
   main_char(X),
   wants(X,water),
   at(water,P), !,
   specify_other_props_loc.
specify_props_loc :-
   main_char(X),
   wants(X,water),
   write_sent(['Where is water?']),
   repeat,   /* Get a valid loc_name or home for answer. */
   read_sent(Ans),
   check_words(Ans),
   extract_props_loc(water,Ans), !,
   specify_other_props_loc.
specify_props_loc :-
   main_char(X),
   wants(X,food),
   food(X,Y),
   at(Y,P), !,
   specify_other_props_loc.
specify_props_loc :-
   main_char(X),
   wants(X,food),
   food(X,Y),
   has(_,Y), !,
   specify_other_props_loc.
specify_props_loc :-
   main_char(X),
   wants(X,food),
   food(X,Y),
   write_sent(['Where is',Y,'?']),
   repeat,   /* Get a valid loc_name or home for the answer. */
   read_sent(Ans),
   check_words(Ans),
   extract_props_loc(Y,Ans), !,
   specify_other_props_loc.
specify_props_loc.  /* Catch all clause */
```

```
extract_props_loc(Y,Ans) :-
   Ans = [Prep,the,Z,.],
   (Prep = in ; Prep = at ; Prep = by ; Prep = on),
   loc_name(Z), !,
   assert(at(Y,Z)).
extract_props_loc(Y,Ans) :-
   Ans = [at,the,home,of,Z,.],
   ischar(Z), !,
   home(Z,P),
   assert(at(Y,P)).
extract_props_loc(Y,Ans) :-
   Ans = [First,has,_,.],
   ischar(First),
   assert(has(First,Y)).
extract_props_loc(Y,Ans) :- s(Ans).


specify_other_props_loc :-
   (Y = water ; foodtype(Y)),
   not(at(Y,_)),
   not(has(_,Y)),
   write_sent(['Tell me the location of other props, if you wish.']),
   write_sent(['Say "This is it.", when finished.']),
   talk.
specify_other_props_loc.

/* If the main character wants food, make sure a foodtype for him has
   been specified.
*/

specify_food :-
   main_char(X),
   not(wants(X,food)), !.
specify_food :-
   main_char(X),
   wants(X,food),
   food(X,Y), !.
specify_food :-
   main_char(X),
   wants(X,food),
   write_sent(['What does',X,'eat?']),
   repeat,   /* Get a legal foodtype. */
   read_sent(Ans),
   check_words(Ans),
   extract_food_facts(X,Ans), !,
   specify_food.

extract_food_facts(X,Ans) :-
   get_first_word(Ans,Y),
   foodtype(Y), !,
   assert(food(X,Y)).
extract_food_facts(X,Ans) :-
   Ans = [Y,eats,Z,.],
   (Y = he ; Y = she ; Y == X),
   foodtype(Z), !,
```

```
   assert(food(X,Z)).

specify_likes :-
   likes(_,_), !.
specify_likes :-
   ischar(Y),          /* Query the user, only if at least 2 characters       */
   not(main_char(Y)),
   write_sent(['please specify which characters like which, if you wish. ']),
   write_sent(['say "This is it." when done.']),
   talk.
specify_likes.       /* Catch all clause                                       */


specify_knows_loc_props :-
   main_char(X),
   wants(X,water),
   knows_loc(X,water), !.
specify_knows_loc_props :-
   main_char(X),
   wants(X,water),
   write_sent(['Does ',X,'know where water is?']),
   repeat,    /* Get a legal response. */
   read_sent(Ans),
   extract_knows_loc(X,Ans,water), !.
specify_knows_loc_props :-
   main_char(X),
   wants(X,food),
   food(X,Y),
   knows_loc(X,Y), !.
specify_knows_loc_props :-
   main_char(X),
   wants(X,food),
   food(X,Y),
   write_sent(['Does',X,'know where',Y,'is?']),
   repeat,    /* Get a legal response. */
   read_sent(Ans),
   extract_knows_loc(X,Ans,Y), !.
specify_knows_loc_props.     /* Catch all clause */

extract_knows_loc(X,Ans,Z) :-
   get_first_word(Ans,A),
   (A = yes ; A = y),
   assert(knows_loc(X,Z)).
extract_knows_loc(X,Ans,Z) :-
   get_first_word(Ans,A),
   (A = no ; A = n),
   knows_loc(_,Z), !.
extract_knows_loc(X,Ans,Z) :-
   get_first_word(Ans,A),
   (A = no ; A = n),
   write_sent(['Who, if anybody, knows where',Z, is?']),
   repeat,
   read_sent(Response),
   get_first_word(Response,Y),
   deal_with(Y,Z), !.
```

```
deal_with(Y,Z) :-
   ischar(Y), !,
   assert(knows_loc(Y,Z)).
deal_with(Y,Z) :-
   (Y = nobody ), ! .



