August 1981

Invoking a Beginner's Aid Processor by Recognizing JCL Goals

Jeffrey C. Shrager

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-81-7.

Invoking a Beginner's Aid Processor by Recognizing JCL Goals

Abstract

Typical help processors are invoked explicitly by the user or implicitly when an error occurs. Often a beginner will not know that he needs help because the inefficient use of commands will get the job done without raising errors. WIZARD is an expert system that recognizes beginner misbehaviors and can automatically start a help transaction.

The WIZARD processor relies on a special purpose, dynamic, pattern matcher directed by a KL–One based knowledge network. An author studies logs of beginner interaction and develops sequence rules which parse and properly identify misbehaviors. Objects that drive the parser to understand VAX DCL commands are coded into the network and a set of semantic programming utilities is used to perform actual goal recognition.

This thesis deals primarily with the implementation of such a goal recognizing expert invocation system. It is the WIZARD documentation and final working report. I discuss the motivations for the design of the system and detail the knowledge base and heuristics that support goal recognition. Some issues of generality are taken up and potential topics for later research are presented which will extend WIZARD'S capabilities.

Comments

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-81-7.

MS-CIS-81-7

UNIVERSITY OF PENNSYLVANIA
THE MOORE SCHOOL OF ELECTRICAL ENGINEERING
SCHOOL OF ENGINEERING AND APPLIED SCIENCE

INVOKING A BEGINNER'S AID PROCESSOR BY RECOGNIZING JCL GOALS

Jeffrey C. Shrager

Philadelphia, Pennsylvania

August 1981

A thesis presented to the Faculty of Engineering and Applied Science in partial fulfillment of the requirements for the degree of Master of Science in Engineering for graduate work in Computer and Information Science.

Timothy W. Finin

Aravind K. Joshi

The work reported here was supported in part by NSF grant number MCS 79-08401.

"If you are trying to get around the snake...
I can give you a hint but it will cost you ten points."

-- Collosal Cave

Abstract

Typical help processors are invoked explicitly by the user or implicitly when an error occurs. Often a beginner will not know that he needs help because the inefficient use of commands will get the job done without raising errors. WIZARD is an expert system that recognizes beginner misbehaviors and can automatically start a help transaction.

The WIZARD processor relies on a special purpose, dynamic, pattern matcher directed by a KL-One based knowledge network. An author studies logs of beginner interactions and develops sequence rules which parse and properly identify misbehaviors. Objects that drive the parser to understand VAX DCL commands are coded into the network and a set of semantic programming utilities is used to perform actual goal recognition.

This thesis deals primarily with the implementation of such a goal recognizing expert invocation system. It is the WIZARD documentation and final working report. I discuss the motivations for the design of the system and detail the knowledge base and heuristics that support goal recognition. Some issues of generality are taken up and potential topics for later research are presented which will extend WIZARD's capabilities.

Acknowledgements

Most importantly, thanks to my supervisor, Tim Finin, for putting up with my various crazy ideas and providing invaluable input into all of them, including this one. I would like to also thank to Ira Winston for succeeding where DEC failed in making VMS a reasonable place to live and Rick Bowen for the figure graphics in this thesis.

Special thanks to my parents without whom I would not have been X (for any X), to Adele Howe who I think cares about me more than I care about myself, to Steve Bagley for grins, and to Lori for Lori.

Preface

This work borders on several areas of software engineering sometimes associated with the cognitive modeling aspect of artificial intelligence. The terminology used to describe portions of the system includes phrases like "knowledge representation" and "understanding". It is important that the reader be aware that I have no intention of trying to relate the data structures or algorithms used by WIZARD to any supposed actual processes in the mind of a human being. My use of AI terminology is purely a historical one. I have derived many of these techniques from research which does claim some sort of cognitive reality and the terminology was carried along for the sake of not renaming the wheel. As far as I am concerned, this work represents research in software engineering and technique, not cognitive simulation.

1.0 Chapter 1: Recognizing a User Assistance Loophole.

No one will argue the utility of a user assistance processor. Such systems are especially useful during initial exploration in a new interactive environment. In this chapter I motivate this work by showing that there is a species of problem not covered by typical user aid paradigms. My solution, WIZARD, is introduced and an outline of its processing is given.

1.1 Common Users Assistance Paradigms.

Unfortunately, most of the help programs currently available are annoyingly anti-social. They are of use only if the user explicitly calls on them or if an error occurs which the system knows how to deal with.

Here are a few exam les of the types of interactions that take place with such aids:

```
$HELP LOGOUT
%The logout command causes...
```

[The "$" will consistently indicate user input and the "%" will show the system's response.]

In the above example, an explicit invocation of the user assistance processor, two assumptions are made that are relevant to this discussion: it is assumed that the user knows how to ask for help and it is assumed that he knows exactly which question to ask.

Following is an example of an error-invoked help transaction:

```
%Error SUCH-AND-SUCH occurred.
%Do you need help?
```

In this case, the user needn't know how to ask for help. An error demanded attention implicitly. Assuming that the help program is somewhat clever, the user needn't know exactly how to work with it. Of course, this interaction is predicated on the user having caused an error.

Various improvements can be made to the above paradigms but their assumptions remain an obstacle to complete user assistance. Following are several of the more common enhanced functions based upon the above behaviors:

The "do what I meant" game:

```
$LOGOUT
%LOGOUT is not a legal command,
%perhaps you meant to type BYE?
```

The combination of the major themes:

```
$INFO LOGOUT
%INFO is not a legal command,
%perhaps you meant to type HELP?$yes
%LOGOUT is not known to the HELP system,
%perhaps you meant to type BYE?$yes
%The BYE command causes...
```

Holding the user's hand:

```
%Welcome to VAX/VMS at The Moore School
%Type HELP if you need it.
```

1.2 The Presumptions of Error or Missed Knowledge.

Unfortunately, all of the above interactions depend upon the user's awareness that he is in need of assistance or the system's ability to recognize mistakes (which most systems do) and respond to them in a helpful manner (which most systems do not). Consider the following example:

Suppose that a beginner wished to change the name of a file in the new system. Knowing about the COPY and DELETE commands he might think to change filename A to B via:

```
$COPY A B
$DELETE A
```

An expert user observing this behavior would probably correct him indicating that he could have simply typed:

```
$RENAME A B
```

to accomplish the same result.

Without benefit of a consultant the user is burdened with a great deal of work in order to learn about such shorthand incantations. He must:

- Recognize the desired function as an unique entity, (Changing the name of a file.)
- Guess that the system designers have provided a way of doing this without having to COPY the file and DELETE it, and,
- Guess how to ask for help about this function.

Even granting the first and second of these obstacles are surmountable, and assuming that he knew how to invoke the help processor, what would he have asked about? In this particular case asking for information on RENAME would have done the trick but there is at least one system where the command used to perform this exact operation is CATALOG -- not as likely a guess. [Some of us are painfully familiar with "PIP B=A/RE".] He might have asked about "changing the name of a file" and a sufficiently intelligent processor might have figured out what was meant. Such cleverness is rare. A simpler way out, assuming again that the beginner thought to ask at all, would be to ask for a list of all help and then hunt around for the RENAME command. This is a clear waste of time.

More importantly, the behavior was perfectly valid and did accomplish the name change. As far as the user is concerned it is perfectly reasonable to go on indefinitely without the RENAME command. No errors occurred which might have triggered a help interaction and there was no reason for the system to think twice about the validity of this COPY+DELETE sequence.

Thus, in this case, neither the user initiated help processor or the error initiated help processor would have been any use at all. It often requires an expert to catch this error of omission. Such persons have trained themselves by word-of-mouth or some other means. These persons can often be found in the guise of a user consultant or highly experienced user. [The pen-name for such an individual is a

Even granting the first and second of these obstacles are surmountable, and assuming that he knew how to invoke the help processor, what would he have asked about? In this particular case asking for information on RENAME would have done the trick but there is at least one system where the command used to perform this exact operation is CATALOG -- not as likely a guess. [Some of us are painfully familiar with "PIP B=A/RE".] He might have asked about "changing the name of a file" and a sufficiently intelligent processor might have figured out what was meant. Such cleverness is rare. A simpler way out, assuming again that the beginner thought to ask at all, would be to ask for a list of all help and then hunt around for the RENAME command. This is a clear waste of time.

More importantly, the behavior was perfectly valid and did accomplish the name change. As far as the user is concerned it is perfectly reasonable to go on indefinitely without the RENAME command. No errors occurred which might have triggered a help interaction and there was no reason for the system to think twice about the validity of this COPY+DELETE sequence.

Thus, in this case, neither the user initiated help processor or the error initiated help processor would have been any use at all. It often requires an expert to catch this error of omission. Such persons have trained themselves by word-of-mouth or some other means. These persons can often be found in the guise of a user consultant or highly experienced user. [The pen-name for such an individual is a

"wizard" thus I have named the system WIZARD and shall refer to it by that name from here on.]

1.3 The Third Paradigm: WIZARD's Informal Introduction.

One can imagine a help processor that would "understand" commands that the user enters and would "recognize" the goal that they are most likely meant to implement. Assuming that the system is clever enough to see that the COPY+DELETE sequence is meant to be a RENAME, it would not be very difficult to have it tell the user about the existence of RENAME or to invoke a separate help processor for this purpose.

The WIZARD interactions might be:

```
$COPY A B
$DELETE A
%Assuming that you wanted to rename the file A to call
%it B you might have simply said: $RENAME A B.  You can
%ask for HELP on the RENAME command by typing $HELP RENAME.
```

1.4 Some Terminology.

A "sequence" is any list of commands to the operating system. [The particular domain of WIZARD is the VAX DCL command language.] I shall refer to the user's long-winded command sequence as a "misbehavior". Each sequence is said to have a "goal" which is the effect that the user wished to achieve through his application of the sequence. In the above example the goal was something like "change the name of the file A to B".

WIZARD is said to "recognize" the goal of a sequence. That is, given a sequence it can decide from the universe of known goals which one(s) were likely intended by the user. The potential misbehaviors, goals, recommended sequences and the relationships between them are meant to be predefined by the human consultant whose job it is to control what WIZARD will recognize and what advice will be distributed.

1.5 Goal Recognition Heuristics.

There are various methods that have been used to perform recognition of the intention from input sequences. Most are driven by pattern matchers of one sort or another. It is clear that some sort of parser is required in order to take the first step of understanding the individual commands. It is the job of that process to transform individual input strings into some internal representation that can be used to drive the goal r cognition process. This will be detailed in chapter 2.

Less clear is the processing that performs recognition over the entire user input. This is the algorithm (or heuristic) that will determine when a help transaction should be invoked. The three approaches that were considered in the process of WIZARD's design were environmental change observation, syntactic analysis, and anticipation.

1.5.1 Environmental Change Observation.

An interesting approach but one which causes several problems is to derive the goal by comparing the environment before the operation with the environment afterward. Thus, the goal "change the name of a file" can be discovered by seeing that a filename has been changed. This is not actually so simple. First, it may be necessary to compare the entire environment in order to derive the goal. In order to distinguish between having renamed an existing file and having simply deleted one file and created an entirely different one it is necessary to compare at least the contents of the new file with the old. Another difficulty in this approach is that it is not simple to define the bounds of the recognition. We must assume that recognition is occurring all the time (every possible change is being recognized) and that something outside of this process causes the recognizer to actually call upon the help system. It is still necessary to have in hand the command images in order that the recognizer not warn the user about RENAME after his just having used a RENAME.

Another more difficult problem with this approach is that it forces the designer to develope a theory of significance in order to determine which of several simultaneous changes to focus on. For example, the COPY+DELETE sequence also updated the creation dates, and took several seconds of CPU time. Suppose that help were available for "how to waste CPU time". Would one prefer to invoke that aid or the help for RENAME?

Environmental information alone is not sufficient and is very difficult to obtain at times (e.g., the contents of a DELETED file). Thus, this approach was not considered for very long. It is probably better applied to text editor environments where the language is simple and the environment is readily at hand.

1.5.2 The Syntactic Approach.

In many ways, this problem is like that confronting a natural language understanding system. WIZARD might seek a match for a "goal pattern" in much the same way that the rule based execution of a transformational grammar seeks the base form of a sentence. The control mechanism may also be the same as a natural language understanding system; ATN driven parsers can be successfully used in this work.

A problem with this approach is multiple sequences may be intertwined. Consider the following example:

```
$COPY A B
$COPY C D
$DELETE A
$DELETE C
```

We would probably want the help (for at least the first of the pair) to be presented even though there is a "noise" command (COPY C D) in the way. The extra commands could be ignored as noise but then the second nested sequence would not be recognized. Suppose that we wanted to warn the user each time he makes this mistake. Then the

second sequence (COPY C, DELETE C) need be analyzed as well as the first. If noise were simply ignored we would lose this second recognition.

1.5.3 Goal Recognition by Anticipation.

It might be possible to modify some standard parsing algorithm to handle these problems but a more general approach is suggested by the above: It seems that two separate and non-communicating processes are taking place in the intertwined recognition (assuming that we would like the redundant warnings). Why not simply start the understanding processes independently of one another. This approach resolves the noise and intertwining difficulties simultaneously. The implementation of such independent processes in a parser environment may be accomplished by anticipation: Entered commands that are earlier parts of sequences cause the parser to be modified such that later commands are understood as the following portions of the sequences under consideration.

To be a bit more detailed: Each time a command is entered it is intercepted and processed by WIZARD. The first step in this processing is to force the command to become an individual of some generic DCL-command and parse it accordingly. The parser searches a semantic net description of the DCL language trying to match, at each DCL-command node, the current command with that generic. If it matches (i.e., the parse succeeds from that node) then the current

command is instantiated as an individual of that generic DCL-command. This is the process that I refer to as "understanding".

The instantiation of an individual causes some set of prescribed actions to take place. These actions change the structure of the network which controls the parser. The effect is that of laying traps for the latter portions of the sequence that is to be recognized. The last action to be invoked in this recognition process is the successful recognition of the goal. That action might include the construction of help text using data that has been passed along in the semantics of the parsing objects. It is very important to notice that this method is driven entirely from the syntax of the incoming commands. No external information is used in the recognition task. We will see that this is a problem for WIZARD in this particular domain.

This method of recognition is not unlike Riesbeck's natural language understanding system [10 and 11] which uses expectation schema to direct the parsing process although the domain and data structure are a great deal different. For Riesbeck; "The mechanism for passing information from one point in the analysis to the other is the expectation. An expectation consists of a specification of a situation and a specification of what to do if that situation is encountered". This matches my thinking exactly. In fact, I strongly recommend references [10] and [11] to the reader interested in the anticipatory recognition approach. Many of the arguments put forth

here are very like Riesbeck's.

1.6 COPY+DELETE Detailed.

Applying this approach to the example above, one can think of the command "COPY A B" as setting a trap which reads "If the command 'DELETE A' is entered, tell the user that a RENAME command might have been more appropriate". Thus, the action attached to "COPY file1 file2" would be "Set a trap that is a DELETE command for file1 with the proper associated actions".

We will see that this is not quite so simple. There are many problems to be overcome in implementation of a working WIZARD. Both the constraints of anticipation and the problems of the domain will mar the apparent simplicity of this approach.

1.7 Forward.

WIZARD is a special purpose program. Its parser includes knowledge specific to the VAX DCL command language. However, the design is such that the parser can be easily changed without tearing apart WIZARD's internals. This constitutes both effective programming practice in general and specifically supports the extensibility required by the anticipation heuristic.

All of the examples that I will mention are in DCL command language and some understanding of a small subset of that language might be necessary to properly interpret some of those examples. Reference [4] may be used as a guide to DCL.

In the following chapters I will lay out the detailed representations and functions required to support goal recognition and the algorithms that process the knowledge base. I explain the motivation for each decision. The purpose of this work was to experiment with goal recognition in this particular domain and to decide what primitive actions and data structures can be used to implement help system invocation as I have described it.

Details of the actual implementation of WIZARD make up the bulk of this work. There are problems resulting from the chosen environment and algorithms, that severly limit WIZARD's utility. I discuss these problems and talk about possible future work including some correction of problems and filling some of the unresolved (or unprogrammed) holes in the system.

A reading note: The language in which WIZARD is implemented, Franz LISP, retains case information and thus all the WIZARD code itself is written in lower case characters. In the interest of clarity I have used upper case characters in this thesis to distinguish special names (such as the names of DCL commands). This becomes troublesome only if the reader is trying to follow along some

of this text with the appropriate appendecies.

2.0 Chapter 2: Understanding DCL commands.

Here I describe the gory details of the WIZARD parser. It can be described as a "dynamic, object driven, knowledge based, command string parser". Hopefully, after having been through this thesis, the reader will understand what is meant by that.

