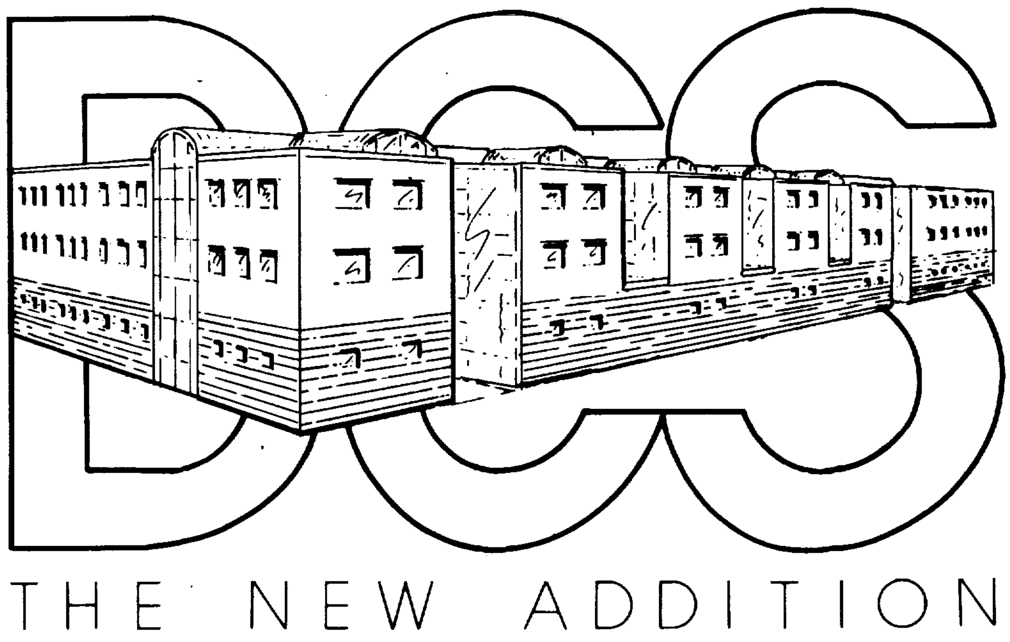
~~N90-21911~~

IN-61 291075 ~~N90-21914~~

DEPARTMENT OF COMPUTER SCIENCE

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

50P



REPORT NO. UIUCDCS-R-89-1535 UILU-ENG-89-1754

QATT: A NATURAL LANGUAGE INTERFACE FOR QPE

by

Douglas Robert-Graham White

August 1989

QATT: A NATURAL LANGUAGE INTERFACE FOR QPE

BY

DOUGLAS ROBERT-GRAHAM WHITE

B.S., Purdue University, 1987

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois, Urbana-Champaign, 1989

Urbana, Illinois

QATT: A NATURAL LANGUAGE INTERFACE FOR QPE

Douglas Robert-Graham White, M.S.
Department of Computer Science
University of Illinois at Urbana-Champaign, 1989

This thesis presents QATT, a natural language interface developed for the Qualitative Process Engine (QPE) system. The major goal of the project was to evaluate the use of a preexisting natural language understanding system designed to be tailored for query processing in multiple domains of application. The other goal of QATT is to provide a comfortable environment in which to query envisionments in order to gain insight into the qualitative behavior of physical systems. It is shown that the use of the preexisting system made possible the development of a reasonably useful interface in a few months.

ACKNOWLEDGEMENTS

I first want to thank my thesis advisor, Ken Forbus, for his motivation, patience and guidance. And many thanks to the members of the Qualitative Reasoning Group, especially Dennis, John, Randy, and Rob for their help and humor. I owe much to Dr. Richard Grace for his encouragement. I also wish to thank AT&T, and my supervisor Mark Dailey, for making my graduate study possible. And special thanks to Michelle for her support and understanding.

This research was supported by National Aeronautics and Space Administration, Contract Number NAG69137, an IBM equipment grant, and AT&T.

The author's current address :

```
AT&T Bell Laboratories
2000 N. Naperville Rd.
Naperville, IL 60566-7033
```

TABLE OF CONTENTS

LIST OF FIGURES

1 INTRODUCTION

This thesis presents QATT, a natural language interface developed for the Qualitative Process Engine (QPE), a qualitative simulator (Forbus, 1988).

The major goal for the project is to evaluate the use of a preexisting natural language understanding system which was designed to be domain-independent. Can off-the-shelf natural language technology be used to quickly generate reasonably useful interfaces? To answer this question I built an interface, attempting to replicate the capabilities of preexisting, landmark systems, such as SOPHIE and LIFER. The other goal of QATT is to provide a comfortable environment in which to query envisionments in order to gain insight into qualitative models. The hope is that with a more friendly interface, QPE will become even more useful and accessible. Throughout the thesis, I will assume that the reader has deep familiarity with Qualitative Process theory (Forbus, 1984) and QPE.

1.1 Potential Users

When building a natural language understanding system, the first consideration should be the characterizing the potential users of the system. Users should be classified by their knowledge of the application domain and by any special requirements they may have. This defines the requirements of the system.

The potential users of QATT can be divided into three classes :

1. Students trying to learn qualitative physics or the internal workings of QPE.

2. QPE programmers who already know QPE's inner workings.

3. People using QPE to model real world systems.

Each class of user brings its own problems to QATT. For students it is paramount to minimize the level of frustration involved in using the interface. Otherwise, they will spend time learning the capabilities of QATT instead of learning about QPE. This requires QATT to accept a wide variety of input sentences. When a sentence is not accepted, the system should explain why it didn't understand. With this feedback, the student need not spend time guessing about what was not understood, and can more easily find an alternative statement that will work. The students may also require more detailed and informative responses, in whole sentences or paragraphs, since they may be unfamiliar with QPE nomenclature and formats.

The frequency of use of QATT by QPE programmers also places requirements on the interface. They require quick responses. And as they become more familiar with the interface, they will also desire customized short-cuts to allow extracting information with minimal key strokes. And finally, they don't want to be forced to wade through long-winded textual responses to find the answers to their queries. They want short, concise responses.

The last class of users of QATT, those using QPE in the field, require all of the above features in an interface. Since they will probably be fairly frequent users, they need low response delays and the power to customize their input. But as QPE novices, they will be prone to entering sentences that the interface cannot understand. Consequently, the interface will need to accept many sentences in the domain, to be highly tolerant of errors, and to provide helpful feedback when sentences are not understood.

QATT attempts to make all classes of users comfortable. Being an experimental tool, though, QATT concentrates on the needs of the constant users of QPE. This is due mainly to the time constraints of the project, but also to the complexity of dealing with novice users, and the problems they cause a natural language interface. However, many features of QATT are implemented for the other kinds of users, and Chapter 4 shows how they make the interface more friendly.

1.2 Grammar Skeleton

QATT was developed from another natural language understanding system called ATT (Martin, 1985). ATT's premise was that a natural language understanding system could be separated into a domain-independent part (grammar, interpreter, etc.), and a domain-dependent part (lexicon). So, with a general grammar skeleton in place, the ATT could in principle be configured for a new application just by redefining the lexicon. For QATT then, I would only need to extend the lexicon to reflect the QPE domain, by adding QPE-specific verbs and nouns, and simply use the existing ATT grammar for parsing.

1.3 Evaluating QATT

A large part of the thesis is the evaluation of this approach. Did the use of this grammar skeleton aid in the interface's development? One measure of the approach's merit is in the amount of effort required to configure the new domain, which included defining the lexicon, and in fact extending the grammar where it was insufficient.

Time constraints ruled out "field tests" on a statistically significant population of users, so my evaluation will of necessity be more analytic. One way to gauge the outcome of this project, is to compare the features of QATT with those of SOPHIE (Burton & Brown, 1979). SOPHIE was an intelligent computer-assisted instructional tool designed to teach troubleshooting electronic circuits. SOPHIE included a successful natural language system that in its time set a new standard of performance. One hopes that as a technology advances, creating new systems becomes easier and easier. SOPHIE was developed in the mid to late 1970's as part of a multi-year, multi-person project. QATT was developed in only a few months by me. Has the technology improved enough to allow SOPHIE-quality interfaces to be rapidly developed? Chapter 5 discusses the outcome.

2 REQUIREMENTS OF NATURAL LANGUAGE INTERFACES

To characterize the needs of a natural language interface, we must first understand what separates good interfaces from bad interfaces. The most important factor is the comfort of the user. If a user is uncomfortable with any interface, it becomes a detriment instead of a helpful feature. In a good interface, the user does not need to think about it at all. With an "invisible" interface, the user can concentrate on the task at hand.

In (Burton & Brown, 1979), Burton and Brown give a list of the requirements of a natural language understanding system. Among these are efficiency, habitability, self-tutoring ability, and awareness of ambiguity. These are detailed below, and in Chapter 5 are used to evaluate the QATT system.

2.1 Efficiency

One thing that users dislike is the delay between the entering of the input, and the eventual response from the system. While a system is "thinking", the user may lose interest, his mind may wander, and by the time the system comes back, he may have forgotten the purpose of the query. Worse yet is the anxiety of a new user, wondering what he could have done wrong as the system crunches away slowly.

Burton and Brown cite (Miller, 1968), whose study showed that response delays of more than two seconds negatively effected the performance of complex tasks on computers. So an interface should try to respond under this two second mark. But, there is a trade off between speed and coverage of the sub-language, as the next section shows.

2.2 Habitability

No system to date can understand all of English. Such a system would be incredibly complex and would have to be infinitely expandable, as the English language is. So natural language understanding systems typically characterize and understand a subset of English. The system should strive for, as (Watt, 1968) puts it, a *habitable* sub-language, or "one in which users can express themselves without straying over the language boundaries into unallowed sentences". The sub-language should also allow the user to make "minor" modifications to an accepted sentence, and still get an accepted sentence. Of course, the specification of "minor" is open to interpretation. But, Brown and Burton give a good example, shown in Figure 2.1. Sentences 2 and 3 seem to be minor variants of sentence 1, so if the system accepts 1, it should also accept sentences 2 and 3. Sentence 4 gives a semantic extension, and should probably also be accepted. Sentence 5, though easily understood in common conversation, is such a permuted variation, that it would probably be considered out of the scope of a habitable system. So, the sub-language should provide more than just adequate coverage of the concepts of the domain: it should maintain a comfortable coverage that will allow users to easily work in the sub-language.

Another feature of habitability is the understanding of context. As a user starts to feel more comfortable with an interface, she will typically use contextual knowledge in her dialog. The user, as she starts to feel that the interface is more and more intelligent, will subconsciously start making

1. *"Is anything wrong?"*
2. *"Is there anything wrong?"*
3. *"Is there something wrong?"*
4. *"Is there anything wrong with Section 3?"*
5. *"Does it look to you as if Sections 3 could have a problem?"*

Figure 2.1: "Minor" modifications to sentences.

1. *"What is the population of Los Angeles?"*
2. *"What is it for San Francisco?"*
3. *"What about San Diego?"*

Figure 2.2: Multiple sentence phenomena.

more assumptions about its abilities. Included in what Burton and Brown call the *multiple sentence phenomena* are the contextual issues of pronominalization, ellipsis, and anaphoric deletion.