specify_knows_loc_char :-
   main_char(X),
   wants(X,sex),
   sex_poss(X,Y),
   likes(X,Y),
   knows_loc(X,Y), !.
specify_knows_loc_char :-
   main_char(X),
   wants(X,sex),
   sex_poss(X,Y),
   write_sent(['Does',X,'know where',Y,'is?']),
   repeat,
   read_sent(Ans),
   extract_knows_loc(X,Ans,Y), !.
specify_knows_loc_char.          /* Catch all clause */
```

```
isis!lng3330[3] pr inout


Apr 20 20:46 1986  inout Page 1


/* Following procedure reads in a sentence */
read_sent([W|Ws]) :- get0(C), readword(C,W,C1), restsent(W,C1,Ws).

/* Given a word and the character after it, read in the rest of the sentence */

restsent(W,_,[]) :- lastword(W), !.
restsent(W,C,[W1|Ws]) :- readword(C,W1,C1), restsent(W1,C1,Ws).

/* Read in a single word, given an initial character, and remembering what
   character came after the word */

readword(C,W,C1) :- single_char(C), !, name(W,[C]), get0(C1).
readword(C,W,C2) :-
       in_word(C,NewC), !,
       get0(C1),
       restword(C1,Cs,C2),
       name(W,[NewC|Cs]).
readword(C,W,C2) :- get0(C1), readword(C1,W,C2).

restword(C,[NewC|Cs],C2) :-
       in_word(C,NewC), !,
       get0(C1),
       restword(C1,Cs,C2).
restword(C,[],C).

/* These characters form words on their own */

single_char(44).     /* , */
single_char(59).     /* ; */
single_char(58).     /* : */
single_char(63).     /* ? */
single_char(33).     /* ! */
single_char(46).     /* . */

/* these characters can appear within a word */
/* The second in_word clause converts characters to lower-case */

in_word(C,C) :- C> 96, C< 123.          /* a b .. z */
in_word(C,L) :- C> 64, C< 91, L is C + 32.  /* A B .. Z */
in_word(C,C) :- C> 47, C< 58.                /* 1 2 .. 9 */
in_word(39,39).                             /* ' */
in_word(45,45).                             /* - */

/* These words terminate a sentence */

lastword('.').
lastword('!').
lastword('?').

/* Write a sentence nicely. This does the opposite of read_sent above.
   Writes on standard output the "sentence" represented by a list of
   atoms. Capitalizes the first word (and known proper nouns ), spaces
   punctuation appropriately, and prints a new-line after sentence.
*/

write_sent([]) :- nl.          /* Handle null sentence                    */
```

```
write_sent([First|Rest]) :-
        cap(First,CapFirst),        /* Capitalize first word              */
        write(CapFirst),
        write_rest(Rest).

write_rest([]) :- nl.               /* Handle end of sentence             */
write_rest([Next|Rest]) :-
        punctuation(Next), !,       /* Don't space before punctuation     */
        write(Next),
        write_rest(Rest).
write_rest([Next|Rest]) :-          /* Capitalize first letter of a name */
        tab(1),                     /* Space before a word                */
        proper_noun(Next), !,
        cap(Next,NewNext),
        write(NewNext),
        write_rest(Rest).
write_rest([Next|Rest]) :-
        tab(1),                    /* Space before a word                 *
        write(Next),
        write_rest(Rest).

cap(Word,CapWord) :-
        name(Word,[First|Rest]), /* Explode word to be capitalized    */
        caplet(First,NewFirst),  /* Capitalize first letter           */
        name(CapWord,[NewFirst|Rest]). /*Put capitalized word back together*/

caplet(C,MapC) :-                  /* Capitalize lower case letter      */
        C>="a", C=<"z", !,
        MapC is C-"a"+"A".
caplet(C,C).


punctuation(',').                  /* punctuation atoms                 */
punctuation(';').
punctuation( :').
punctuation('?').
punctuation('!').
punctuation('.').
```

```
/* This is the file kbase, containing hard wired relations.                    */

food(X,Y) :-
    species(X,Z),
    eat(Z,Y).

home(C,H) :-
    species(C,X),
    live(X,Y),          /* species X lives in hometype Y                       */
    buildhome(C,Y,H). /* H is the instance of home built for character C */

opp_sex(X,Y) :-
    male(X),female(Y).
opp_sex(X,Y) :-
    female(X),male(Y).

sex_poss(X,Y) :-
    opp_sex(X,Y),
    species(X,Z),
    species(Y,Z).

knows_loc(X,Y) :-
    ischar(X),
    (ischar(Y); foodtype(Y); (Y = water)),
    at(X,P),
    at(Y,P).
knows_loc(X,Y) :-
    has(X,Y).