The task of WIZARD's goal recognition algorithm can be divided into two gross parts: First, the system must input and "understand" DCL commands. This chapter deals with the method of processing those commands. Later chapters will deal with the second part: recognition of goals from sequences of commands.

2.1 What Understanding Means.

I use the terms understanding and recognition in somewhat the opposite sense that one might think normal. Understanding applies to the parsing of strings whereas recognition applies to overall goals. This is due to the unfortunate name, "goal recognition", chosen by my predecessors in the field, to describe the latter process. I have tried to keep the terms separate.

The parser of a compiler can be said to accept correct programs in its language. In WIZARD command understanding is more than simply parsing the input commands. An author has written into WIZARD information about the DCL command formats as generic objects in the knowledge base. WIZARD understands an input string as a particular

DCL command, according to what it knows about DCL commands, and names this string according to the names associated with the objects in the database. There is a "meaning" associated with each command in the form of a LISP expression called the "semantics". The semantics are involved in the second half of the process. This will be discussed in detail later.

For example, The command "COPY A B" matches the form that WIZARD associates with a "Copy-command". Thus, this particular string is understood as an instance of "Copy-command" and is parsed according to the pattern that is associated with that object. An instance is a duplicate of the generic with specific string parts inserted at its leaves. It is named and included in the semantic net then the semantics associated with the Copy-command generic object are invoked.

2.2 How a String is Parsed: The Naming Algorithm.

Understanding actually takes place in a somewhat upside down way: the name associated with an input is based upon where the string fits properly into the net. In order to understand how parsing takes place one must understand the data structure of the knowledge net from which WIZARD's control derives.

2.2.1 Objects and Their Parsing Roles.

Each name that might be used to describe a string exists in a database which I shall call the "semantic network" or "knowledge net". This is a type of representation that relies upon uniquely named objects and descriptions of the relationships between them. The network is hierarchically organized. That is, there are "super objects" which have "sub objects" that are special cases of the former. These special cases are called "specializers".

This representation is not novel. The terminology and basic structure that I use were suggested by, but are not identical to those described by, Brachman in [2] and [3].

Figure 1 shows a portion of the WIZARD semantic net. Each object is represented by an ellipse. The boxes are "roles" of an object. I will discuss their use momentarily. Arrows that connect objects are called "is a" links. In figure 1 "File-deletion-command" is a "DCL-command". Arrows that connect lower level roles up to higher level roles are called "role filling links". Arrows that connect objects to roles (from the ellipse to a box) are the "role specifications". Arrows that link roles to objects or specifications (strings, numbers, etc) are called value restrictions (V/R). Again in figure 1, the roles of DCL-command are "Command-name" and "Command-argument". The value of command-argument is restricted to being a "List-of-filenames".

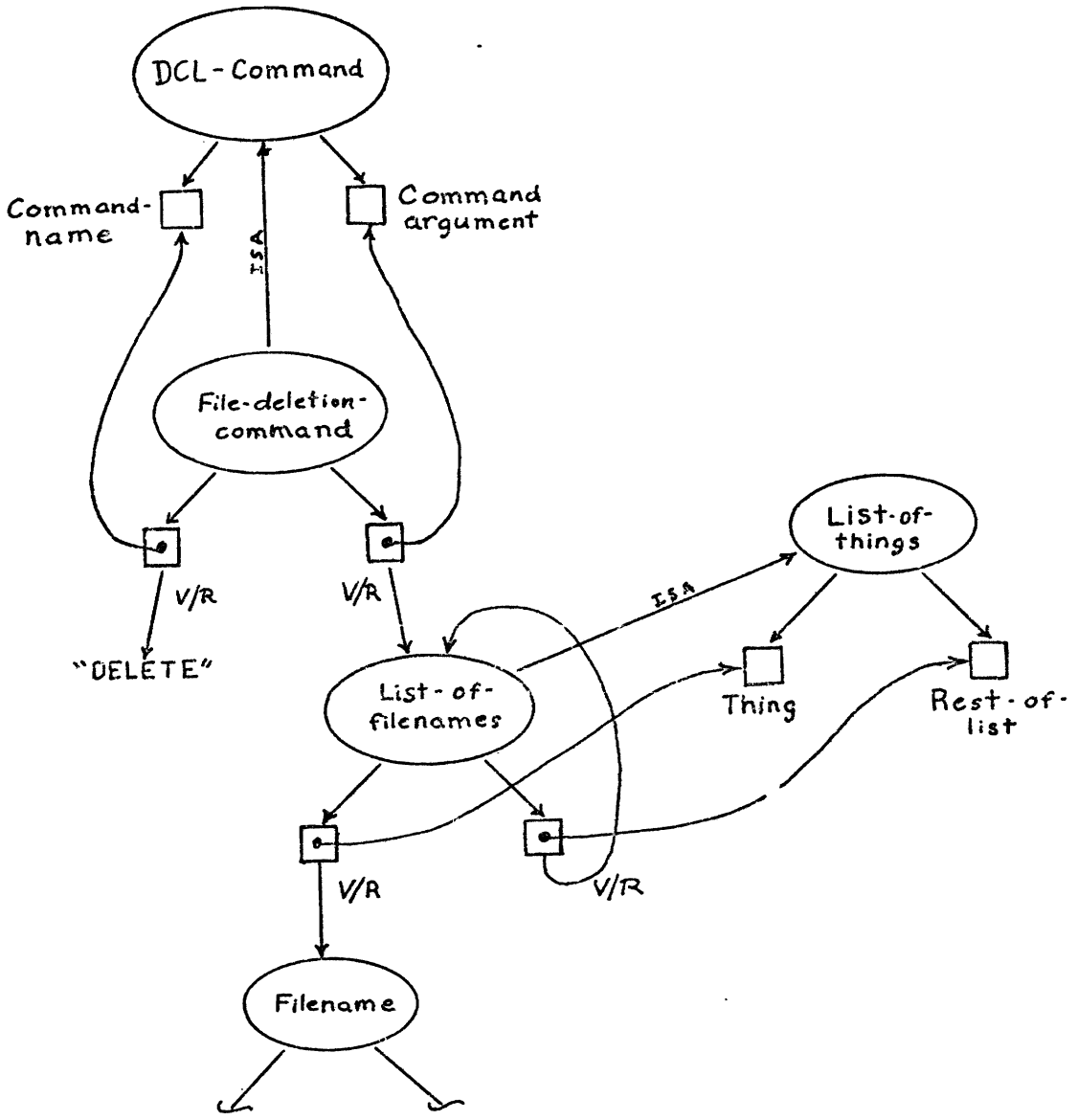

Figure 1: Part of the Parser's Knowledge of DELETE-COMMAND

The main object in the WIZARD network is the "DCL-command". The roles of a super object (one that has no further superior) are expected to be lambda expressions that cause parts of the input string to be selected and bound to those roles. For example:

```
Input: "DELETE PHOO,BEAR"
Object: DCL-command
   Command-name selects: "DELETE"
   Command-argument selects: "PHOO,BEAR" (the rest)
```

"File-deletion-command" is a special case of DCL-command. Its roles (as must the roles of any specializer) indicate constraints upon the form of the bindings selected by its superior. Thus, continuing the above example and remaining in figure 1:

```
DCL-command has parsed:
   Command-name: "DELETE"
   Command-argument: "PHOO,BEAR"

File-deletion-command requires
   Command-name="DELETE" (it does)
   Command-argument=a List-of-filenames which must match
                    "PHOO,BEAR"
```

Now we must proceed back up to the super object "List-of-things" in order to process the argument of the command and try to fit it into the slot whose value is constrained to be a List-of-things where each thing is to be a Filename.

2.2.2 Recursive Objects -- How Parsing Terminates.

The object "List-of-things" in figure 1, is a "recursive object". That is, the value/restriction of one of its roles is an object of the same type as List-of-things. This could cause the parser to go into an infinite loop trying to push down into the string with this recursive specification. I have abritrarily specified two cases in which parsing will stop: the roles have all been successfully filled (or some role cannot be filled in which case the parse fails), or, second, the string runs out. In the latter case, the parse succeeds and all remaining roles are filled by NIL.

It is not possible to eradicate recursive objects from the DCL-command language without severely limiting its conciseness. DCL uses comma-delimited lists frequently. We shall see in the discussion of semantics that these List-of-things objects play a major role in the actions that I have selected.

2.2.3 Network Search.

The function that drives the parser expects to see only the roles of the object under consideration. The result of the parse is a single list in which each role has been paired with the portion of the input string that was matched by that role. A higher level function passes the parsing roles to the parsing function and then processes its result. The order that objects are passed for matching by the

role parser is depth first in the net. That is, before the string is tried against a particular superior object, all of its specializers are attempted.

All possible successful parsings of the string are returned but their order is as discussed just above. Thus, for example, the command:

```
$DELETE PHOO,BEAR
```

would first be parsed as:

```
Delete-command with roles
        Command-name: "DELETE"
        Command-argument: a List-of-things
                Thing: "PHOO"
                Rest-of-list: a List-of-things
                        Thing: "BEAR"
                        Rest-of-list: NIL
```

and then also as:

```
DCL-command with roles
        Command-name: "DELETE"
        Command-argument: "PHOO,BEAR"
```

I will discuss the reason that all possible parses are returned later. Briefly, one means of goal recognition might include scanning the history of entered commands. One would want to be able to obtain the most specific interpretation (which would be the CAR of that element of the history list) at first glance.

2.2.4 Instantiation of Objects: The Name of a Command.

Each successful parse causes an instance of the generic object, whose roles were used to drive the parser, to appear. That is, a new object is created that represents the particular instance of the specializer that matches this string. Instances are named by appending a unique five-digit identifier to the name of the object which successfully matched the string.

An instance of each of the value restriction objects for this parse is also created. Figure 2 shows the way that the above delete command would be instantiated. [Instance objects are doubly lined.] The application of instantiation to sub objects also occurs depth first. Thus, the subordinate value/restrictions of an instance are created before the object itself. Also, each possible parsing of a command is instantiated individually. Thus, we are left with both File-deletion-command-00004 and some instance of DCL-command from the preceeding example. These hold implications that will be discussed in the section on semantics.

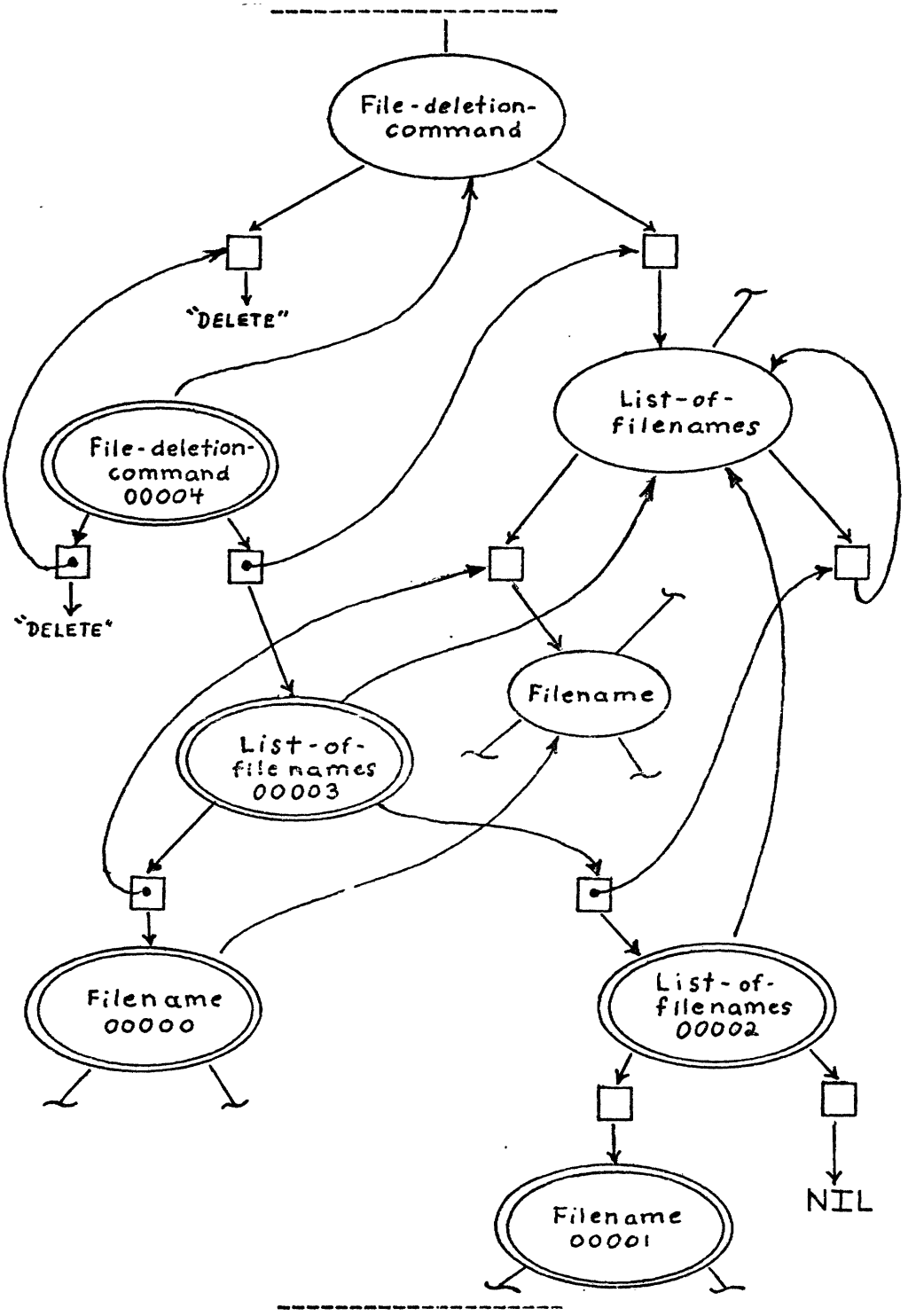

Figure 2: An Instantiation of "DELETE PHOO,BEAR"

2.2.5 Specializers vs Instances: Access Between Objects.

Every network object is one of:

Top-Level generic: An object with no more general one above it. In figure 1, DCL-command and List-of-things are Top Level objects. A Top Level object's IS-A link is NIL.

Specializer: An object that is a special case of a higher level (possibly Top-Level) object.

Template: An object that is not really in the network but which looks like a specializer and is used to create new specializers by copying.

Instance: A very specific object that represents some actually entered command. Understanding of a command causes an instance to be added to the network.

A freshly made instance has no specializers, templates, or instances below it but its name is added to the "instances" list of the object whose roles parsed this string. The depth first parsing routine looks only at the specializers in order to process an object. Thus, instances do not enter into the parsing search. In fact, the form of the instance object is slightly different than that of a specializer and passing an instance to the parser will result in unpredictable behavior. Every network object contains within it a pointer to its superior so that access both upward (always one to one) and downward (potentially one to many) are achieved. A top level object has no superiors in the network (the pointer is NIL).

2.3 What to do with Instances After Understanding.

WIZARD's general processing scheme can be summarized as follows:

1) Read a command.
2) Try to understand the command as some instance of an object or objects in the network.
3) Instantiate the objects that represent the command.
   At each instantiation evaluate the semantics of the specializer to which each instance is attached.
4) Add the name(s) of the instance(s) to the command history.
5) Goto (1)

In addition to the parsing roles, each object has a set of "semantic actions". These are in the form of an s-expression that the author has attached to a specializer. The process of instantiation causes the actions associated with the superior to which this instance was bound to be evaluated. It is this evaluation that drives the goal recognition process. The actions and support for goal recognition are discussed in chapters 4 and 5.

2.4 Problems with the Parsing Algorithm.

As mentioned briefly above, the parser is very special purpose and suffers from difficulties that might make it unusable in other domains. These problems stem from the simplicity of the parser relative to the DCL language and from a lack of generality of communication between DCL and LISP and within DCL itself.

2.4.1 Case Matching and Abbreviation.

There are two simple difficulties in the current parser. These are best described as unimplemented sections and code can be easily added to correct them. First, the internal code of WIZARD expects the input to be in lower case. Thus, if the user enters commands shifted they will not be properly understood. A case translation algorithm can be added to the input control in order to account for this difficulty. All input would be down shifted to match the internal form.

The second problem is slightly more trouble but not outstanding; Some DCL commands can be abbreviated. That is, all of DEL, DELETE, DELE, etc are valid forms of the DELETE command. The simplest solution to this difficulty is to cause the matching expression of the parsing object to accept all the possible abbreviations.

2.4.2 One Position Look ahead Parsing.

A set of parsing utility functions is provided for use in the top-level parsing expressions. These functions permit the expression to read forward one character in the input buffer, look at the next character in the buffer, skip spaces in the input buffer, drop a character into the result string, copy characters to the result until a certain character is seen, or return the remainder of the input string as the result.