Pronominalization is the use of a pronoun for some referent noun. The only way to define the referent of the pronoun is to evaluate the context of the conversation, and even then there may be ambiguities. Figure 2.2 shows an example from (Burton & Brown, 1979). In sentence 2, *it* refers to *population*, but without contextual knowledge, sentence 2 is not clear. And what if the response to sentence 1 made reference to the increase of the population of Los Angeles as being due to the beautiful weather there. Then, the *it* in sentence 2 could be considered as referring to the weather in San Francisco. Resolving such ambiguous references can be extremely difficult.

Users will often begin to use pronouns as they grow familiar with the system, and it is important to allow this. To accommodate pronominalization requires recognizing it, maintaining context, and providing for the possibility of ambiguous reference.

Ellipsis is another multiple sentence phenomena. Sentence 3 of figure 2.2 provides an example. Here the system would need to recognize that *San Diego* is a noun, but the system must also determine how the current context dictates the noun's usage (i.e. it will not take the place of *population* in sentence 1, but rather *Los Angeles*).

So ellipsis is the implicit substitution of one element of a sentence for another, based on context. The difficulties with ellipses include recognizing the elliptical reference, which may be only a sentence fragment, and then finding its place in the current context.

Deletion of a part of a sentence also leads to problems. A user may unknowingly omit a meaningful part of a sentence, and the system must hypothesize about what is missing. In sentence 3 of figure 2.2, there is no reference to population; not even with a pronoun. So the system must recognize that a constituent is missing, and then make its best guess as to what that missing constituent is.

Habitability can be characterized as flexibility in the sub-language of the application domain. A habitable system will strive to make the user as comfortable as possible, and itself as inconspicuous as possible. This will require an adequate coverage of the sub-language, as well as the ability to fill in information from context.

2.3 Self-Tutoring

As a user converses with an interface for the first few times, he undergoes a learning process that helps him to characterize the sub-language of the system. As this process continues, the user will subconsciously limit his interaction to what he thinks is the system's sub-language. The goal of a good interface is to make this learning process as fast and painless as possible.

Providing meaningful feedback on mistakes, gives the user the best chance to learn the sub-language. If the system simply states that it cannot understand the input, then the user is forced to hypothesize what the error was, and then test this hypothesis. Or worse yet, they may simply give up on the query, rather than trying to get the computer to understand it. But, if the system tries to explain why the input was not understood, the user can adjust his input accordingly.

2.4 Awareness of Ambiguity

In nearly all domains, the possibility of ambiguity can arise in a conversation. This can occur when the user asks a question, understanding it one way, but the system takes it another way and answers accordingly. For example (again from (Burton & Brown, 1979)):
"Was John believed to have been shot by Fred?"
The sentence can be understood as Fred shooting John, or Fred believing that John has been shot. Both of these interpretations are valid and without complex context analysis, the correct interpretation cannot be definitely chosen.

So it is important for the system to be explicit in its responses. A simple *"Yes"* response may lead to the user thinking that the system understands the question in a completely different way than it actually does. Explicitly stating its beliefs by responding with :
"Yes, John was believed to have shot Fred.", or *"Yes, Fred believed that John was shot."*
would be more clear and helpful.

2.5 Convenience Features

Some problems with natural language interfaces have nothing to do with their lexical coverage or their contextual knowledge. Often, the most irritating aspects of a conversation are input oriented. Hendrix gives some ideas about how to alleviate some of this frustration in (Hendrix, 1977). He uses paraphrases, synonyms, spelling correction, and access to the host language to make the user more powerful and comfortable.

Paraphrases and *synonyms* allow the user to customize his interface. If he is used to referring to the *"Department of Computer Science"* as *"CS"*, he can add to the system the word *"CS"* and tell the system that it is a synonym for the *"Department of Computer Science"*. Similarly, if the system accepts :

"List the salary of each member of the Department of Computer Science."
but the user doesn't want to type that for each department, he can define a paraphrase like:
"Salary CS."
which will be interpreted by the system as meaning the same as the original longer sentence. The paraphrase should do more than substitute one set of words for another. It should generalize the types of words used, and allow further use of the paraphrase. So the user might enter similar paraphrases that can be interpreted, like:
"Age EE."
which will print the age of each member of the Electrical Engineering Department. These features help make the user feel more comfortable by allowing him to define his own environment. They also increase the user's efficiency by allowing them to define short-cuts.

Allowing short cuts and synonyms may also allow the system to function while accepting a smaller sub-language. The idea is to provide some means with which the information can be obtained, and rely on the paraphrases and synonyms to allow the user to tailor the input language. For example, if the system doesn't understand the user's preferred phrasing:
"The red blocks are supported by what?"
but does understand :
"What supports the red blocks?"
a paraphrase allows the system to understand his way of asking the question.

Spelling correction is also an important tool. If a user is not a terribly good typist, she will make many typing errors in her input. And few things are more irritating than having to retype a long, or even not so long, sentence just because of a typing error. To alleviate this, spelling correction should be used to spot misspelled words, and then replace them with their corrected form. Such a simple procedure can add much power and utility to an interface. An important addition should be made though. The replacement should be explicitly stated, to avoid user confusion if an unexpected replacement occurs.

A final convenience feature is access to the host language. Frequent users of a system will want to do other things outside the interface, like loading files, reading mail, or checking the time. Some common activities, such as loading files, should be incorporated into the system's sub-language explicitly. But predicting every necessity is impossible, so a simple command that will put the user temporarily into the host language can help to make the interface more comfortable.

In summary, a natural language interface needs to be efficient, otherwise the long response delays may prove detrimental to the user's performance. It should also understand a habitable sub-language that will allow the user to comfortably converse in the domain. The system should provide feedback upon not understanding a sentence to help the user to learn the boundaries of the sub-language. The system should be careful to avoid misunderstandings due to ambiguity by explicitly stating its responses so as to reflect its understanding. And finally, some "user friendly" conveniences like paraphrases and spelling correction will give the user more power in the interface, and allow him to customize the interface to accommodate his sub-language.

3 ATT OVERVIEW

The Augmented Transition Tree (ATT) was a Master's project by Bruce Martin (Martin, 1985). The premise was that a natural language understanding system could be divided into domain-dependent and domain-independent parts. His thinking was that this could lead to the development of a general *grammar skeleton* that would parse simple English commands and questions, and call the domain-dependent lexicon functions to respond to the user.

ATT is a specialization of Woods' Augmented Transition Network (ATN) (Woods, 1970). The code was developed from a simple example program in (Winston & Horn, 1984), and was tested in a blocks-world domain. The idea of an ATN is similar to that of a finite state machine. The nodes in an ATN correspond to deeper parsing ATN's, and the arcs correspond to words parsed by the network. The *augmentation* comes with the addition of *tests* on the arcs that conditionalize their use. There is also the ability to build structures during the parse of a sentence to represent the ATN's interpretation of it.

The ATT specializes the ATN approach by not allowing branches to remerge, hence the "Tree" designation. ATT also does no explicit structure building while parsing. The ends of branches are analogous to end states in a finite state machine. Figure 3.1 shows a top level graph of the ATT used as the grammar in the system and is explained in the next section which summarizes the ATT's operation.

3.1 Operation of the ATT

The main feature of ATT is the separation of the domain-dependent lexicon from the domain-independent grammar and parsing mechanisms. The general skeleton consists of a grammar core that parses some simple questions, assertions, and commands. It also consists of an interpreter (or in the case of QATT a compiler, also developed from an example in (Winston & Horn, 1984)), and means of maintaining the lexicon, elliptical references, pronouns, and other features. The lexicon contains all of the information needed for the specific application, such as the verb and noun definitions, and the response generating functions.

Understanding a sentence starts with using the ATT to parse it. In figure 3.1 we see that the highest level in the grammar is called `Interface`. Transitions are based on characteristics of the input sentence. If the input sentence consists only of the word *"Lisp"*, then the lisp arc is taken to handle a LISP interaction. If the input contains a *"?"*, then the question arc is taken. Otherwise, the command arc is taken. This type of testing and transition making continues until the input sentence is consumed and the appropriate syntactic information is computed.

At the end of a successful parse, the interpreter should find itself at the end of a branch in the tree. There it should find a call to the Respond function that will determine what domain-dependent response function to call, using the syntactic information gathered in the parse. The chosen function will then be called to respond to the user. This allows all of the idiosyncratic information to be taken out of the syntactic knowledge of the grammar, and into the lexicon. Figure 3.2 shows how the ATT divides the system into domain-dependent and domain-independent parts. So, in theory, only the domain-dependent parts, the lexicon and the response functions, need to be supplied to the ATT in order to prepare it for natural language understanding in the new domain.

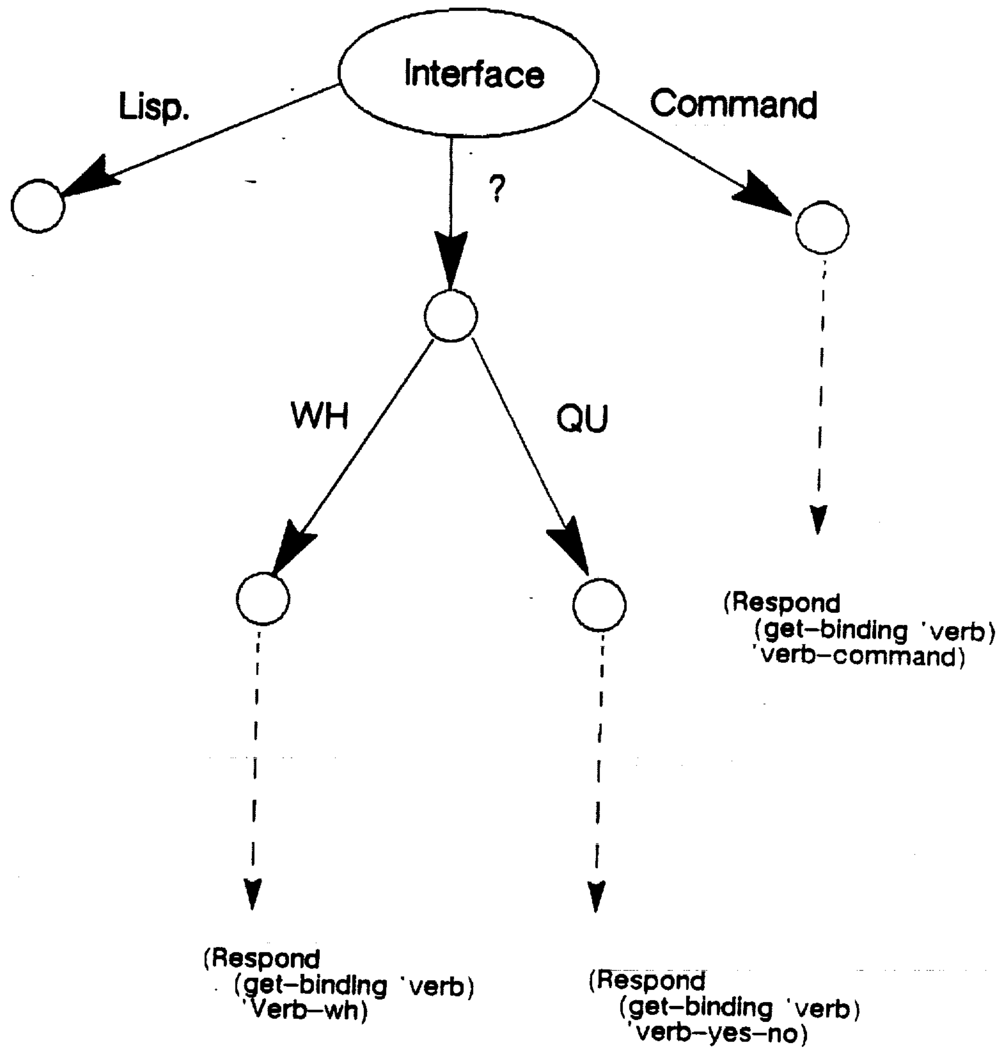

Interface
Lisp.
?
Command
WH
QU
(Respond (get-binding 'verb) 'verb-command)
(Respond (get-binding 'verb) 'Verb-wh)
(Respond (get-binding 'verb) 'verb-yes-no)