print_knows_loc(X,Y) :-
    at(Y,Q),
    knows_loc(X,Y), !,
    det_aux_verb(Y,V),
    write_sent([X,'knew',Y,V,'at the',Q,'. ]).
print_knows_loc(X,Y) :-
    det_aux_verb(Y,V),
    write_sent([X,'did not know where',Y,V,'.']).


proper_noun(X) :-     /* Character-names are proper nouns                      */
    ischar(X).

det_nom_pronoun(X,Y) :-
    male(X), !, (Y = he).
det_nom_pronoun(X,Y) :-
    female(X), !, (Y = she).

det_obj_pronoun(X,Y) :-
    male(X), !, (Y = him).
det_obj_pronoun(X,Y) :-
    female(X), !, (Y = her).

det_poss_pronoun(X,Y) :-
    male(X), !, (Y = his).
det_poss_pronoun(X,Y) :-
    female(X), !, (Y = her).
```

```
det_aux_verb(X,Y) :-
    singular(X), !,
    (Y = was).
det_aux_verb(X,Y) :-
    plural(X), !,
    (Y = were).


ptrans(W,W,X,Y) :-          /* Transport W from X to Y                    */
    retract(at(W,X)),
    assert(at(W,Y)).

/* desire_ptrans(X,Y,P,Q) does the following : character X is at P.
   Character Y is at Q. X wants to go where Y is. If both are at the
   same location, an appropriate statement, indicating this is printed.
   If not and X knows where Y is, X goes there. If X does not know where
   Y is, he/she goes to the home of Y, provided such a home exists.
*/

desire_ptrans(X,Y,P,Q) :-
    (P == Q), !,
    write_sent(['Both',X, and',Y,'were at the',P, .']).
desire_ptrans(X,Y,P,Q) :-
    (P \== Q), !,
    dest(X,Y,P,Q).

dest(X,Y,P,Q) :-
     knows_loc(X,Y), !,
     ptrans(X,X,P,Q),
     print_desire1(X,Y,P,Q).
dest(X,Y,P,Q) :-
     home(Y,R),
     ptrans(X,X,P,R),
     print_desire1(X,Y).


atrans(X,Y,Z) :-            /* Transfer possession of Z from X to Y      */
     retract(has(X,Z)),
     assert(has(Y,Z)).

take(X,Z) :-                /* X takes Z                                 */
     at(Z,P),
     retract(at(Z,P)),
     assert(has(X,Z)),
     write_sent([X,'took the',Z,'from the',P,'. ]).

give(X,Y,Z) :-              /* X gives Z to Y                            */
     has(X,Z), !,
     atrans(X,Y,Z),
     write_sent([X,'gave the',Z,'to',Y,'.']).
give(X,Y,Z) :-
     at(Z,P),
     home(X,P),
     at(X,P), !,
     take(X,Z),
```

```
     atrans(X,Y,Z),
     write_sent([X,'gave the',Z, to',Y,'.']).
give(X,Y,Z) :-                  /* X goes home to get Z to give to Y          */
     at(Z,P),
     home(X,P),
     not(at(X,P)), !,
     ptrans(X,X,_,P),
     write_sent([X,'went home to get the',Z,'.']),
     take(X,Z),
     at(Y,Q),
     ptrans(X,X,P,Q),
     atrans(X,Y,Z),
     write_sent([X,'returned to the',Q, and gave the',Z, to',Y,'.']).
```

```
append([],L,L).                 /* Append 1st argument to 2nd, return in 3rd */
append([X|L1],L2,[X|L3]) :-
      append(L1,L2,L3).
```

```
/* This is the file named story. The story goal calls the various procedures
   to put together the story.
*/

story :-
   main_char(X),
   wants(X,food),
   at(X,P),
   write_sent(['Once upon a time',X,'was by the',P, and was hungry.']),
   food(X,Z),
   sat_hunger(X),
   write_sent(['The end. ]).
story :-
   main_char(X),
   wants(X,water),
   at(X,P),
   write_sent(['Once upon a time',X,'was by the',P, and was thirsty. ]),
   sat_thirst(X),
   write_sent(['The end.']).
story :-
   main_char(X),
   wants(X,sex),
   at(X,P),
   write_sent(['Once upon a time',X,'was by the',P,'and wanted to make love. ])
   sat_desire(X),
   write_sent(['The end.']).
```

