A METHODOLOGY FOR REPRESENTING NATURAL LANGUAGE EXPRESSIONS IN VARIABLE-VALUE LOGIC

Anthony R. Nowicki

MLI 88-4[illegible] 12
TR-10-88

A publication of *Machine Learning & Inference Laboratory*
Artificial Intelligence Center
George Mason University
Fairfax, VA 22030 USA
(703) 764-6259

Editor: R. S. Michalski
Assistant Editor: J. Zhang

MLI Reports replaces *ISG Reports* published until
December, 1987 by the Artificial Intelligence Laboratory
Department of Computer Science, University of Illinois
Urbana, IL 61801.

A METHODOLOGY FOR REPRESENTING NATURAL LANGUAGE EXPRESSIONS IN VARIABLE-VALUED LOGIC

Anthony R. Nowicki

**Martin Marietta Corp.
Denver, CO 80222**

**MLI 88-4
TR-10-88**

June 1988

This report is based on the author's Master Thesis completed in the ISG group of the Department of Computer Science of the University of Illinois at Urbana-Champaign. The thesis is done under the guidance of Professor R. S. Michalski.

A METHODOLOGY FOR REPRESENTING NATURAL LANGUAGE EXPRESSIONS IN VARIABLE-VALUED LOGIC

ABSTRACT

As Artificial Intelligence systems and in particular expert systems become more common, the number of diverse users increases. The design of interfaces for these systems must take this into account. An interface is required for any system, for without the interface, the system would have no means of communication with its users. As these systems become more readily available for a wide range of tasks, more are being used by casual users, those who do not use the system day in and day out, and naive users, those who may be regular users but may not have a complete understanding of the system and do not profess to know or even care. These users wish to simple answer. Users usually want to communicate in a language that they already know. This becomes even more critical when the casual user is considered, they are more likely to reject or forget a specialized input language.

A system is described, the PARALINC system, which implements a natural language interface designed to interact with expert and learning systems developed by the Intelligent Systems Group at the University of Illinois, Urbana. Systems of particular interest include the EXPLORER project, an integrated learning and inference system, the INDUCE system, a machine learning program, and the ADVISE system, a meta-expert system. The expert and learning systems to which this interface will serve are systems based on an information representation in a variant of predicate calculus called variable-valued logic. This calculus is a typed, first-order logic which additional features not found in a standard predicate calculus.

ACKNOWLEDGEMENTS

I would like to first thank Professor R. S. Michalski for his support throughout this research. His suggestions and interest in this research helped maintain my interest throughout. There are a number persons to whom I owe much gratitude. Brian Falkenhainer has offered much support and encouragement. Thanks to jeff Becker and Bob Stepp for their helpful comments.

This research was supported in part by the NSF under DCR 84-06801, in part by the Office of Naval Research under grants N00014-82-K-0186, in part by the Defence Advanced Research Projects Agency under grant, administered by the Office of Naval Research, No. N00014-87-K-0874, in part by the Office of Naval Research under grant No. N00014-88-K-0226, and in part by the Office of Naval Research under grant No. N00014-88-K-0397.

1. INTRODUCTION

As Artificial Intelligence (AI) systems and in particular expert systems become more common, the number of diverse users increases. The design of interfaces for these systems must take this into account. An interface is required for any system, for without the interface, the system would have no means of communication with its users. As these systems become more readily available for a wide range of tasks, more are being used by casual users, those who do not use the system day in and day out, and naive users, those who may be regular users but may not have a complete understanding of the system and do not profess to know or even care. These users wish to simply enter the appropriate information and obtain a useful answer. Users usually want to communicate in a language that they already know. This becomes even more critical when the casual user is considered, they are more likely to reject or forget a specialized input language.

A system is described, the PARALINC (PARAphrasing natural Language IN predicate Calculus) system, which implements a natural language[1] interface designed to interact with expert and learning systems developed by the Intelligent Systems Group at the University of Illinois, Urbana. Systems of particular interest include the EXPLORER project [Becker, 1985b], an integrated learning and inference system, the INDUCE system [Hoff, Michalski, and Stepp, 1983], a machine learning program, and the ADVISE system [Michalski, 1983], a meta-expert system. The expert and learning systems to which this interface will serve are systems based on an information representation in a variant of predicate calculus called variable-valued logic (VL). This calculus is a typed, first-order logic with additional features not found in a standard predicate calculus.

Figure 1 depicts the structure of a typical system. The component of particular interest in this thesis is the interface between the user and the host system, more specifically, the input side of the system. There may or may not be an analogous component for output. This interface is the component which handles the often difficult problem of translating the *user oriented language* into a *host oriented language*.

[1] Although natural language can refer to many languages, natural language within the scope of this thesis refers strictly to the English language, and more precisely, a subset of the English language.

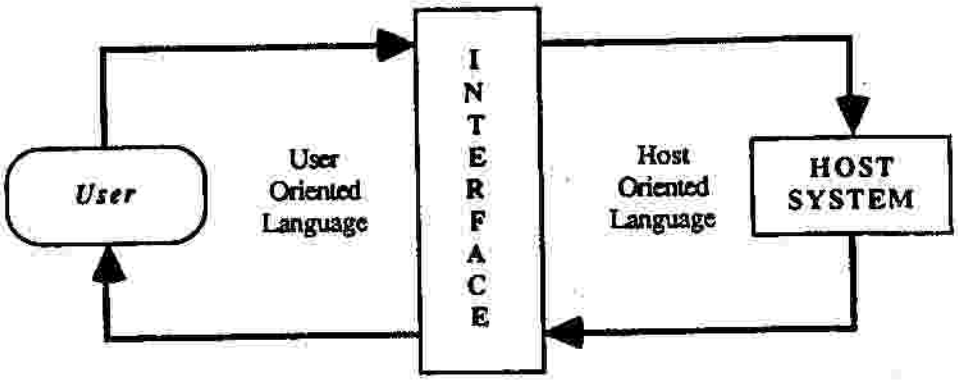


Figure 1. Top Level View of a Typical System

The user oriented language accepted by PARALINC is natural language, the standard method of exchanging information between people. The host languages are the variable-valued languages VL_1 (variable-valued logic one), VL_2 (variable-valued logic two), and Annotated Predicate Calculus (APC), each being a representation language used by a family of logic-based expert and learning systems. Both of these languages contain features which make them attractive for this system.

1.1. Why Use Natural Language for an Interface?

One's first response to "Why use natural language?" may be "why not?". As flippant as that may sound, it drives home the point "Use what you know best." There are a number of reasons for choosing natural language for an interface language. The primary reasons are based on familiarity and experience. Even though a user of a system may not be familiar with all of its details, he must be familiar with the language used for information exchange. Therefore, there must be a simple yet expressive means for any user and the system to interact. No matter how good the system, if an average user cannot interact with it efficiently, the system is not being used to its potential.

Familiarity and experience with a language convey a sense of friendliness. A friendly system in one which makes the interchanges simple and understandable. It is a known fact that people are often fearful of computers, and in some cases rightly so. The system should not place a burden on the user, especially the intermittent user, for this slows down the user and often requires him to learn a new "language" for which to exchange information with the system. People have a tendency to reject learning something new if they have something that will already "make do". A goal of any user interface is to reduce the learning time or training involved with the system. This points to a need for a means of interaction that is common to all including the naive user. For example, expert systems by design are meant to assist and in some cases replace the need for a human expert in some domain. This would imply that the users of such a system may not be as well versed as an expert and would thus have less detailed knowledge about the domain or the system. In most cases, the expert system's formalism, reasoning algorithms, and system language are hidden to the end user making a system language seem foreign. Natural language is a language which is already familiar to all users, making it an obvious choice for a means of interaction.

Another goal of any interface system is to assist the user and make the man/machine interaction go as smoothly as possible. If the system language is used for the exchange, the simplicity of the interaction may be impacted. A typical system language is primarily designed with the system in mind and not the user. As an example, a detailed interaction which may require a number of individual requests in a host language may be contained in a single natural language input. In many cases, a single sentence or expression in natural language will transform into more than one equivalent logical host expression. Although the system language may have the ability to represent a more complicated request concisely, one must remember that a natural language interface is for the general user. A user who may not known all of the details of the system or its language.

It is often argued that the use of natural language introduces uncertainty. In many cases it makes the system appear more friendly to the user, but in some cases, it may frustrate the user if the input is misunderstood. Although there is inherent ambiguity in natural language input, there are a number factors which reduce a potentially open-ended problem into a tractable one. First, the domains of host systems are typically constrained. This in turn constrains the user vocabulary. In most cases, only a subset of natural language is needed due the limited domain or fixed representation language of the system. Secondly, due to the nature of the exchange and the type of user involved in the interaction, the input is agained constrained. When input consists of short interchanges, the user has a tendency to adhere to rules of good English when formulating queries or generating imperatives. When the user does violate grammatical rules, the violation is often easily detected and the missing information can be filled in as appropriate. An example of this is the common use of ellipsis when expanding on the subject or object of the previous sentence.

Having described natural language as the interface language of choice, one may ask, "why isn't natural language used for all of the current systems?" One answer is that the design and development of a natural language processing system takes time, and in many cases, such a system is never fully completed. A refinement process occurs throughout the lifetime of the system. Also, many systems are being used by people who appreciate the capabilities and the conciseness of an alternate language. A well designed interface should accommodate more sophisticated users. An interface should not make these users suffer with a more verbose and potentially ambiguous language, after all, the system language is tailored to the system. Despite this, the end users of many systems, particularly expert systems, are the naive, those who desire for an easy program to use and understand. Expert systems, intelligent tutoring systems, and database retrieval systems are example systems for which this type of interface technology best applies.

1.2. Why Variable-Valued Logic?

Natural language may be a good choice as an interface language, however, the language chosen for representing the information must also be considered. There is a need to find a language both expressive and concise enough to represent the full intent of the user but should tend towards the comprehensible. Any language chosen as the host oriented language must be capable of representing the details specified in the user oriented language in an understandable way. In [Michalski, 1983], he presents the *comprehensibility postulate*. It states that

> The results of computer induction should be symbolic descriptions of given entities, semantically and structurally similar to those a human expert might produce observing the same entities. Components of these descriptions should be comprehensible as single "chunks" of information, directly interpretable in natural language, and should relate quantitative and qualitative concepts in an integrated fashion.

The comprehensibility postulate not only states why a variable-valued logic is a good choice for the host system language but also indicates that natural language is the best choice for the interaction with the user by stating that "the information should be directly interpretable in natural language." In many systems, the representation language is designed for efficiency rather than comprehensibility. Although such a representation is efficient for program processing, it is not efficient or convenient for the average user. The representation language chosen for PARALINC exhibits both comprehensibility and conciseness.

The issue of comprehensibility is also important when one considers a man/machine interaction. Verification of the user's input is important for reducing the number of errors and to instill a feeling of confidence in the user that the system has understood the request. If the resultant structure is too complicated, the user cannot verify that the input was understood. Variable-valued logics serves both as inputs to the host systems and an efficient means of representing natural language inputs that are understandable and verifiable to the user.

1.3. Related Work

The topic of this thesis has its roots in a number of areas of interest of the Artificial Intelligence and Computational Linguistics communities. Relevant areas of interest for this work include Natural Language Processing (NLP) and Natural Language Understanding (NLU), interface systems in natural language, and systems that use a variable-valued logic formalism.