Figure 3.1: Top level view of QATT grammar

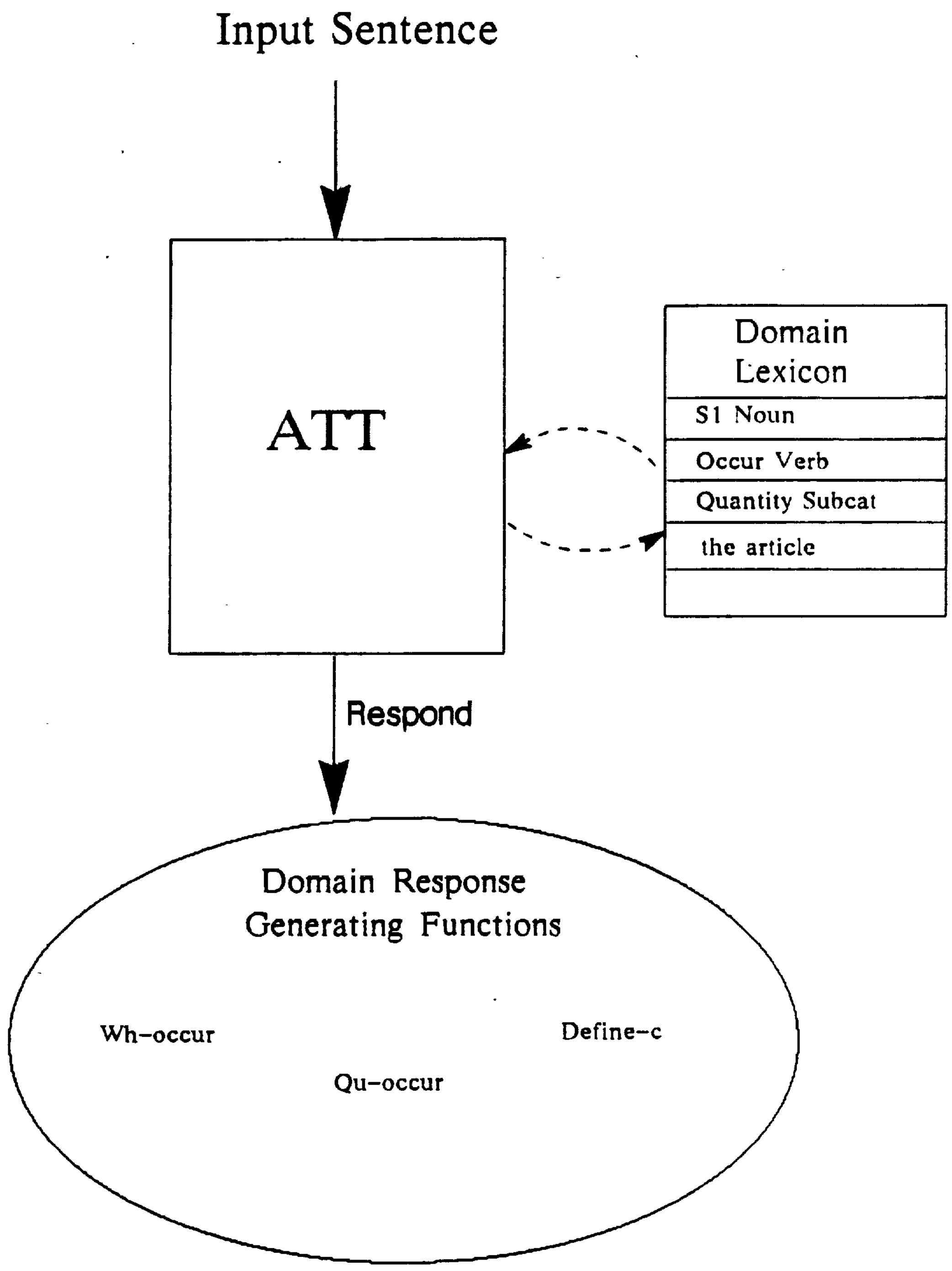


Figure 3.2: The separation of domain-dependent and domain-independent parts in ATT

1. *"What are the red blocks?"*
2. *"The green pyramid is huge."*
3. *"Put the blue brick in the large box."*
4. *"Is the orange pyramid on the table?"*

Figure 3.3: Some sentences understood by ATT

3.2 Grammar

The general grammar provided by ATT was tested using a blocks-world domain. So it is heavily biased toward simple questions and commands. It is also a *syntactic* grammar, rather than *semantic*. That means that the parsing is done on a purely syntactic basis. This make sense since it was meant to cover varied application domains.

Some examples of sentences understood by ATT are shown in figure 3.3. These sentences, and most of those understood by the ATT's language, refer to either a specific object, or a class of objects, characterized by their (previously defined) properties. Later we will see how this caused problems in the QATT implementation.

The grammar is linear in nature; it can only parse left to right and cannot look ahead or back easily. This causes problems with some nested references and discontinuous morphemes, but this is of little harm in QATT. The tree structure also makes arbitrary length conjunctions or disjunctions difficult. And there are also various other quirks in the grammar that posed problems when applying it to the QPE domain. These are described in Chapter 5.

3.3 Representation

ATT's grammar uses an *extensional* representation scheme, as opposed to *intensional*. For example, the phrase *"the red blocks"* would be interpreted in an intensional scheme as *all things that are blocks and red*. An extensional scheme instead provides a list of the red blocks. So rather than coming up with a definition of the meaning of the phrase, ATT finds the actual objects in the domain that fit the meaning of the phrase.

The questions and commands accepted by ATT are represented as function calls to appropriate domain-dependent response functions. These functions provide the responses to the users. The *entities* in the domain are represented by symbols with various properties and values attached to them. So, *"the red blocks"* will be represented by a list of all symbols that have the two properties of being red and being a block. The question *"What are the red blocks?"* will be represented by a call to the domain function that handles *WH* questions for the main verb *is*, and it will be given the list of red block symbols as an argument.

3.4 Response Functions

The domain-dependent response functions are the expressive parts of the system. The set of these functions is the true definition of exactly what the scope of the interaction with the user can be. The functions are keyed to verbs; that is, they are used to respond to sentences with their verb as the main verb. The functions for each verb and their arguments are included in the lexicon as part of the verb's definition. Upon successful parsing of the input sentence, the grammar having determined the main verb and sentence type, the **Respond** function gathers the appropriate arguments for the response function, and then calls that function with its required arguments.

As an example, let's take the question *"What are the blocks?"* Since the main verb is *is*, and the sentence is a *Wh-* question, the function called is `WH-IS`. `WH-IS` takes arguments like `SUBJECT`, `OBJECT`, `PREPOSITION`, `OBJECT-OF-PREPOSITION`, and `WH` (for the type of wh-question ; what, how, who, etc). **Respond** must then gather all of these arguments, of which only the `SUBJECT` is bound in our example. Then `WH-IS` is called with its arguments, and by seeing that `WH` is *what*, that only the `SUBJECT` is bound, and that the `SUBJECT` is a list of nouns, it will proceed to print these nouns on the screen. The response would have done something completely different if the `WH` had been *where* instead. So these domain functions are usually large conditional structures, and they should be carefully engineered to inform users of their capabilities.

Once the grammar is capable of parsing the sentence and providing the correct verb function with the appropriate arguments, it is up to this function to decipher the actions to be taken based on the arguments given. So the response functions require the most programming effort and attention to detail.

Now that we have a basic understanding of the natural language understanding skeleton that was the foundation for `QATT`, we next examine the `QATT` implementation in detail and the problems encountered using the `ATT`.

4 QATT OVERVIEW

This chapter examines the features of the QATT implementation; specifically, its representation of QPE objects, the verbs used for the sub-language, and the implementation of its major features.

4.1 Query Space

In order to define a sub-language for QATT, it was necessary to first characterize a *query space* for QPE; that is, a set of question types that could easily be answered by a QPE envisionment. Once this query space is defined, a set of verbs can be selected to cover it.

The first type of interaction with QATT concerns simple information about the entities in the lexicon and the envisionment. For example, "*What are the quantities?*" and "*Display the verbs?*". This type of interaction typically produces a list of the specified objects.

Another type of interaction with QPE asks about the qualitative change of quantities. For instance, "*When is the amount of water in G increasing?*" or the more general "*How can the amount of water in G change?*" For these cases, a list of states should be produced for each of the qualitative changes in the quantity. Along with qualitative changes go questions of influence, such as "*What influences the amount of water in G?*" Here the response should be a list of processes or views and their effect on the quantity.

For processes, the questions will center around in what states they are active. "*When is PI2 active?*" should be responded to with a list of the states in which the process is active. Asking what quantities a process can affect, as in "*What does PI2 influence?*" should produce a list of the quantities that the process causes to increase or decrease.

Questions about limit hypotheses focus on the conditions under which they may occur. "*When can LH2 occur?*" would give a list of the states that satisfy the limit hypothesis' starting environments. "*What can LH2 lead to?*" would respond with a list of possible end states of the limit hypothesis.

Qualitative states of a system can contain a great deal of information, and so will be the subject of many queries. Most queries concern the properties of a state, like "*What is the duration of s3?*" and "*Is S3 an end state?*". There are also questions about transitions to and from other states; "*What states have no transitions out?*", "*Does S3 have transitions in?*", "*Can S3 occur from S2?*".

Finally, in order to examine large envisionments, it is often necessary to isolate the subset of states that conform to certain specifications. This smaller set may then be examined more carefully. To do this, some sort of filtering mechanism is needed. So a command like "*Define FOO as the set of states where the amount of water in G is increasing and the flow of water from F to G is active.*" should force FOO to refer to the set of states which have both of these properties.

That is the basic query space covered by QATT. It concentrates on the specification and manipulation of sets of states, and provides reasonable insight into the envisionment. Now we will see how this query space is accommodated by the lexicon.

4.2 Representation of QPE Data

To provide a lexicon which the ATT grammar can use in parsing, we must add words relevant to the QPE domain. This includes verbs that might be used in the discourse, nouns to represent the QPE objects, and subcategories that provide a means of referring to nouns by their type.