These functions provide a simple parsing capability which is inadequate to process some parts of the DCL command set. I have handled some of these as special cases by making use of the possible predicate expression in the top-level parsing object representation. For example: in order to distinguish the form "device:filename" from "filename" without causing filename to be mistaken as a device name one can define a special function which will look for the ":" in the input string. This string is available to the parsing expressions as an exploded list.

2.4.3 Successful recognition of failed commands.

One of the major philosophical premises upon which WIZARD is based is that the goal recognition is predicated only upon successful and correctly formed DCL commands. In the ideal case, the sub process to which WIZARD passes the commands for actual invocation would return an error code and only those commands which generated no errors would be considered (parsed etc). Unfortunately, VMS and LISP do not communicate well with one another and such error codes are not immediately available.

If WIZARD were to try to understand syntactically or semantically incorrect commands the parser, which is not a syntax analyzer, would try to find a legal place to put these strings even though they do not necessarily make any sense. The results of the application of the goal recognition processing to such malformed instances is probably

unpredictable. Any help generated from such illegal commands used in goal recognition would certainly be wrong or misleading.

2.4.4 Partial Failure of List Operations.

An additional problem arises from the method of operation of DCL commands when lists are involved. Under some circumstances the command processor will process list arguments even though some of the members might cause an error. For example: if the files A and C exist, but not B, the command:

```
$DELETE A,B,C
```

will work for the two existing files but cause an error for the deletion of B. Do we or do we not want to accept that command for processing in WIZARD? It is not trivial to correct or detect the potential of such an error without duplicating most of the DCL error logic. This also leads to possible misunderstandings between WIZARD and DCL.

2.4.5 Misunderstanding Commands.

It is possible that a command will be misunderstood as a simpler form. That is, for example, suppose that a command to delete a single file were in the network. It would probably understand commands like "DELETE WINNING.PHOO" perfectly well but would tend to misunderstand (that is, match when it should not have) the command "DELETE WINNING.PHOO,BEAR" as deleting the single file whose name is

"WINNING" and whose extention is "PHOO,BEAR".

There are two possible means of avoiding this problem. The easiest is to be sure that there are no such simple objects in the network. Sometimes objects like that were meant to be later steps in a recognition sequence and should have been templates rather than specializers. The other approach is simply to be quite careful about what objects are in the network and what they will match. The above example can be avoided by making the expression that parses filenames clever enough to look for illegal characters in the extent field and fail if they occur. This takes more time in coding and execution but is certainly the more general, and recommended, solution.

3.0 Chapter 3: Details of the Object Representation.

In this chapter I motivate and detail the subparts of network objects. Some of these parts have been mentioned in chapters 1 and 2. The motivation for other parts of the objects will not become clear until semantic actions are discussed. I place this chapter here because it is too detailed to go very early in the thesis but the terminology it explains is necessary in order to understand the action of the goal recognizer.

As indicated in the preface, the objects that WIZARD knows of are not meant to directly represent concepts in the actual knowledge of the creator of the network. Therefore, I have chosen to include in the representation of objects anything that was necessary in order to support the parser, goal recognition, etc. It turns out that not all that much information is required.

Objects have both value based data (that is, parts of the object's list representation) and property based information. In general value based information is static once the object is created but property based data changes as a result of external network operations. These are explained separately since they serve different purposes. In reading this chapter, it might be useful to remove appendix B, the primary DCL network code, and keep it nearby.

3.1 Value Based Contents of Objects.

Every object in the WIZARD knowledge base contains exactly the same parts although some of them are unused in special cases. For example, instances of a specializer do not use the mask field and this field is NIL in those objects. Links (arrows in the figures) are made by including the name of the object to which the link connects.

Value based parts of objects typically are copied in case of instantiation or creation of a new specializer and are not changed thereafter. The recursive functions in WIZARD's internal code deal with construction and processing of these value based parts directly.

Figure 3 represents the general object schema. Solid lines show the value based parts of the object (parsing roles, mask, semantics, etc). The dashed lines indicate information kept in properties associated with the object (instances and specializers). Figure 4 shows the lisp representation of the general object. Note the substructure of the various fields. The rest of this chapter discusses these fields in detail.

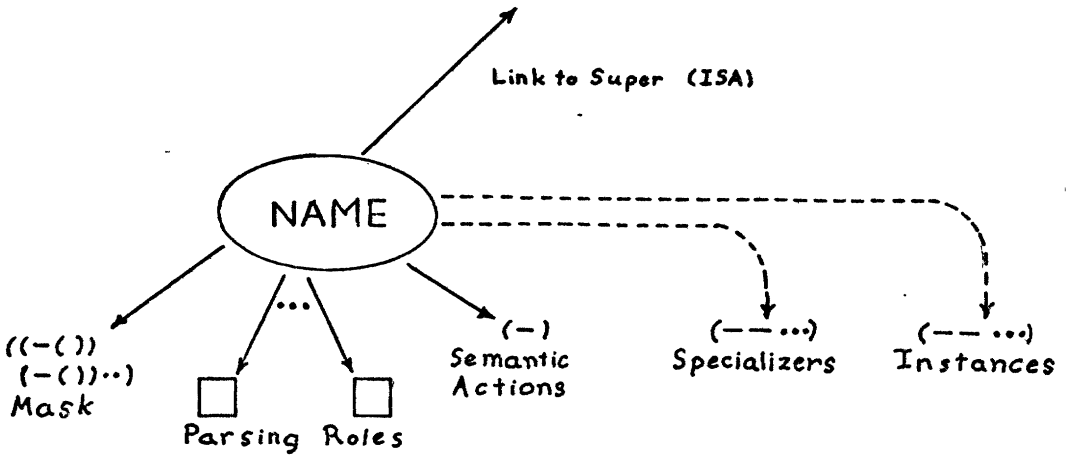

Figure 3: A diagram of the General Object.

```
(object-name
   ; "type" specification field
   (IS)                                [for a top level object]
   (SPECIALIZES super-obj)             [for an original specializer]
   (SPECIALIZES super-obj origin)      [for a dynamic specializer]
   (INSTANTIATES supe -object)         [for an instance]
   (TEMPLATE super-ob ect)             [for a template]
   ; "parsing roles" ield (name Value/Restriction)
   ((name-of-role
           (parsing function)          [for a top level]
           ()                          [indicates "must be empty"]
           (predicate)                 [to match string directly]
           atom                        [to match exactly]
           (A sub-object)              [point to a lower object]
     )
    (...) ...)
   ; Semantics
   (expression to be evaluated)
   ; Name/path mask
   ( (name (path specification)) ...)
)
```

Figure 4: LISP Syntax of the General Object.

3.1.1 The Object Specifications Field.

Given an object name (or value) one can decide whether it is a top level, specializer, template, or instance and if not a top level, exactly what object is the given's superior. The specifications field has a constant "type" that is one of "IS", "INSTANTIATES", "TEMPLATE" or "SPECIALIZES". The superior is the second element if the object is a specializer, template, or instance.

The action of updating the parser dynamically (due to the evaluation of some semantic action at some point) can cause new specializers to be created. This is done by copying from a template object. An object that was created in this way includes a pointer back to the template from which it was kloned. This pointer is the third optional element of the specification field. It is used by the "object matcher" (discussed later).

3.1.2 The Parsing Roles.

Chapter 2 discussed in detail the use of the parsing roles in command understanding. The roles are kept in an association list between the name of the role and its value restriction. That value restriction is a parsing function in the case of an IS type object. In non-top-level objects, if the V/R is a list then it is a pointer to the name of the object whose roles will be used to subspecialize this role or a function which will be a predicate applied to the string

under consideration. A slight inconsistency between the form of an instance vs a specializer occurs here in that this list will have the form "(A name)" for a specializer but the form "(name)" for an instance. The purpose of the "A" is to differentiate predicates (that may begin with LAMBDA) from pointers to sub object. Functions cannot occur in an instance so the "A" keyword is not needed. This notation was suggested by Finin [9].

If the V/R is NIL then the string that is being matched must be empty. The remaining case, an atom, gives the exact value that the string must match.

3.1.3 The Semantics.

The semantics field is simply an expression that will be evaluated whenever an instance of this specializer is created. A disscussion of its use will make up the better part of the remainder of this thesis. The semantic expression is the main controller for the goal recognition process.

3.1.4 The Name/Path Mask.

It is often necessary to extract subparts of an instance that may be below the particular node in hand. It is possible to uniquely specify any sub-element of an instance (that is not arbitrarily deep in a List-of-things) by a path through the roles of this object and

its inferiors. For example, in figure 2, the instance of File-deletion-command, we might want to refer to the first filename in the List-of-filenames that is its Command-argument. We would have to specify the path:

(Command-argument Thing)

in order to access it. The name/path mask is a shorthand list which associates names with paths. It simply eases the task of access to parts of objects by letting the author specify the name instead of having to enter the whole path list.

3.2 Property Lists of Instances and Specializers.

The lists of the instances and specializers of objects are typically updated and scanned rather than being constructed and decomposed. Thus, they are not a part of the object itself but, rather, are kept in the properties INSTANCES and SPECIALIZERS on the name of the super object. These are simply lists of names (pointers).

The action of instantiation adds the name of the instance to the front of the INSTANCES property of the superior. Likewise, dynamically created SPECIALIZERS are added to the front of the specializers property of the superior.

3.3 The Utility of Recursive Objects: List-of-things.

DCL-commands make heavy use of lists of things. For example, most commands that apply to a single filename, like DELETE, apply also to a list of such names in the way that MAP applies to a list in LISP. It is necessary to be able to understand and work with such lists in a uniform manner. Finin [12] discusses the necessity of such lists and much of the design of WIZARD deals specifically with this topic.

In the semantic network, a list of things is a sequence separated by commas. If one wants to understand lists separated by something besides a comma, a new parsing object can be easily created. Commas appear most frequently as DCL-command list separators. [For the sake of consistency the PRINT command can have a list of things in which the delimiter is a plus-sign (+).]

The recursive object List-of-things has, as its last role, a pointer that must be another instance of List-of-things (or NIL to terminate the list). The normal action of the parser (as discussed in the previous chapter) will understand commands by separating the input string into separate "things" and building an instance of such a recursive list. Figure 2 (in chapter 2) shows how the list-of-things breaks up a list of filenames.

The semantic actions of an object can access either the entire list (by specifying a path to its head) or can extract any desired element if the exact position in the list of that element is known. It turns out to be more useful to simply select the entire list and then use special mapping utilities to process the members individually.

3.4 Primitive Network Objects.

Appendix B is a listing of some of the objects that are predefined in the semantic network. I have written their forms manually. Examples of WIZARD's processing will refer to these objects by name as the superiors of special parsing objects.

As previously noted, the main object is "DCL-command". This is "main" because it is the node that is passed to the parser in order to begin the process of understanding a command. All command forms have this object as their ultimate top level.

4.0 Chapter 4: Designing Goal Recognition Sequences.

In this chapter I deal with the high-level form of the semantic actions that drive WIZARD's goal recognition. The semantics portion of objects is used to write goal recognition "programs". These actually have much the same flavor that standard programs have. One can think of this section as an introduction to programming concepts for programmers that will be using the WIZARD semantics language. I will speak in terms of actions in the semantic network (like changing the values of variables in Pascal) and of flowcharts for goal recognition that are somewhat analogous to flowcharting in an iterative language.

4.1 General Approach.

As was previously mentioned, the parser that undertakes the understanding of DCL-commands is dynamic. That is, it can be updated by the addition or deletion of objects that understand commands. Since understanding is controlled solely by the objects in the semantic network, addition and deletion of objects will affect the operation of the parser.

If a specializer is added to the network in the correct place, it will act to understand commands that match its form (as discussed in chapter 2). If an extant specializer is detached from the network then commands that it would have understood are no longer understood

as they would have been before that object was deleted.

It should be clear that only the addition and removal of specializers affects the understanding process. The addition and removal of instances or templates would not serve any purpose since they do not take part in the understanding search. Top level objects should not be deleted.

The human author, who is responsible for WIZARD's functionality, writes goal recognition sequences by specifying the objects that will understand command strings and the actions that are to be performed upon successful matching. The actual process of the recognition of a goal takes place as a direct result of understanding some command. That command is usually the last in a particular sequence. This last command will be referred to as the "terminal command" for the recognition sequence. A help transaction is usually invoked by the semantics of a terminal object. Take, for example, the degenerate sequence which contains only one command: "$LOGOUT". The goal of this sequence is to terminate the session. If our only purpose in recognizing this sequence was to bid farewell to the user who is about to be logged off, we could embed a print in the semantics of the object which will recognize the LOGOUT command and have it say "see you later".

4.2 Specializer Templates.

Sequences are typically longer than one command in length. The method of dealing with longer sequences is to manually plant in the network the object which will understand the first command in the sequence. It will be that object's job to activate the objects that will understand the latter commands in the sequence. Objects are activated by being copied from a template.

A template is an object that does not itself match any string but can be duplicated and modified so that it will match something. The reason that a template does not act to match anything is that it is not a specializer, in the sense of being named in some superior's SPECIALIZERS list, until it is kloned.

For example: Suppose that we wanted to match the sequence:

```
$PRINT <filename-x>
$DELETE <filename-x>
```

where the same filename is to be specified. [This sequence might actually be used to match a misbehavior for PRINT with the /DELETE qualifier.] The author must have put the following objects into the network:

```
Print-command: Match "$PRINT <filename>"
   Make a new copy of Delete-command replacing the hole for
   a filename with the <filename>.

Delete-command [template]: Match "$DELETE ---"
   Tell the user that he could simply type "$PRINT/DELETE"
```

The successful matching of Print-command will cause activation of its semantics. They specify the creation of a new object from the template Delete-command, derived by filling the hole (indicated above by "---") with a filename. That process will cause the template to be copied to a new object which is a specializer. This new object now exists in the network and successful matching of it will cause the activation of its semantics to print the warning. The activation of that Delete-command copy is what we would refer to as the successful recognition of the goal under consideration.

4.3 Construction of Specilizers from Templates.

In the above example it was necessary to have in hand the actual name of the file that was printed in a form that would enable us to make a specilizer of the correct form to match that name. This is done by extracting V/R objects from the instance of Print-command that was that particular PRINT command and inserting these into the unfilled positions in the new object under construction. Parts of instances are accessed by paths that are named as discussed in the previous chapter. The template (or one of its superiors) has a name/path field also and a part of the semantic capability is to insert a particular at a named location in the new object. That particular would have been selected by name from the instance of Print-command that was in activation at that moment.

4.4 Dealing with Lists of Things.

Life is not quite as simple as the above example might suggest. Most of the complications derive from the occurrence of lists of things in the command. Consider the following two sequences:

```
$PRINT FILE1,FILE2,FILE3
$DELETE FILE2

$PRINT FILE2
$DELETE FILE1,FILE2,FILE3
```

In both cases we would like to be able to inform the user that the simpler: "PRINT FILE2/DELETE" would have sufficed. No simple command anticipation scheme will match the second of the set of commands to activate goal recognition. In WIZARD, such cases are handled by two mechanisms that are essentially complimentary.

4.4.1 Unwinding Lists into Multiple Traps.

The first method deals with the first of our examples. Specifically, the list of filenames that is the command argument of the print command is unwound in the semantics of Print-command into as many new Delete-command copies as as are required to cover all the possible deletions. This is done by mapping the template copying operation over the List-of-filenames that was formed from the parsing of Print-command. The result of this process is three new Delete-command specializers in the network, one for each of the files named in the first argument of the PRINT command.

4.4.2 Searching the Net for Matching Objects.

The method of handling the second problem case (in which a list is used in a command for which there might be an individual trap) is to again map through the list of filenames. This time instead of copying a template for each filename, we see whether there is a parsing object in the net that will match the formed object. Presumably such an object has been created by some previous activation (such as the just previous PRINT command). If such an object is found then its semantic expression is activated as if the object had matched an incoming string. Since each new object (formed from a template) includes a pointer to the template from which it came, it is a simple matter to tell which objects should be sought as a match for the current object.

4.5 Detaching Parsing Objects.

In addition to adding new parsing objects to the network, it must be possible to delete objects. We can see this need in the following undersirable behavior of WIZARD:

```
$RENAME A B
$COPY C D
$DELETE C
%You can use RENAME...
```

Obviously, the user knows about the RENAME command so he must have had some non-obvious motive for issuing the second and third commands. We do not want to suggest the use of RENAME if he already has demonstrated knowledge of that command.

The way I have chosen to deal with this is to permit the semantics of the object that would match a RENAME command to deactivate the object which would match COPY and begin the recognition of the COPY+DELETE sequence above.