Natural Language Processing and Natural Language Understanding have been active research areas within the field of AI from the very beginning. Much of these efforts grew out of the view that in order to exhibit intelligence, a machine should be capable of conversing with the user in the user's language. Research within these fields is concerned with the more theoretical aspects of understanding natural language expressions by capturing the correct meaning of the expression and to hopefully learn about the nature of language and information exchange. In many cases, information is exchanged and intent is understood using only a subset of a grammatically correct expression. Issues addressed in these fields include the handling of incomplete sentences, pronoun references, ellipsis, gapping forms, and the various issues of semantic and syntactic processing. Researchers such as Chomsky, Marcus, Woods, Winograd, Schank, and many others have addressed many of these issues. Much of their work serves as a good reference for those interested in more theoretical issues of natural language processing. In addition, researchers in the field of Intelligent Man Machine Interfaces (IMMI) are dedicated to improving all interface technologies for computer-based systems, including natural language and graphics-based interactions.

Expert systems and data retrieval systems are prime examples of systems which have effectively used natural language interfaces. These systems are currently making there way into the general user community. Although many expert systems are quite useful on their own, users found it easier to converse with systems when the systems understood natural language. Natural language systems like XCALIBUR [Carbonell, et.al., 1983], the natural language interface for the R1 (XCON) computer configuration expert system, became almost required when the system made it out into the user community. In fact, systems such as Bobrow's STUDENT [Bobrow, 1968] or Novak's ISAAC [Novak, 1976] problem solvers had a natural language interface as part of their initial design.

Some of the more known natural language interface systems include the LIFER system [Hendrix, 1977], an applications oriented system for creating natural language interfaces for computer programs. The LUNAR system [Woods, 1972], a natural language database retrieval system for the analysis of the Apollo moon rocks, and the PLANES system [Waltz, 1977], a large relational database retrieval system for aircraft flight and maintenance information. Many of these and additional examples can be found in [Waltz, 1977], where a survey of over 50 natural language interface research efforts is presented. It is clear from this list that natural language understanding is a popular area of research and will continue to be.

In addition to the many references on natural language understanding research, much effort has gone into the theory and design of parsers for computer languages and compilers [Aho and Ullman, 1972]. Many issues concerning grammars and parsing are addressed in these areas.

Of obvious importance to this research is the use of variable-valued logics as a means of representing the internal knowledge. There are a number of systems that make use of these languages. The systems currently using a variable-valued language include INDUCE, GEM [Reinke, 1984], CLUSTER [Stepp, 1984], and ADVISE [Michalski and Baskin, 1983]. The eventual goal for PARALINC is to provide a uniform interface to all of these systems. The difficulty of this is that each system exhibits enough differences in their interaction that it is difficult to retrofit a generic interface approach to them.

Other translation systems and inference systems have been designed for variable-valued logics. The PARA [PARA, 1983] system performed a VL to natural language translation. When performing this translation, PARA transformed a more structured language into a more flexible by the use of simple pattern matches. A system was also developed which translated APC into PROLOG [Pua, 1984a]. In addition, experiments were performed on an inference system developed for APC [Pua, 1984b].

The work described in this thesis addresses the use of variable-valued languages as a means of representing natural language expressions. The merging of these two areas will help in defining both the interfaces required by VL-based systems and the meaning of a VL expressions in natural language.

1.4. Thesis Overview

Section 2 of this thesis contains a detailed description of one of the variable-valued logics, the Annotated Predicate Calculus (APC) language, as presented in [Michalski, 1983]. The reader unfamiliar with the APC language is encouraged to read Section 2 prior to subsequent sections. An understanding of the language and its representation will help to better understand the process of transforming natural language input into VL expressions. Section 3 introduces the overall architecture of the PARALINC system and describes the theory and operation of each of its major components. Examples are cited which are drawn directly from a working version of PARALINC. A summary and conclusion of this research is presented in Section 4.

2. THE REPRESENTATION LANGUAGE

The formalism chosen to represent the natural language inputs is a variant of predicate calculus described as *Variable-Valued* (VL) logic. The VL languages used by various systems developed by the Intelligent Systems Group have evolved over the years. The first VL formalism is VL_1 (variable-valued logic one) [Michalski, 1974]. VL_1 has been successfully used in learning and expert systems including PLANT/ds [Reinke, 1983], and ExceL [Becker, 1985c].

The successor of VL_1 is the language VL_2. VL_2 has all the features of VL_1 plus some additional capabilities allowing for a more complex selector form. The VL_2 language has been successfully used in programs like INDUCE [Hoff, Michalski and Stepp 1983], and CLUSTER [Stepp, 1984].

The newest and most extensive VL language is APC (Annotated Predicate Calculus). It was first presented in Michalski [Michalski, 1983] and is the building block of a current project to integrate an interactive learning and expert system, the EXPLORER system [Becker, 1985b].

The APC language is a typed predicate calculus with additional expressive power. Being the predecessor of VL_1 and VL_2, APC has the ability to represent any description that can be expressed in either language but contains additional capabilities not found in VL_1 or VL_2. These additional features add both a functional and expressive power that lends itself to representing expressions, particularly English sentences, as well as being general enough to be used by a family of inductive learning and expert systems. Because APC subsumes both VL_1 and VL_2, only a description of the APC formalism is presented.

2.1. The APC Language

The APC language is an extension of predicate calculus that contains a number of novel forms along with an attached *annotation* for each predicate, variable, and function. The annotation stores information such as the type of the function and attributes. Example annotations may include the domains of the values, or the range of the value sets for functions.

2.1.1. APC Form

Terms in the APC language can be both elementary and compound. An *elementary term* (e-term) is the same as a term in predicate calculus. E-terms may be constants, variables, or function symbols followed by a list of arguments that are themselves e-terms. Some example e-terms are:

$$\text{BOX}_1 \qquad \text{Population(US)}$$

A *compound term* (c-term) is a composite of e-terms or a function symbol in which one or more arguments are compound terms. Composition of e-terms is described as the *internal disjunction* ($\vee$) or the *internal conjunction* (&) of e-terms. The following are examples of compound terms:

$$\text{BALL}_1 \vee \text{BOX}_1 \qquad \text{Population(US \& CANADA)}$$

Compound terms do not have a truth status as traditional predicates do but rather are used only as arguments in predicates. Within APC, the logical operators $\vee$ and & can also be applied to e-terms as well as predicates. Arguments of functional terms that are composites can be transformed into a composite of elementary terms. If f is a functional symbol having n arguments, A representing $n-1$ of them, and the n'th being the conjunction or disjunction of two elementary terms t_1 and t_2, then the composite term can be expanded by the application of the following *term-rewrite* rules:

$$f(t_1 \vee t_2, A) \leftrightarrow f(t_1, A) \vee f(t_2, A)$$
$$f(t_1 \,\&\, t_2, A) \leftrightarrow f(t_1, A) \,\&\, f(t_2, A)$$

A sample application of the rewrite rules is:

$$\text{Population(US \& CANADA)} \iff \text{Population(US) \& Population(CANADA)}$$

When A is a composite and contains both internal conjunctions and internal disjunctions, the rewrite rules are applied in such a manner as to expand the internal disjunction first. A conjunctive form is more binding than the disjunctive.

Predicates in APC, like terms, can be either elementary or compound. An *elementary predicate* (e-predicate) is a predicate form in which the arguments of the predicate are e-terms. *Compound predicates* (c-predicates) are predicates whose arguments are c-terms. An example e-predicate is:

$$\text{Went(FRED, STORE)}$$

An example c-predicate would be:

Went(FRED & BARNEY, POOL_HALL $\vee$ BOWLING_ALLEY)

There also exists a set of rules to transform a compound predicate into one or more elementary predicates. The equivalence preserving *reformulation rules* for predicates are:

$$\begin{aligned} P(t_1 \vee t_2, A) &\models P(t_1, A) \vee P(t_2, A) \\ P(t_1 \,\&\, t_2, A) &\models P(t_1, A) \,\&\, P(t_2, A) \end{aligned}$$

where P represents an arbitrary predicate and A, t_1, and t_2 having similar meanings as in the term-rewriting rules. As an example, the reformulation rules as applied to:

Went(FRED & BARNEY, POOL_HALL $\vee$ BOWLING_ALLEY)

would generate the elementary predicates:

Went(FRED, POOL_HALL) & Went(BARNEY, POOL_HALL) $\vee$
Went(FRED, BOWLING_ALLEY) & Went(BARNEY, BOWLING_ALLEY)

The application order of the reformulation rules is of importance. The application of the disjunctive rule is performed prior to the rule that transforms a conjunctive form, much like the application order for the rewrite rules for terms. This follows our intuitive notion of the meaning of expressions that have both & and $\vee$ logical connectives. The & is considered more binding than the $\vee$ connective, therefore, the application of the rules in the above order produces the more desirable result. In the example, the ordering suggests that FRED and BARNEY went to either the POOL HALL or the BOWLING ALLEY together.

The APC language contains all forms available in first order predicate logic but also has a special form know as the *selector*. Selectors are predicates that express relations using an infix notation. This is often a more comprehensible representation than a standard predicate. The form of a selector is a relational statement defined as:

$$[\,Term_1 \text{ REL } Term_2\,]$$

where Term$_1$ (the *reference*) and Term$_2$ (the *referee*) are terms. *REL* represents one of the relational symbols: $=$, $\neq$, $>$, $<$, $\leq$, $\geq$. The equivalent predicate calculus representation would take the form *REL(Term$_1$, Term$_2$)*. If Term$_1$ is a non-constant e-term and Term$_2$ is a constant with REL being one of

$=$ or $\neq$, the selector is an *elementary selector*. An example elementary selector is:

$$[\text{Population(US)} = 1{,}000]$$

Selectors that have a non-constant referee and a conjunction or disjunction of constants from the domain of the referee's e-term are *referential selectors*. Referential selectors can have any of the possible relational symbols. An example referential selector is:

$$[\text{Color}(\text{BALL}_1) = \text{RED} \vee \text{BLUE}]$$

Note that predicates can be formally described as a selector of the form:

$$[\,Term = \text{TRUE}] \quad \text{where } Term \text{ is a predicate}$$

Compound selectors have referees and references of any term form. The relation symbol can be any of the possible relational symbols. Some example compound selectors are:

$$[\text{height(FRED} \vee \text{BARNEY)} > 5]$$
$$[\text{Population(US \& CANADA)} > \text{Population(FRANCE \& BRITAIN)}]$$

Compound selectors are often transformed using the term-rewriting rules and the predicate transformation rules before being used by the learning system or expert system.

Expressions in the APC language are composed of selectors and logical connectives. The basis of an APC expression is the selector and in fact a selector is itself an APC expression. APC expressions have the standard logical expression connectives negation, conjunction, disjunction, implication, and equivalence ($\neg$, &, $\vee$, $=>$, $<=>$), along with a new operation called the *exception*. The exception operator ($\leftthreetimes$) is defined as the symmetric difference of the two APC expressions:

$$F_1 \leftthreetimes F_2 \;\equiv\; (\neg F_2 \Rightarrow F_1) \,\&\, (F_2 \Rightarrow \neg F_1).$$

and reads F_1 *except* when F_2. In the case that the expression F is negated ($\neg$F) and it is an elementary selector having a relation symbol of $=$, the relation is redefined as $\neq$.

Often in inductive learning, the APC expression is of a special form. The form consists of a conjunction of selectors and is denoted a *complex*. A complex is often referred to as an *event* in the *event space* and represents one input example. Complexes can be combined to form *decision rules*. A decision rules consists of the disjunction of one or more complexes and an implication and takes the form:

$$C_1 \vee C_2 \vee \ldots \vee C_n \;\Rightarrow\; D$$

where C_i are complexes and D is a decision. A *decision* is a single elementary selector having a value in the decision domain.