4.2.1 Verbs

Two kinds of verbs are used by QATT. First there are domain-independent verbs. *"Be"* is used extensively for all of the *"What is..."* questions. *"Display"* is used in commands to show the elements of some data type. *"Load"* and *"save"* are used for file manipulation and with paraphrases or synonyms. *"Set"* and *"Reset"* are for toggling QATT system flags. *"Use"* is used for synonym creation and for window manipulation. Finally, *"Erase"* is used to remove elements from the QATT lexicon.

Several verbs were added for the QPE domain. *"Define"* is used for the state filtering commands. *"Change"*, *"influence"*, *"increase"*, and *"decrease"* are all used for questions about the changes of quantities and the effects of processes. *"Occur"*, *"hold"*, and *"happen"* are for questions about limit hypotheses. And finally *"lead"* is used for questions about what states are attainable from other states and the ending states of transitions.

4.2.2 Nouns

The nouns used in QATT generally parallel the objects in the QPE domain. Each noun has several properties, and can also be part of a subcategory. For example, the noun *"PI0"* is a member of the PROCESS subcategory. Additionally, each noun has two printing routines that specify how it is to be displayed to the user for different levels of detail.

The first nouns are the QATT *flags* which allow the user to control global parameters. The only flags now implemented are for controlling the level of detail in the output of responses. Each flag has associated with it a :VAL field that holds its current value.

The properties of the QATT nouns corresponding to QPE objects are extracted from the envisionment. The extracted properties of quantities include the processes and views which directly and indirectly influence them. The states in which the quantity is increasing, decreasing, and constant are also computed. For states, its status, duration, environments, and active process and view instances, along with transitions to and from it are extracted.

Quantities are represented in the lexicon by nouns. For QATT to have a single symbol to identify with a given quantity, each is given a name (i.e. Q0, Q1, ...). To provide a means to match a quantity with a phrase like *"...the amount of water in G..."*, an :INFO field is used to hold the key words found in the quantity object, such as (amount-of water liquid G).

Nouns for states are given the properties status, duration, active-pis, active-vis, assumptions, and lists of transitions to and from the state. To keep a pointer to the QPE object, there is a :QPE-SIT field. A state may also be an end state (no transitions out), an eden state (no transitions in), or an isolated state (no transitions in or out). These properties are added to the state noun as appropriate.

Limit hypothesis nouns include the start-envs and end-envs which point to the environments that the limit hypothesis can occur in and lead to respectively. Process and view nouns only have the :INFO field that is used much like that in the quantity nouns to match processes with phrases.

4.2.3 Subcategories

Each type of noun must have an associated *subcategory*. These are used by QATT to identify groups of nouns based on their properties. The QATT subcategories are *quantity*, *state*, *transition*, *environment*, *process*, *view*, and *set*. Subcategories are also used to refer to the fields of nouns, like *status*, *influencer*, and *start-env*. Several general properties of nouns are also encoded as

```
8 ?--what is the status of s3?
<PARSED>
STATUS of S3 is R-COMPLETE.

9 ?--duration?
DURATION of S3 is INTERVAL.

10 ?--s2?
DURATION of S2 is INTERVAL.
```

Figure 4.1: Example of the elliptical capabilities of QATT.

subcategories, such as *end* for states and *empty* for sets. The information for each subcategory must also be duplicated for its plural form.

Stored with each subcategory is the list of nouns having it as a property. This is a feature of the ATT. So, for example, the subcategory QUANTITY, contains a list of all of the quantity nouns. This listing of members of subcategories is necessary since the ATT uses an extensional representation.

4.3 QATT Features

This section covers the implementation and limits of the major features of QATT. It begins by examining those features that are domain-independent, such as paraphrases and ellipsis. Then it examines those that are particular to the QPE domain.

4.3.1 Domain-Independent Features

Most of the domain-independent features deal with the context of discourse. These have been implemented in a general manner so as to be useful in any ATT application.

Ellipsis was partially implemented in ATT. The mechanics for maintaining a copy of the parsing of the last sentence was in place, but it made many errors. ATT would try to place the elliptical reference into one of two noun slots; subject or object. The QATT ellipsis implementation added the possibility of the noun occupying the slot for the object of preposition. It also checks to see that the slot was actually used in the previous context, unlike ATT. This allowed dialogs like that shown in figure 4.1.

The ATT context mechanism also allowed limited *pronominalization*. This is accomplished by maintaining a `*last-noun*` variable, and substituting its value for the pronoun, as in figure 4.2. This was left unchanged for QATT.

Access to the host language was also partially implemented by ATT through the acceptance of the simple sentence *"Lisp."* This command awaits input, and then evaluates that input as a lisp expression. QATT added the ability to load and save files directly from the grammar. Having such common actions available through the grammar may increase the user's confidence in the interface. But it is necessary to give the user raw access to the host, since it is impossible to provide for all types of desired interaction. Figure 4.3 illustrates.

```
26 ?--what is the amount-of water in G?
<PARSED>
  AMOUNT-OF(C-S(WATER LIQUID G))

27 ?--what influences it?
<PARSED>
AMOUNT-OF(C-S(WATER LIQUID G)) is influenced indirectly by
(CONTAINED-STUFF(C-S(WATER LIQUID G)))
```

Figure 4.2: Example of the pronominalization capabilities of QATT.

```
16 ?--load "/u/white/new-att/qpe-att/string.lisp"!
<PARSED>
;;; Loading source file "/u/white/new-att/qpe-att/string.lisp"

17 ?--lisp.

-> (* 3.14159 3)
9.42477
```

Figure 4.3: Example of the host language access capabilities of QATT.

```
52 ?--does sO have transitons?
<PARSED>
No transitions in to SO.
No transitions out of SO.

 (Replaced TRANSITONS with TRANSITIONS)

33 ?--show Foo.

 I am stuck on the word SHOW.

 Do you have a replacement word? (word or <enter to fail>) :display

 Got it, DISPLAY for SHOW, thanks.

Should I consider SHOW a synonym for DISPLAY? (Y or N) y
Trying to add SHOW as a synonym for DISPLAY
Adding synonym SHOW for (DISPLAY)
<PARSED>
      .
      .
      .
```

Figure 4.4: Example of the spelling correction capabilities of QATT.

Spelling correction was added in QATT. It works as follows. All words in the lexicon are encoded in a correction table using the Soundex algorithm (Knuth, 1973). When a sentence fails to parse, QATT looks for any unknown words. For each unknown word, its soundex code is computed and used to fetch all words with the same code, which constitutes the words in the lexicon the unknown word is a possible misspelling of. The sentence is re-parsed with each candidate in turn until one of them is understood. If no acceptable substitution is found, the user is prompted for one. Then she is asked if her replacement word is a synonym for the unknown word. If so, then that synonym is added. This allows the interface to learn new terms in the domain without explicit synonym creation by the user. (See figure 4.4)

The synonym capability was also added for QATT. It allows the interface to learn new terms in the sub-language and to provide short cuts for the user. It is implemented with a table that matches the synonym with its associated list of words. When a sentence is input, it is first searched for known synonyms, and these are replaced by their associated words. For efficiency, only one-word synonyms are allowed, since searching the sentence for arbitrarily long sequences of words would be too time consuming. Synonyms may be defined from unrecognized symbols, as above, or by an explicit "*Synonym.*" call, or in the grammar with the "*Use*" command. Examples are shown in figure 4.5.

```
27 ?--synonym.
Words      :amount-of-in of water in f
Synonym    :amt-g
Adding synonym AMT-G for (AMOUNT-OF-IN OF WATER IN F)

28 ?--how can amt-g change?
<PARSED>
AMOUNT-OF-IN(WATER LIQUID F) is increasing in (S3)
AMOUNT-OF-IN(WATER LIQUID F) is decreasing in (S5)
AMOUNT-OF-IN(WATER LIQUID F) is constant in (S0 S1 S2 S4)

29 ?--use amt-of for amount-of.
<PARSED>
Adding synonym AMT-OF for (AMOUNT-OF)

30 ?--what is the amt-of water in f?
<PARSED>
  AMOUNT-OF(C-S(WATER LIQUID F))
```

Figure 4.5: Example of the synonym capabilities of QATT.

Paraphrases were also implemented for QATT. These provide the ability for users to customize their environment by providing alternative ways to say something. Figure 4.6 shows the use of a paraphrase. The example sentence is parsed, and then common words are found in the paraphrase. These words are then generalized to their lexicon classification and the part of the sentence they represented, so the paraphrase can accept a larger variety of paraphrases. In the example, "*duration*" was generalized to any sequence of subcategories, and "*s3*" to any object of preposition. Then a segment of an ATT branch that will parse the paraphrase and interpret it as the example sentence is created and incorporated into a list of similar segments for other paraphrases. A short description of this segment is displayed to the user showing how the paraphrase was interpreted. These paraphrases may also be saved and loaded to maintain a user's environment across sessions.

4.3.2 Domain-Dependent Features

These features are either specific to the QPE domain or are implemented in the domain-dependent response functions, and hence not a part of the ATT skeleton.

Feedback to the user is divided into two parts. An ATT domain-independent part tells the user, upon not understanding a sentence, how much of the sentence the interface did understand, with the idea that the rest is incorrect. This will help users identify exactly where the error may be, rather than just telling them that the sentence is wrong. The domain-dependent part of feedback comes in the response functions. Each response function contains branching conditionals that key on values of parameters. If a combination of values is not accounted for, but the grammar parsed the sentence, then the response function should explain to the user why a response is not

```
19 ?--paraphrase.
Sentence   :what is the duration of s3?
Paraphrase :duration s3?
<PARSED>

 require generalized SUBCATS (for DURATION)
 require generalized OBJOP (for S3)
 require ?

20 ?--duration s3?
<PARSED>
DURATION of S3 is INTERVAL.

21 ?--duration and status s2?
<PARSED>
DURATION of S2 is INTERVAL.
STATUS of S2 is R-COMPLETE.
```

Figure 4.6: Example of the paraphrase capabilities of QATT.

available. This explanation often requires printing some background information about the domain or implementation status.

Feedback provides one of the tutorial abilities of the interface. QATT's other main tutorial feature is its access to the commands, nouns, quantities, and any other noun or subcategory. This allows even a novice to quickly get a rough idea of the capabilities of the system. Also, the use of system variables to control the level of output detail can also help the user to understand the domain more easily. Figure 4.7 demonstrates some of QATT's feedback and tutorial capabilities.

Ambiguity handling also must be done in the response functions. This is handled by making responses to the user explicit as to their meaning and the interfaces understanding of the question. QATT should never respond with a *"Yes."*, but should always qualify its answer, like *"Yes, S3 is an end state."* Further, when a response function makes an assumption in the case of anaphoric deletion, that assumption should be made clear. For example, *"What can lh3 lead to?"* is assumed to refer to the states that lh3 leads to. The response makes this clear: *"LH3 leads to the following states..."*

QATT makes available the use of different streams for output. This allows the user to choose where the system's responses are to be displayed. This was added to take advantage of Symbolics' Scroll Windows for viewing detailed output, but may also be used for other purposes, such as appending output to a file. To implement this, the *"Use"* verb was extended and window nouns were added with properties of LEFT, RIGHT, and BOTTOM. By selecting distinct windows for different responses, it was also possible to compare two responses side by side. Similar nouns could also be added for output to files.