Another example in which object deactivation serves us is in preventing multiple activation of the assistance message. That is, suppose there would be reason to issue the same command several times. If this command matches the terminal object for some goal recognition then the user is going to cause the activation of the help system each time the command is issued. This is somewhat undesirable (although it would certainly get the oint across).

In this case, the g neric object that was activated to recognize the goal would remove itself from the network. Perhaps it would also act to remove all other traps from the net that would give, now redundant, aid by telling the user the same thing.

4.6 Flow of Control Charting.

I have found it personally useful to diagram the semantic actions that I will apply to a goal recognition task. Figure 5 shows the "flowchart" (for lack of a better term) that represents the recognition of the PRINT+DELETE sequence discussed above. I will use this informalism to illustrate all recognition examples. This is not meant to be a detailed description of the recognition process but merely a visual aid indicating the sort of action that will take place.

The actions are represented by arrows in the figure 5. Solid arrows indicate the creation of a dynamic specilizer to match later commands in the sequence. Crossed arrows (—✕→) indicate deactivation of a parsing object. Note that all the dynamically created objects are self deactivating. A dashed arrow (- - - →) indicates object search rather than object creation. Wherever list unwinding takes place there is a one-to-many mapping via solid or dashed arrows. Lists of things are denoted by sequences followed by an elipsis ($\xi_n, \ldots$). Objects that are in the initial network (not created dynamically) are noted to the left by a double arrow (⟹).

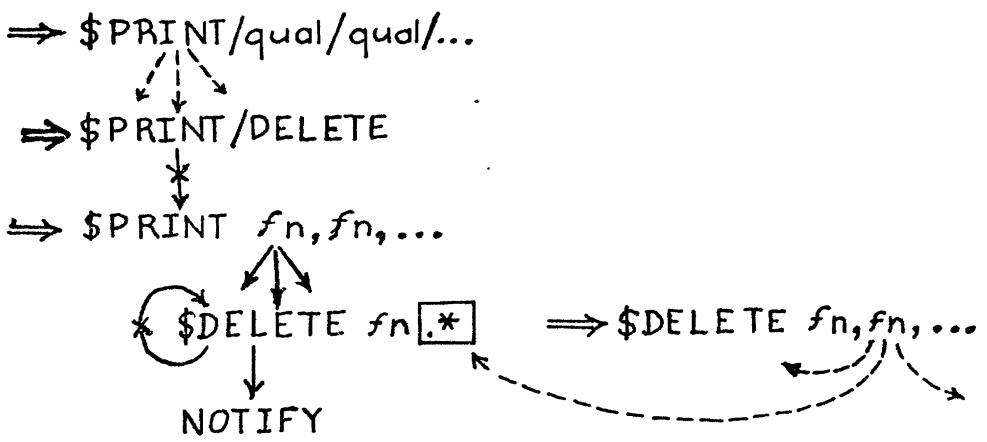

Figure 5: Flowchart for the PRINT+DELETE Recognition.

If the command PRINT/DELETE has ever been used, we need to deactivate the normally potential recognition of the PRINT+DELETE sequence. It would obviously not make much sense to tell the user what he knows already. This particular step is not so simple since it is difficult to recognize PRINT/DELETE. It might have had intervening qualifiers (such as: "PRINT/NOFLAG/DELETE" etc. In order to perform this task, it is necessary to unwind the list-of-qualifiers to the PRINT command and search in the network for a PRINT/DELETE object. The semantics of THAT object will cause the deactivation of the recognition sequence head -- not the general print command activation.

4.7 Considerations of Command Order.

One of the advantages that WIZARD holds over a straight pattern matcher is its that it will automatically ignore anything that it has not been told about. This turns out to be of use in many cases where commands that might intervene between parts of the sequence might not affect it in any way. For example, if the user interspersed a TYPE command between the COPY and DELETE commands above then we would probably want to ignore it even if the file typed was one of those copied. If we did not want to ignore commands that accessed those files, that capability exists as well by simply placing a parsing trap for that filename.

It is not always the case that we can tell syntactically whether a command will have an effect on our sequence recognition. Consider the following example of redirecting the output of the DIRECTORY command:

```
$ASSIGN S.TMP SYS$OUTPUT
$DIRECTORY *.*
$DEASSIGN SYS$OUTPUT
```

Should be recognized as attempting to capture the output of the DIR command in a file. The /OUTPUT= qualifier can be used in order to do this as follows:

```
$DIRECTORY/OUTPUT=S.TMP
```

In this case, commands that intervene between the two end commands affect the result only in case they generate output. It would be ludicrous to have the semantics of the ASSIGN command lay

traps for every command that might generate output. An approach beyond straight expectation is required to correctly recognizing this sequence.

4.7.1 Hunting the Command History List.

As was mentioned: a command history is retained which contains a list of all the interpretations of the commands entered. The way chosen to approach the general problem suggested above is by pattern matching against the command history list. Each member of the history list is a list of the names of the instances created for each parsed command. Thus, the commands in the ASSIGN+DIR example above might create the following history:

```
(       (assign-command-00001
         dcl-command-00003)

        (dir-command-00005-00008
         dcl-comma id-00010)

        (deassign-- :ommand-00006-00012
         dcl-comma id-00013)
)
```

This data is stored in the global variable "cmd-history".

Note that the latter command have two id numbers since they were matched by dynamically created objects that had their own numbers. The rule of name generation causes an additional id number to be appended to the parsing object's name. Also note that the most specific interpretation is the first in the sublist. DCL-command will be the last interpretation in any sublist since it is the start node

for the depth-first-search and succeeding in matching any input.

Utilities for matching against the command history list are not provided as WIZARD primitives but it is a reasonably simple matter to map down the "cmd-history" variable with any desired search. Thus, an additional criterion for the activation of the recognition (attached to the last command: DEASSIGN) in the ASSIGN+DIR example would be that the locations of the commands instantiated were next to one another. There are any number of other codings for this test even to the point of having some list of the commands that create output and seeing that one of them was not between the understanding objects.

4.8 The Form that Help Takes.

This work deals primarily with activation of the help processor rather than the form that the help itself will take. I have specifically avoided this topic but there are issues in the design of those interactions that rely heavily upon the command understanding stages. In particular, it is important to know, for example, which of several possible filenames was the one which activated the help request. The availability of this type of information can help improve the help interactions by clarifying the context in which the goal recognition succeeded.

WIZARD provides a way of accessing parts of instance objects. This can be used in one of two ways to pass the context of the recognition to later steps.

4.8.1 Passing Information in Properties.

The most straightforward method of passing context information is to simply stuff it into a property attached to the newly created parsing objects. For example, when the list of filenames in the COPY command that began a COPY+DELETE sequence is unwound, the name of the file at hand in each step of the unwinding can be put onto the property list of the DELETE parsing object created for that filename. Then all that the recognition process need do is get that name (and any other information that was squirreled away for its benefit).

4.8.2 Extracting Information from this Instance.

The other means of gaining context information is to lookup specific strings from the instance objects by simply using the WIZARD path access utility to dig out the proper leaf. This method can only be applied if the desired data is a part or subordinate of the object in hand. Such things as the time of day of the initial command match are clearly not going to be accessible in this manner and will have to be passed in a property.

4.9 Two Examples.

Here I include two examples of semantic programming techniques. We have seen these before as examples throughout this work and will see them again in detail in the next chapter. The treatment here is more concerned with the style of semantic programming needed to support their recognition. I apply techniques discussed throughout this chapter.

4.9.1 COPY+DELETE => RENAME

This first example is to recognize the sequence from which the idea of WIZARD first came. That is, recognize when a user is renaming a file by application of the COPY command and subsequent deletion (by DELETE command) of one of the source files from the copy. Figure 6 represents this recognition program. It is much the same problem as the PRINT+DELETE sequence mentioned earlier in this chapter.

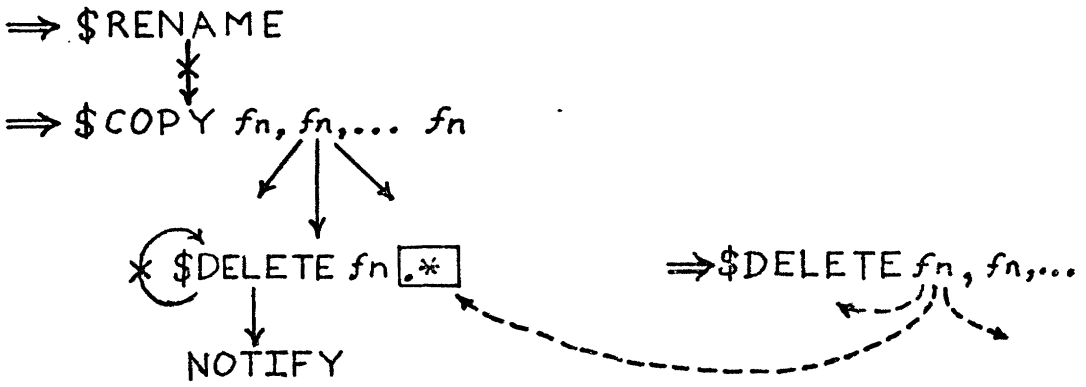

Figure 6: Flowchart for the COPY+DELETE Recognition.

If the user ever uses a RENAME command then remove the potential to begin this recognition (since he clearly already knows what we have to tell him). When a COPY command is encountered, set traps for a possible deletion of each individual file. Also, whenever a delete command is encountered, unwind the list of files to be deleted and search the net for an extant DELETE command parsing object which will match the filenames in the list. Each delete command object detaches itself upon activation.

We would like to present some specifics about the context of the recognition if one of the delete objects is activated. Thus, we will have to pass along the name of the file that was copied and the name to which it was copied. The way that these are actually passed will be detailed later.

4.9.2 ASSIGN SYS$OUTPUT+DIR => DIR/OUTPUT=

This is a three step recognition program. It is rather complex. The reader should refer to figure 7, the flowchart for this process.

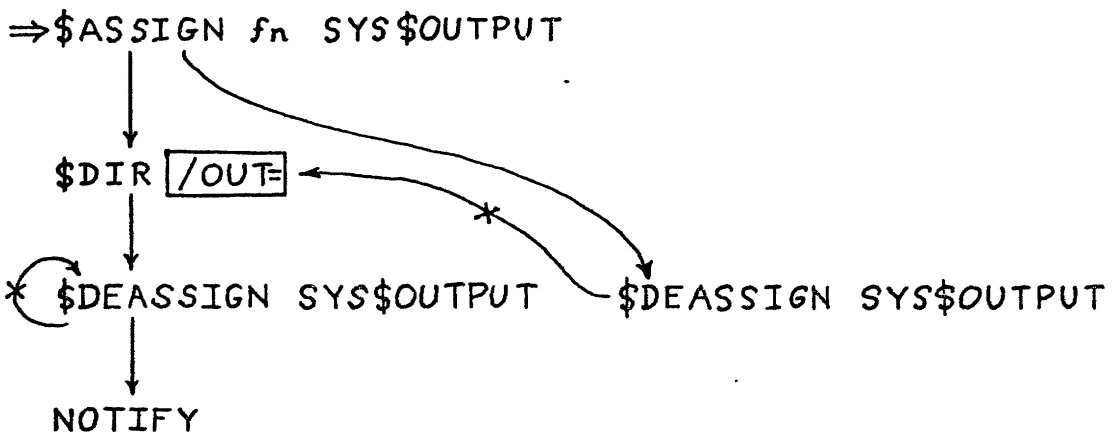

Figure 7: Flowchart for the ASSIGN+DIR Recognition.

The initial object for this sequence lays two traps, branching two possible paths of continuation. The first, the expected path, is a trap for a DIR command that does not have /OUT= as a qualifier. If this is located, then a trap for the terminal DEASSIGN command is laid. Activation of that terminal trap will cause activation of the notification (for the DIR/OUT=) command, only if there was only one command between the ASSIGN and DEASSIGN commands. Thus, if there were any other commands between the two, the user aid is not invoked. In any case, activation of the terminal DEASSIGN command detaches itself.

The other path spawned from the initial ASSIGN command is a trap for a matching DEASSIGN command. Should that take place, the trap created for the DIR command is deactivated along with this DEASSIGN command trap. Thus, if an ASSIGN+DEASSIGN pair occurs without having an interposed DIR command, the DIR trap is removed. If it were not removed there would be the possibility of a spurious recognition in the case of a later application of DIR command and a DEASSIGN command. Note that the name of the DIR command must be passed to this DEASSIGN command trap in order for this deactivation to take place.

This DEASSIGN trap will be activated in addition to the DEASSIGN trap from the other path in case a DIR command was issued. This is convenient, in this case, because it causes the cleanup of the DIR command trap for both paths.

5.0 Chapter 5: Detailed Primitive Semantics.

Whereas the previous chapter might be subtitled "The Art of Semantic Programming (volume I)", this chapter might bear the title "A WIZARD programmer's manual". Once the recognition plan has been outlined as discussed in Chapter 4, the author can apply the specifics in this chapter to make WIZARD perform the intended recognition.

Since the base programming language for semantic programs is LISP it is necessary that the author of WIZARD recognition programs be familiar with LISP. I will assume such familiarity in this chapter. Although the semantic actions are reasonably simple to use, they are only utilities and need to work with some amount of user written LISP code.

5.1 Objects vs Names.

Every object in the network has a unique name. Thus, given the name of an object one can get its body (the value). The opposite is also true (see chapter 3). As far as the semantic actions are concerned, a name is as good as its value and vice versa. The utilities all subscribe to the philosophy that no matter how many times the author is told to use a name here or a value there he will reverse them at some point. Thus, the first thing that each action utility does is to force the argument passed to be of the type that that function needs regardless of its initial form. I will use the

term "object" always to refer to the thing to be passed.

5.2 The Semantic Actions.

The semantic actions are a set of utility functions that are used by the author to write goal recognition programs. They are used as parts of the "semantics" expression. The particular functions provided are:

(lookup object path)
Find a subobject of something by path specification.
(apply-to-list object function)
Map the specified function over the given List-of-things binding the name of each thing to the function argument.
(new-copy template bindinglist)
Create a dynamic parsing object by copying a template object replacing subparts by specifics from the bindinglist.
(find-copy template bindinglist)
Make an object (as in new-copy) but instead of adding it to the net search for a matching object already extant.
(detach object)
Remove an object from the network.

5.2.1 Finding Parts of Objects.

Given an object in the network it is possible to extract the subobjects or specifics that make it up. The LOOKUP function provides the means of finding parts of objects by path specification. The form of a path has been explained in section 3.1.4. Handed an object and a path list, this function will return the thing (an object or atom) that resides at the end of the path.

5.2.2 Unwinding Lists of Things.

Paths obviously cannot be used to specify arbitrarily deeply nested things in a List-of-things. The way lists are handled is to unwind them. LOOKUP is used to find the top of the list and then this is passed to APPLY-TO-LIST. That function acts somewhat like MAPCAR: it takes the list and a function (typically a LAMBDA expression) and applies that function to each thing in the list.

APPLY-TO-LIST continues through the list until the result of some application fails to return NIL. That is, if the result of the application of the function to the thing is non-NIL then that value is returned as the result of the entire application and the APPLY-TO-LIST terminates. This provides a convenient way of using list unwinding to search through the list for some single object using the applied expression as a predicate. If the end of the list is reached, APPLY-TO-LIST returns NIL.

5.2.3 Creating a New Object.

New parsing objects are dynamically created specializers in the network. They are created by copying a template object and replacing holes in that template with specifics. Template objects must have been put into the network previously. The template should contain paths to each hole in its name/path mask field. Each hole should have a name/path pair. The second argument to the NEW-COPY function is a

list associating each name in the name/path mask with the value which is to fill that hole.

For example: consider the name/path mask which might be in a DELETE command template that will match the deletion of a specific file.

```
( (name (Command-argument Name))
  (ext (Command-argument Ext)) )
```

This might be bound by the bindinglist:

```
`( (name ,(lookup instancel
                        '(Command-argument First-thing Name))
   )
  (ext ,(lookup instancel
                        '(Command-argument First-thing Ext))
   )
 )
```

The above example demonstrates many techniques commonly applied in the use of the semantic actions. First, note the use of LOOKUP to select the specific to be bound into the object under construction. [It may be supposed that they are being selected from a COPY command where the two arguments to the command were parsed into a First-thing and a Second-thing.] The backquote macro comes in quite handy in these constructions since it permits us to form the bindinglist almost directly.