```
isis!lng9330[6] pr thirst_rules


Jun  2 06:31 1986  thirst_rules Page 1


/* This is the file thirst_rules. The goal sat_thirst(X) is invoked by the
   character X to satisfy his/her thirst.
*/

sat_thirst(X) :-
    at(X,P),
    at(water,P), !,
    print_thirst1(X),        /* Print the location of water                */
    print_thirst2(X),        /* X drank water                              */
    retract(wants(X,water)).
sat_thirst(X) :-
    knows_loc(X,water), !,
    at(X,P),
    at(water,Q),
    ptrans(X,X,P,Q),
    print_thirst3(X),       /* X knew where water was                      */
    print_thirst2(X),       /* X drank water etc.                          *
    retract(wants(X,water)).
/* If X has something to trade in return for knowing where water is, he
   or she tries to do so.
*/
sat_thirst(X) :-
    trade_filter1(X,Y,Z),
    sat_thirst(X,Y,Z).
sat_thirst(X) :-
    trade_filter2(X,Y,Z),
    sat_thirst(X,Y,Z).
/* If main character X does not know where water is and has not been able
   to trade something for the knowledge of the location of water, he finds
   a character Y he/she likes and asks for the location of water. If Y
   likes X and knows where water is, he/she tells X the location of water.
*/      .
sat_thirst(X) :-
    filter1_thirst(X,Y,P,Q),
    sat_thirst(X,Y,P,Q).
sat_thirst(X) :-
    filter2_thirst(X,Y,P,Q),
    sat_thirst(X,Y,P,Q).
/* The difference between two filters is that in the first X and Y are at
   the same location and in the second they are not.
*/
sat_thirst(X) :-
    det_poss_pronoun(X,Y),
    write_sent([X,'was not able to quench',Y,'thirst.']).


trade_filter1(X,Y,Z) :-    /* X has something to trade                     */
    has(X,Z),
    ischar(Y),
    wants(Y,food),
    food(Y,Z).

trade_filter2(X,Y,Z) :-    /* X has something at home, he/she can trade    */
    ischar(Y),
    wants(Y,food),
    food(Y,Z),
```

```
    at(Z,P),
    home(X,P).

filter1_thirst(X,Y,P,Q) :-
    likes(X,Y),
    at(X,P),
    at(Y,Q),
    (P == Q).

filter2_thirst(X,Y,P,Q) :-
    likes(X,Y),
    at(X,P),
    at(Y,Q),
    (P \== Q).
/* Above arrangement ensures that P == Q case, requiring no ptrans is
   tried first. This is sensible because X should first seek information
   from the animals at the same location before he/she makes a trip to
   ask someone else.
*/

sat_thirst(X,Y,Z) :-
    knows_loc(X,Y),
    knows_loc(Y,water), !,
    at(X,P),at(Y,Q),
    print_trade_thought(X,Y,Z),
    desire_ptrans(X,Y,P,Q),
    print_trade_food_info(X,Y,Z,water),
    give(X,Y,Z),                          /* X gave Z to Y                    */
    tells_loc(Y,X,water),                 /* Y tells X the location of water */
    at(water,R),
    ptrans(X,X,_,R),
    print_thirst7(X),
    print_thirst2(X),
    retract(wants(X,water)).
sat_thirst(X,Y,Z) :-
    knows_loc(X,Y),
    not(knows_loc(Y,water)), !,
    not(knows_not_loc(X,Y,water)),
/* X does not know that Y does not know where water is                        */
    at(X,P),at(Y,Q),
    print_trade_thought(X,Y,Z),
    desire_ptrans(X,Y,P,Q),
    print_trade_food_info(X,Y,Z,water),
    print_thirst8(Y,water),
    print_thirst8(X,Y,water),
    assert(knows_not_loc(X,Y,water)), !,
/* X now knows that Y does not know where water is                            */
    fail.
sat_thirst(X,Y,Z) :-
    not(knows_loc(X,Y)),
    at(water,S),
    knows_loc(Y,water),
    at(X,P),at(Y,Q),
    home(Y,R),
    (R == Q),
    print_trade_thought(X,Y,Z),
```

```
    desire_ptrans(X,Y,P,Q),
    print_trade_food_info(X,Y,Z,water),
    give(X,Y,Z),
    tells_loc(Y,X,water),              /* Y tells X where water is          */
    ptrans(X,X,_,S),
    print_thirst7(X),
    print_thirst2(X),
    retract(wants(X,water)).
sat_thirst(X,Y,Z) :-
    not(knows_loc(X,Y)),
    at(X,P),at(Y,Q),
    home(Y,R),
    not(knows_not_home(X,Y)),          /* X does not know Y is not home     */
    (Q \== R),
    print_trade_thought(X,Y,Z),
    desire_ptrans(X,Y,P,Q),
    print_desire1(Y),                  /* Y was not home                    */
    assert(knows_not_home(X,Y)), !,
    fail.