By far the most complicated feature of QATT is the state filtering feature. This allows the user to define sets of states that conform to certain characteristics. This new set may then be examined

```
12 ?--what quantities does lh3 lead to?
<PARSED>
I can only answer questions about states or envs for LEAD.

13 ?--what are the processes?
<PARSED>
  LIQUID-FLOW(WATER G F P1)
  LIQUID-FLOW(WATER F G P1)

14 ?--what are the commands?
<PARSED>
  SAVE DEFINE USE LET CALL SET RESET

18 ?--what is A?
I don't understand.
I got as far as :(WHAT IS)
```

Figure 4.7: Example of the tutorial capabilities of QATT.

more closely as desired. The problem with the implementation of this feature came from the ATT grammar. It was simply not capable of handling such a complex command.

The first step in the filter implementation was getting the grammar to accept phrases that corresponded to quantities, processes, and limit hypotheses. To do this, I added to the grammar semantic extensions of quantity, process, and limit-hypothesis phrases. These were, unlike the rest of the grammar, semantically defined, not syntactically. To match a quantity, for instance, I required that words appearing in the quantity description in QPE be used in the phrase. For example, the QPE quantity AMOUNT-OF(WATER,LIQUID,G) could be matched by the phrase *"the amount of water in G"*, since the words *"amount"*, *"water"*, and *"G"* all appear in the quantity description.

To do this in a manner that would extend across QPE domain models, the use of prepositions to explicitly refer to properties of nouns was not feasible. Instead, I used the above textual matching method. This method, though, was not sufficient in some cases. For example, the phrase *"flow of water from F to G"* could match the processes LIQUID-FLOW(WATER,G,F,PIPE1) and LIQUID-FLOW(WATER,F,G,PIPE1). The prepositions must be used to help disambiguate this phrase. To do this in a QPE-domain-independent manner, each QPE domain model must define, for each type of process or quantity that may cause such ambiguity, the prepositions that are keyed to argument positions. For example, for LIQUID-FLOW the preposition *"(of)"* may be used to refer to the first argument, *"(from, between)"* for the second, *"(to, between)"* the third, and *"(in, through)"* for the fourth. With this information, the phrases in figure 4.8 can be disambiguated. If the phrase cannot be disambiguated, the user is asked to choose from a list of the possible matches.

But the filtering commands still need to parse specifications for these objects, like *"increasing"* for quantities, or *"active"* for processes. And then the name of the set has be parsed and stored. Then, once the command can actually be parsed, the response function is responsible for finding

- *"flow of water from F to G"* ⇒
  `LIQUID-FLOW(WATER,F,G,?path)`

- *"flow of water between F and G through PIPE1"* ⇒
  `LIQUID-FLOW(WATER,F,G,PIPE1)` ∪ `LIQUID-FLOW(WATER,G,F,PIPE1)`

- *"flow of water to G"* ⇒
  `LIQUID-FLOW(WATER,?source,G,?path)`

Figure 4.8: Phrases mapped to disambiguated objects.

```
31 ?--define FOO as the set of states with the amount-of water in F increasing
and water flowing from G to F.
<PARSED>
 Set: FOO
 Elements: (S3)
 Size: 1
Set FOO

32 ?-- Show FOO.
  S3
```

Figure 4.9: Example of the state filtering command.

the elements of the set that correspond to the specifications, and then including the set in the lexicon. An example of this interaction is shown in figure 4.9.

5 ANALYSIS

The previous chapter summarized the features of QATT and the major additions made to the ATT. This chapter evaluates the interface along the requirements of a good natural language interface and compares its capabilities with those of SOHPIE. But, of course, the real test of the system will come when people try to use it routinely. Finally, some general comments are made about the suitability of using an ATT as a foundation for constructing a natural language interface, based on these comparisons and my experience building QATT.

5.1 QATT as a Natural Language Interface

To judge the results of the QATT interface, I will evaluate its habitability, efficiency, handling of context, self-tutoring abilities, awareness of ambiguity, and its convenience features.

5.1.1 Habitability

SOPHIE's strongest feature is habitability. This would seem necessary, since the interface is used in an educational environment with novice users. And since, due to time constraints, QATT was aimed at the frequent users of QPE, habitability suffered somewhat.

In the limited tests of QATT, it seems to cover the query space well when the user knows what types of questions it understands, as there are several ways to ask these questions that are accepted. But in some cases, especially the set filtering command, the format accepted is quite rigid. The lower coverage of QATT is due to two factors. First is the syntactic nature of the grammar. A semantic grammar is more flexible, and can be made to parse more varied sentence formats, because it looks for semantic components, not syntactic constituents. But, since the ATT grammar is designed to be domain-independent, it clearly cannot be semantically oriented. The QATT grammar, being syntactic, needs to take explicit measures to accept syntactic orderings for all sentence types, blind to their semantic content. A semantic grammar gains leverage by knowing what types of sentence formats make sense for specific kinds of phrases. For example, a semantic specification of a phrase for *measurement* in an electronics domain can expect to see a measurable quantity, followed by a preposition, and then some part or place where the measurement is to be taken (Burton & Brown, 1979). But a syntactic grammar will have no such leverage, and must parse the phrase with no expectations about its content. To do this in a domain-independent manner, all grammatical word orderings would need to be represented in the syntactic grammar. So to have the wide coverage of SOPHIE, QATT would have to accept virtually any grammatical syntactic sequence. And this is, of course, not feasible for as diverse a language as English.

The second factor for QATT's lower coverage is limited development time. Much of the coverage provided by the interface is implemented in the domain-dependent response functions. If the sentence is parsed by the grammar, it is up to the response function to provide the response. With limited time, some queries that are parsed by the grammar were not implemented in the response functions. So this is not a weakness of the QATT system, but rather a time constraint problem. With more time, the response functions could be extended to handle those parsed queries.

5.1.2 Efficiency

To gain coverage in the ATT grammar, many additions were made in the form of optional branches. The prime example is the various ways to phrase *"[To] what [states] does LHS lead [to]?"*, where the

[]'s denote optional words. Allowing for such varied parses increases the coverage of the grammar, but necessarily decreases the efficiency of the interface, since it must repeatedly follow the wrong branch past optional nodes. But this is the price to be paid for wide coverage. QATT cut optional branches to a minimum.

In the blocks world, ATT's typical response time was around 4 seconds. QATT responds to short (5 to 8 word) queries in an average of 3 seconds when run on a Symbolics or an IBM-RT. In the worst case, an unparsed set filtering command, responses are around 6 seconds. These times could be improved by pruning the ATT grammar to better fit the QPE query space, but this would be violating the goal of the project to develop the interface from a preexisting grammar.

5.1.3 Context

The revisions to the ATT context mechanisms for QATT have put it nearly at the level of SOPHIE. QATT has a record of the "last-noun" used in the context. This last-noun is then used as the referent of subsequent pronouns. So in the sentence *"What influences it?"*, *"it"* is assumed to refer to the value of `*last-noun*`.

SOPHIE is able to handle context references like (Burton & Brown, 1979): *"Set the voltage control to .8?"*, *"What is the current thru R9?"*, *"What is it with it set to .9?"*. In the third sentence, it is able to determine that the first *"it"* refers to *"current"*, and the second refers to *"voltage control"*. The difference comes from SOPHIE's use of its semantic grammar to predict missing or pronominalized constituents. QATT can only hypothesize from syntactic information.

QATT's ellipsis handling is nearly as powerful as SOPHIE's. QATT assumes the elliptical references are always to nouns. SOPHIE allows elliptical references to prepositional phrases, as in the sequence *"What is the base current of Q3?"*, *"Thru the emitter?"*. The QATT ellipsis mechanism could be configured to handle this type of reference, but as of yet it was not found necessary.

SOPHIE uses its semantic grammar to make assumptions as to the content of missing constituents in anaphoric deletions. By noting the possible semantic portion missing in a sentence, it is able to make intelligent guesses as to its referent. For example, if there were a semantic rule for *"lead to"* questions about limit-hypotheses, it might look like:

`What states does <limit-hypothesis> lead to?`

If the input sentence is *"What does LH3 lead to?"*, the grammar can assume that the missing constituent is *"states"*. In QATT, the guessing is left up to the response functions. The sentence must still be parsed by the grammar, and if a constituent is unbound, then the response function can simply make an assumption about what it is. This assumption is "hard-coded" into the function, and has no notion of context.

QATT could also use a notion of "last-state" for context. For example, the sequence *"What state does LH3 lead to?"*, *"Can S3 occur from there?"* could be disambiguated with this information. This was not implemented mainly due to the time constraints, but also due to my reluctance to make domain-dependent alterations to the ATT grammar.

5.1.4 Self-Tutoring

Being an educational tool, SOPHIE is extremely strong in its self-tutorial abilities. It not only tells the student why it could not understand a sentence, but it is also able to explain to the student why a sentence might not make sense in the domain. This ability is partially implemented in QATT and is incorporated in the response functions, the equivalent of SOPHIE's *specialists*. QATT's tutorial abilities could be improved by simply expanding the explanations of misunderstandings in these response functions. It is not shown in (Burton & Brown, 1979) that SOPHIE provides feedback when a sentence is not successfully parsed, such as the QATT's *"I got as far as ..."*. LIFER provides this and goes a step further by suggesting possible categories of words that might fit into the sentence to make it understandable.

5.1.5 Ambiguity Awareness

Like SOPHIE, QATT responses have been carefully worded so as to make any implicit assumptions clear to the user. If the context mechanisms for QATT were extended to include the "last-state" notion above, then there could possibly be more chance of ambiguous references. But as long as responses are explicit about their suppositions, this should not cause a problem.

5.1.6 Convenience Features

QATT's strong suit is the convenience features that it provides to the user. These are the synonyms, paraphrases, access to LISP, and spelling correction. This again is mainly due to the focus of the interface on the the frequent QPE users. Since SOPHIE is not as strong as LIFER in this area, I will use LIFER as the benchmark.

Access to the host language is simple enough. LIFER provides a command to access LISP, but it does not allow for host interaction through the grammar. QATT presently only accepts loading and saving of files in the grammar, but these commands were very easy to incorporate, and others could be added just as easily. Executing such commands inside the grammar makes the interface more helpful to the user.

The QATT synonym facility appears to work as well as LIFER's. Its only problem is the inability to handle multi-word synonyms, such as *"level in C"* for *"the level of water in can C"*. This was an efficiency consideration, since searching for arbitrarily long strings in the sentence is so time consuming. Having synonym definition possible from the grammar, which is done in LIFER as well, further aids habitability.

The paraphrase ability of QATT is limited when compared to LIFER. LIFER is able to not only paraphrase whole sentences, but it can also find hidden paraphrases in these sentences. For example, using *"Salary for CS Faculty"* for *"Print the salary of everyone in the Computer Science Department."*, LIFER would build a large paraphrase for the short sentence, and also build a sub-paraphrase that matches *"CS Faculty"* with *"everyone in the Computer Science Department."*. QATT can only paraphrase at the sentence level, but does generalize enough to make these paraphrases useful in other contexts, as shown in figure 4.6.