The APC language has all of the quantifications that are available in a standard predicate calculus, but also has an additional form known as *numerical quantification.* Numerical quantification of an APC expression is expressed as:

$$\exists\ (I)\ v,\ F[v]$$

where I is the *index set* denoting a set of integers, and $F[v]$ is an arbitrary APC expression having v as a free variable. An example of numerical quantification is:

$$\exists\ (5..10)\ v,\ F[v]$$

and is read, "there exists five to ten" values of v for which $F[v]$ holds. The index set can also take the form

$$\exists\ (n_1 \vee n_2 \vee .. \vee n_n)\ v,\ F[v]$$

and is read as "there exists n_1 or n_2 or ... n_n" values of v for which $F[v]$ holds. Note that the existential quantification is subsumed by numerical quantification as defined by:

$$\exists\ v,\ F[v] \ \equiv\ \exists\ (\geq 1)\ v,\ F[v]$$

Universal quantification can also be represented with numerical quantification as:

$$\forall\ v,\ F[v] \ \equiv\ \exists (k)\ v,\ F[v]$$

where k is the number of possible values that v can take on.

2.1.2. APC Annotation

The annotation of a predicate, variable, or function (collectively called *descriptor*) contains information relevant to a particular problem. The annotation may describe the concept represented by the descriptor, or its relationship to other concepts and by describing the relevance of the descriptor given other descriptor definitions. The annotation may include problem-oriented information such as those presented in [Michalski, 1983]. Some of the relevant annotations and their functionality in PARALINC include:

- A specification of the domain and the type of the descriptor. *(aids in the parameter selection).*
- A specification of the constraints and the relationships between the descriptor and other descriptors. *(Used to verify relationships and existence with respect to other descriptions formed).*
- A characterization of the objects to which the descriptor is applicable (i.e., a characterization of the possible arguments). *(aids in the parameter selection).*
- A specification of the descriptor class containing the given descriptor that is the parent node in a generalization hierarchy of descriptors (e.g., descriptors 'length', 'width', and 'height', can have a parent node denoting 'dimension'). *(aids in the parameter selection and selector identification and selection)* .

2.1.3. APC Types

If the problem domain is well specified, a set describing the potential values achievable by the descriptor can be described. This *value set* is called the *domain* of the descriptor. The domain type description can describe the domain of the value that the selector can take on and the domain of the arguments of any predicate. Domains of the variables and arguments can be in one of the nominal, linear, cyclic, or structure domain. In the case of inductive learning, the domain is called upon to limit the extent to which a descriptor can be generalized.

- The *nominal* domain consists of a set of unordered symbolic values. For example, the attribute *color* can take on the values *red, blue, green,....*
- *Linear* domains consist of an order set of values, either numeric or symbolic which can be realized by a term or variable. For example, the attribute *length* may have values *10..100 meters*, or the attribute *size* may have values *minute..large*. The ".." is the range operator and represents the set of values found between the two given values.
- The *cyclic* domain is much like the linear except that the first element can follow the last. For example, the attribute *day* can take on the values equal to the days of the week.

The *structured* domain consists of a hierarchy of unordered values much like a tree structure. For example, *parrot*, *cockatiel*, and *parakeet* are siblings of the node *exotic birds* for attribute *birds*.

2.2. APC as a Means of Representing Natural Language

From the description of the language, it can be seen that APC exhibits a high degree of expressiveness yet remains concise and comprehensible. A comparison of expressive power between APC, first-order predicate logic, and clausal form are discussed in [Goldfain, 1985]. A few simple examples will suffice to demonstrate the expressive power of the APC language and how it can be represented in natural language.

The expressiveness of symbolic and nominal domains can be seen in the example APC representation for "the color of ball$_1$ is red or blue or green".

$$[\mathrm{Color}(\mathrm{BALL}_1) = \mathrm{RED} \vee \mathrm{BLUE} \vee \mathrm{GREEN}].$$

The equivalent predicate logic representation would be:

$$\mathrm{Color}(\mathrm{BALL}_1,\mathrm{RED}) \vee \mathrm{Color}(\mathrm{BALL}_1,\mathrm{BLUE}) \vee \mathrm{Color}(\mathrm{BALL}_1,\mathrm{GREEN})$$

When expressions have values in a linear domain, the conciseness of an APC expression is even greater. The APC expression:

$$[\mathrm{Height}(\mathrm{Tower}) = 100 \ .. \ 300]$$

would be expressed equivalently in predicate logic as:

$$\mathrm{Height}(\mathrm{Tower},\ 100) \vee \mathrm{Height}(\mathrm{Tower},\ 100+\textit{step size}) \vee \ldots \vee \mathrm{Height}(\mathrm{Tower},\ 300)$$

But would be expressed in natural language as simply:

The height of the tower is 100 to 300.

The number of disjunctively combined predicates would be equal to the number of individual steps required to get from the lower bound to the upper. In APC, the step value is specified in the domain of the associated APC expression and is part of the annotation of the expression. The alteration of the step size has no effect on the physical description of the APC expression (only on the inference and learning algorithms which have access to the annotation of the descriptor) but would directly alter the description for a predicate logic equivalent.

Considering some of the features of the APC language, natural language can be more easily represented in APC than other logic forms. Details of how VL expressions of natural language input can be constructed will be presented in Section 3.

3. THE PARALINC SYSTEM

The PARALINC system is a system which accepts input from the user in a natural language format, translates it into a host system command or VL expressions, and makes it available to the host system. The system serves as an interface to a class of systems using the variable-valued logic formalism described in Section 2. The two major components of the system are the PARSER and the VL Generator. It is in these components where the majority of the processing occurs.

3.1. System Overview

Figure 2 depicts the overall architecture and emphasizes the major components and their interactions. The modular design of PARALINC was adopted in order to maintain the highest degree of system flexibility. The major components of the system consist of both data files and processing components.

Individual data files exist for the *GRAMMAR*, the host system *PRAGMATICS*, and the *DICTIONARY*. The GRAMMAR file contains the definitions of the structure of the input language. Grammars used in this program can be either simple *context-free* grammars or context sensitive grammars. Even though a grammar may be simple, it is often broad enough to describe the majority of any input form for this system. The PRAGMATICS file contains the detailed knowledge about the host system including its command language and input grammer. Typical inputs to a system consist of both input data and the host commands. The PRAGMATICS file is used by the Command Processor to distinguish between these types. The final input file is the DICTIONARY. The DICTIONARY contains the descriptions of all words and domains that are processed by the system. A detailed description of these files will be discussed in the next section.

The major processing components are the Parser, the Interactive Dictionary Editor, the Command Processor, and the two output generators, the Host System Generator, and the VL Generator. A logical flow through the system begins when the user supplies a natural language input to the Parser. Together with the grammar and the dictionary entries, a parse of the input is performed. The parsed input is then made available to the Command Processor. The Command Processor determines whether the input is a

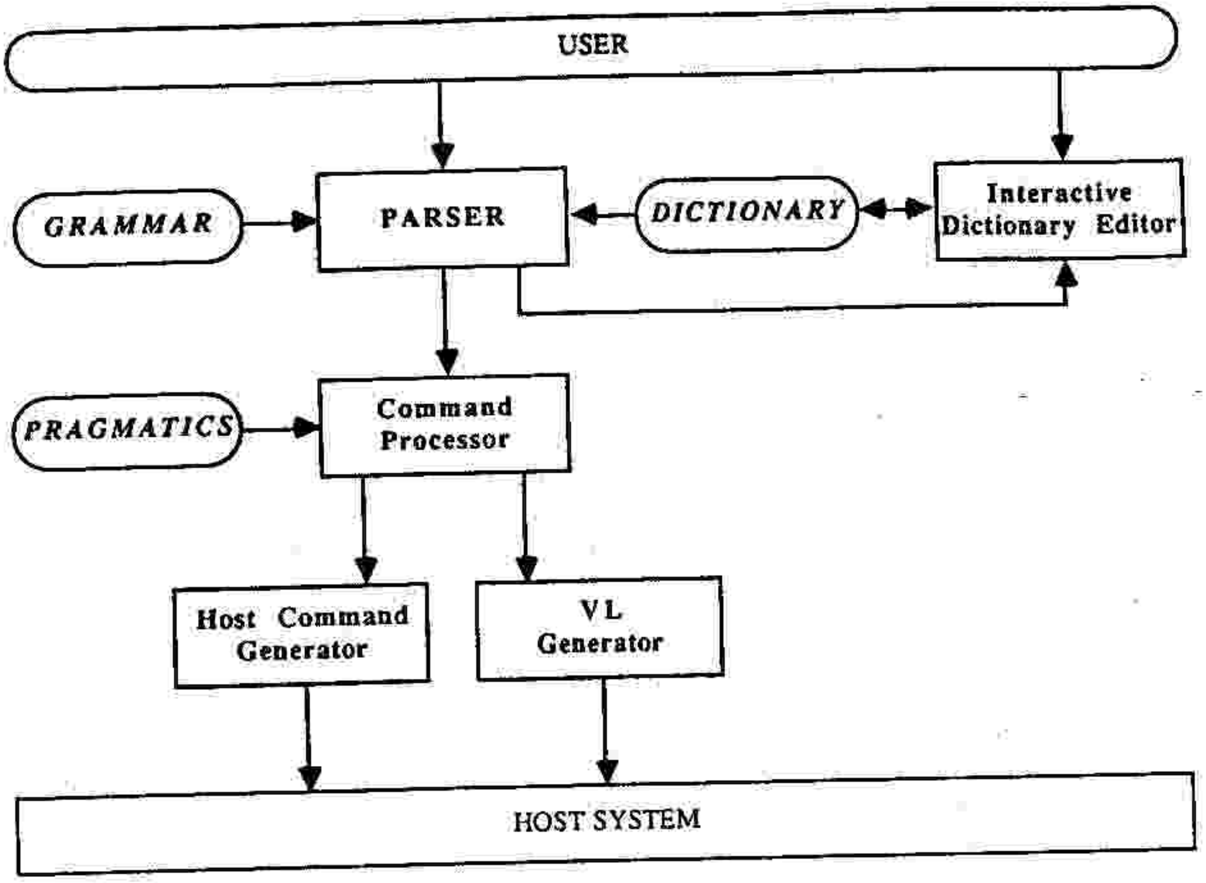


Figure 2. The PARALINC Architecture

host system command or input data in VL form. Having made this decision, the appropriate module accepts the parse and generates the input to the host system. More detail on the process of each of these components will be presented in subsequent sections.

3.2. The Dictionary and the Interactive Dictionary Editor

The dictionary maintains the definitions of each word of the input form. The definition of each word contains a set of information required by different components of PARALINC. The knowledge required by the parser is of course dependent upon the parsing technique and may actually dictate the number and type of slots within each dictionary element. Figure 3(a) presents the skeleton form for a definition used in the PARALINC system. Figure 3(b) contains an example definition for a sample word. The *root* and *variants* slots appear in all dictionary entries. The variant to a word may be the root in past tense, plural form, etc. When a syntactic parser is used, the lexical category or categories (the *lex* slot) of the word is required.

Another slot within the dictionary entry is the type or *domain* slot. This contains information about the word and how it maps into a typed hierarchy or set membership relation. The type slot of the word is used during the construction of the VL forms and performs much the same function as the domain description associated with an APC selector annotation (described in Section 2.1.2). As an example, the words SQUARE, ROUND, and RECTILINEAR may be in the domain SHAPE. The *vl-form* slot within each definition contains a template of the VL expression that has a root in this word. Arguments (selector references and referees) within this template are handled in a similar manner.