sat_thirst(X,Y,P,Q) :-
    at(water,S),
    knows_loc(X,Y),
    knows_loc(Y,water),
    likes(Y,X), !,
    print_desire2(X,Y),                /* X liked Y                         */
    desire_ptrans(X,Y,P,Q),
    print_thirst4(X,Y,water),          /* X asked Y where water was         */
    print_desire2(Y,X),
    tells_loc(Y,X,water),
    ptrans(X,X,_,S),
    print_thirst7(X),                  /* X goes where water is             */
    print_thirst2(X),                  /* X drinks the water etc.           */
    retract(wants(X,water)).
sat_thirst(X,Y,P,Q) :-
    knows_loc(Y,water),
    knows_loc(X,Y),
    not(likes(Y,X)), !,
    print_desire2(X,Y),
    desire_ptrans(X,Y,P,Q),
    print_thirst4(X,Y,water),
    print_thirst5(X,Y,water),
    fail.
sat_thirst(X,Y,P,Q) :-
    knows_loc(X,Y),
    not(knows_loc(Y,water)),
    not(knows_not_loc(X,Y,water)), !,
/* X does not know that Y does not know where water is                        */
    desire_ptrans(X,Y,P,Q),
    print_desire2(X,Y),
    print_thirst4(X,Y,water),
    print_thirst6(Y,water),
    assert(knows_not_loc(X,Y,water)),
    fail.
sat_thirst(X,Y,P,Q) :-
    not(knows_loc(X,Y)),
```

```
    knows_loc(Y,water),
    home(Y,R),
    (R == Q),
    at(water,S),
    likes(Y,X), !,
    print_desire2(X,Y),
    desire_ptrans(X,Y,P,Q),
    print_desire2(Y),
    print_thirst4(X,Y,water),
    print_desire2(Y,X),
    tells_loc(Y,X,water),
    ptrans(X,X,_,S),
    print_thirst7(X),
    print_thirst2(X),
    retract(wants(X,water)).
sat_thirst(X,Y,P,Q) :-
    not(knows_loc(X,Y)),
    (knows_loc(Y,water)),
    home(Y,R),
    (R == Q),
    not(likes(Y,X)), !,
    print_desire2(X,Y),
    desire_ptrans(X,Y,P,Q),
    print_desire2(Y),
    print_thirst4(X,Y,water),
    print_thirst5(X,Y,water),
    fail.
sat_thirst(X,Y,P,Q) :-
    not(knows_loc(X,Y)),
    not(knows_loc(Y,water)),
    not(knows_not_loc(X,Y,water)),
/* X does not know that Y does not know where water is                    */
    home(Y,R),
    (R == Q), !,
    print_desire2(X,Y),
    desire_ptrans(X,Y,P,Q),
    print_desire2(Y),
    print_thirst4(X,Y,water),
    print_thirst6(Y,water),
    assert(knows_not_loc(X,Y,water)),
/* X now knows that Y does not know where water is                        */
    fail.
sat_thirst(X,Y,P,Q) :-
    not(knows_loc(X,Y)),
    home(Y,R),
    (R \== Q),
    not(knows_not_home(X,Y)), !,
    print_thirst6(X,water),      /* X did not know where water was        */
    print_desire2(X,Y),          /* X liked Y                             */
    desire_ptrans(X,Y,P,Q),
    print_desire1(Y),            /* Y was not nome                        */
    assert(knows_not_home(X,Y)),/* X now knows Y is not home              */
    fail.

print_trade_thought(X,Y,Z) :-
    det_nom_pronoun(X,T),
```

```
    det_obj_pronoun(X,U),
    det_nom_pronoun(Y,V),
    write_sent([X,'thought',T,'would give',Z, to',Y, if',V,'told',U,
    'where water was.' ]).

print_trade_food_info(X,Y,Z,U) :-
    det_nom_pronoun(X,T),
    det_obj_pronoun(X,W),
    det_obj_pronoun(Y,V),
    write_sent([X,'told',Y,'that',T,'would give',V,'the',Z, if',Y,'would tell',
    W,'the location of',U,'.']).

tells_loc(Y,X,Z) :-          /* Y told X the location of Z                    */
    det_nom_pronoun(Y,T),
    knows_loc(Y,Z),
    at(Z,P),
    assert(knows_loc(X,Z)),
    write_sent([Y,'knew where',Z,'was.']),
    write_sent(['So',T,'told',X,', ',Z,'was at the',P, .']).



print_thirst1(X) :-
    at(water,P),
    write_sent(['There was water in the',P,'.']).

print_thirst2(X) :-
    at(water,P),
    det_nom_pronoun(X,Y),
    write_sent([X,'drank water from the',P,'.']),
    write_sent([Y,'was no longer thirsty.']).

print_thirst3(X) :-
    at(water,P),
    det_nom_pronoun(X,Y),
    write_sent([X,'knew water was in the',P,'.']),
    write_sent([Y,'went to the',P, . ]).

print_thirst4(X,Y,Z) :-
    write_sent([X,'asked',Y,'where',Z,'was','.']).

print_thirst5(X,Y,Z) :-
   det_nom_pronoun(Y,T),
   write_sent(['Since',Y,'did not like',X,T,'did not tell ,X,'where',Z,'was.']).

print_thirst6(Y,Z) :-
    write_sent([Y,'did not know where',Z,'was. ]).