LIFER handles paraphrase definition exclusively inside its grammar. So a definition would be something like *"Use [paraphrase] for [sentence]."*. QATT was not implemented this way because of the complexity of re-configuring the grammar. Instead, the simpler "fill-in-the-blank" approach was used. This may tend to decrease flexibility of the interface, but the time needed to accommodate definitions in the grammar could not be justified.

The Soundex algorithm used for QATT's spelling correction does not seem appropriate for typing errors. The algorithm was developed for airline reservation systems that experienced problems with misspelled names, not because of typing errors, but due to letters sounding the same over the phone. So Soundex maps strings of letters to strings that may sound the same. LIFER and SOPHIE both use a spelling correction algorithm borrowed from INTERLISP. This method looks for transposed letters and double strikes, making it more suitable for finding typing errors.

QATT does however come back to the user if no successful substitutions were found in the spelling hash table. It also makes synonym definition immediately available to the user in that case. And like LIFER and SOPHIE, it does inform the user of any substitutions made.

One possible enhancement to the spelling correction would be to keep the system from substituting for words that are unknown, not because they are misspelled or new terms, but because they represent objects that don't exist in the domain. For example, in an envisionment with only 5 states, *"Display S19?"* will actually display S0, and then tell the user that the substitution was made. This determination as to what words are semantically valid but don't refer to anything, would need to be domain-dependent though, and was not implemented. An alternative solution would be to ask the user if a proposed substitution is acceptable before responding.

5.1.7 Summary

To get QATT as close to the level of SOPHIE as feasible would require approximately three months of refinement of the domain-dependent parts of the interface. The domain-independent parts, with the exception of the grammar, appear to be roughly as capable as those of SOPHIE. But for the grammar to achieve the coverage of SOPHIE, it would need to be extensively altered and tuned for the QPE domain. And in so doing, the efficiency of the interface could also be improved by trimming branches for the QPE query space. This would have violated the spirit of the project, which was to take a general grammar and build on it a QPE natural language interface. However, while QATT is not as robust as SOPHIE's interface, it appears to be reasonably useful.

5.2 Use of General Grammar Skeleton

Initially, using ATT as the basis for the QATT interface seemed to offer a fast means to development. And it also looked as if I could construct this natural language interface without any formal linguistic training. The results, though not perfect, are very promising.

The ATT grammar was implicitly biased toward the blocks world domain that it was tested on. Most notably, the prepositional phrases were assumed to be restrictive. For example, in *"The block in the box on the table..."*, the prepositional phrase *"in the box"* restricts the blocks considered, and *"on the table"* restricts the boxes considered. The extensional representation of ATT also demanded that these restrictions be explicitly recorded in the properties of each object. So in our example, the blocks that matched would need to have BOXn in their field :IN, where BOXn is a box with TABLE in its :ON field.

An important goal was to make QATT work with any QP domain model that QPE could simulate. So to handle prepositional phrases, a new type of non-restrictive prepositional phrase parse was devised that merely parses the prepositional phrase and returns its contents without regard to any restrictions. The restrictions are then worked out using the ordered prepositions that are supplied for each QPE domain model, as shown in section 1.3.

Every application is bound to face similar obstacles when trying to fit a "general" grammar to a specific application. This will be the case until the (unlikely to be soon) invention of a complete natural language understanding system.

Until then, to gain coverage beyond the scope of the original grammar, a linguistic novice is forced to manipulate the grammar to fit her needs. Inevitably, this will lead to many optional branches, as in QATT, or possibly even reduced coverage as the grammar is hacked at by the programmer.

However, once the query space is accepted by the grammar, the division of domain-dependent and domain-independent parts of the interface make implementation of reasonable responses easy. By planning ahead for required properties of nouns, and exactly which nouns will be implemented, the lexicon can be quickly developed and serve as the data base for all of the responses. Verbs can be added to expand the interface's coverage, and the detail of responses can be changed as the need arises. As time permits or as the need arises, the programmer can work on the response functions without touching the grammar or lexicon. With a few hours of work, a verb can be added to the system and all of its response functions can be debugged to provide adequate responses to the new sentences.

In a few months, I was able to develop a reasonable natural language interface using this general grammar skeleton, with minimal linguistic experience. The only real difficulty came in manipulating the grammar to achieve greater coverage of the query space, and in devising new methods of parsing when the general grammar failed to meet my needs. Most of the work done in the project, the increased coverage, the context knowledge, and the convenience features, were extensions of the general skeleton, and would transfer to other domains. Using the general skeleton made possible the development of the interface in a matter of months rather than years.

5.3 What is Missing

QATT remains wide open for enhancements. Many improvements would be simple, but could not be accomplished with my time constraints. One example is displaying the paraphrases and synonyms in an easy to read format. Increasing the coverage of the response functions would also be easy, and could increase the system's habitability. A more appropriate spelling correction algorithm, like the one in INTERLISP, would not be hard to implement. And if the system could respond to *"Help!"* with a short explanation of the interface's capabilities and commands, it might help the novice user.

Some enhancements would take more integration but would not be too difficult to implement. Linking QATT with ZGRAPH, a graphical display utility, could allow the user to point to states, as if to say *"That one."*. QPE could also be run from QATT. Providing a word completion capability that would find a known word and display it once enough characters have been entered to disambiguate it, could make the interface more habitable and reduce typing errors. And using internal interface routines to fetch envisionment data rather than explicitly copying it into the lexicon would make the domain-dependent part of QATT more modular and easily modified.

And there are more complicated, theoretical extensions that would require significant work. One promising extension would be the use of a text generation system that could provide english text from semantic descriptions derived in the response functions. Another would be to incorporate a "user model" that monitors a user's knowledge of the domain and of the interface, and adjusts responses accordingly. And extending ATT to build a syntactic structure of the input sentence could provide more information to the response functions, and allow the paraphrase utility to capture nested paraphrases as LIFER does.

REFERENCES

Burton, R. R and Brown, J. S. Toward a natural-language capability for computer-assited instruction. *Procedure for Instructional Systems Development*, 1979.

Forbus, K. Qualitative process theory. *Artificial Intelligence*, **24**:85–168, 1984.

Forbus, K. QPE: a study in assumption-based truth maintenance. *International Journal of Artificial Intelligence in Engineering*, 1988.

Hendrix, G. G. Lifer: a natural language interface facility. *SIGART Newsletter*, **61**, 1977.

Knuth, D. E. *The Art of Computer Understanding, second edition*, pages 391–392. Volume 3, Addison-Wesley, 1973.

Martin, B. The limitations of augmented transition tree interpreters as natural language interfaces. *Master's Thesis, University of Illinois*, 1985.

Miller, R. Response time in man-computer conversational transactions. *AFIPS Conference Proceedings*, 1968.

Watt, W. Habitability. *American Documentation*, **19**, 1968.

Winston and Horn. *LISP second edition*. Addison-Wesley, 1984.

Woods, W. Transition network grammars for natural language analysis. *CACM*, **13**, 1970.