In the above case, the COPY command from which the filename parts are being selected is supposed to be a copy of only one filename. If we wished to have a new copy of the DELETE command object formed for each instance of a filename in a list of filenames (a List-of-things

where each thing is a filename) then we would have to embed the entire construction in an APPLY-TO-LIST as follows:

```
(apply-to-list
 (lookup this-instance '(Command-argument First-thing))
 '(lambda (obj)
        (new-copy Delete-template
         `((name ,(lookup obj '(Name)))
           (ext  ,(lookup obj '(Ext)))
          ))
        () ; result must be NIL
   )
)
```

Notice that a NIL is returned by the LAMBDA expression. This is so that APPLY-TO-LIST continues to process the entire list.

5.2.3.1 Naming the Current Instance.

The value of the global THIS-INSTANCE in the above expression is bound for the duration of the semantic evaluation, to the object that was instantiated in orde to cause this evaluation. This is needed in order to access portions of the command that was just entered.

5.2.3.2 Communicating to Later Steps.

NEW-COPY returns the name of the newly created specializer. If we wished to pass some information to the semantics of that potential activation, we could enclose the application of NEW-COPY in a PUTPROP expression and hang associated information onto it at will. For example, if we wished to pass the current time of day to the latter activation, we might write:

(putprop (new-copy ...) 'time-of-day (timeexpr))

5.2.3.3 Specifying with Subobjects.

In all of the above examples, I have been filling the template holes with specific parts of an extant object selected out by LOOKUP. It is important to note that this does not work quite as simply if the object with which we are filling the holes is another specializer (say, a complete Filename). Recall that the form of an instance (such as that which is the value of THIS-INSTANCE) is different from that of the specilizer that is under construction. It may be necessary to embed object names in lists that begin "(A ...)" in order that the parser properly process the specializer.

This is also necessary if new copies are to be embedded within other new copies. This particular embedding of new copy applications is typically not necessary since the template should be a complete object with all sub specifiers in place.

5.2.4 Finding a Matching Object.

Exactly analogous to NEW-COPY; one can hunt the network for an extant parsing object instead of creating one. This is used specifically in unwinding lists "backward" to see whether some trap has been set for a particular subcommand issued as a result of a list-of-things used within the current command.

For example, we wished to be able to cause the following behavior:

```
$COPY A B
$DELETE X,A,Y
%If you are trying to change the name of A...
```

The semantics of the COPY command laid traps for the specific command "DELETE A" but this command never appeared. Rather, it was embedded in the list deletion of the three files X,A,Y. In the semantics of that latter command we would want to put the expression:

```
(apply-to-list (lookup this-instance '(Command-argument))
 '(lambda (fname)
   (find-copy Delete-template
        `((name ,fname))
    )
   ()
   )
)
```

Here a subobject of the instance (the Filename object that was bound by the mapping action of APPLY-TO-LIST) is replaced for the NAME specifier in the same template from which the delete parsing object was created. The FIND-COPY utility will attempt to match all extant objects which were derived from Delete-template with this virtual object ("virtual" in the sense that it is not actually named and attached to the net but, rather, is simply being used to match extant objects).

The objects to be tested can be found via the ORIGIN field in the specification element of network objects (see chapter 3). Only the objects hanging from the template's superior need to be searched and only those that originated from the Delete-template itself.

5.2.5 Detaching an Object.

The DETACH utility is used to remove an object from the network. It is given the object and returns NIL. That object is REMOBed from the LISP workspace and extracted from the property lists of its parent. Objects below the one being detached are <u>not</u> REMOBed explicitly although their existence will not affect the speed of the parser unless they have been explicitly arranged as specializers of some other father. If such is the case then the author might wish to arrange to have these extra objects explicitly detached.

In order to perform self detachment, we must know the name of the parsing object which matched causing the instantiation of the current object. This name is kept in the global variable: PARSING-OBJECT which has a value for the duration of the evaluation of that object's semantics.

5.3 Two Examples Detailed.

Appendix C contains a slightly stripped form of the semantics code that performs the recognitions for the COPY+DELETE and ASSIGN+DIR goals. This appendix together with Appendix B, the primary network objects, form a complete and functional knowledge base for those recognitions. These recognizers are stripped in the sense that they only perform the recognition on complete command names (e.g., "del" is not an acceptable substitute for "delete").

there are several points to note in reading this code:

The ASSIGN+DIR recognition is invoked only if the original ASSIGN command is exactly 2 commands from the terminal DEASSIGN command thus handling the problem of interposing commands.

The VMS DELETE command requires a version number on the filename. This must be ignored in order to properly match with the filenames from the COPY command. Since it is not specified in the object sl-fileform, and it is not forced to any value (or NIL) by the semantics of sl-copy-command, it will be properly ignored in parsing or matching.

The filename (name.ext) is constructed by the sl-copy-command semantics and passed in the property 'filename' to the DELETE command. This is then used in the help message. That name could have just as simply been pulled out of the DELETE parsing object at notification time.

6.0 Chapter 6: Open Problems and Loose Ends.

In case it is not yet obvious to the reader, WIZARD is not perfect in concept or implementation. I can wave my hands at some of the difficulties and include them in "future directions". The more practical problems (sometimes known as design errors or, more concisely, bugs) are more difficult to explain away.

In this chapter, I will try to lay out what WIZARD might have been as well as what it cannot be without completely abandoning the anticipation method of goal recognition. I claim no excuse for having failed to implement something as "cleanly" or "correctly" as it might have been. Sometimes considerations of time prevented such revamping. Sometimes a minor restructuring in one area would demand a major restructuring in another and so the burden of the difficulty was left to the author/user.

6.1 Implementation Rest ictions (i.e., Bugs).

I begin the discussion of problems with those that are most clear: the bugs. These are not bugs in the sense of causing LISP error traps but rather misconsidered design of parts of the WIZARD system. Some of these problems have been discussed already in chapter 2 with relation to the simplicity of the parsing scheme. I will not reiterate those particular problems here.

6.1.1 Conflict of Object Form.

As discussed briefly in chapter 3, the form that an object takes differs between a specializer (or template) and an instance. We might often wish to attach a sub object that is not atomic to a hole in a template. This template will later be copied into a specializer. Herein lies the bug: the form of a sub object role filler in an instance is a list whose sole element is the name of the sub instance. In the case of a specializer, however, sub object role fillers are formed by a list like "(A ...)". The "A" form is necessary in order to distinguish a LAMBDA expression from a sub object. Instances should also be in this "A" format so that, as far at the matcher and parser are concerned, an instance object is not different from a specializer V/R. Currently, if one wishes to use an instance derived sub object to fill a hole in a template, it is necessary to create a new copy of that object as well and insert atomicly selected parts of the object in hand into that new copy. Only template derived specializers are valid parsing objects.

6.1.2 Redundant Parsing: An Efficiency Issue.

A problem with efficiency of the WIZARD parser lies in the action of the depth first search system. The current implementation begins all the way down at the leaves and backs up to the leaf's superior to preparse the string. This process is repeated for each leaf in the net. That is, for example, Copy-command and Delete-command are both

DCL commands. Thus, the expressions in the roles of the DCL-command objects are run for both of those sub specializations and each evaluation of DCL-command will clearly return the same result--a redundant and time-consuming operation.

There are two ways of avoiding this redundancy. The first involves recoding the parser so that as depth first search proceeds down (looking for a leaf at which to begin grinding) it will parse the string with the superior objects and pass the predigested selections to the leaf when one was finally found. Thus, the leaf parsing would not need to go to its superior before attempting to sub specialize its role bindings. The bindings would be immediately available.

The second method of fixing this inefficiency is somewhat less clean but does not involve rewriting the parser. That is to use memoing of some sort in order to have to evaluate the lexical functions of the superiors only once for each string.

6.2 Problems With the DCL Domain.

Next I will discuss some of the more difficult problems stemming from the chosen environment and the assumption of totally syntactic goal recognition. It turns out that only very limited recognition can be done on that basis due to non-transparent syntactic structures which expand into other structures. There are several areas in which information must be imported into the recognition process.

6.2.1 Wildcard Filename Compression.

DCL supports "wild card" filename specifications. That is, certain forms of filename are a shorthand for a whole list of actual files. For example, if one's directory contains the files "PHOO.BEAR" and "POO.BAR" the form: "P*.B*" is replaced internally with both those names. WIZARD has no provision for importing information such as the contents of the user's directory and would have to provide a pattern expansion function to handle such cases.

This particular type of imported information is not actually totally unreasonable and, in fact, work to handle exactly this expansion is currently in progress.

6.2.2 Symbolic Replacement and Command Files.

Another feature of DCL is the ability to replace any command word (actual or fictitious) with some other string. This information is kept in internal DCL tables and is not immediately available to WIZARD. Along with this facility, the more advanced user can write complete files whose contents are performed in place of a command (via an "@" prefix which is usually hidden in a string macro replacement).

For example, the user might create a file, CONFUSE.COM whose contents are a PRINT command followed by a DELETE command (as in a previous example). Then he might specify that whenever he used the command: "RENAME" it was to execute the CONFUSE.COM file. Thus, the

sequence:

```
$RENAME PHOO
```

would actually issue:

```
$PRINT PHOO
$DELETE PHOO
```

This would clearly confuse WIZARD entirely if it could not first get to the symbolic replacement tables which specify that RENAME=@CONFUSE.COM and, secondly, read that file and interpret its contents.

6.3 Theoretical Problems.

I now turn to more interesting problems. These are less like deficiencies and more like things that I did not think very hard about. In research, such things are qualified as either "not within the domain of this thesis" or as "future directions". In keeping with tradition, I mention here some of each.

6.3.1 Arguments of Deactivation.

In the current design of WIZARD, the only way that help transactions which would normally occur will not be invoked is if some other activation has caused the objects which would have generated those messages to be explicitly detached from the network. There are some simple rules that have been suggested for some sort of implicit deactivation. These rules have been argued in specific cases but they

have not been implemented because it is not clear that they will not cause undesirable effects in other cases. Some of the rules are:

6.3.1.1 Persistence of Dynamic Objects.

Once a COPY command has been issued, how long should the trap for a subsequent DELETE command remain? It would be somewhat unnerving to return three weeks later and issue the terminal DELETE so that the COPY+DELETE goal actions are invoked. In the mean time the user has probably learned of RENAME (either himself or via WIZARD) and needn't see that help message. That trap is simply taking up space and time and is meaningless after some number of sessions.

First, it should be clear that the mechanisms are available in WIZARD to store pointer to all the DELETE trapping objects in some global variable and then detach them all when a RENAME command is issued. This would correct the difficulty if the user figures out rename without the aid of WIZARD.

6.3.1.2 General Self-Deactivation.

It has been argued that all dynamically created objects should be restricted by some general deactivation rules. These are that every dynamic object detach itself upon its activation (so that it is not accidentally reactivated, causing the same help message or hanging a duplicate new object into the network) and that all objects created as

a result of a given unwinding process deactivate all its brothers when it is activated. These suggestions are both aimed at specifically avoiding redundancy of help messages and in some cases they seem to improve WIZARD's behavior.

It is not clear, however, that we would want such rules in all cases. Consider the case in which the node which would be deactivated is not a terminal node but, rather, spawns new objects that are dependent upon the input form. If this object were to self deactivate, only the first of the possible activations of that interim step would be actually executed.

6.3.1.3 User Profiles.

Closely associated with the duration of dynamic objects is the question of what knowledge should be preserved from session to session or over longer periods of time. It is not even clear that, in the case of our COPY+DELETE sequence having been detached by the use of a RENAME command, that sequence should not be restored. If the user still persists in using COPY+DELETE misbehaviors then we might either question his memory or question the potential of our misunderstanding some side effect of that sequence.

The stored knowledge base becomes a profile of the knowledge of the user and can be analyzed to determine how far along he has come or how quickly he is progressing. It also provides a log of all commands

issued and can thus be used in protocol analysis to, perhaps, actually teach WIZARD about this particular user.

6.3.2 Reconstruction of Strings for Help Messages.

WIZARD is really a help processor invocation system. I have not been especially concerned with the actual help interactions. This is primarily because I do not think that I could have done justice to both topics. There are, however, some simple applications of the WIZARD data structure and logic that might be useful in the construction of the actual help senarios.

All the context information that the help processor needs must either be passed along with the WIZARD program (in properties) or else extracted from the object whose activation invoked the help interaction. It is sometimes necessary to break apart objects in order to extract their parts for the help messages. For example, in order to print a properly formed filename in the COPY+DELETE help message it was necessary to extract the first part of the name and the extension separately and rebuild the filename manually (via string concatentation).

This should not be necessary. All the information exists in the objects in the network to reconstruct them into strings by simply reconconcatentating the parts in some uniform way. A list of filenames is a list-of-things where each thing is a filename. Thus,

in order to make the string from which a list of filenames was derived, simply insert commas between the reconstructed filenames. The process is not quite this simplistic because the top-level expressions might lose information (although probably not anything relevant) and there would have to be some rebuilding function that performed the catentations. Otherwise, we would have to go in and invert the operation of the LISP expressions that parsed the strings to begin with and this is certainly not a simple task.

6.4 Automatic Generation of Semantics.

Given that the WIZARD paradigm is the perfect user assistance frontend (of course), it remains only to clean up some of the messy programming that is involved in making it work properly. The reader, by this time, has probably reached the conclusion that it takes some very careful planning and a great deal of experience in order to write working WIZARD programs. It would be nice if an author could simply enter his flowchart in some higher level language and have it automatically converted into parser objects, templates, and associated semantics. In fact, it is a short hop from some less lispy object syntax to the actual network objects. The conversion is rather less well defined for semantic actions. One would have to include provisions for passing information on the side in properties. It is not beyond speculation, however, that a programming language could be designed which was compiled into wizard objects complete with

semantics.

The next, more interesting and difficult step, is to provide an "author's workbench". This utility would permit an author to review logs of beginner sessions and simply indicate which parts of the interaction are misbehaviors and what the distinguished sequence should be. The workbench utility would then code the WIZARD programs itself.

7.0 Chapter 7: Postmotivations and Possible Universes.

Although the goals of WIZARD are modest the general topic of the application of goal recognition to user assistance is in much need of research. I think that many of the ideas argued in this thesis, those ideas for which WIZARD is a test bed, deserve more thought than the industry or academia have yet put forth. In this, last, chapter, I try to justify (to myself as well as the reader) the time spent on this research and to demonstrate that such research is actually of some interest.

7.1 Advantages of Anticipation.

The goal recognition approach used here is not a novel technique. The use of anticipation to recognize tactics has been applied to natural language understanding (a la Riesbeck) among other areas. It is generally accepted as a reasonable approach in some cases. I originally chose this technique for several reasons. One is that it works particularly well in a non-iterative task domain (like a command language). Another reason is that it is simpler and more general, to my mind, to construct programs in order to recognize things than to design large and hairy patterns that are matched against entire sets of user logs.

7.1.1 Interspersed and Intertwined Commands

One specific advantage of the anticipatory approach over, for example, a pattern matcher is that ignorance of commands that do not affect the recognition is a built in feature. For example, as far as WIZARD is concerned the following are identical:

```
$COPY A B              $COPY A B
$PRINT C               $DELETE A
$DELETE A
```

Additionally, overlapping goal recognition programs are automatically handled as in the following example:

```
$COPY A B
$PRINT C
$DELETE A
%The RENAME command may be used...
$DELETE C
%The PRINT/DELETE command...
```

Another reasons for choosing the anticipatory scheme was that it is reasonably simple to describe the sequencing that becomes the goal recognition program. That is, an expert user can actually anticipate misbehaviors of beginners in the same sort of I/O behavior that WIZARD seems to use. Often a tutor will be able to anticipate what his student will do wrong before the mistake is made. WIZARD's heuristic depends upon this exact type of anticipatory ability.

A human being (a tutor of some lesson) learns to anticipate misbehaviors by having experienced them many times. WIZARD is taught explicitly which misbehaviors to seek out and recognize. It is not

too far an imaginative leap to think of WIZARD learning about such common mispractices by observation. I think that this might be accomplished by extension of WIZARD's knowledge and perception to include changes in the environment and by having WIZARD learn which commands effect which changes.

7.2 Knowledge Representation.

My final major motivation for the use of anticipation was that it fits well into the scheme of knowledge representation. KL-One is a system whose properties have always fascinated me but for which I had found little utility. The need for a dynamically modifiable pattern matcher gave me the opportunity explore knowledge representation. I do not think that it is a necessary feature of WIZARD. Snobol might be made to do somewhat the same type of recognition but the descriptions of the DCL language would have been much more complex and the unavailability of the power of a built in programming language would have made the anticipation programs that much more difficult.

The semantic network is designed to be extensible, simple and easy to use due to the natural way in which objects are described. I have found it to be a more than adequate tool for exactly the type of pattern matching that this work required. The ease with which I was able to describe DCL commands and have WIZARD understand them proved out the thesis that KL-One like data descriptions are useful and interesting.