An additional slot of a word structure, the *typo*, contains common misspellings for the word. This provides for a simple error correction process for any known misspellings. One of the features which many feel is essential for a friendly, robust interface is to accept slight variants to a correct form [Waltz, 1983]. The correct form may be violated in two ways. A simple violation may occur if one or more words are misspelled. This violation can be overcome if the offending words are replaced with words that are known to the system. The other violation may occur because of missing sentence components. One example of an improper structure form involves the use of ellipsis. Ellipsis is a technique in which the user drops parts of a sentence assuming a substitution of a like form from a previous sentence. This problem has intra-sentence impact and is best handled by the combination of the syntactic parser component, a semantic analyzer, and a history of the input. Although simple, the approach used for misspellings can resolve a

DICTIONARY ELEMENT STRUCTURE

root	*- root word of definition*
variant	*- variants of word (e.g. past tense, etc.)*
synonym	*- synonym for root word*
typo	*- typical misspellings*
lex	*- lexical category(s) fo word*
domain	*- domain description of word*
type	*- data type (e.g. linear, nominal, etc.)*
vl-form	*- VL form word takes on*

(a)

DEFINITION for USHAPE

root	: ushape
variant	: (ushaped)
synonym	: ()
typo	: ()
lex	: adj
domain	: SHAPE
type	: nominal
vl-form	: [Shape(*obj*) = ushape]

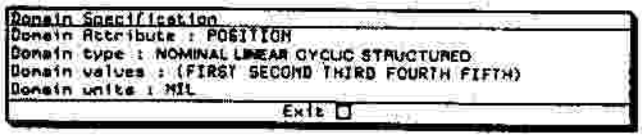
(b)

Figure 3. Dictionary Data Structure and Example

large number of errors that would otherwise lead to a nonrecognizable expression. The *synonym* slot of the structure serves much the same purpose as the typo slot differing only in relative priorities.

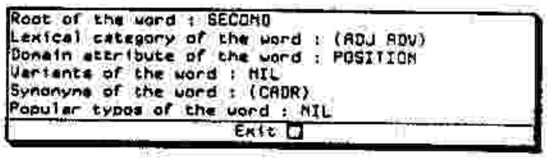
Dictionary and domain objects are defined as structures in the PARALINC implementation. The domains are maintained on a list, the dictionary elements are maintained in a hashtable. The use of a hashtable is well suited for the saving and retrieval of dictionary elements. When a new word is created, an element is added to the hashtable keyed off of the root of the word. In addition, redundant definitions are added for each of the entries in the synonym, variants, and typo slots. In these cases, an entry is created only if another definition having a root as that word does not already exist. If a variant was added and a root word is later added, the new word would overwrite the variant of the previous word. This approach ensures that variants are of lower priority than other root words. A hashtable implementation also enables a consistent retrieval time for all dictionary elements during parsing or IDE interactions.

The Interactive Dictionary Editor (IDE) component is a facility which allows the user to create, inspect, alter, save, and retrieve dictionary and domain elements. In order to emphasize flexibility and user friendliness, this component takes advantage of various interactive techniques such as pop-up menus and multiple window presentations. Figure 4 presents two menus accessible from the IDE. Figure 4(a) shows the variable-value window used to edit and create domain descriptions. Figure 4(b) is a variable-value menu used for creating and editing dictionary entries. An added feature of the dictionary editor component is the ability to store and retrieve dictionarys and domains from disk files. In this way,



```
Domain Specification
Domain Attribute : POSITION
Domain type : NOMINAL LINEAR CYCLIC STRUCTURED
Domain values : (FIRST SECOND THIRD FOURTH FIFTH)
Domain units : NIL
```

(a)



```
Root of the word : SECOND
Lexical category of the word : (ADJ ADV)
Domain attribute of the word : POSITION
Variants of the word : NIL
Synonyms of the word : (CADR)
Popular typos of the word : NIL
```

(b)

Figure 4. Domain and Dictionary Specification Menus

individual information can exist for either different problem domains or different host systems.

3.3. The PARSER

Nearly all natural language systems must process their input with a parser. The natural language parser is the first major component in the process of transforming natural language into VL expressions. the parsing process of the system was chosen carefully because it is one of the key components of the PARALINC system. The parser for PARALINC was selected after examining issues of natural language parsing such as the recognition of incomplete sentences and the efficiency of various parsing strategies.

3.3.1. Some Issues of Natural Language Parsing

When any natural language processing system is discussed, one can never escape the issues of *syntax* and *semantics* as well as bottom-up versus top-down parsing. A syntactic parser recognizes a sentence by giving the form a *structure*. The parser operates using the structure or grammar of the language. A grammar is a set of rules which specify legal combinations of words or constituents into other constituents. Syntactically, words are defined as belonging to one or more lexical categories or as described in English textbooks, *parts of speech*. Semantic parsing is described as giving a sentence *meaning*. The focus is not placed as much on the legal structure but rather the meaning behind the sentence.

Although reliable, complete systems have been developed using only one of either technique, it has been found that the combination of syntax and semantics overcomes many deficiencies associated with each individually. One of the reasons to use a combination of both syntactic and semantic components centers around the efficiency of the parser. The LUNAR system [Woods, Kaplan, and Nash-Webber, 1972] employed a semantic post-processor to validate the syntactic structure. The SHRDLU system [Winograd, 1972] also processed the input by examining it syntactically, however, a semantic component was called upon to examine the intermediate structures for *semantic validity*. This interleaved approach was used both for validation and efficiency. These systems like many others used the combined approach for efficiency. Marcus [Marcus, 1980] takes this one step further by stating that the combination is essential for deterministic parsing of English.

If a syntactic parser is preferred, a choice must be made between a purely top-down, a purely bottom-up, or a combination of both bottom-up and top-down parsing. A top-down parser begins the process by examining the structure to determine if a top level constituent has already been recognized. If the highest level constituent does not currently exist, the components required to construct the higher level are determined. The process recurs on each of these components until ground elements or *facts* are recognized. In the case of natural language, facts within this process are assigned at the single word level. Intermediate levels contain descriptions of constituents such as noun phrases, verb phrases, etc., and the top level constituent is most often the sentence. Upon returning, the lower level constituents are assimilated into successively higher levels, eventually achieving the highest level constituent. A comparison can be drawn to the production system control structure within expert systems. The top-down process is similar to *backward chaining.*

Even in simple grammars it can be seen that a number of alternate rules exist for each constituent. In a pure top-down approach, alternatives are explored depth first. If a rule is unsuccessful at the ground level, a process of *backtracking* must occur before alternate rules are explored. Backtracking effectively forgets any work that has already been performed. An additional drawback to the top-down parse is that incomplete forms, those not returning to the highest level, would never be found by this process. The parser would eventually halt without finding any parse even though a complete parse may be as little as one word away.

The bottom-up parser starts at the lowest level or ground elements and constructs more encompassing constituents. This approach is analogous to a *forward chaining* rule system which exhibits a data-driven behavior. This combining process continues until no new grammar rules apply to the current set of constituents.

As with the top-down parse, the bottom-up parser exhibits some drawbacks. In the case of the bottom-up parser, much processing time may be expended building constituents that may never contribute to a successful parse. But unlike the top-down approach, intermediate constituents are available in the event that the parse does not complete successfully. This makes it possible for a semantic post-process to fill in the gaps.

The distinction between top-down, bottom-up and the saving of intermediate results is not as clean as described. In many cases the saving of intermediate results is a by-product of the representation scheme as must as the process. Here is where the advantages of a chart-based parser is realized.

Issues such as these have been a matter of contention for as long as natural language processing research has been performed. These issues have been iterated by many over the years, but there has never been an agreement on the preferred approach. A number of systems have been designed and developed using any combination of approaches. Winograd [Winograd, 1983] has a volume dedicated to the syntactic nature of language. The second volume will address the semantic component of language.

3.3.2. The PARALINC Parser

Having reviewed many of these issues, it appears that the choice of parsing strategies depends on factors such as the intended user, the rules of interaction for the system, and the extent of the vocabulary. In this thesis, I describe one approach for the PARSER component. This approach makes use of bottom-up a chart parser.

The chart parser was chosen for the initial implementation of PARALINC for a number of reasons. First, the usual inputs to the systems of interest are terse, single sentences, rather than a sequence of sentences or paragraphs. References made to previous sentences are minimal. The syntactic approach is both simpler and effective for this type of interchange. Secondly, a logic like expression presented in natural language has a tendency to be rather constrained and syntactically correct. The semantic processing of the input form is introduced at the time of VL generation. Although one approach was chosen, the modularity of the design creates a high degree of flexibility over the choice of parsing strategies. Various parsing systems such as chart parsers, ATNs [Woods, 1970], shift reduce parsers, and MARCUS parsers [Marcus, 1980] can all be "plugged" into the current architecture.

The PARSER is composed of two submodules, the first is the pre-parser. This component utilizes the dictionary and the user input to create the internal structure. The second component operates on this structure with the aid of a grammar as it builds a parse of the input. An example grammar appears in Figure 5. Figure 6 expands upon the detail of the PARSER component. The primary output of the PARSER is the chart and its annotated parse trees. In addition to these inputs and outputs, the parser is also coupled to the Interactive Dictionary Editor. When an unknown word is encountered in the input, the user has three options in which to proceed. The first is to call upon the IDE to allow the user to define the new word. Following this process, the parse continues. The second alternative prompts the user for an alternate word. This option may be used in the case that the word was an unknown misspelling for a known word. The last option is to abort the parsing process and return to the top level of PARALINC.

The chart parser makes use of a *chart* for storing intermediate parses. This approach enables the parser to make use of constituents that have already been derived. The storage eliminates the time

SS	<-	S
SS	<-	S *conn* SS
S	<-	NP VP
NP	<-	*det* NP1
NP	<-	NP1
NP1	<-	*noun*
NP1	<-	ADV-ADJ NP1
NP1	<-	ADV-ADJ *conn* ADV-ADJ NP1
NP1	<-	NP1 PP
ADV-ADJ	<-	*adj*
ADV-ADJ	<-	*adv* ADV-ADJ
PP	<-	*prep* NP
VP	<-	*verb*
VP	<-	*verb* *adv*
VP	<-	*verb* NP
VP	<-	VP PP

Figure 5. A Sample Grammar for PARALINC

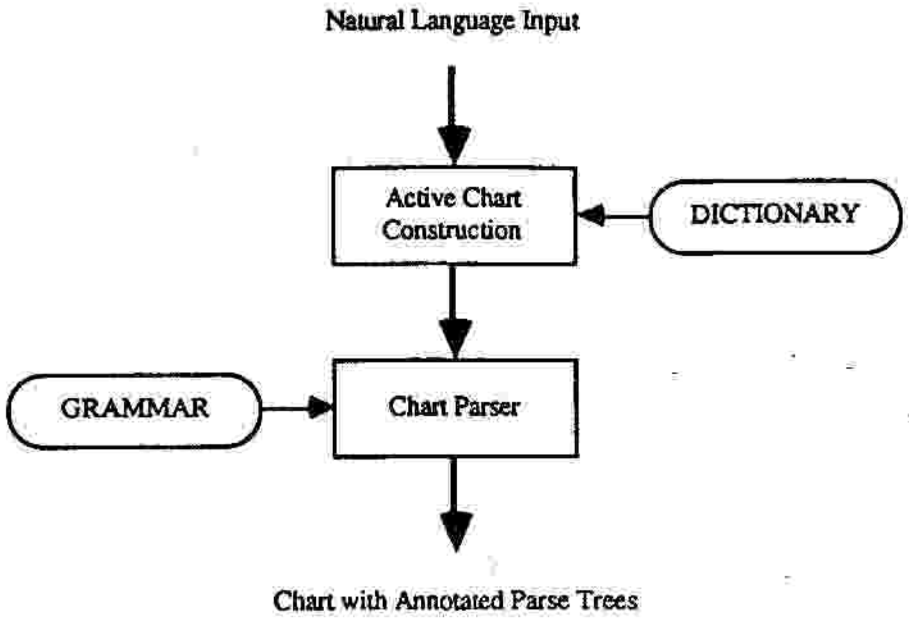


Figure 6. The PARSER Functional Components and Files

consuming and wasteful process of redoing work that has previously been performed. Consider how wasteful it may be to almost fully construct a higher level constituent only to fail and have to potentially rediscover a majority of the constituent again when approaching the problem using a slight variant of the failed grammar rule. A chart parser has an obvious advantage over classic backtracking systems which may repeatedly rederive a constituent. With the use of a chart, constituents may already be available during alternate parses.