A SAMPLE DIALOG WITH QATT

```
> (init-att)
Loading data...
;;; Loading source file "/u/white/new-att/qpe-att/q-init.lisp"
Initializing
 Adding verbs
 Adding QPE data
;;; Loading source file "/u/white/new-att/qpe-att/spec.lisp"

 Adding parts
;;; Loading source file "/u/white/new-att/qpe-att/ord-preps.lisp"

Initialized.
#P"/u/white/new-att/qpe-att/q-init.lisp"
> (start)

0 ?--what are the flags?
<PARSED>
    DETAIL     NIL
    SHOW-TYPE  NIL
    FORM       T

1 ?--what are the commands?
<PARSED>
  SAVE DEFINE USE LET CALL SET RESET

2 ?--what are the states?
<PARSED>
  S0  S1  S2  S3  S4  S5
3 ?--set detail.
<PARSED>
DETAIL set.

4 ?--what are the quantities?
<PARSED>
  VOLUME(C-S(WATER LIQUID G))
  VOLUME(C-S(WATER LIQUID F))
  TOP-HEIGHT(G)
  TOP-HEIGHT(F)
  TEMPERATURE(C-S(WATER LIQUID G))
  TEMPERATURE(C-S(WATER LIQUID F))
  TBOIL(WATER G)
          .
          .
          .
  AMOUNT-OF-IN(WATER LIQUID G)
  AMOUNT-OF-IN(WATER LIQUID F)
  AMOUNT-OF(C-S(WATER LIQUID G))
  AMOUNT-OF(C-S(WATER LIQUID F))

5 ?--what are the processes?
<PARSED>
  LIQUID-FLOW(WATER G F P1)
  LIQUID-FLOW(WATER F G P1)

6 ?--what is resettable?
<PARSED>
    DETAIL    (Provides detailed output) T
    SHOW-TYPE (Show the type of a datum) NIL
    FORM      (Form output)              T

7 ?--reset detail.
<PARSED>
DETAIL reset.

8 ?--what is the status of s3?
<PARSED>
STATUS of S3 is R-COMPLETE.

9 ?--duration?
DURATION of S3 is INTERVAL.

10 ?--s2?
DURATION of S2 is INTERVAL.

11 ?--what are the end states?
<PARSED>
  S4
12 ?--eden states?
  S3  S5
13 ?--what is A?
I don't understand.
I got as far as :(WHAT IS)

14 ?--what quantities does lh0 lead to?
<PARSED>
I can only answer questions about states or envs for LEAD.

15 ?--load "/u/white/new-att/qpe-att/strings.lisp"!
<PARSED>
I can't find /u/white/new-att/qpe-att/strings.lisp
```

```
16 ?--load "/u/white/new-att/qpe-att/string.lisp"!
<PARSED>
;;; Loading source file "/u/white/new-att/qpe-att/string.lisp"

17 ?--lisp.

-> (* 3.14159 3)
9.42477

18 ?--what is the duration of s3?
<PARSED>
DURATION of S3 is INTERVAL.

19 ?--paraphrase.
Sentence   :what is the duration of s3?
Paraphrase :duration s3?
<PARSED>
Foregoing response

 require generalized SUBCATS (for DURATION)
 require generalized OBJOP (for S3)
 require ?

20 ?--duration s3?
<PARSED>
DURATION of S3 is INTERVAL.

21 ?--duration and status s2?
<PARSED>
DURATION of S2 is INTERVAL.
STATUS of S2 is R-COMPLETE.

22 ?--s1?
DURATION of S1 is INTERVAL.
STATUS of S1 is R-COMPLETE.

23 ?--end-env lh0?
<PARSED>
LHO doesn't have END-ENV.

24 ?--end-envs lh0?
<PARSED>
END-ENVS of LHO is ENV-185.

25 ?--what is the amount-of water in G?
```

```
<PARSED>
  AMOUNT-OF(C-S(WATER LIQUID G))

26 ?--what influences it?
<PARSED>
AMOUNT-OF(C-S(WATER LIQUID G)) is influenced indirectly by  (CONTAINED-STUFF(C-S(WATER
LIQUID G)))

27 ?--synonym.
Words       :amount-of-in of water in f
Synonym     :amt-g
Adding synonym AMT-G for (AMOUNT-OF-IN OF WATER IN F)

28 ?--how can amt-g change?
<PARSED>
AMOUNT-OF-IN(WATER LIQUID F) is increasing in (S3)
AMOUNT-OF-IN(WATER LIQUID F) is decreasing in (S5)
AMOUNT-OF-IN(WATER LIQUID F) is constant in (SO S1 S2 S4)

29 ?--use amt-of for amount-of.
<PARSED>
Adding synonym AMT-OF for (AMOUNT-OF)

30 ?--what is the amt-of water in f?
<PARSED>
  AMOUNT-OF(C-S(WATER LIQUID F))

31 ?--define FOO as the set of states with the amount of water in F increasing and
water flowing from G to F.
<PARSED>

Please choose one by number to resolve ambiguity:
   1  AMOUNT-OF(C-S(WATER LIQUID F))
   2  AMOUNT-OF-IN(WATER LIQUID F)
   3: ALL

CHOICE:1
Merci
 Set: FOO
 Elements: (S3)
 Size: 1
Set FOO

32 ?--set detail.
<PARSED>
DETAIL set.
```

```
33 ?--show Foo.

 I am stuck on the word SHOW.

 Do you have a replacement word? (word or <enter to fail>) :display

 Got it, DISPLAY for SHOW, thanks.

Should I consider SHOW a synonym for DISPLAY? (Y or N) y
Trying to add SHOW as a synonym for DISPLAY
Adding synonym SHOW for (DISPLAY)
<PARSED>

Sclass S3, 1 situations:
   Status = R-COMPLETE, Duration = INTERVAL
   IS:QPE,C-S(WATER,LIQUID,G),C-S(WATER,LIQUID,F)
    VS: VI0: CONTAINED-STUFF(C-S(WATER,LIQUID,G))
        VI1: CONTAINED-STUFF(C-S(WATER,LIQUID,F))
    PS: PI0: LIQUID-FLOW(WATER,G,F,P1)
    --  Environments --

 Env ENV-203:
  A[AMOUNT-OF-IN(WATER,LIQUID,G)]>ZERO
  A[AMOUNT-OF-IN(WATER,LIQUID,F)]>ZERO
  A[PRESSURE(C-S(WATER,LIQUID,F))]<A[PRESSURE(C-S(WATER,LIQUID,G))]
  A[FLOW-RATE(PI0)]>ZERO
  A[FLOW-RATE(PI1)]??ZERO
  ALIGNED(P1)
  ENFORCE(QUANTITY-EXISTENCE)

34 ?--reset detail.
<PARSED>
DETAIL reset.

35 ?--what can lh2 lead to?
<PARSED>
LH2 leads to the following states
  S4
36 ?--what does lh2 lead to?
<PARSED>
LH2 leads to the following states
  S4
37 ?--what states does lh2 lead to?
<PARSED>
LH2 leads to the following states
```

```
  S4
38 ?--to what states does lh2 lead?
<PARSED>
LH2 leads to the following states
  S4
39 ?--to what can lh2 lead?
<PARSED>
LH2 leads to the following states
  S4
40 ?--what envs does lh2 lead to?
<PARSED>
LH2 leads to the following environment:
ENV-193

41 ?--what happens after lh2?
<PARSED>
LH2 leads to the following states
  S4
42 ?--what happens after lh2 occurs?
<PARSED>
LH2 leads to the following states
  S4
43 ?--what must hold for lh2 to occur?
<PARSED>
For LH2 to OCCUR  one of the following environments must hold :
  ENV-172

44 ?--what holds before lh2?
<PARSED>
Before LH2 occurs the following envirionments may hold :
  ENV-172

45 ?--can s1 occur from s0?
<PARSED>
No S1 cannot occur from S0

46 ?--can s3 lead to s4?
<PARSED>
Yes, S3 can lead to S4 directly.

47 ?--can s4 occur from s3?
<PARSED>
Yes, S4 can occur from S3 directly.

48 ?--can lh2 lead to an end state?
<PARSED>
```

```
LH2 leads to the following specified states (i.e. end states) :
  S4
49 ?--is s0 an end state?
<PARSED>
yes S0 is (a) (END STATE)

50 ?--what states have transitions in?
<PARSED>
The following states have transitions IN :
  S4

51 ?--what states have transitions out?
<PARSED>
The following states have transitions OUT :
  S3  S5

52 ?--does s0 have transitons?
<PARSED>
No transitions in to S0.
No transitions out of S0.

 (Replaced TRANSITONS with TRANSITIONS)
53 ?--what corresponds to the amount-of water in g?
I am not able to handle correspondences.

54 ?--q
Do you want to save any paraphrases or synonyms? (Y or N) n
NIL
>
```