Another feature of which I did not take full advantage is the capability of building a detail model of what a user, who has been running under WIZARD, knows about the command environment. It is my feeling that this is a virtually unlimited area of exploration in the user assistance area.

7.3 Advantages of WIZARD as a Help Invocation Paradigm.

If nothing else is gained from this thesis, it is my hope that the reader will recognize the importance of goal recognition in user assistance. Although I stated in the preface that I would avoid speculation about psychology, I do think that the general recognition paradigm is useful in describing the behavior of a real user consultant (I do not think that consultants use anticipatory methods but that is not important to this point).

Clearly, when one human being helps another with anything, the helper has some idea of what his student is trying to do. Trial and error training is useful only to a point. That point comes when the user knows how to get the job done and avoid silly errors (that is, syntax and trivial semantics). There is only so far that an entirely trial and error taught individual can go in an environment as rich as an operating system and after a time it is useful to have an expert suggest appropriate techniques of efficiency and cleanliness.

Goal directed help systems can be applied much more generally than I have done here. Imagine interactive programing environments that give the user a bit of programming guidance. These are not particularly novel ideas. I have concentrated on a very specific domain in hopes of being able to get interesting behavior. WIZARD is a functioning, albeit slow, implementation of exactly what I had imagined this work to produce.

Appendix A

The WIZARD Code

It is not necessary to read code in order to understand WIZARD's functionality. The bulk of this thesis discusses the algorithms implemented by the functions detailed in this appendix. I do, however, feel that it is important to people interested in working on WIZARD that the LISP code be properly documented and explained in detail.

The functions are divided into major sections of the system. Each section is briefly discussed and then the functions detailed. A short discussion of how the function fits into the WIZARD framework is included. If LISP scares you, simply ignore the code and read the comments. I have tried to be consistent about code style and commenting. Some functions are commented internally in order to explain non-transparent techniques. Comments always describe the code immediately following the comment.

WIZARD was developed and runs under John Foderaro's Franz LISP environment from Berkeley as modified by Lars Ericson at Carnegie Mellon. The system must be run in that environment.

The WIZARD Top-Level.

When WIZARD begins execution, the user seems to be talking directly to VAX DCL. The "$" DCL prompt is generated by WIZARD's top level control function and input is passed thru to the command system. The goal recognition subsystem "watches" commands as they are passed to DCL. A major assumption is made here that the command in not in error. There is no straight-forward way of retrieving error codes from DCL commands and thus I do not do so.

Since characters that are special to LISP (such as brackets and periods) are a normal part of the DCL command syntax, WIZARD temporarily replaces the LISP reader syntax array with a modified array in which those characters are treated as normal characters. This essentially kills LISP for use after WIZARD has been activated. That is only a problem if there is an error. The system supporter should be able to use LISP if an error occurs in WIZARD. In order to correct this problem, the old version of the syntax array is copied into a prog variable and the error demon knows to replace it in case of error.

The top level controller simply reads strings, issues the DCL command, and calls the grinder (parser) passing it the input command string and the DCL-command network object to process the string against as described in chapter 2. The list of successful parses for this string is then appended to the command history and a new command is read.

```
;
(setq cmd-history ())
;
;                          >>> wizard <<<
;
; The read-eval(dcl)-print[not really] loop for the wizard system.
; The reader has to be screwed with in order to get all the chars
; that DCL wants to see |[]| etc.  The error processors are also
; fixed so that the read tables are replaced on break or error.
;
(defun wizard ()
 (prog (hold-break table-holder hold-err)
    ; Save the old read table in case we want to put it back.  This is
    ; primarily for debugging purposes.
    (setq table-holder (makereadtable ()))
    ; Fix the reader to accept all chars.
    (set-wizard-reader)
    ; set up error handler so that WIZARD recovers properly (sort of)
```

```
    (setq hold-break (getd 'break))
    (putd 'break (getd 'wizard-break))
    (setq hold-err (getd 'err))
    (putd 'err (getd 'wizard-error))
    ; The MAIN LOOP !!!
  input
    (patom "
$")
    ; Translate the input line into a dcl command
    ; and run it thru the wizard processor.
    (ZARDOZ (list-->dcl (lineread '$)))
    (go input)
  ))
;
;                          >> ZARDOZ <<
;
; This is the wizard main driver.  Run the command and try to insert it
; into the semantic net if no execution errors are detected.  The error
; detection logic is currently non-existent.
;
(defun ZARDOZ (cmd)
 (prog ()
 ; Issue the command thru Ira's DCL link.
 (dcl cmd)
 ; If an error occurs then simply return.
 (cond ((dcl-error) (return ())) )
 ; OKAY -- insert the command in the net (this is where WIZARD processing
 ; takes effect!) and then append the list of applicable parsings into the
 ; command object history list.
 (setq cmd-history (append cmd-history (list (sn-insert 'dcl-command cmd))))
 ;
 ))
;
;                       >> list-->dcl <<
;
; Translates a list of atoms to the string representation of the entire list
; as a sentence with intervening spaces.
;
(defun list-->dcl (l)
 (cond ((equal (length l) 1) (car l))
       (t (concat (car l)
                  (concat " " (list-->dcl (cdr l))) ))
    ))
;
;                          >> dcl-error <<
;
; This function SHOULD return "t" if a command error was detected so that
; WIZARD knows not to parse commands that were in error, but the linkage is
; not yet in so it is currently a stub.
```

```
;
(defun dcl-error () nil)
;
;                    >> set-wizard-reader <<
;
; This function fixes the characters that should be inputtable to DCL but
; that LISP wants to use specially.
;
(defun set-wizard-reader ()
 (setsyntax '|'| 2)
 (setsyntax '|[| 2)
 (setsyntax '|]| 2)
 (setsyntax '|.| 2)
 (setsyntax '|,| 2)
 (setsyntax '|;| 2)
 (setsyntax '|(| 2)
 (setsyntax '|"| 2)
 (setsyntax '|)| 2)
 (setsyntax '|/| 2)
 )
;
;                       >> wizard-error <<
;
; Fix the error controller so that we return with life reset when an error
; occurs.
;
(defun wizard-error fexpr (a)
      (apply 'msg a)
      (putd 'err hold-err)
      (putd 'break hold-break)
      (setq readtable table-holder)
      (apply 'err a)
   )
;
;                       >> wizard-break <<
;
(defun wizard-break fexpr (a)
      (apply 'wizard-error a)
   )
```

Parser Control.

These functions are the parser controller and parse utility functions for use in top level selector functions. Lexical functions have the exploded string in a variable called STRING and the result is constructed in the variable RESULT. The utilities available are:

GRAB - Returns the next character in the string and removes that character from the input.
PEEK - Also returns the next input character but does not remove it.
DROP <c> - Put a character into the output string (RESULT).
SEEK <c-list> - Perform GRABs until the first character in the string is one of those in the list of characters (c-list).
REST - Copies all the rest of the characters into RESULT. This also stops processing of this parsing function.
DONE - Stop processing this parsing function and return RESULT.
SKIP-SPACES - Drops chars until the first character in STRING is not a space.

The RESULT of parsing is passed back via THROWs and CATCHs so the author of a lexical function needn't return a result from the function. The use of REST or DONE will terminate the parsing function properly. If an attempt is made (via PEEK, GRAB, etc) to get a character beyond the end of the string a DONE is forced.

```
;
;                          >>> parse <<<
;
; The main driver.  Explodes the string and the applies the parsing
; routine.  Results are collected and returned with the names of the
; routines.  This function actually takes a list of the names of the
; parsing routines and the string.
;
;               (parse '(name cqual args) "string")
;
; In actual use, the names will more likely be lambda exprs.
;
(defun parse (proc-list string)
 (prog (result)
  (setq string (explodec string))
  (return (parsesubr proc-list))
 )
)
;
```

```
;                          >>> parsesubr <<<
;
; The recursion thru this routine applies the eating functions.
; Parsing stops when we either run out of string or run out of pattern.
;
(defun parsesubr (p)
 (setq result ())
       ; If we're out of parsing functions then halt.
 (cond ((null p) ())
       ; If we're out of string then fill out the remainder of the roles
       ; with the empty string.
       ((null string)
      (cons (list (car p) "")
            (parsesubr (cdr p))))
       ; Actually parse the current string against the next role function.
       (t (cons (list (car p)
                      ; The parser will throw us a result from the application
                      ; of the named (or lambdaed) lexical routine.
                      (copy (catch (apply (car p) ()) 'parser-result)))
                (parsesubr (cdr p))
           ) )
   )
)
;
;                        Parse utility routines.
;
; Use the following primitives to recognize and "suck up" input characters:
;
;  1. Peek at the next character in the input stream (peek)
;  2. Suck up the next character in the input stream (grab)
;  3. Return with the string as is and the portion of the input that
;     was matched as the result of the function. (done)
;  4. Take in the rest of the input string. (rest)
;
; The functions use the global variables STRING and RESULT to represent
; incoming string and the string to be returned.
;
;                              >>> peek <<<
;
; Look at the next character of input without removing it from the
; list.
;
(defun peek ()
 (cond ((null string) (done))
       (t (get_pname (car string)))
 )
)
;
;                              >>> grab <<<
```

```
;
; Return the next char and remove it from the list.
;
(defun grab ()
 (cond ((null string) (done))
       (t (get_pname (car (list (car string) (setq string (cdr string)) ))) )
  ))
;
;                          >>> done <<<
;
; Return RESULT from this parse proc by throwing it to the waiting catch.
; This is done so that we can get out of any prog loops that the caller
; has jammed in the way when the string runs out.
;
(defun done () (throw (get_pname (implode result)) parser-result))
;
;                          >>> rest <<<
;
; Copy the remainder of string onto the result list.
;
(defun rest () (setq result (append result string)))
;
; The following auxilliary functions are also provided for convenience of the
; procedures:
;
;   (skip-spaces) ... Skip over and throw out spaces in the input string.
;   (drop char) ... Add the named character to what will be the result
;                   of this application.
;   (seek char) ... Do (drop (grab)) until the named char is seen.  That
;                   char will be left in the first po ition of the input
;                   string (for the next peek or grab).
;
;                       >>> skip-spaces <<<
;
; Luse all spaces in the front of the string.
;
(defun skip-spaces ()
 (do ((c (peek) (progn (grab) (peek))))
     ((not (equal c " ")) ())
 )
)
;
;                          >>> drop <<<
;
; Put a character into the result string.
;
(defun drop (c) (setq result (append result (list c))) )
;
;                          >>> seek <<<
```

```
;
; Look thru the string for any of the characters named and drop all the
; characters in the way into the result.
; If the user calls this with an atomic result, it is clever enough to
; listify the atom.
;
(defun seek (c)
       ; If the wip gave up an atom then make it a list for member.
 (cond ((atom c) (seek (list c)) )
       (t (do ((x (peek) (progn (drop (grab)) (peek))))
              ((member x c) ()) ) )
 )
)
```

Object Access Utilities.

This set of functions is like InterLISP record descriptions for the parts of objects. They are value based selectors and selection and update functions for property based information. The FORCE-TYPE object/name coersion utility is also here. That is used in the semantic actions to force a name to be the object or vice versa.

```
;
;                  Functions to decompose objects
;
;                        >>> object-name <<<
;
; Each object has a unique identifier that locates it's top preciscely in
; the network.  This function takes an object body to its name.
;
(defun object-name (obj)
      (car obj)
)
;
;                             >>> specs <<<
;
; The object's specification section indicates whether this is an instance of
; or a specializer of some other (higher) object.  If not then the specs list
; will be simply (is).
;
(defun specs (obj)
      (cadr obj)
)
;
;                           >>> spec-type <<<
;
; Passed an object body, this function tells exactly the type of
; specification one of: SPECIALISES, IS, or INSTANTIATES.
;
(defun spec-type (obj)
      (car (specs obj))
)
;
;                         >>> super-object <<<
;
; For a non-generic object, this function tells which object the current
; one is an instance of if any.
;
(defun super-object (obj)
      (cadr (specs obj))
```

```
)
;
;                          >>> origin <<<
;
; The third element of the specifications list (if there is a third element)
; will contain the name of the object from which this object was constructed.
; Presumably the construction was performed by (new-copy). This exists mainly
; for the efficiency of the function (find-copy) so that only the objects that
; have an appropriate origin tag need be searched.
;
(defun origin (obj)
      (caddr (specs obj))
)
;
;                        >>> parse-body <<<
;
; Every object has a description of the expressions that can be used by the
; parser and specialization logic to match string elements to this object. In
; the case of a generic (IS) these functions must be lexical analyzers. See
; the net-manager and parser for a more detailed discussion.
;
(defun parse-body (obj)
      (caddr obj)
)
;
;                          >>> semantics <<<
;
; Given an object, this function indicates what the side effect of making
; an instance of this object is. Every object has semantics although they
; may be nil in many cases.
;
(defun semantics (obj)
      (cadddr obj)
)
;
;                            >>> mask <<<
;
; The mask is used to locate lower parts of an
; object by associating an externally available name with a path by which
; the lookup processors can get to parts of the object.
;
(defun mask (obj)
      (caddddr obj)
)
;
;                          >>> specializers <<<
;
; Now we are into the things that hang near objects but are not part of
; them per-se. Most of these associated things are in the PLIST of the
```

```
; object and thus these functions require the NAME of the object rather than
; the body.
;
; specializers are the sub-concepts that hang from a generic.  They are also
; generics.  The contents of this property are arranged at net main loading
; time by the net loading functions or by action of (new-copy).
;
(defun specializers (obj)
      (get obj 'specializers)
)
;
;                          >>> instances <<<
;
; Instances are much like specializers execept that the parser does not see
; instances in the paring process and instances are created at run time rather
; than at load time.  The parser makes an instance of a generic when the parse
; succeeds.
;
(defun instances (obj)
      (get  obj 'instances)
)
;
;                    Functions to update objects
;
;                          >>> add-specializer <<<
;
; The name of the specializer is added to the front of the name of its father.
; specializers are specializers that will be searched by the parser.
;
(defun add-specializer (object-name specializer-name)
      (property-update object-name specializer-name 'specializers)
)
;
;                         >>> add-instance <<<
;
; An instance is exactly like a specializer expect that they are not scanned
; by the MSS parser.  This adds the name of the instance genned by the parser
; to the father node.
;
(defun add-instance (object-name instance-id)
      (property-update object-name instance-id 'instances)
)
;
;                         >> property-update <<
;
; A utility that jams a new name on the front of the property specified in the
; object specified.  Used by add-specializer and add-instance.
;
(defun property-update (object-name x property)
```

```
      (putprop object-name
               (cons x (get object-name property))
               property
        )
)
;
;                          >>> set-obj <<<
;
(defun set-obj (name specs parse semantics mask)
      (set name
             (list
                    name
                    specs
                    parse
                    semantics
                    mask
             )
      )
)
;
;                          >>> force-type <<<
;
; Used to coerce a name to its object or vice versa.  The first arg is the
; name of the var to be forced (via set) into the type specified by the
; second arg.  That is, either "name" or "object".  This is used primarily
; in the semantics in order to protect authors from making stupid errors.
; I have avoided using it within the WIZARD code because it would be too
; slow and, more importantly, if there is a mismatch within WIZARD then
; something might be screwed up and I'd prefer it bomb out.
;
(defun force-type fexpr (a)
       ; If the given is an atom ...
 (cond ((atom (eval (car a)))
            ; ... and caller wants name then leave it alone.
      (cond ((equal (cadr a) 'name) ())
            ; Otherwise its wrong and has to be coerced!
            (t (set (car a) (eval (eval (car a)))))
        ))
       ; For lists ... (i.e., objects)
             ; ... if caller wants a name the get the name from the list.
       (t (cond ((equal (cadr a) 'name)
                 (set (car a) (object-name (eval (car a)))))
             ; else ok -- leave her alone.
             (t ())
        ))
 )
)
```

Network Management.

These are the DCL network intializer functions and the parse grinder that puts incoming strings into the network as instances. The first few functions (net-construct, etc) are simply used at WIZARD startup to put the original objects into the net. Each object is hooked into the net properly and the properties that point an object to it children (instances or specializers) are updated properly. It is important that the father go into the network before its children so that the updating occur correctly.