print_thirst7(X) :-
    at(water,P),
    write_sent([X,'went to the',P, .']).

print_thirst8(X,Y,P) :-
    det_nom_pronoun(Y,Z),
    write_sent(['So',Z,'could not tell',X,'the location of',P, .']).
```

```
isis!lng9330[1] pr hunger_rules

Mar 30 14:27 1986  hunger_rules Page 1

/* This is the file hunger_rules containing the rules for satisfying
   hunger.
*/

sat_hunger(X) :-            /* X has food Z                          */
   has(X,Z),
   food(X,Z), !,
   print_hunger1(X,Z),
   print_hunger2(X,Z),
   retract(has(X,Z)),
   retract(wants(X,food)).
sat_hunger(X) :-           /* X has food Z at home                   */
   food(X,Z),
   at(Z,P),
   home(X,P), !,
   hunger_ptrans(X,P,Q),
   print_hunger3(X,Z),
   print_hunger2(X,Z),
   retract(at(Z,P)),
   retract(wants(X,food)).
sat_hunger(X) :-
   hunger_ptrans(X,P,Q),          /* X went home to look for food      */
   print_hunger4(X),              /* X had no food at home             */
   fail.
sat_hunger(X) :-
   food(X,Y),
   at(Y,P),
   not(home(_,P)),       /* Food Y is at a location that isn't anyone's home */
   at(X,Q),
   knows_loc(X,Y), !,
   print_knows_loc(X,Y),
   hunger1_ptrans(X,Q,P),/* X goes from Q to P to find food            */
   take(X,Y),
   print_hunger2(X,Y),
   retract(has(X,Y)),
   retract(wants(X,food)).
sat_hunger(X) :-
   food_trade_filter1(X,Y,Z,F),
   sat_hunger1(X,Y,Z,F).
sat_hunger(X) :-
   food_trade_filter2(X,Y,Z,F,Q),
   sat_hunger2(X,Y,Z,F,Q).
sat_hunger(X) :-
   food_trade_filter3(X,Y,Z,F,P,Q),
   sat_hunger3(X,Y,Z,F,P,Q).
sat_hunger(X) :-
   food_trade_filter4(X,Y,Z,F,P),
   sat_hunger4(X,Y,Z,F,P).
sat_hunger(X) :-
   det_poss_pronoun(X,Y),
   write_sent([X,'was not able to get any food to satisfy',Y,'hunger.']).


food_trade_filter1(X,Y,Z,F) :-
   has(X,Y),
   food(Z,Y),
```

```
   wants(Z,food),
   has(Z,F),
   food(X,F).

food_trade_filter2(X,Y,Z,F,Q) :-
   has(X,Y),
   food(Z,Y),
   wants(Z,food),
   food(X,F),
   at(F,Q),
   home(Z,Q).

food_trade_filter3(X,Y,Z,F,P,Q) :-
   food(Z,Y),
   at(Y,P),
   home(X,P),
   wants(Z,food),
   food(X,F),
   at(F,Q),
   home(Z,Q).

food_trade_filter4(X,Y,Z,F,P) :-
   food(Z,Y),
   at(Y,P),
   home(X,P),
   wants(Z,food),
   food(X,F),
   has(Z,F).

sat_hunger1(X,Y,Z,F) :-
   (knows_loc(X,Z) ; (at(Z,Q),home(Z,Q))),
   print_trade_desire(X,Y,Z), /* X wants to trade Y for food with Z   */
   at(X,P),
   desire_ptrans(X,Z,P,Q),   /* X goes where Y is or to Y's home      */
   print_trade_food(X,Y,Z),
   give(X,Z,Y),
   give(Z,X,F),
   print_hunger2(X,F),
   retract(wants(X,food)).
sat_hunger1(X,Y,Z,F) :-
   not(knows_loc(X,Z)),
   at(X,P),
   at(Z,Q),
   not(home(Z,Q)),
   print_trade_desire(X,Y,Z),
   desire_ptrans(X,Z,P,Q),
   print_desire1(Z), !,          /* Z was not home                       */
   fail.

sat_hunger2(X,Y,Z,F,Q) :-        /* X is the main_character              */
   (knows_loc(X,Z) ; at(Z,Q)),   /* Y is the thing X wants to trade      */
   at(X,P),                      /* Z is the prospective trade-partner   */
   at(Z,R),                      /* X eats F which is at Q , the home of Z */
   print_trade_desire(X,Y,Z),
   desire_ptrans(X,Z,P,R),
   print_trade_food(X,Y,Z),
```

```
   give(X,Z,Y),
   give(Z,X,F),
   print_hunger2(X,F),
   retract(wants(X,food)).
sat_hunger2(X,Y,Z,F,Q) :-
   not(knows_loc(X,Z)),
   at(X,P),
   at(Z,R),
   (Q \== R),
   print_trade_desire(X,Y,Z),
   desire_ptrans(X,Z,P,R),
   print_desire1(Z), !,
   fail.