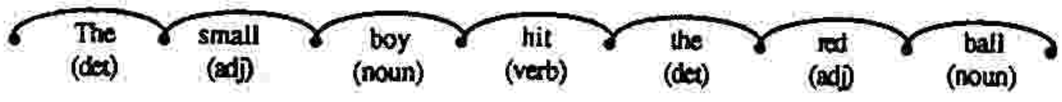
Because natural language is inherently ambiguous, many parsers must judge the goodness of the partial parse in order to determine what next to explore. This interpretation process is interleaved with the parsing process, potentially judging goodness with incomplete information. An approach to disambiguation is the *delay* or *wait and see* parsers such as ATNs [Woods, 1970] or Marcus style parsers [Marcus, 1980]. These parsers maintain intermediate constituents internally until no ambiguity remains. The constituent is then added to the result. The chart parser offers an alternative to this approach. Rather than judging goodness during parsing, all legal parses are found and stored within the chart data structure. The judgement of goodness is deferred to a post-parse semantic review. This approach is particularly suited for the PARALINC system because the semantic interpretation of the parses is inherent in the construction of the VL expressions.

3.3.3. The Chart Parser Chart Structure

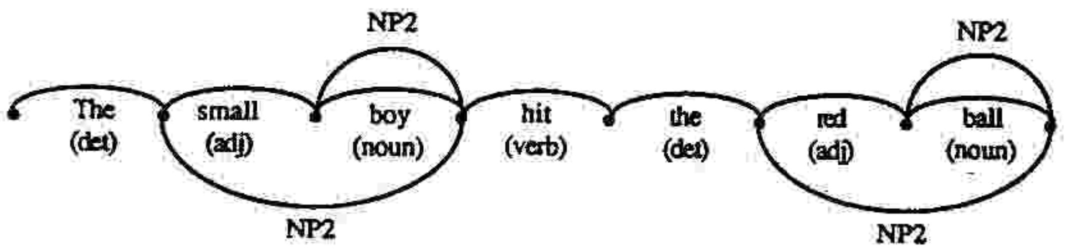
The chart parser is based on the data structure used during the parsing process. The chart is a graph with a set of *vertices* and *edges* or *arcs*. The vertices represent points in the sentence with unique points at the start and end as well as points between each word. The edges of the chart represent the constituents of the sentence. As new constituents are found, additional edges are added to the chart. The initial chart contains only vertices and the edges corresponding to the lexical categories for individual words. Figure 7(a) depicts an initial chart for a sentence.

3.3.4. The Chart Parser Process

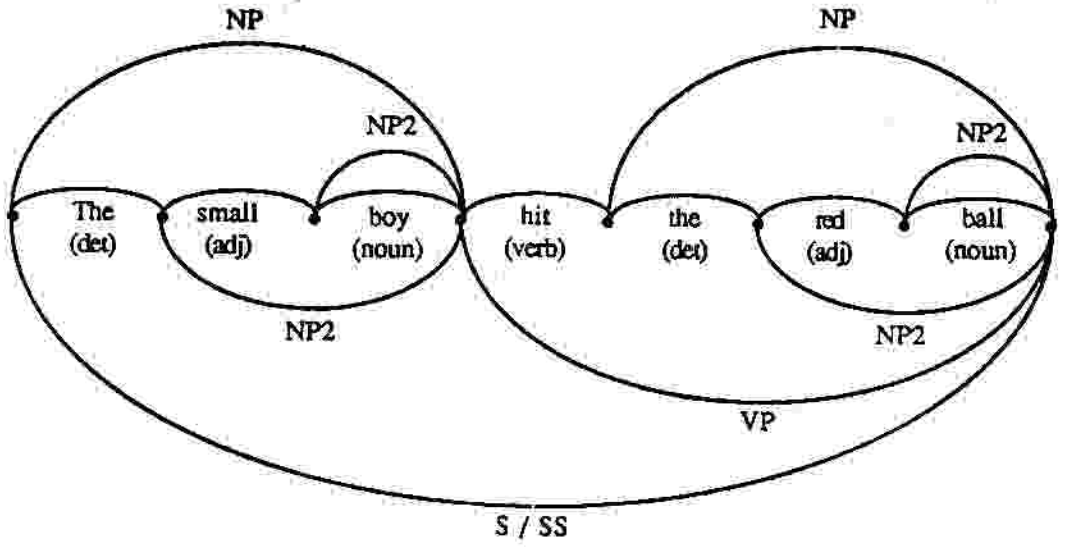
The initial chart is the starting point for the the parsing process. The chart parser executes by examining the grammar and the chart in order to determine if any constituents can be combined into higher level constituents. If a higher level description is found, the edge is created for that constituent and attached to the chart at the appropriate vertices. Figure 7(b) shows the same chart with some intermediate constituent edges added (note that the grammer presented in Figure 5 is the grammar used in this example). Figure 7(c) contains the complete chart at the conclusion of the process. The final chart is displayed graphically on the interface screen of the PARALINC system. The detailed algorithm of the chart parser is presented in Figure 8.



The initial chart
(a)



Partial chart
(b)



Complete chart
(c)

Figure 7. The Active Chart at Stages of the Parser Operation

```
PARSE ( active_chart , grammar )
begin
    match ::= match_grammar ( grammar , active_chart )

    if match then
        new_edge ::= create_edge ( match )
        update_constituents_in_edge ( new_edge , match )
        active_chart ::= add_edge ( new_edge , active_chart )
        PARSE ( active_chart , grammar )

    else
        RETURN
end
```

Figure 8. The Chart Parser Algorithm

The actual parsing process operates much like a production system where the grammar functions as the rule set and the application of a rule creates a new constituent and attaches it into the chart. In this implementation, the parse is performed bottom-up and terminates when no grammar rules are found to apply to the current chart. The grammar rules are maintained in a discrimination net. During the parsing process, a pointer follows the branches of the net until terminal or *value* nodes are encountered. A value node contains the constituent formed when that node of the network is reached. Figure 9 contains both the LISP form of the network and the graphical form. Note that the outlined nodes of the net represent value nodes in the grammar. Additional descriptions of chart parsers can be found in a number of texts including [Varile, 1983] and [Winograd, 1983]. Both of these texts describe the chart as a storage of intermediate results, however, the parsing strategy is most often presented as top-down.

```
(NIL
  (S (VALUE SS)
     (CONN
       (SS (VALUE SS))))
  (NP
    (VP (VALUE S)))
  (DET
    (NP1 (VALUE NP)))
  (NP1
    (VALUE NP)
    (PP (VALUE NP1)))
  (NOUN (VALUE NP1))
  (ADV-ADJ
    (NP1 (VALUE NP1))
    (CONN
      (ADV-ADJ
        (NP1 (VALUE NP1)))))
  (ADJ (VALUE ADV-ADJ))
  (ADV
    (ADV-ADJ (VALUE ADV-ADJ)))
  (PREP
    (NP (VALUE PP)))
  (VERB (VALUE VP)
        (ADV (VALUE VP))
        (NP (VALUE VP)))
  (VP
    (PP (VALUE VP))))
```

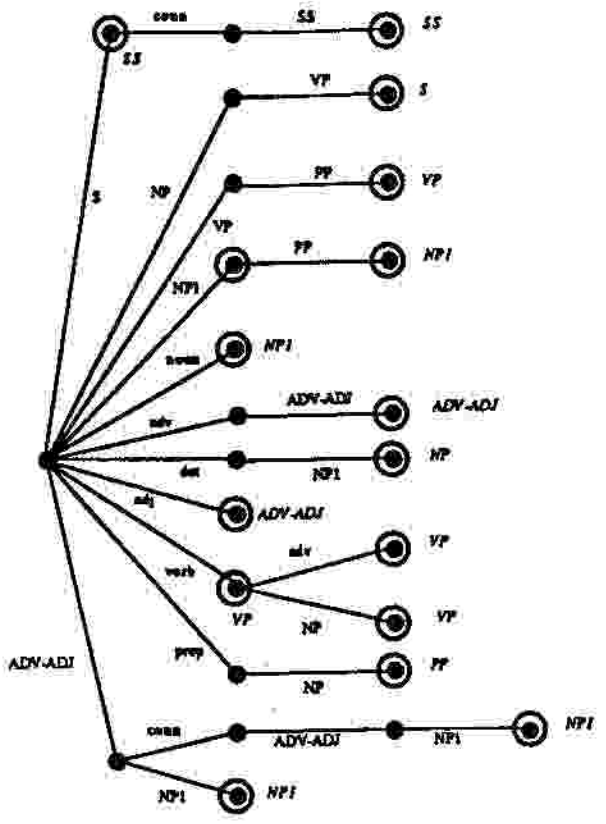


Figure 9. The Grammar and Discrimination Network

3.4. The COMMAND PROCESSOR

The Command Processor (CP) is the first component of the system which operates with the knowledge of the host system. The parse trees, which are maintained as an annotation of the chart edges, are the input to this component. The output of the Command Processor becomes input to either the Host Command Generator or the VL Generator, the choice is made as part of the internal process of the CP. The sole function of the Command Processor is to determine if the user supplied input is a host system command or input data for the host system.

The recognition of a paraphrase of a host system command often proves to be a difficult task. The difficulty of this process lies in the fact that the PARALINC system has been developed for different host systems. In addition, the development of PARALINC occurred after the development of most of the host systems, therefore, each of the host systems has a potentially different command language, each with its own quirk and nuances. Being more of a engineering issue rather than a research issue, the requirement to interface to a number of different systems was best handled by utilizing a simple pattern match between the specification of host system commands and the input.

In order to determine whether the input is a host system command, the CP must have knowledge about all host commands which may be supplied to the system. This is the information that is maintained in the PRAGMATICS file. The pragmatics file contains information about what the system should recognize and what the system should generate. Figure 10 contains examples of some of the relevant pragmatics associated with the INDUCE system [Hoff, Michalski and Stepp, 1983] and a description of their function. The descriptions accompanying the pragmatics are useful because they supply semantic information abouttthe host system commands much like the vl-form of a word definition supplies semantic information about the words of a paraphrased VL expression. These descriptions may be of use when the CP must decide on the type of input.

If the input is recognized as a paraphrase of a VL expression, the input is gated to the VL Generator. If the input is recognized as being a host system command, the input and the appropriate pragmatics are supplied to the Host Command Generator.

3.5. The HOST COMMAND GENERATOR

The style and type of the commands generated for each host system differ and as such require a different predefined model of the commands. The Host Command Generator is the module which generates these commands for the host system.