The set of functions beginning with GRIND are the real heart of WIZARD. They process a string against the DCL-command object in the network and cause the parsing and instantiation to occur. The grinder also causes the evaluation of the semantics to occur thus running the goal recognition system.

```
;
;                       >>> net-construct <<<
;
; The function that inserts objects into the net.  The objects are
; simply setqed into their respective names. Each object gets arranged
; with a property 'specializers, which will contain a list of all objects that
; directly specialize this object, and a property 'instances which will
; contain all objects parsed by the object in hand.
; The generics must be input before their specializations.
;
(defun net-construct fexpr (nlist)
      ; Map over the list of passed objects and insert each into the net.
      (mapcar 'net-insert nlist)
)
;
;                          >> net-insert <<
;
; The work routine for net-construct.  Takes one object at a time and jams
; them into the network.
;
(defun net-insert (n)
      ; Put it into the net.
      (set (object-name n) n)
      ; Set up the specializers and instances lists to NIL initally.
      ; Since get returns nil if there was nothing there this is a bit
      ; redundant but cleaner.  It also permits reloading the net in order
      ; to clean it out.
      (putprop (object-name n) () 'specializers)
      (putprop (object-name n) () 'instances)
```

```
      ; If this object specializes something else, add its name to
      ; it father's specializers list.
      (cond ((equal 'specializes (spec-type n))
             (add-specializer (super-object n) (object-name n)) )
      )
)
;
;                        >>> sn-insert <<<
;
; This function takes a pointer to the top node in the net of the type
; of object to be analyzed and a string.  A Bottom-up search is performed
; on the entire structure below the named object and the string's parse is
; inserted as an instance of ALL successful parses!
;
(defun sn-insert (startobj string)
      ; Insertsub expects to be getting a list of specializers.  This (list)
      ; makes it think that the start-obj is the only child of some virtual
      ; higher concept. Like religion.
      (insertsub (list startobj))
)
;
;                          >> insertsub <<
;
; The depth first search is driven by this function.  Standard hack: first
; do the kiddies then do the parent then to the brothers.
;
(defun insertsub (obj)
       ; If father had no specializers then I must be a figment of the DFS's
       ; imagination.  Pop back so that it can do my father now.
 (cond ((null obj) ())
          ; Inskeep retains the search results.
                ; Do the kids of the first node...
       (t (inskeep (insertsub (specializers (car obj)))
                   ; Exapand-obj instantiates the results of the grinder.
                                ; The grinder is the parsing controller.
             (inskeep (expand-obj (grind (car obj) (eval (car obj)) string))
                      ; Lastly do the following brothers.
                     (insertsub (cdr obj))
           )
         )
       )
    )
)
;
;                          >> inskeep <<
;
; This is used primarily to conconcatentate only successful parses into the
; history list.  It flattens the funny looking things that can come out
; of the DFS algorithm.
```

```
;
(defun inskeep (item rest)
    (cond ((null item) rest)
        ((null rest) (list item))
        ((atom item) (cons item rest))
        (t (append item rest))
       )
)
;
;                          >>> grind <<<
;
; This function does the legendary bottom-up top down search that is the
; parsing algorithm.  It is bottom up in the sense that it begins from
; some object that is subspecifies a higher object and goes after the
; objects above for the parser driving functions.  It is top down in
; that it goes depth-first into the object's own roles.  Hand this function
; a string and and the name of some object in the net.  Every object is
; either a generic (begins with "is") or a specializer (begins with
; "specializes").  If it is a specializer then we parse the string at
; hand according to the superconcept and then reparse it according to
; the "with" restirctions of the roles on this object.
;
; The main grind function sets a catch in case there is a mismatch deep
; in the net.  If there is then the pattern matcher will fail and throw
; back to this level.
;
(defun grind (name object string)
 (catch (grind-sub name object string) 'grinder-flag) )
;
;                          >> grind-sub <<
;
(defun grind-sub (name object string)
 (cond ((null string) ()
       ; If this is a to, level then simply stop here and parse the
       ; roles at this level.  Return the result for specialization by the
       ; caller if it so desires.
       ((equal 'is (spec-type object))
                                ; Parse-driver only wants to see the roles.
      (cons name (parse-driver (parse-body object) string)) )
       ; If this object specializes another object then go and parse its
       ; father then apply the additonal role restrictions at this level to
       ; the father's parsed roles.
       ((equal 'specializes (spec-type object))
      (cons name
            ; specialization is done by matching the roles of the object
            ; at hand to the result of the grind of this object's father.
            (specialize
                (cdr (grind-sub (super-object object)
                                (eval (super-object object))
```

```
                                string
                        ))
                ; Feed the roles to the specializer as well.
                (parse-body object)
              )
        ) )
  )
)
;
;                       >> parse-driver <<
;
; This function is called to run an object and a string thru the parser.
; it constructs a parse list and then unconstructs the parsed pieces into
; just the names of the roles and the items that they match.
;
(defun parse-driver (p s)
      ; Rewind the parse result.  ; Unwind the roles for parsing.
      ; I.e., reinsert the names. ; ((name lex)...)  --> (lex lex...)
      (disembed-parse-list (parse (mapcar 'cadr p) s) p)
)
;
;                     >> disembed-parse-list <<
;
; Reconstructs the result of parsing with the names replaced for the lex
; functions.
;
(defun disembed-parse-list (l p)
 (cond ((null p) ())
       (t (cons (list (caar p) (cadar l))
                (disembed-parse-list (cdr l) (cdr p))
        ) )
   )
)
;
;                       >>> specialize <<<
;
; Take the result of a ground string and process each parsed element
; by applying the context of the with clause included in this specialization.
; With parts are either an atom which must be equal the ground data, ()
; which means that the ground data must be empty, or a list which indicates
; that the grinder should be called recursively to grind this string with
; the named object.  These lists are in the form (a <objectname>). The
; "a" is just for show.
;
(defun specialize (r s)
 (cond ((null r) ())
       ; For each element in the result of the parse, if there is a special
       ; role to be applied, apply that role restiction.
       ((assoc (caar r) s)
```

```
                          ; specsub does the actual role/value restriction.
      (cons (list (caar r) (specsub (cadr (assoc (caar r) s))
                                    (cadar r)
                                    )
                  )
            (specialize (cdr r) s)
       ))
       ; If no specializer is named for this role then simply accept it and
       ; go on to the next one.
       (t (cons (car r)
                (specialize (cdr r) s) ))
  )
)
;
;                          >> specsub <<
;
; This does the real work of specialization.  The second member of the role
; restiction is one of:
;     () - indicating that this role must be unfilled,
;     atomic - indicating that an exact match is required,
;              a list of the form "(a ...)" indicating that "..."
;              is the name of an object which this will be
;              subprocessed by, or,
;     another list which is assumed to be a monadic predicate
;             which will test the string.
;
(defun specsub (sp st)
       ; If the spec is () then the string must be empty.
 (cond ((null sp) (cond ((equal "" st) "")
                        (t (grind-fail)) ))
       ; If the string is nil then something's wrong.  Since the spec was
       ; not also "".  This is a succeding match beleive it or not!
       ((equal "" st) ())
       ; If the spec is atomic then they have to be equal.
       ((atom sp) (cond ((equal sp st) sp)
                        (t (grind-fail)) ))
       ; Aha!  Is there a sub object to do further grinding?  If so then go.
       ((equal 'a (car sp))
      (grind-sub (cadr sp) (eval (cadr sp)) st))
       ; I don;t understand it so simply try to apply the predicate and hope
       ; that they luser knows what he's doing.
       (t
      (cond ((apply sp (list st)) st)
            (t (grind-fail)) ))
  )
)
;
;                          >> grind-fail <<
;
```

```
; Called when a grind mismatch occurs.  This causes the grinder to stop cold
; and go on to the next possible parsing object.
;
(defun grind-fail () (throw () grinder-flag) )
;
;                        >>> expand-obj <<<
;
; After the grinder has decided exactly where to put
; the object, this function takes the compacted form of the instance
; (as returned from the grinder) and instantiates each object on the correct
; superconcepts.  The result is a pointer to the top node.
;
; The object comes into this function in the form:
;
;     (name (rolename filler) (rolename filler)...)
;
(defun expand-obj (r)
       ; If the grinder failed then forget it.
 (cond ((null r) ())
                                ; Do the roles FIRST.  Then...
        ; ... do the top.
       (t (make-instance (car r) (expand-roles (cdr r))) )
  )
)
;
;                       >> expand-roles <<
;
; The roles are each analyzed and either instantiated in place (if they are
; not lists) or are recursively attached to sub instances.
;
(defun expand-roles (roles)
 (cond ((null roles) ())
       ; If this filler is a list then expand its sub concept.
       ((listp (cadar roles))
            ; Reattach the name to the sub concept instance.
      (cons (list (caar roles) (list (expand-obj (cadar roles))))
            ; And then do the rest of the roles.
            (expand-roles (cdr roles)) ) )
       ; Atomic or string values needn't have instances made of them.
       (t (cons (car roles) (expand-roles (cdr roles)) ) )
  )
)
;
;                       >>> make-instance <<<
;
; Make-instance and set-instace form an instance name from a gensym
; value and the name of this object.  It is then attached to the generic.
;
(defun make-instance (super roles)
```

```
      (set-instance
              super
              roles
              (gen-name super)
         )
)
;
;                        >> set-instance <<
;
(defun set-instance (super roles instance)
      (set-obj instance                          ; object name
               (list 'instantiates super)        ; specifier
               roles                             ; parse expr
               ()                                ; semantics
               ()                                ; mask
         )
      (add-instance super instance)
      ; When the object is instantiated... execute the semantic
      ; component.  The semantics have the var "parsing-object"
        ; available in order
      ; to name the particular superior specializer under consideration.
      ; Also, "this-instance" is the name of this particular instance.
      (eval-semantics super instance (semantics (eval super)))
      ; Return the name of the new instance from this function
      instance
)
;
;                      >> eval-semantics <<
;
(defun eval-semantics (parsing-object this-instance semantics)
      (eval semantics)
)
;
;                          >> gen-name <<
;
; Takes the prefix that you would like to see on the new name and adds a
; unique 5 digit numerical tag to it.
;
(defun gen-name (prefix)
      (implode (append (explode prefix)
                       (cons '- (cdr (explode (gensym))))
                  )
         )
)
```

Semantic Action Utilities

These are the programmers utilities for goal recognition processing. They are discussed in detail in chapter 5. Each function demands arguments that are network objects or names in a certain form. The FORCE-TYPE function (from the record package) is used to perform coersion when it is needed.

```
;
;                        >>> lookup <<<
;
; Given an object and a path list, this function will go down the object
; and extract the value that matches the path specification.  For example,
; in the object that matched "del a,b" the list-of-filenames "a,b" would
; be found at the path specified as '(command-argument).
;
(defun lookup (object path)
 ; Make sure that the object is not a name.  This is typically a result
 ; of having called it from an apply-to-list but might be an error.
 (force-type object object)
       ; If we've run out of path then this is it!
 (cond ((null path) object)
       ; If there is more path to travel, select the correct branch
       ; from this point and then continue processing at that role.
       (t (lookup-step (cadr (assoc (car path) (parse-body object)))
                       (cdr path)
          ))
   )
)
;
;                        >> lookup-step <<
;
; If we are at the end of the line for this role then simply return the
; atom that it at the end of the search path.  This might be a failure
; but currently is not error flagged.
;
(defun lookup-step (role path)
       ; A role out of steam will have an atomic binder.
 (cond ((atom role) role)
       ; Otherwise get the next level object and recur.
       (t (lookup (eval (car role)) path))
  )
)
;
;                        >>> apply-to-list <<<
;
```

```
; This is a version of apply that applys the function to all the elements
; of a list-of-things.  It continues down the last role of the object until
; the value of the function is t.  If it hits the end then it returns nil.
; The fn should take one arg that will be applied to the atom that is the
; name of the "thing" role filler.
;
(defun apply-to-list (object fn)
 (cond ((null object) ())
              ; Apply fn and stop if it returns anything but ().
       (t (cond ((apply fn (cadar (parse-body object))))
              ; Otherwise select the next "thing" from the list and recur.
              (t (apply-to-list (eval (caadadr (parse-body object))) fn) )
          ))
  )
)
;
;                          >>> new-copy <<<
;
; Given any object and a binding list
; this function will make a new object and insert it into the network.
; The binding list is in the form ((name replacement) (name replacement)...)
; The names are matched to the mask names in the object passed and then
; the replacement objects are put in the new object in the places indicated
; by the mask paths associated with each name.
;
(defun new-copy (obj binders)
      (make-copy obj (super-object obj) (path-expand binders obj))
)
;
;                          >> path-expand <<
;
; The binding list (as name-replacement pairs) has to be translated
; into path-replacement pairs.  This function performs that conversion by
; mapping through the list of bindings and doing a get-path on each name.
; The path search is anchored at the current object and proceeds up to the
; top level super concept.
;
(defun path-expand (bindlist obj)
      (mapcar '(lambda (binding)
                                    ; get-path does most of the work here.
                        (cons (cadr (get-path obj (car binding)))
                              (cdr binding)
                        )
                )
              bindlist
        )
)
;
;                          >>> get-path <<<
```

```
;
; This function takes an object and a path identifier that is an element
; of a mask in one of the object's superiors.  If the name is not found in
; that object or in one of it's superiors up to the IS concept then nil is
; returned.  The path is returned otherwise.  The search is performed from the
; bottom up so there may be multiple occurrences of a name that are chenged
; toward the base.
;
(defun get-path (obj pathid)
       ; If the path name is in the current element then simply return
       ; the rolename-rolefiller pair.
 (cond ((assoc pathid (mask obj)))
       ; See if the top is this one.  If so then an illegal path has been
       ; specified and an error SHOULD be returned (but nil is instead).
       ((equal 'is (spec-type obj)) ())
       ; Try to get the path from this guy's father.
       (t (get-path (eval (super-object obj)) pathid) )
   )
)
;
;                          >> make-copy <<
;
; This function takes an object and an expanded bindingpath-replacement list
; It copies the old object into all new instances by instantiating this node
; and then instantiating each of the parse-body-role nodes that represent the
; subconcepts that make up this concept.
;
(defun make-copy (node super bindlist)
      (set-special
              (object-name node)
              super
              ; Expand the roles of this object into specializers as well.
              (copy-roles (parse-body node) bindlist)
              ; A new name is genned for this parsing object.  Thus any
              ; instances of this object will have double numbers when they
              ; have names genned.
              (gen-name super)
              ; The semantics of the old node are copied.
              (semantics node)
        )
)
;
;                          >> set-special <<
;
; Called in order to add a parsing object to the network.  The new object
; is assumed to specialize its super.  This is NOT an instance.
;
(defun set-special (origin super roles name semantics)
      (set-obj name
```

```
                (list 'specializes super origin)
                roles
                semantics
                ()
           )
        ; Add the name of the new parsing object as a specialier of the
        ; parent so that the parsing search finds it.
        (add-specializer super name)
        ; This function returns the name as a result.
        name
)
;
;                          >> copy-roles <<
;
; Go thru each member of the parse-body of this object and expand each role
; by either returning the object that is its replacement (if there is one
; specified) or a copy of the sub concept to which this role expands.
;
(defun copy-roles (roles bindlist)
 (cond ((null roles) ())
       ; If a binding has been specified for this role then insert that value
       ; in place of whatever the author originally had in this place.
       ((find-binding (caar roles) bindlist)
        (cons (list (caar roles)
                    ; find-binding is repeated (sorry) and could probably be
                    ; replaced by some memoing or a lambda bind later.
                    (find-binding (caar roles) bindlist) )
              ; Do the rest of the roles as well
              (copy-roles (cdr roles) bindlist)
        ))
       ; If the specification of this role is a sub-object (a ---) then make
       ; a new copy of that object as well and reinsert that name here.
       ((and (listp (cadar roles)) (equal 'a (caadar roles)) )
        (cons (list (caar roles)
                    (list 'a
                       ; Recursively call make-copy on the subobject
                       (make-copy (eval (cadadar roles))
                           ; Make-copy needs the super of the sub also.
                           (super-object (eval (cadadar roles)))
                           ; Pass only those bindings that apply to this role.
                           (applicable-binders (caar roles) bindlist)
                         )
                       )          ;  close (list 'a...)
                    )
              ; Again, be sure to do the rest of the roles.
              (copy-roles (cdr roles) bindlist)
        ))
       ; There are no applicable bindings and there is no subobject.  This
       ; role is probably filled by either an atom or some lambda expr.
```

```
        ; Just copy it as is.
        (t (cons (car roles) (copy-roles (cdr roles) bindlist)))
 )
)
;
;                    >> applicable-binders <<
;
; This takes a role name and the binding list and returns only those binders
; that might apply to this name.  The cdr of the paths of those binders
; is returned so that it can be reused immediately.
;
(defun applicable-binders (name bindlist)
 (cond ((null bindlist) ())
        ; The binders in role "foo" that start "(foo ---)" are selected and...
        ((equal (caaar bindlist) name)
             ; ... "foo" is pulled out of the path list.
       (cons (cons (cdaar bindlist) (cdar bindlist))
             (applicable-binders name (cdr bindlist))
         ))
        ; Skip any that don't begin "(foo ---)"
        (t (applicable-binders name (cdr bindlist)) )
 )
)
;
;                        >> down-level <<
;
; Used to move to the next position in the paths listed in the bindlist.
; Each path is CDRed.
;
(defun down-level (bindings)
      (mapcar '(lambda (b) (cons (cdar b) (cdr b))) b.ndings)
)
;
;                        >> find-binding <<
;
; Searchs throught the bindlist and returns either () or the object to act
; to replace the current sub-part.
;
(defun find-binding (name bindlist)
 (cond ((null bindlist) ())
        ; The qualifications for being a valid binder are that it is of the
        ; form exactly "(foo)" for role "foo".  Longer lists are for deeper
        ; paths.
        ((and (equal 1 (length (caar bindlist)))
              (equal name (caaar bindlist))       )
       (cadar bindlist))
        (t (find-binding name (cdr bindlist)))
    )
)
```

```
;
;                          >>> detach <<<
;
; Removes an object from the network.  This is typically used in the semantic
; actions in order to stop the processing of traps. Note that only specializers
; can be removed from the net and this function assumse that it has been handed
; a specializer [type:(specializes foo)].
;
(defun detach (object)
      ; Make sure that the author passed us an object rather than a name.
      (force-type object object)
      ; Tell daddy that we're leaving home.
      (remove-specializer (super-object object) (object-name object))
      ; Remove the name itself thus unhooking all the lower stuff.
      (remob (object-name object))
)
;
;                          >> remove-specializer <<
;
; This is used to remove the name of a specializer that is going to be remmed
; from the specializer property of its father.  It is pretty straightforward.
;
(defun remove-specializer (super obname)
      (putprop super
               (rem-name-from-list (get super 'specializers) obname)
               'specializers
         )
)
;
;                     >> rem-name-from-list <<
;
; Standard pull an atom from a list of atoms by name.
; This should be a standard lisp function.
;
(defun rem-name-from-list (l n)
 (cond ((null l) ())
       ((equal n (car l))
      (cdr l))
       (t (cons (car l) (rem-name-from-list (cdr l) n)) )
  )
)
;
;                          >>> find-copy <<<
;
; This is like new-copy except that instead of instantiating a specializer
; in the place specified it tries to match the new object with existing
; specializers hanging from the superior that the new copy would be hung from.
; The 'origin' field of the object specification is used to decide which
; specializers of the superior are to be tested.  Only ones that were derived
```

```
; from this particular spec are tried for reasons of efficiency.
; If it matches then the sematics of the object that is matched are evaled.
; The "parsing-object" variable is set temporarily to the name of the
; specializer being activated so that its semantics work properly.
;
; It is important to note that this function is comparing the content of
; an instance with the content of a specializer so it has to be real careful
; about the form of the roles of the specifier vs the binders.
;
(defun find-copy (obj binders)
 ; Make sure that the author gave us the right type.
 (force-type obj object)
              ; Use the names in the mask field to replace the named parts
              ; with their paths for processing.
 (find-copy-sub (path-expand binders obj)
              ; Pass the list of objects to be searched.  I.e., the kids of
              ; the superior to this object.
              (specializers (super-object obj))
              (object-name obj)
    )
)
;
;                        >> find-copy-sub <<
;
; Map down the list of the "brethren" of the matching node and compare the
; ones that derived from this specializer.
;
(defun find-copy-sub (bindlist searchlist restriction)
 (cond ((null searchlist) ())
       (t (match-object (car searchlist)
                        ; Make the name into an object.
                        (eval (car searchlist))
                        ; Restrict the matcher so that only those objects
                        ; whose origin was the node with which we are doing
                        ; the comparison are compared.
                        restriction
                        bindlist
              )
        ; As always... cdr down the list.
        (find-copy-sub bindlist (cdr searchlist) restriction)
       )
  )
)
;
;                        >> match-object <<
;
; If the object under consideration matches the object in hand then do
; the semantics of the EXISTING object.  Note that only the binders are
; really relevant to this comparison since the object in hand derived from
```

```
; the object under consideration by application of the same set of binders.
;
(defun match-object (name obj restriction bindlist)
       ; This is restricted to only those whose origin is the object in hand.
 (cond ((equal (origin obj) restriction)
       (cond ((compare-obj (parse-body obj) bindlist)
              ; Bind the name of the super object to "parsing-object"
              ; and nil to
              ; "this-instance" then eval the semantics of the super.
              (eval-semantics name () (semantics obj)))
             (t ())
         )
       )
        (t ())
   )
)
;
;                          >> compare-obj <<
;
; this guy does the control work for matching two objects.  It takes the
; parse bodies of the two objects and the binding replacement list and does
; the comparison.
;
(defun compare-obj (roles bindlist)
 (cond ((null roles) t)
       ; If you can find a binder for this role then match the role's value
       ; with the bind replacement value.
       ((find-binding (caar roles) bindlist)
           ; Match the binder value with the role it matched.
       (and (match-binder
              (parse-body (eval (find-binding (caar roles) bindlist)))
              (parse-body (eval (cadadar roles)))
             )
           ; Be sure to consider the goodness of fit of the rest of the
           ; roles as well.
           (compare-obj (cdr roles) bindlist)
        ))
       ; If there's no binder then try to rest of the roles and dive into
       ; this one's value cell.
             ; Comparing the rest of the roles.  This is done first beacuse
             ; the job of diving might be considerable in a list of things.
       (t (and (compare-obj (cdr roles) bindlist)
             ; Trying to match role values.  If the role has a list cadr
             ; then dive into it and compare some more.
             (cond ((listp (cadar roles))
                    (compare-obj (parse-body (eval (cadadar roles)))
                                 ; Take the binders that matter with you.
                                 (applicable-binders (caar roles) bindlist)
                     )
```

```
                    )
                   (t t)
                )
            ))
    )
)
;
;                        >> match-binder <<
;
; The actual comparison between two objects is done here.  Remember that
; obj1 is an instance and obj2 is a specializer that theoretically
; bound that instance.  Thus there is all sorts of cruft in obj2 that won't
; be in obj1 but that has to be check anyhow.  If there is no role in one
; that matches the other then it is an unspecified role and matches by
; definition.
;
(defun match-binder (obj1 obj2)
 (cond ((null obj1) t)
       (t (and (match-binder (cdr obj1) obj2)
               ; This is the relevant line.  Compare the value of this role
               ; with the value of the things that bound it.
               (match-role (car obj1) (assoc (caar obj1) obj2))
          ))
  )
)
;
;                          >> match-role <<
;
(defun match-role (r1 r2)
       ; They can be directly equal or...
 (cond ((equal r1 r2) t)
       ; If r2 is nil then this role was unspecified
       ((null r2) t)
       ; If not lists then they are just plain wrong here and now.
       ((not (listp (cadr r1))) ())
       ; ...we have to dive into their subobjects.
       (t (match-binder (parse-body (eval (cadr r1)))
                        (parse-body (eval (cadadr r2)))
             ))
    )
)
```