sat_hunger3(X,Y,Z,F,P,Q) :-
   (knows_loc(X,Z) ; at(Z,Q)),
   at(X,R),
   at(Z,S),
   print_trade_desire(X,Y,Z),
   desire_ptrans(X,Z,R,S),
   print_trade_food(X,Y,Z),
   give(X,Z,Y),
   give(Z,X,F),
   print_hunger2(X,F),
   retract(wants(X,food)).
sat_hunger3(X,Y,Z,F,P,Q) :-
   not(knows_loc(X,Z)),
   at(Z,S),
   (S \== Q),
   at(X,R),
   print_trade_desire(X,Y,Z),
   desire_ptrans(X,Z,R,S),
   print_desire1(Z), !,
   fail.

sat_hunger4(X,Y,Z,F,P) :-
   (knows_loc(X,Z) ; (at(Z,Q),home(Z,Q))),
   at(X,R),
   at(Z,S),
   print_trade_desire(X,Y,Z),
   desire_ptrans(X,Z,R,S),
   print_trade_food(X,Y,Z),
   give(X,Z,Y),
   give(Z,X,F),
   print_hunger2(X,F),
   retract(wants(X,food)).
sat_hunger4(X,Y,Z,F,P) :-
   not(knows_loc(X,Z)),
   at(X,R),
   at(Z,S),
   home(Z,Q),
   (S \== Q),
   print_trade_desire(X,Y,Z),
   desire_ptrans(X,Z,R,S),
   print_desire1(Z), !,
   fail.
```

```
print_trade_desire(X,Y,Z) :-
   det_nom_pronoun(X,T),
   det_obj_pronoun(X,U),
   det_nom_pronoun(Z,V),
   write_sent([X,'thought',T,'would give the',Y,'to',Z,'if',V,'would give',U,
   'some food.']).

print_trade_food(X,Y,Z) :-
   det_nom_pronoun(X,T),
   det_obj_pronoun(X,W),
   det_obj_pronoun(Z,V),
   write_sent([X,'told',Z,'that',T,'would give',V,'the',Y,'if',Z,'would give',
   W,'some food.']).


hunger_ptrans(X,P,Q) :-
   home(X,P),
   at(X,Q),
   (P == Q), !.
hunger_ptrans(X,P,Q) :-
   home(X,P),
   at(X,Q),
   (P \== Q), !,
   ptrans(X,X,Q,P),
   write_sent([X,'went home to look for food.']).

hunger1_ptrans(X,P,Q) :-
   ptrans(X,X,P,Q),
   write_sent([X,'went to the',Q,'.']).



print_hunger1(X,Z) :-
   write_sent([X,'had the',Z,'.']).

print_hunger2(X,Z) :-
   write_sent([X,'ate the',Z,'.']),
   det_aux_verb(Z,V),
   write_sent(['there',V,'no more',Z,'.']),
   write_sent([X,'was not hungry any more.']).

print_hunger3(X,Z) :-
   write_sent([X,'had the',Z,'at home.']).

print_hunger4(X) :-
   det_nom_pronoun(X,Y),
   write_sent([Y,'had no food at home.']).
```

```
isis!lng9330[5] pr love_rules

Feb 19 16:54 1986  love_rules Page 1


/* This is the file love_rules, cntaining the rulebase for telling the
   love stories.
*/

sat_desire(X) :-
    filter_desire1(X,Y,P,Q),
    sat_desire(X,Y,P,Q).
sat_desire(X) :-
    filter_desire2(X,Y,P,Q),
    sat_desire(X,Y,P,Q).
sat_desire(X) :-
    write_sent([X,'was not successful in finding love.']).

filter_desire1(X,Y,P,Q):-
    ischar(Y),
    sex_poss(X,Y),
    not(rejected(X,Y)),  /* X not previously rejected by Y.          */
    at(X,P),
    at(Y,Q),
    (P == Q).
filter_desire2(X,Y,P,Q) :-
    ischar(Y),
    sex_poss(X,Y),
    not(rejected(X,Y)),
    at(X,P),
    at(Y,Q),
    (P \== Q).