The commands to the system are typically short, abbreviated imperatives such as "set *parameter* to *value*", "run", "quit", etc., or interrogatives that either elicit help from the host system or retrieve valuss of particular parameters. The Command Processor performed a simple matching process in order to

High Level System Commands	Description
H	*Get HELP*
M	*Modify rule base*
L	*Enter logically derived descriptor*
E	*Enter domain generalization structures*
A	*Enter an arithmetic derived descriptor*
N	*Add arithmetic derived descriptor to rule base*
X	*Enter a logical derived descriptor into rule base*
C	*Cover a set of formulas*
V	*Enter VL_1 mode*
P	*Enter parameter examination and modification mode*

Figure 10. Example High Level Commands for the INDUCE System

determine which of these commands was specified by the user. This matching process is driven by keywords in the input that might suggest a system command. The Host Command Generator creates the complete command by filling in information such as parameter values.

In addition to the regular command set, many systems require an initialization process to be run at start-up time. The pragmatics file also contains this information. When the PARALINC system is first initialized or when the user wishes to reinitialize the host system, the host system initialization information defines what host system initialization process must be performed.

In addition to accepting paraphrased system commands in natural language, the PARALINC system takes advantage of additional features available on workstations by providing a host system command menu. Research in intelligent man/machine interfaces has soown that users may prefer menu-oriented inputs for simple commands or for parameter selection. If this approach is adopted as the only means of specifying host system commands there would be a clear delineation between the host system pragmatics and the input data to that system. This approach would eliminate the CP and allow the parser output to go directly to the VL generator. Any system command or parameter specification would be performed through a user/menu interaction.

3.6. The VL GENERATOR

The generation of VL expressions from the natural language input has been the focus of the effort in PARALINC. The VL Generator is the component of the system which implements this process.

The VL Generator is the component of the system which transforms the parsed input into an expression or expressions in a variable-valued logic language. Together with the parse tree obtained from the parser and the dictionary elements within that tree, the VL generator creates the expressions. In general, generation rules exist for natural language phrases such as verb phrases, adjectives, adverbs, prepositional phrases, and sentence forms, but the rules of generation and combination are dependent on the grammar and word definitions. There are a set of rules associated with each grammar set which describe this process. These sets of rules are loaded with the grammar at grammar initialization time.

The multiple parses which may exist following the chart parser operation are interpreted semantically by the VL Generator as part of the translation process. Any ambiguity is resolved by determining the VL interpretations that satisfy the vl-forms within the parse tree constituents. The result of this process is a set of VL expressions which semantically describe the input.

3.6.1. VL Generation

The translations process is driven by the dictionary entries for each word. Annotated within the dictionary entry is a slot which describes the type and form of the VL expression (described in Section 3.1). The vl-form slot contains a template of the corresponding VL expression. For example, the dictionary entry for the adjective "small" contains the VL-form:

$$[\mathrm{Size}(\mathit{object}) = \mathrm{SMALL}].$$

The functor *Size* represents a linear domain having a value of *small*. In addition, type descriptors for the references and referees of the selector are specified in the selector form. These types are used to determine legal values for each parameter. In this example, a word being modified by "small", the parameter of size, must be of type *object* in the type hierarchy. The word definition contains a type description which is used to perform these tests. The VL Generator constructs the expressions by first retrieving these templates and based on the requirements for each variable and value within the template, obtains the fillers from other words or phrases. The rules accompanying the grammar describe this process. The validity of any VL form that is generated is determined by the template matching process. The domains and properties of the words ensure correct values.

The lexical categories and the semantic interpretations of the words dictate what are the templates used for each word. The lexical categories and grammatical phrases of particular interest in the generation of a VL expression are the noun phrase, the verb phrase, and the prepositional phrase, as well as adverbial phrases and adjectives. This approach based on lexical categories is similar to the approach by McCord [McCord, 1985] who addresses the role of lexical categories in the generation of LFL expressions. LFL is a predicate logic used as a semantic representation language for natural language.

The information embodied within these templates and the grammar combination rules can be compared to the conceptual dependency (CD) [Schank and Riesbeck, 1981] language that was developed to capture the meaning of natural language exchanges. It is similar to the APC representation although somewhat less formal. Through the introduction of predicates such as PTRANS, MTRANS, ATRANS, etc., a typed hierarchy has been introduced where the selection of the desired CD is actually the search for the domain of the set which describes the particular word or phrase. Consider the sentence:

The man ate the cake.

This would translate into the CD form *(INGEST ...)*. Likewise, the same CD form would result for the sentence *The man drank the milk.*. Within the PARALINC system, a similar process occurs. The templates within the words contain VL expression descriptions. In the case of a word which represents a predicate or function, the domain of the word is used as the actual functor. This is a mapping much like the word to CD mapping. Charniak and McDermott [Charniak and McDermott, 1985] discuss CDs in light of a logic formalism. They point out that CDs are actually a logic formalism wrapped in a nonformalistic approach. They call one representation a notational variant of the other.

Subsequent subsections will discuss the role of a number of lexical categories in the construction of VL expressions.

3.6.2. Noun Phrase Forms

Nouns in the input sequence correspond to terms in APC. Section 2 described terms as being elementary and compound. Elementary terms are constants, variables, or function symbols followed by a list of e-term arguments. Compound terms are composites of e-terms where the combining operations are either the conjunction or disjunction of terms. Nouns are at the root of many of the constituents such as noun phrases and prepositional phrases and function as terms in the VL form.

Terms in the APC expression are typically used as arguments to predicates and values of selectors. The connectives found in the grammar rules for noun phrases correspond to "and" and "or" combinations of nouns, proper nouns, or noun phrases and are an indicator of a compound term. If a noun phrase constituent contains more than one noun phrase combined with either "and" or "or", the list of nouns must be reorganized to reflect the precedence of the conjunctions and disjunctions. The conjunctive operator is defined as having a higher precedence than the disjunctive operator, thus, the list appears as a single disjunction of one or more conjunctions. This ordering is consistent with the description of the compound predicate reformulation rules and compound term rewrite rules of Section 2.1 where it was noted that this order represents the more natural ordering exhibited by people. For example, the partial sentence:

"*... the ball and the box or the block or the pyramid ...*"

would be grouped as:

```
(<or>
 (<and> "the ball" "the box")
 (<and> "the block")
 (<and> "the pyramid"))
```

Most dictionary entries for a noun do not have APC expressions associated with them but rather assume roles in other expressions.

3.5.3. Adjective and Adverbial Forms

The lexical categories of adjective and adverbial modification of an adjective are of importance in selector construction. A noun may be modified by an adjective or adverb adjective combination. An adjective, along with the noun being modified, creates a selector that describes the relationship. Although the number of modifiers of a noun is arbitrary, the system treats each adjective and adverb individually and generates a selector of the form:

$$[\mathit{modifier\_domain}(\mathit{modified\_noun}) = \mathit{modifier\_value}].$$

The sequence "red ball" would be represented by the selector:

$$[\mathrm{Color(BALL) = RED}]$$

A sequence of two or more modifiers of the same noun would generate a conjunction of selectors, each representing a distinct modifier. Consider the sequence "large blue box". The resulting selectors would be:

$$[\mathrm{Color(BOX) = BLUE}]\ \&\ [\mathrm{Size(BOX) = LARGE}].$$

The only form of consistency testing currently provided by the system is for redundant selectors. For example, the sequence "red red box" would generate only one selector. However, other semantic discrepancies may arise. Consider the sequence "red blue box". The selectors generated would be:

$$[\mathrm{Color(BOX) = RED}]\ \&\ [\mathrm{Color(BOX) = BLUE}].$$

This seems contradictory and indeed it is, but the system makes the assumption that the user input is accurate. The alternative is for the system to detect similar[2] selectors and either choose one by some predefined priority or prompt the user for either one or both to be included in the final expression. The assignment of priorities may be part of the annotation of the form. The priority may describe whether different values are legal within one expression or that some values are mutually exclusive. This solution, although more flexible, requires user intervention or the specification of the priorities.

[2] A selector is similar to another if the functor definition, the number of arguments, and the domains of the arguments are the same but the argument value(s) are different.

3.6.4. Prepositional Phrase Forms

Because of the many roles which a prepositional phrase can play, additional semantic interpretations must be made in order to determine how the prepositional phrase is used with respect to the VL form to be generated. Most often, prepositions describe a relationship between objects. But the relationship must be understood before the VL form can be created. Consider the two phrases with embedded prepositional phrases:

the ball on the box and *the size of the ball ...*

In the first example, the preposition is describing a relationship between to objects. In the second example, the noun within the prepositional phrase is actually the root object and the word being modified is an attribute of the object with an as yet unspecified value. Forms such as these must be recognized during VL generation. An example recognition rule is one which examines the prepositional phrase for adjective like descriptors:

```
IF
$prep$ = "of" and $noun_2$ has an attribute of $noun_1$
THEN
VL form is $[Noun_1(noun_2) = object]$.
```

Forms which describe relationships take the form of predicate selectors in APC. A predicate selector is a selector in which the value is not explicitly given but is assumed to be boolean *TRUE*. As can be seen by the above example and the grammar, prepositional phrases can either be attached to the nearest noun phrase, to the verb phrase, or can function as an adjective/prepositional phrase combination that describes a linear domain range. The prepositional phrase to linear domain specification and the prepositional phrase modifying the verb will be addressed as part of interpretation of sentence forms.

When a prepositional phrase modifies a noun phrase a relationship is described. A predicate is constructed for this relationship which takes the form:

$[predicate\_form\_of\_preposition\ (noun_1, noun_2)]$.

Like other word definitions, the selector associated with the preposition is part of the vl-form slot of the preposition and the VL predicate form describes the legal modifiers which can act as arguments to the predicate. For example, the prepositional phrase "on the box" in the phrase "the ball on the box" would describe a predicate of the form:

[Ontop(BALL, BOX)].

In the case of prepositional phrases, prepositions have different meanings depending on the word being modified and the noun phrase that is part of the prepositional phrase. A test is generated to determine what is the correct form based on actual words and word domains. This test is performed before a predicate or any other selector form is constructed from the prepositional phrase.

3.6.5. Verb Phrases and Sentence Forms

The rules that correspond to the grammar for adjectives, adverbs, and prepositional phrases are rather straight forward. Any of the validity checking that is performed is done during selector generation or through a simple rule. In the case of verb phrases and sentences, the rules become more complicated because of the nature of the VL expressions and the varying forms which can be generated at higher levels in the grammar.

The transformations described above show how selectors are constructed for specific lexical categories or phrases which are components of a complete sentence. There is also potential for VL expression construction for the sentence. The elements of interest in this process are the list of noun phrases for both the object and the subject of the sentence, the verbs and adverbs, the prepositional phrases which modify verbs, and the adjective preposition combinations which describes a numeric quantifier or numeric domain. In addition, determiners, articles, and certain connectives are processed at the sentence level. The processing of the sentence may create selectors much like the lower level phrase processing but may also create higher level *rules* and *complexes* describing the sentence.

The recognition of APC quantifications occurs at the sentence level. Example quantifications may be the universal quantifier or the existential quantifier, as well as the APC specific numeric quantifiers. Universal or existential quantifiers are found in phrases such as "for all ...", "there is ...", or "all ...". In order to establish numeric quantifiers phrases such as "six balls bounced" or "three to five boxes are on the table" must be recognized as numeric type specifiers. The recognition of these forms is similar to the recognition of object/attribute pairs in prepositional phrases. The only difference being the object of modification. The parse must be examined for words in the numeric domain and sequences such as "*a* to *b*". Section 2.1.1 described the numeric quantifiers as well as presenting examples of each in natural language. An example of a sentence which has a linear domain component is:

"*The waves are six to ten feet high*".