B GRAMMAR EXTENSIONS FOR QATT

```
;;; -*- Package: ATT; Syntax: Common-Lisp -*-

;;; Defn Comm - DRGW
;;; These are meant to parse commands used to define sets of objects.
;;; Define X <as, to be> [].
;;; Call [] X.
;;; Let X be [].
;;; [] : quantity-phrase
;;;      the set of Y's <with, in which>
;;;   : quantity-phrase <increasing, decreasing, constant>
;;;      process-phrase  <present, active>
;;;       and .   (possibly ORs later) (possibly NOTs later)
;;; "Call the level in G LEV-G."
;;; "Define FOO as the set of states in which the flow rate through PIPE1 is increasing."
;;; "Let BAR be the set of states with a flow into G <present> and boiling in F."

;;; Defn Comm
;;; Top level of Definition command parsing.
(defrecord DEFN-COMM
  ((branch (DEFINE
           (test-word (lambda (x) t) NAME)
           ;; add word to spell table DRGW 7/6
           (test (or (spell-insert (get-binding 'name)) t))
           (parse defn-comm-connect)
           (parse gaggle)
           (parse punctuation)
           (parse-result-if-end (respond 'define 'verb-command)))
          (CALL
           (parse gaggle)
           (test-word (lambda (x) t) NAME)
           ;; add word to spell table DRGW 7/6
           (test (or (spell-insert (get-binding 'name)) t))
           (parse punctuation)
           (parse-result-if-end (respond 'define 'verb-command)))
          (LET
           (test-word (lambda (x) t) NAME)
            ;; add word to spell table DGRW 7/6
            (test (or (spell-insert (get-binding 'name)) t))
            (parse defn-comm-connect)
            (parse gaggle)
            (parse punctuation)
            (parse-result-if-end (respond 'define 'verb-command) )))))

;;; Gaggle (terrible name, I know)
;;; parses the specification for the set of things being gathered.
;;; i.e. "...Set of states with <SPEC>..."
;;; (As in a gaggle of geese).
(defrecord GAGGLE
  ((branch ((parse-optional article)
            set of
            (test-word (lambda (x) t) GAG-TYPE)
            (parse-optional gag-connect)
            (parse gag-specs)
            (parse-result (cons  'SET
                                 (cons (get-binding 'gag-type)
                                       (list (get-binding 'gag-specs)))))
           ((parse quantity-phrase)
            (parse-result (get-binding 'quantity-phrase)))
           ((parse process-phrase)
            (parse-result (get-binding 'process-phrase))))))

;;; Gag Specs
;;; parses specifications for set membership
(defrecord gag-specs
  ((branch ((parse gag-spec)
            (one-of and or)
            (test (bind 'gag1 (get-binding 'gag-spec)))
            (parse gag-specs)
            (parse-result (cons (get-binding 'gag1)
                                (get-binding 'gag-specs))))
           ((parse gag-spec)
            (parse-result (list (get-binding 'gag-spec)))))))

;;; Gag Spec
;;; parses a single specification for set membership
(defrecord GAG-SPEC
  ((branch ((parse good-quantity-phrase)
            (parse gag-spec-q-spec)
            (parse-result (cons (get-binding 'gag-spec-q-spec)
                                (get-binding 'quantity-phrase))))
           ((parse good-process-phrase)
            (parse gag-spec-p-spec)
            (parse-result (cons (get-binding 'gag-spec-p-spec)
                                (get-binding 'process-phrase))))
           ((parse good-proc-v-phrase)    ;;Doesn't require a p-spec
            (parse-result (cons (find-p-spec (get-binding 'good-proc-v-phrase))
                                (get-binding 'good-proc-v-phrase))))
           ((parse lh-phrase)
            (parse gag-spec-lh-spec)
            (parse-result (cons (get-binding 'gag-spec-lh-spec)
                              (get-binding 'lh-phrase))))))


;;; Gag Spec Q Spec
;;; parses a specifying word for quantities (like increasing).
(defrecord GAG-SPEC-Q-SPEC
  ((branch ((parse-optional preposed-aux)  ; is increasing, increases -> val of increase
            (parse verb)
            (test (and (pget (get-binding 'verb) 'verb)
                       (member 'q-spec
                               (lexical-subcat (pget (get-binding 'verb) 'verb)))))
            (parse-result
             (get (verb-key (lexical-info (pget (get-binding 'verb) 'verb)))
                               'val)))
           ((parse-optional preposed-aux)
            (test-word (lambda (x)
                         (member x (lexical-info (pget 'q-spec 'subcat)))) spec)
            (parse-result (get (get-binding 'spec) 'val))))))

;;; Gag Spec P Spec
;;; parses a specifying word for processes (like active).
(defrecord GAG-SPEC-P-SPEC
  ((branch ((parse-optional preposed-aux)
            (parse verb)
            (test (and (pget (get-binding 'verb) 'verb)
                       (member 'p-spec (lexical-subcat
                                         (pget (get-binding 'verb) 'verb)))))
            (parse-result
             (get (verb-key (lexical-info (pget (get-binding 'verb) 'verb))) 'val)))
           ((parse-optional preposed-aux)
            (test-word (lambda (x)
                         (member x (lexical-info (pget 'p-spec 'subcat)))) spec)
            (parse-result (get (get-binding 'spec) 'val))))))

;;; GAG-SPEC-LH-SPEC parses a specifying word for LHs (like occur).
(defrecord GAG-SPEC-LH-SPEC
  ((branch ((parse-optional preposed-aux)
            (parse verb)
            (test (and (pget (get-binding 'verb) 'verb)
                       (member 'lh-spec
                               (lexical-subcat (pget (get-binding 'verb) 'verb)))))
            (parse-result
             (get (verb-key (lexical-info (pget (get-binding 'verb) 'verb))) 'val)))
           ((parse-optional preposed-aux)
            (parse neg)
            (parse verb)
           (test (and (pget (get-binding 'verb) 'verb)
                      (member 'lh-spec
                              (lexical-subcat (pget (get-binding 'verb) 'verb)))))
           (parse-result (- 0 (get (verb-key
                                     (lexical-info (pget (get-binding 'verb) 'verb)))
                                   'val))))
          ((parse-optional preposed-aux)
           (test-word (lambda (x)
                        (member x (lexical-info (pget 'lh-spec 'subcat)))) spec)
           (parse-result (get (get-binding 'spec) 'val))) )))

;;; Defn Comm Connect
;;; parses connecting words for definition commands
(defrecord defn-comm-connect
  ((branch (be (parse-result 'be))
           (to be (parse-result 'to-be))
           (as (parse-result 'as)))))

;;; Gag Connect
;;; parses connecting words for gaggles.
(defrecord gag-connect
  ((branch (with (parse-result 'with))
           (where (parse-result 'where))
           (in which (parse-result 'in-which))
           (that (parse-result 'that))    ;;DRGW 6/26
           (that have (parse-result 'have)))))


;;; -*- Package: ATT; Syntax: Common-Lisp -*-

;;; QPE-ATT specific grammar enhancements for quantities - DRGW

;;; Good Quantity Phrase
;;; Requires a Q to be found from Q-Phrase
(defrecord good-quantity-phrase
  ((parse quantity-phrase)
   (test (find-quantity (get-binding 'quantity-phrase)))
   (parse-result (get-binding 'quantity-phrase))))

;;; Quantity Phrase
;;; Parses a phrase that refers to a Quantity
;;; i.e. "... amount of water in can1..."
(defrecord QUANTITY-PHRASE
  ((branch
    ((parse quantity-p)
     (test (not (prep-next? 'quantity-p)))
     (parse-result (get-binding 'quantity-p)))
    ((rebind)
     (parse quantity-p)
     (test (prep-next? 'quantity-p))
     (test (bind 'quant1 (get-binding 'quantity-p)))
     (parse prep)
     (test (bind 'prep1 (get-binding 'prep)))
     (branch
             ((parse quantity-phrase)
              (parse-result (list (get-binding 'quant1)
                                  (get-binding 'prep1)
                                  (get-binding 'quantity-phrase))))
             ((parse good-process-phrase)
               (parse-result (list (get-binding 'quant1)
                                   (get-binding 'prep1)
                                   (get-binding 'good-process-phrase))))  )))))

;;; Quantity P
;;; Gathers quantity type words and eats articles.
(defrecord quantity-p
    ((parse-optional article)
     (parse quantity-words)
     (parse-result (get-binding 'quantity-word))))

;;; Quantity Words
;;; Gathers consequtive Quantity words (maybe "and")
(defrecord quantity-words
  ((branch
    ((parse quantity-word)
     (parse-result (list (get-binding 'quantity-word))))
    ;; flow rate
    ((parse quantity-word)
     (test (bind 'q-word1 (get-binding 'quantity-word)))
     (parse quantity-words)
     (parse-result (cons (get-binding 'q-word1)
                         (get-binding 'quantity-words))))
    ;; flow and pressure
    ((parse quantity-word)
     (test (bind 'q-word1 (get-binding 'quantity-word)))
     and
     (parse quantity-words)
     (parse-result (cons (get-binding 'q-word1)
                         (cons 'and
                               (get-binding 'quantity-words)))))))

;;; Quantity Word
;;; Parses a single Quantity word.
;;; Test makes sure that the subcat is a quantity word
;;;  (Set in Init-Qs())
(defrecord quantity-word
  ((parse subcat)
   (test (and (pget (get-binding 'subcat) 'subcat)
              (listp (lexical-subcat (pget (get-binding 'subcat) 'subcat)))
              (member 'quant (lexical-subcat (pget (get-binding 'subcat) 'subcat)))))
   (parse-result (get-binding 'subcat))))

;;; Prepp Obj
;;; parses prepp and returns its object
;;; NOTE: This had to be added to avoid ATT's
;;; insistance on restrictive Prepps.
(defrecord prepp-obj
  ((parse prep)              ; using prepp sends it down noun-with-adj
   (parse-optional article)
   (parse subcat)
   (parse-result (list (get-binding 'prep)
                       (get-binding 'subcat)))) )

;;; Prepp Objs
;;; Parses multiple Prepp-objs
(defrecord prepp-objs
  ((branch ((parse prepp-obj)
            (test (prep-next? nil))
            (test (bind 'po1 (get-binding 'prepp-obj)))
            (parse prepp-objs)
            (parse-result (cons (get-binding 'po1)
                                (get-binding 'prepp-objs))))
           ((parse prepp-obj)
            (parse-result (get-binding 'prepp-obj))))))

;;; Prep Next?
;;; Test to see if the next wrod is a Prep
;;; or the last word ended with a prep (i.e. amount-of)
(defun prep-next? (bound-part)
  (cond ((prep? (car remaining-words)) t)
        ((and bound-part
              (prep? (car (last (dehyph (get-binding bound-part))))) ; amount-of case
              (push (car (last (dehyph (get-binding bound-part))))
                    remaining-words))))

;;; -*- Package: ATT; Syntax: Common-Lisp -*-

;;; QPE-ATT specific grammar enhancements for processes - DRGW
;;; Good Process Phrase
;;; Requires a Process to be found from phrase
(defrecord good-process-phrase
  ((branch ;; the Flow of water from A to B
           ((parse process-phrase)
            (test (setq res (find-process (get-binding 'process-phrase))))
            (parse-result (get-binding 'process-phrase)))
           ;; water is flowing from A to B
           ((parse PROC-V-PHRASE)
            (parse-result (get-binding 'proc-v-phrase))) )))

;;; Good Process V Phrase
;;; Requires a Process to be found from the V-Phrase
(defrecord good-proc-v-phrase
  ((parse PROC-V-PHRASE)
   (test (find-process (get-binding 'proc-v-phrase)))
   (parse-result (get-binding 'proc-v-phrase))))

;;; Process Phrase
;;; Parses a phrase that might refer to a process of the form
;;;  "... FLow of water from F to G..."
(defrecord PROCESS-PHRASE
  ((branch
    ((parse process-p)
     (test (not (prep-next? 'process-p)))
     (parse-result (list 'PROC (get-binding 'process-p))))
    ((rebind)
     (parse process-p)
     (test (prep-next? 'process-p))
     (test (bind 'proc1 (get-binding 'process-p)))
     (parse prep)
     (test (bind 'prep1 (get-binding 'prep)))
     (parse process-phrase)
     (parse-result (list 'PROC
                         (list (get-binding 'proc1)
                               (get-binding 'prep1)
                               (get-binding 'process-phrase))))))))

;;; Process V Phrase
;;; parses a process phrase where the process key word is
;;; used as a Verb in the phrase.
;;; i.e. water is flowing from A to B.
;;; NOTE: This verb must be added to lexicon!
(defrecord PROC-V-PHRASE
  ((parse-optional article)
    (parse process-words)
    (parse-optional preposed-aux)
    (parse proc-verb)
    (parse prepp-objs)
    (parse-result (list (if (member 'neg *tense*) 'NOT)
                        (verb-key (lexical-info (pget (get-binding 'proc-verb) 'verb)))
                        (get-binding 'prepp-objs)))))

;;; Proc Verb
;;; Parses a process verb like "Flowing"
(defrecord proc-verb
  ((parse-optional preposed-aux)
   (parse-optional neg)
   (parse verb)
   (test (member 'PROC-VERB (lexical-subcat (pget (get-binding 'verb) 'verb))))
   (parse-result (get-binding 'verb))))

;;; Process P
;;; parses multiple process words and articles
(defrecord process-p
  ((parse-optional article)
   (parse process-words)
   (parse-result (get-binding 'process-word))))

;;; Process Word
;;; Parse a process word
;;; Subcat must have 'PROC as a :subcat,
;;; Put in in Init-PsVs().
(defrecord process-word
  ((parse subcat)
   (test (and (pget (get-binding 'subcat) 'subcat)
              (listp (lexical-subcat (pget (get-binding 'subcat) 'subcat)))
              (member 'proc (lexical-subcat (pget (get-binding 'subcat) 'subcat)))))
   (parse-result (get-binding 'subcat))))

;;; Process Words
;;; Parses multiple process words
(defrecord process-words
  ((branch
     ((parse process-word)
      (parse-result (list (get-binding 'process-word))))
     ;pumped flow
     ((parse process-word)
      (test (bind 'p-word1 (get-binding 'process-word)))
      (parse process-words)
      (parse-result (cons (get-binding 'p-word1)
                          (get-binding 'process-words))))
      ;;pumped and flow (It could happen)
      ((parse process-word)
       (test (bind 'p-word1 (get-binding 'process-word)))
       and
       (parse process-words)
       (parse-result (cons (get-binding 'p-word1)
                          (cons 'and
                                (get-binding 'process-words))))) )))

;;; -*- Package: ATT; Syntax: Common-Lisp -*-

;;; QPE-ATT specific grammar enhancements for Limit Hypotheses - DRGW

;;; Lh Phrase
;;; Parses limit hypothesis phrases
;;; NOTE for now just requires some lh-words
(defrecord LH-PHRASE
  ((parse lh-words)
   (parse-result (get-binding 'lh-words))))

;;; Lh Words
;;; Parses multiple LH words
(defrecord lh-words
  ((branch ((parse lh-word)
            (test (bind 'lh1 (get-binding 'lh-word)))
            (parse lh-words)
            (parse-result (cons (get-binding 'lh1)
                                (get-binding 'lh-words))))
           ((parse lh-word)
            (parse-result (get-binding 'lh-word))))))

;;; Lh Word
;;; Parses one Lh word
;;; Requires the subcat to have 'limit-hypothesis in its :info
(defrecord lh-word
  ((test-word (lambda (w)
                (member w (lexical-info
                           (pget 'limit-hypothesis 'subcat)))) lhw)
   (parse-result (get-binding 'lhw))))

;;; Lh Verb
;;; Parses a verb disignated as being a possible LH spec
;;; i.e. "...occurs..." or "... happens ..."
(defrecord lh-verb
  ((parse verb)
(test (member 'LH-VERB (lexical-subcat (pget (get-binding 'verb) 'verb))))
(parse-result (get-binding 'verb)))
```

BIBLIOGRAPHIC DATA SHEET	1. Report No. UIUCDCS-R-89-1535	2.	3. Recipient's Accession No.
4. Title and Subtitle QATT: ,A NATURAL LANGUAGE INTERFACE FOR QPE			5. Report Date August 1989
			6.
7. Author(s) Douglas Robert-Graham White			8. Performing Organization Rept. No. R-89-1535
9. Performing Organization Name and Address Dept. of Computer Science 1304 W. Springfield Avenue Urbana, IL 61801			10. Project/Task/Work Unit No.
			11. Contract/Grant No.
12. Sponsoring Organization Name and Address			13. Type of Report & Period Covered Technical
			14.
15. Supplementary Notes			

16. Abstracts

This thesis presents QATT, a natural language interface developed for the Qualitative Process Engine (QPE) system. The major goal of the project was to evaluate the use of a preexis ng natural language understanding system designed to be tailored for query processing in multiple domains of application. The other goal of QATT is to provide a comfortable environment in which to query envisionments in order to gain insight into the qualitative behavior of physical systems. It is shown that the use of the preexisting system made possible the development of a reasonably useful interface in a few months.

17. Key Words and Document Analysis. 17a. Descriptors

natural language
qualitative physics
interfaces

17b. Identifiers/Open-Ended Terms

17c. COSATI Field/Group

18. Availability Statement unlimited	19. Security Class (This Report) UNCLASSIFIED	21. No. of Pages 50
	20. Security Class (This Page UNCLASSIFIED	22. Price

FORM NTIS-35 (10-70) USCOMM-DC 40329-P71