sat_desire(X,Y,P,Q) :-
    not(likes(X,Y)), !,
    print_desire4(X,Y),  /* X could ask Y but he/she did not like Y.  */
    fail.
sat_desire(X,Y,P,Q) :-
    likes(X,Y),
    knows_loc(X,Y),
    likes(Y,X),
    wants(Y,sex), !,
    print_desire2(X,Y),  /* X liked Y.                                */
    desire_ptrans(X,Y,P,Q),/* X went where Y is or to the home of Y.  */
    print_desire3(X,Y),  /* X asked Y for love.                       */
    print_desire5(X,Y),  /* Y agreed because Y liked X and was in mood*/
    retract(wants(Y,sex)),
    retract(wants(X,sex)).
sat_desire(X,Y,P,Q) :-
    likes(X,Y),
    knows_loc(X,Y),
    likes(Y,X),
    not(wants(Y,sex)), !,
    print_desire2(X,Y),  /* X liked Y.                                */
    desire_ptrans(X,Y,P,Q),/*X went where Y was or to the home of Y.  */
    print_desire3(X,Y),  /* X asked Y for love.                       */
    print_desire6(X,Y),  /* Rejection because Y not in mood.          */
    assert(rejected(X,Y)),/* Record X's rejection by Y.               */
    fail.
sat_desire(X,Y,P,Q) :-
    likes(X,Y),
```

```
    knows_loc(X,Y),
    not(likes(Y,X)), !,
    print_desire2(X,Y),    /* X liked Y.                                    */
    desire_ptrans(X,Y,P,Q),/* Takes care of X's physical transportation*/
    print_desire3(X,Y),    /* X asked Y for love.                           */
    print_desire7(X,Y),    /* Y refused as he/she did not like X.           */
    assert(rejected(X,Y)), /* X rejected by Y.                              */
    fail.
sat_desire(X,Y,P,Q) :-
    likes(X,Y),
    not(knows_loc(X,Y)),   home(Y,R),
    (R == Q),
    likes(Y,X),
    wants(Y,sex), !,
    print_desire2(X,Y),  /* X liked Y.                                      */

desire_ptrans(X,Y,P,Q),
   print_desire2(Y),    /* Y was at home.                                   */
   print_desire3(X,Y),  /* X asked Y for love.                              */
   print_desire5(X,Y),  /* Y agreed as he/she is in mood and likes X.       */
   retract(wants(Y,sex)),
   retract(wants(X,sex)).
sat_desire(X,Y,P,Q) :-
   likes(X,Y),
   not(knows_loc(X,Y)),
   home(Y,R),
   (R == Q),
   likes(Y,X),
   not(wants(Y,sex)), !,
   print_desire2(X,Y), /* X liked Y.                                        */
   desire_ptrans(X,Y,P,Q),
   print_desire2(Y),    /* Y was at home.                                   */
   print_desire3(X,Y), /* X asked Y for love.                              */
   print_desire6(X,Y), /* Y liked X but was not in mood.                   */
   assert(rejected(X,Y)),
   fail.
sat_desire(X,Y,P,Q) :-
   likes(X,Y),
   not(knows_loc(X,Y)),
   home(Y,R),
   (R == Q),
   not(likes(Y,X)), !,
   print_desire2(X,Y), /* X liked Y.                                        */
   desire_ptrans(X,Y,P,Q),
   print_desire2(Y),    /* Y was home.                                      */
   print_desire3(X,Y), /* X asked Y for love.                              */
   print_desire7(X,Y), /* rejection because Y did not like X.              */
   assert(rejected(X,Y)),
   fail.
sat_desire(X,Y,P,Q) :-
   likes(X,Y),
   not(knows_loc(X,Y)),
   home(Y,R),
   (R \== Q), !,
   print_desire2(X,Y), /* X liked Y.                                        */
   desire_ptrans(X,Y,P,Q),
```

```
   print_desire1(Y),    /* Y was not home.                                  */
   fail.


print_desire1(Y) :-
   det_nom_pronoun(Y,Z),
   write_sent([Z,'was not home.']).

print_desire2(Y) :-
   det_nom_pronoun(Y,Z),
   write_sent([Z,'was at home. ]).

print_desire1(X,Y) :-
   det_nom_pronoun(X,Z),
   write_sent([Z,'went to the home of',Y,'.']).

print_desire2(X,Y) :-
   write_sent([X,'liked',Y,'.']).

print_desire3(X,Y) :-
   det_nom_pronoun(Y,Z),
   write_sent([X,'asked',Y, if',Z,'would like to make love.']).

print_desire4(X,Y) :-
   det_nom_pronoun(X,Z),
   det_obj_pronoun(Y,T),
   write_sent([X,'could ask',Y,'but',Z,'did not like',T,'.']).

print_desire5(X,Y) :-
  det_nom_pronoun(Y,Z),
  write_sent([Y,'agreed because',Z, liked',X,'and was in mood.']).

print_desire6(X,Y) :-
  det_nom_pronoun(Y,Z),
  write_sent([Y,'liked',X,'but',Z,'was not in mood.']).
  write_sent(['Therefore',Z,'refused.']).

print_desire7(X,Y) :-
   det_nom_pronoun(Y,Z),
   write_sent([Y,'refused because',Z,'did not like',X, .']).

print_desire1(X,Y,P,Q) :-
   det_nom_pronoun(X,Z),
   write_sent([X,'knew',Y,'was at the',Q,'so',Z,'went there.']).
```