As with other representations such as CDs, the construction of many expressions are verbal-based. The relationship keys off of the verb and draws from the additional information provided by the other sentence elements. Action verbs, like adjectives or prepositional phrases, have a domain associated with them. The processing of input forms which contain verbs is similar to the process of VL generation for adjectives or prepositional phrases. A predicate or selector results from this process. For example, the sentence "The boy hit the large ball" would translate into the two selectors:

$$[\text{Size(BALL)} = \text{LARGE}]\ \&\ [\text{Struck(BOY, BALL)}]$$

verbs of the form *be* often play the role of assignment or a specification of a value for an attribute. Consider the sentence "The ball is red". The verb "is" indicates that this an assignment. The domain of "red" and the noun "ball" indicate that this form should be treated exactly like the sequence "the red ball" and produce the identical selector.

Connectives such as "and" and "or" were discussed as part of the noun phrase descriptions. In addition to this, "and" and "or" are also part of a set of connectives which modify sentences and have a direct relationship with APC rule and complex specifications. Additional connectives that fall into this category include "but", "however", "although", and "in addition". The VL generation process recognizes these forms and creates the correct VL forms in each case.

Determiners such as "what" in the input suggest that the form may be a host system command. Determiners such as "If" or "when" may be either a host command or the start of a rule description. The VL generator examines the parse to intercept these keywords and to process them accordingly using semantic review of the remainder of the parse.

3.7. The PARALINC Implementation

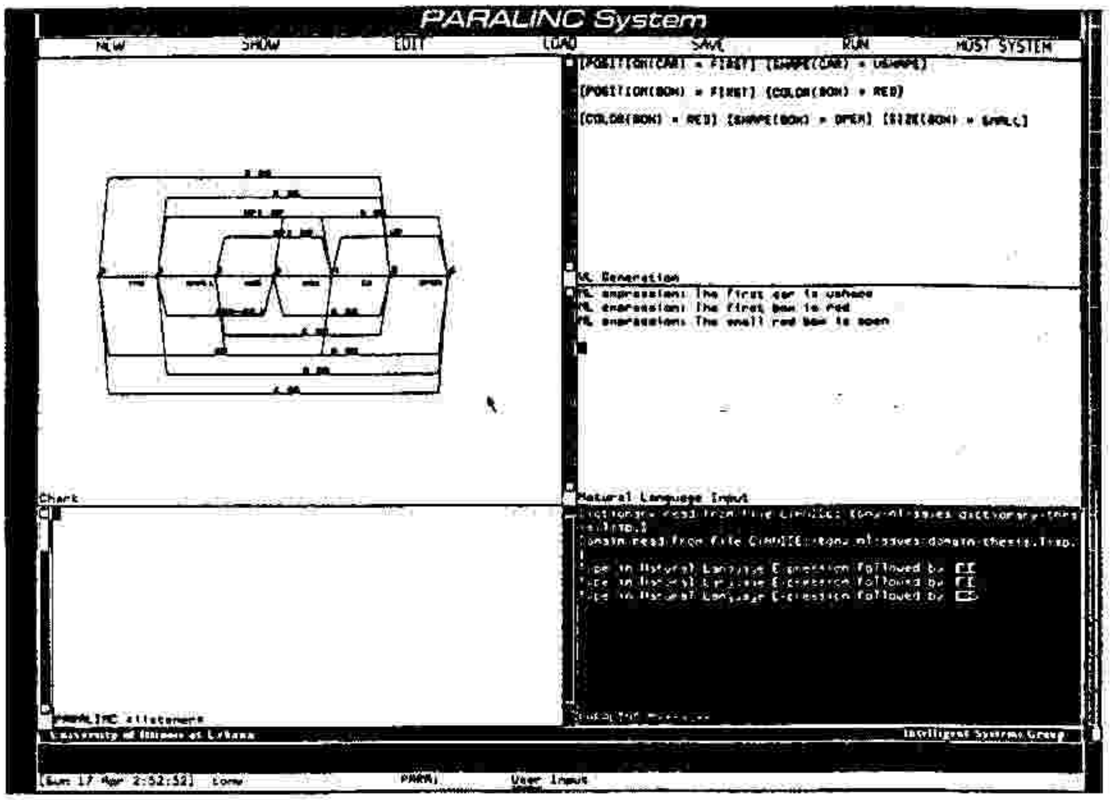
The PARALINC system is a Common Lisp implementation of the theory of transforming natural language input into VL expressions. The system was initially prototyped in Franz Lisp on a Sun Microsystems workstation. The current system runs on a Symbolics Lisp Machine and takes advantage of the many interface utilities offered by that system. The interactions with the system are performed through a mix of menu and keyboard input. The interface provides a means for the user to create, delete, and inspect the contents of the DICTIONARY and the DOMAINS databases as well as parsing and generating VL expressions. In addition, graphical output of the parsing process is provided to the user through a visualization of the active chart. Additional windows provide a trace of the system operation. Figure 11 presents the interface to the system. Various menus which were presented in Section 3.2 are accessed through the global pull-down menu across the top of the screen.

3.7.1. Examples

The previous sections discussed the theory of transforming natural language input into VL expressions. In this section, an example interaction of the PARALINC system is given in order to demonstrate the process. The PARALINC system currently maintains data files for both the structured objects example and the TRAINS example found in [Stepp, 1984]. These examples describe objects which have attributes and interelationships. The example cited is a subsets of a popular test case for inductive learning systems like INDUCE or the conceptual clustering system CLUSTER.

3.7.2. The Structured Objects

The structured objects example contains a data set which describes the interelationships of objects such as next-to and on-top-of. Attributes of the examples include the shape of the objects, the texture, and size. The assumed host system for this example is the INDUCE [Hoff, Michalski, and Stepp, 1983] inductive learning program. Figure 10 presented a short description of the high level command that INDUCE accepts. Figure 12 presents a subset of an interaction with INDUCE during the initial data description process. The italicized comments in the right column give the original INDUCE commands



```
[POSITION(CAR) = FIRST] [SHAPE(CAR) = USHAPE]
[POSITION(BOX) = FIRST] [COLOR(BOX) = RED]
[COLOR(BOX) = RED] [SHAPE(BOX) = OPEN] [SIZE(BOX) = SMALL]
```

```
NL expression: The first car is ushape
NL expression: The first box is red
NL expression: The small red box is open
```

```
Dictionary read from file [illegible] dictionary [illegible]
Domain read from file [illegible] domain thesis [illegible]
Type in Natural Language Expression followed by [illegible]
Type in Natural Language Expression followed by [illegible]
Type in Natural Language Expression followed by [illegible]
```

```
[Sun 17 Apr 2:52:52]    PARA:    User Input
```

Figure 11. The PARALINC Interface Screen

which are paraphrased by the natural language input in the left column.

3.8. Results

Experimentation with the PARALINC system has shown varying results. The complexity of the generation process and thus the natural language understanding can be attributed to a number of factors in the PARALINC system. The transformation process begins with the parser. This is the basic and first level of understanding. The semantic component of generation operates within the scope of the results of

PARALINC INTERACTION	HOST SYSTEM EQUIVALENT
Enter domain generalization rule.	*E<cr>*
If the shape is a circle or an ellipse then then shape is curved.	*[shape=circle,ellipse] => [shape=curved]*
If the shape is a triangle or a square or a diamond or a rectangle then the shape is a polygon	*[shape=triangle,square,diamond,rectangle] => [shape=polygon]*
Add rule to rule base.	*M<cr> A<cr>*
If the size of P1 is small to medium and the size of P2 is large then D is 1.	*[size(p1)=small .. medium][size(p2)=large] => [D = 1]*
Delete rule.	*M<cr> D<cr>*
Add rule.	*M<cr> A<cr>*
If a medium diamond P1 is on P2 and P1 is clear and a medium shaded P2 is a circle and P2 is on P3 and P3 is clear and large and ushaped and P2 is in P3 then D is 1.	*[size(p1)=medium][shape(p1)=diamond] [ontop(p1,p2)][texture(p1)=clear] [size(p2)=medium][texture(p2)=shaded] [shape(p2)=circle][ontop(p2,p3)] [texture(p2)=clear][size(p2)=large] [shape(p2)=ushape][inside(p2,p3)] => [D = 1]*
. . .	
Set parameter meta to 0.	*P meta 0 <cr> Q<cr>*
Print parameters.	*P para <cr> Q <cr>*
. . .	

Figure 12. Example Interaction for the Structured Objects Example

the syntactic parse. The semantic interpretation rules are defined such that they recognize structures and word types within a known syntactic structure.

Forms which are considered relatively easy for the system to transform adhere very well to the definition of the grammar and the VL forms within the dictionary elements. The generation of VL expressions using the lower level rules such as the adverb/adjective modifications and prepositional phrases are forms which have direct, unambiguous translations. As indicated in Section 3.6, there are no specific semantic analysis rules associated with these constructs but rather all validation and interpretation resides within the construction of the VL forms. An example of a simple input successfully transformed is:

The open car$_1$ in front of car$_2$ is long.

With the resulting VL form of:

$$[\text{Infront}(\text{CAR}_1, \text{CAR}_2)] \ \& \ [\text{Length}(\text{CAR}_1) = \text{LONG}]$$

A sentence may be considered complex by a user because of ambiguity or simply because of size. The recognition of a form using a syntactic parser is not effected by the length of the input. In some cases a human may find a sentence long-winded, but the system would be unimpeded by its length. For example, a sentence composed of a number of connected forms and numerous modifiers such as:

> *The big red shaded circle is on the small smooth blue box and the tall shiny triangle is next to the box and the large open rectangle is on the circle.*

would be successfully translated as:

[Size(CIRCLE) = BIG] & [Texture(CIRCLE) = SHADED] & [Color(circle) = RED] &
[Size(BOX) = SMALL] & [Texture(BOX) = SMOOTH] & [Color(BOX) = BLUE] &
[Height(TRIANGLE) = TALL] & [Texture(TRIANGLE) = SHINY] &
[Nextto(TRIANGLE, BOX)] &
[Size(RECTANGLE) = LARGE] & [Shape(RECTANGLE) = OPEN] &
[Ontop(RECTANGLE, CIRCLE)]

Figure 12 contained an example of a long expression for the TRAINS example.

The transformation process may break down when some component of the system fails to succeed. Failures of the system may occur in two different ways. First, the sentence may fail to successfully parse. In this situation, no generation of VL expressions may be possible or the generation may be limited to subsets of the original form which represents the lower level generation capabilities. Another failure may occur if the sentence is successfully parse but the VL expression which results may not accurately reflect the users intentions. This most often occurs when the sentence is ambiguous or semantic interpretation of the form is incorrect.

An example of a sentence which may be ambiguous to many natural language processing systems is demonstrated with the use of the preposition "on". Consider the sentence:

The boy hit the ball on the hill.

This sentence is an often cited example of a sentence which can be interpreted in two ways. It is unique if the sentence read either "the boy on the hill hit the ball" or "the boy hit the ball onto the hill". The ambiguity not only occurs here but also is associated with the act of hitting the ball. In the second interpretation, the balls final resting place is on the hill, but the sentence may also be interpreted such that the ball's initial position was on the hill but the final position is unknown. Various interpretations of this sentence may be expressed as

$$[\text{Location(BOY)} = \text{HILL}] \;\&\; [\text{Struck(BOY, BALL)}]$$
$$\textit{or}$$
$$[\text{Struck(BOY, BALL)}] \;\&\; [\text{Location(BALL)} = \text{HILL}]$$

but neither has any reference to the temporally related locations of the ball before or after the hit. Temporal relations must be derived based not only on the content of the input but an understanding of that content.

Another example of a sentence which is not fully translated is:

The red car has three wheels.