Appendix B

Primary DCL Network Objects

```
(net-construct
;
;
;                    Main generic decriptions:
;
; dcl-command
; list-of-things
; filename (and fileform)
; list-of-filenames
;
(dcl-command (is)
      (       (command-name
                      (lambda ()
                       (prog (c)
                             (skip-spaces)
                      loop   (setq c (peek))
                             (or (equal c " ")
                                 (equal c "/")
                                 (null (drop (grab)))
                                 (go loop)
                               )
                             (done)
                        ))
                 ) ; Close command-name

               (command-qualifier
                       (lambda ()
                         (prog (c)
                             (skip-spaces)
                      loop   (setq c (peek))
                             (cond ((not (equal c "/")) (done)) )
                      iloop  (drop (grab))
                             (skip-spaces)
                             (seek '(" " "="))
                             (skip-spaces)
                             (cond ((equal "=" (peek)) (go iloop)))
                             (go loop)
                        ))
                 ) ; Close command-qualifier

               (command-argument
                       (lambda ()
                         (skip-spaces)
                         (rest)
```

```
                        (done)
                        )
                ) ; close command-argument
        ) ; close parsing body

      () ; semantics

      () ; mask

   ) ; close dcl-command
(list-of-things (is)
      (       (thing
                      (lambda () (seek ",") (done))
                ) ; close thing

              (rest-of-list
                      (lambda () (grab) (rest) (done))
                ) ; close rest-of-list
        ) ; Close parsing body

      () ; Semantics
      () ; mask

   ) ; close list-of-things
;
(list-of-filenames (specializes list-of-things)
      (       (thing (a filename))
              (rest-of-list (a list-of-filenames))
        )

      () ; Semantics
      () ; mask

   )
(fileform (is)
      (       (device (lambda ()
                              (cond ((not (member '|:| string)) (done)))
                              (seek ":")
                              (grab)
                              (done)
                        )
                )
              (location (lambda ()
                              (cond ((not (equal "[" (peek))) (done)))
                              (grab)
                              (seek "]")
                              (grab)
                              (done)
                        )
```

```
                )
               (name (lambda ()
                             (seek ".") (grab) (done))
                )
               (ext (lambda ()
                             (seek '("." ";")) (grab) (done))
                )
               (version (lambda () (rest) (done)) )
          ) ; Close parsing body

        () ; Semantics
        () ; mask

    ) ; close fileform
;
; The relation between fileform and filename is a function of the operation
; of the net searcher.  It assumes that all sub-objects have some super and
; will only take lexical scanning fns from the father.  Thus, in order to
; make the parser process down the string into subpieces we have to give
; it a father node to get lexical fns from.  Fileform is that father.
;
(filename (specializes fileform)
        ()
        ()
        ()
        ) ; close filename
;
) ; close net-construct
```

Appendix C

Example Complete Parsing Objects

This appendix includes the actual code of the COPY+DELETE and one possible construction of the ASSIGN+DIR sequence recognition programs. Their description can be found in chapter 5. Normally, these objects would be surounded by a NET-CONSTRUCT call in order to place thier object parts into the network. Note that these are stipped down versions of the actual objects. They take only the full form of the command names and some of the communications has been removed in order to clarify the semantic actions a bit.

I maintain a labeling convention in which each sequence recognition program has a "sequence number" that is prefixed to all objects that take part in that recogntion. Thus, the COPY+DELETE sequence is "s1". Some objects are more general than only a particular sequence but not sufficiently general to include in the primary network objects. These are not prefixed with a sequence number (e.g., Pair-of-things).

```
;
;
;       The COPY/DELETE misbehavior that we speak of so often
;
; The commands captured by this sequence are:
;
;       $COPY <> ...
;       $DELETE <>
;
; and warn of the RENAME command. Also, the use of a rename command
; deactivates the top trapping copy.
;
(s1-copy-command (specializes dcl-command)
      (       (command-name "copy")
              (command-qualifier ())
              (command-argument (a copy-pair))
         ) ; Close parsing body

 (progn (apply-to-list (lookup this-instance '(command-argument first-thing))
       '(lambda (name)
             (putprop
              (new-copy s1-single-file-delete-command
               `((name ,(lookup (eval name) '(name)))
                 (ext  ,(lookup (eval name) '(ext)))
                )
```

```
                ) ; close new-copy
                (concat (lookup name '(name))
                        (concat "." (lookup name '(ext))))
                'filename
                ) ; Close putprop
               () ; force apply-to-list to continue
             ) ; close lambda
           ) ; close apply-to-list
         ) ; close progn

      () ; mask

   )
;
(pair-of-things (is)
      (       (first-thing
                      (lambda ()
                              (seek " ")
                              (done)
                         )
                )
                (second-thing
                      (lambda ()
                              (skip-spaces)
                              (rest)
                              (done)
                         )
                )
        ) ; Close parsing body

      () ; Semantics
      () ; mask

   )
;
(copy-pair (specializes pair-of-things)
      (       (first-thing (a list-of-filenames))
              (second-thing (a list-of-filenames))
        )

      () ; Semantics
      () ; mask

   )
;
(delete-command (specializes dcl-command)
      (       (command-name "delete")  )      ; Parsing body
      ()      ; Semantics
      ()      ; mask
```

```
   )
;
;
(file-deletion-command (specializes delete-command)
      (       (command-qualifier ())  ; Parsing body
              (command-argument (a list-of-filenames))
        ) ; Close Parsing body

 (progn (apply-to-list (lookup this-instance '(command-argument))
       '(lambda (name)
         (find-copy sl-single-file-delete-command
              `((file ,name))
           )
         ()
         )
        ) ; close apply-to-list
    ) ; close progn

      ((filenames (command-argument))) ; mask

   )
;
(sl-single-file-delete-command (template delete-command)
      (       (command-name "delete")
              (command-qualifier ())
              (command-argument (a sl-filename))
        )
   (progn
    (patom "The RENAME command can be used to change the name of ")
    (print (get parsing-object 'filename))
    (patom "
Look into $HELP RENAME for more info.")
    (detach parsing-object)
    )
      ((file (command-argument))
       (name (command-argument name))
       (ext  (command-argument ext))
      )
   )
;
(sl-filename (template fileform)
      ()
      ()
      ()
      ) ; close filename
;
(sl-rename-command (specializes dcl-command)
      (       (command-name "rename")
              (command-argument (a copy-pair))
```

```
          )
       (detach s1-copy-command)
       ()
   )
;
;                  Assign sys$output sequence
;
; This misbehavior is meant to recognize command streams like:
;
;     $assign s.tmp sys$output
;     $dir
;     $deassign sys$output
;
; The distinguished sequence in this case is "dir/output=s.tmp"
;
(s2-assign-command (specializes dcl-command)
      (       (command-name "assign")
              (command-qualifier ())
              (command-argument (a s2-assign-pair))
         )
   (prog (s2-temp)
      (putprop (setq s2-temp (new-copy s2-dir-command () ))
           (length cmd-history)
           'where)
      (putprop (new-copy s2-general-deassign
                   ()) s2-temp 'dircmnd)
     )
      ()
   ) ; close s2-assign-command
;
(s2-assign-pair (specializes pair-of-things)
      (       (first-thing (a filename))
              (second-thing "sys$output")
         )
      ()
      ()
   ) ; close s2-assign-pair
;
(s2-dir-command (template dcl-command)
      (       (command-name "dir")
              (command-qualifier ())
         )
   (progn
      (putprop
      (new-copy s2-deassign-command () )
      (get parsing-object 'where)
      'where)
    )
      ()
```

```
)
;
(s2-deassign-command (template dcl-command)
      (       (command-name "deassign")
              (command-argument "sys$output")
      )
   (progn
     (cond ((equal 2 (- (length cmd-history) (get parsing-object 'where)))
          (patom "The /out=filename option can be used to direct the
output of the DIR command to a file.
See HELP DIR/OUT for more info.")))
      (detach parsing-object)
   )
   ()
 )

;
(s2-general-deassign (template dcl-command)
      ( (command-name "deassign") )
      (progn (detach (get parsing-object 'dircmnd))
             (detach parsing-object)
      )
      ()
)
```

Bibliography

[1] Ball, Eugene, and Phil Hayes; "Representation of Task-Specific Knowledge in a Gracefully Interacting User Interface"; CMU CS dept., in AAAI August 1980.

[2] Brachman, Ron, et al.; "KL-One Reference Manual"; BBN Report no. 3848, July 1978.

[3] Brachman, Ron; "A Structural Paradigm for Representing Knowledge"; BBN Report no. 3605, May 1978.

[4] Digital Equipment Corp.; "VAX/VMS Command Language User's Guide"; DEC Order no. AA-D023B-TE, March 1980.

[5] Foderaro, John K. [edited by Lars Ericson] "The FRANZ LISP Manual"; by Berkeley and CMU, 1980.

[6] Mark, William; "Rule Based Inference in Large Knowledge Bases"; USC/Information Sciences Institute, published in AAAI August 1980.

[7] Genesereth, Michael R.; "The Role of Plans in Automated Consultation"; Laboratory for Computer Science, MIT.

[8] Sacerdoti, Earl D.; "A Structure for Plans and Behavior"; SRI project report no. 3805; August 1975.

[9] Finin, Timothy; "The Semantic Interpretation of Nominal Compounds"; University of Illinois Coordinated Science Laboratory research report no. T-96; 1980

[10] Riesbeck, C. and R. Schank; "Comprehension by Computer: Expectation Based Analysis of Sentences in Context"; Yale University CS research report no. 84.

[11] Riesbeck, C; "Computational Understanding: Analysis of Sentences and Context"; 1974 working paper from Instituto per gli Studi Semantici e Cognitivi; Castagnola, Switzerland.

[12] Finin, Tim, et al; "Jets: Achieving Completeness Through Coverage and Closure"; IJCAI-79.

[13] Bagley, Steven and Jeff Shrager; "LISP: An Introduction"; Moore School Computing Facility, University of Pennsylvania, 1980.