The form is difficult because of the cardinal modifier of wheels. The system would correctly create the selector for "red" but would have difficulty generating a selector of the form:

$$[\text{Number\_Of\_Wheels(CAR)} = 3].$$

Such an interpretation requires the VL generator to create selectors based on multiple words within a phrase. In this example, the generation process must recognize that the combination of "three" and "wheels" describe a single selector. In general, this error will occur for all selectors which are based on a combination of descriptions, and in most cases, such as cardinals, there may be unique functors for each combination of descriptions. Each combination must be enumerated unless a more general selector is used.

such as:

$$\text{Cardinality}(\textit{object}) = n$$

It appears that the failures of the system can be attributed to either the parsing process, where a user would have control by means of augmenting the grammar, or by the VL generation process. Additional semantic processing may be necessary to overcome these generation failures through the introduction of additional VL generation rules. Input forms which have a clear VL interpretation because of grammar or dictionary semantics are the most easily recognized.

4. CONCLUSIONS

The work described here has focused on the process of creating the common language rather than providing an interface tool. The theory presented in Section 3 has been implemented and tested on a number of examples. Logic languages and in particular variable-valued logics seem well suited for representing input expressed in natural language. However, the flexibility of such a system is directly related to the capabilities and requirements of the host system. When a host system accepts a very broad range of inputs, the interface system grows in complexity. A reliable process of translating natural language to variable-valued logics has been demonstrated for systems which have a well defined command set and vocabulary.

Features of natural language and variable-valued logics were brought together in the PARALINC system because of different advantages due to each. An obvious advantage of any natural language interface is that it introduces a sense of familiarity to users who may have otherwise rejected a system. VL languages have an advantage because of comprehensibility and conciseness.

4.1. Limitations

Although there are many advantages to having a natural language interfaced coupled to a host system, there are also limitations introduced. One of the limitations which occurs with all natural language systems is the robustness of the natural language interpretation process. Limitations may be introduced because of the form or content of the expression. A form violation will occur if the user supplies either a form which is valid English but not part of the grammar definition or by using invalid grammatical forms. A solution adopted by many natural language systems is to introduce a level of semantic processing on top of the parsing process. PARALINC encodes this process with rules embedded within the grammar rule set or higher level translation rules such as those in Section 3.5.4.

The content of the form may be limited because of the finite dictionary. Simply expanding the dictionary may not be a solution to a limitation of content. Often, the dictionary has to be very specific about the host system command set and the accepted vocabulary. It is often the case that the dictionary is not only specific to each host system but must also reflect individual domains within each system. In practice, each dictionary may have to be highly tuned to each example and system. The goal of making a general interface system for a number of systems was partially met through the use of a common grammar. The PARALINC system as a whole remains in tact regardless of the host system. but the price is paid by the DICTIONARY and the PRAGMATICS files.

Natural language does not solve all of the interaction problems of an interface. Consider the expert system that has is decision rules generated using machine learning techniques. If the rules are to be considered an accurate representation for the decision class, the number of examples may be quite large. When the examples are many and the rules have a large number of parameters, the natural language paraphrasing of all of the examples may be as tedious as creating the rules in the host system language. Systems for database retrieval or interactive expert systems that tend to be highly interactive with a short exchange of information at any one time are the best candidates for a natural language interface. Examples have shown that the natural language interface did not dramatically reduce the amount of input but the input is in a form that is more familiar to a larger class of system users.

4.2. Future Directions

A logical extension to this system is to incorporate a natural language generation system. The PARA system [PARA, 1983] was an initial attempt at this but a much more robust generation capability can be created if generation works in harmony with the natural language recognition system. This approach would allow both the recognition and generation systems to access the dictionary and the associated annotations.

The addition of a word learning mechanism would add to the system's robustness and friendliness. The current approach escapes to the IDE component whenever a word is not known. An alternate approach would be to mark unrecognized words at chart creation time, but to allow the parse to proceed in the hopes that the context, both syntactic and semantic, can provide some information for the dictionary entry. As the dictionary grows, one would hope that more slots are automatically filled in.

Through various experiments with earlier versions of PARALINC, it has been discovered that the best interface for many systems should take advantage of both natural language and menu oriented inputs.

The best separation appears to be at the system command and the input data levels. Future systems should accommodate both forms of interaction.

In any natural language processing system there is always room for improvement. The grammars used by these systems can always be expanded and the ability to recognize incomplete forms can be made more reliable. PARALINC is no different in this respect. Such additions would increase the reliability of this system and make it an even more attractive interface tool. Through the use of a common interface and data representation approach, systems like PARALINC may be the basis for future VL based systems. A logical extension of a shared representation is a shared interface.

REFERENCES

[PARA, 1983]
—, "PARA: Paraphrase VL Rules," User's Manual and Internal Report of the Intelligent Systems Group, University of Illinois, Department of Computer Science, 1983.

[Aho and Ullman, 1972]
Aho,A.V., and J.D.Ullman, *The Theory of Parsing, Translation, and Compiling, vol 1*, Prentice-Hall, Publishers, 1972.

[Becker, 1985a]
Becker,J.M., "Macros and Utilities for Franz Lisp," UIUCDCS-F-85-937, ISG 85-7, University of Illinois, Department of Computer Science, 1985.

[Becker, 1985b]
Becker,J.M., "THE EXPLORER SYSTEM Progress Report", ISG Internal Report, University of Illinois, Department of Computer Science, 1985.

[Becker, 1985c]
Becker,J.M., "Inductive Learning of Decision Rules with Exceptions: Methodology and Experimentation," M.S. Thesis, University of Illinois, Department of Computer Science, 1985.

[Berwick, 1982]
Berwick,R.C., "Locality Principles and the Acquisition of Syntactic Knowledge," Ph.D. Thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, 1982.

[Brown, Burton, and deKleer, 1982]
Brown,J.S., R.R.Burton, and J.deKleer, "Pedagogical, Natural Language and Knowledge Engineering Techniques in SOPHIE I,II and III," In D.Sleeman and J.S.Brown (Eds.), *Intelligent Tutoring Systems*, Academic Press, Publishers, 1982.

[Carbonell, et.al., 1983]
Carbonell,J.G., W.M.Boggs, M.L.Maudlin, P.G.Anick, "The Excalibur Project: A Natural Language Interface to Expert Systems," *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*, 1983.

[Charniak, Riesbeck, and McDermott, 1980]
Charniak,E., C.K.Riesbeck, and D.V.McDermott, *Artificial Intelligence Programming*, Lawrence Erlbaum Associates, Publishers, 1980.

[Charniak and McDermott, 1985]
Charniak,E., and D.V.McDermott, *Introduction to Artificial Intelligence*, Addison Wesley, Publishers, 1985.

[Hoff, Michalski, and Stepp, 1983]
Hoff,W., R.S.Michalski, and R.E.Stepp, "INDUCE 2: A Program for Learning Structural Descriptions from Examples," UIUCDCS-F-83-904, ISG-83-4, University of Illinois, Department of Computer Science, 1983.

[Goldfain, 1985]
Goldfain,M.S., "A Translation System from Prolog Clausal Form into Annotated Predicate Calculus," ISG Report, University of Illinois, Department of Computer Science, 1985.

[King, 1983]
King,M., *Parsing Natural Language*, M.King (ed), Academic Press, Publishers, 1983.

[McCord, 1985]
McCord,M.C., "Semantics of Natural Language: LFL," Presented at IBM Europe Institute, Oberlech, Austria, 1985.

[Marcus, 1980]
Marcus,M.P., *A Theory of Syntactic Recognition for Natural Language*, The MIT Press, Publishers, 1980.

[Michalski, 1974]
Michalski,R.S., "Variable-Valued Logic: System VL1," *Proceedings of the 1974 International Symposium on Multiple-Valued Logic*, West Virginia University, Morgantown, West Virginia, 1974.

[Michalski and Chilausky, 1980]
Michalski,R.S., and R.L.Chilausky, "Learning by Being Told and Learning from Examples: an Experimental Comparison of Two Methods of Knowledge Acquisition in the Context of Developing an Expert System for Soybean Disease Diagnosis," *International Journal of Policy Analysis and Information Systems*, Vol. 4, No. 2, 1980.

[Michalski and Larson, 1978]
Michalski,R.S., and J.B.Larson, "Selection of most representative training examples and incremental generation of VL1 hypotheses: The underlying methodology and the description of programs ESEL and AQ11," UIUCDCS-R-78-867, University of Illinois, Department of Computer Science, May 1978.

[Michalski, 1983]
Michalski,R.S., "A Theory and Methodology of Inductive Learning," In R.S.Michalski, J.G.Carbonell, and T.M.Mitchell (Eds.), *Machine Learning - An Artificial Intelligence Approach*, Tioga Publishers, 1983.

[Michalski, Davis, Bisht, and Sinclair, 1983]
Michalski, R.S., J.H.Davis, V.S.Bisht, and J.B.Sinclair, "A Computer-Based Advisory System for Diagnosing Soybean Diseases in Illinois," *Plant Disease*, April, 1983.

[Michalski and Baskin, 1983]
Michalski,R.S., and A.B.Baskin, "Integrated Multiple Knowledge Representations and Learning Capabilities in an Expert System: The ADVISE System," *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*, 1983.

[Nilsson, 1980]
Nilsson,N.J., *Principles of Artificial Intelligence*, Tioga Publishers, 1980.

Pua, 1984a]
Pua, K.-E., "Translating Annotated Predicate Calculus to PROLOG", UIUCDCS-F-84-918, ISG 84-1, University of Illinois, Department of Computer Science, 1984.

[Pua, 1984b]
Pua, K.-E., "An Annotated Predicate Calculus Inference System (APCIS)", UIUCDCS-F-84-928, ISG 84-8, University of Illinois, Department of Computer Science, 1984.

[Reinke, 1983]
Reinke,R.E., "PLANT/ds: An Expert System for Diagnosing Soybean Diseases Common in Illinois. User's Guide and Program Description," UIUCDCS-F-83-911, University of Illinois, Department of Computer Science, 1983.

[Reinke, 1984]
Reinke,R.E., "Knowledge Acquisition and Refinement Tools for the Advise Meta-Expert System," UIUCDCS-F-84-921, ISG-84-4, M.S. Thesis, University of Illinois, Department of Computer Science, 1984.

[Schank and Riesbeck, 1981]
Schank,R.C., C.K.Riesbeck, *Inside Computer Understanding*, Lawrence Erlbaum Associates, Publishers, 1981.

[Shieber, 1983]
Shieber,S.M., "Sentence Disambiguation by a Shift-Reduce Parsing Technique," *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*, 1983.

[Stepp, 1984]
Stepp,R.E. III, "Conjunctive Conceptual Clustering: A Methodology and Experimentation," PhD Thesis, UIUCDCS-R-84-1189, University of Illinois, Department of Computer Science, 1984.

[Varile, 1983]
Varile,G.B., "Charts: a Data Structure for Parsing," in *Parsing Natural Language*, M.King (ed),Academic Press, Publishers, 1983.

[Waltz, 1978]
Waltz,D.L., "An English Language Question Answering System for a Large Relational Database," in *Communications of the ACM*, Volume 21, Number 7, 1978.

[Winograd, 1972]
Winograd,T., *Understanding Natural Language*, Academic Press, Publishers, 1972.

[Winograd, 1983]
Winograd,T., *Language as a Cognitive Process: Syntax*. Addison-Wesley, Publishers, 1983.

[Winston, 1984]
Winston,P.H., *Artificial Intelligence*, Second Edition, Addison Wesley, Publishers, 1984.

[Woods, 1970]
Woods,W.A., "Transition Network Grammars for Natural Language Analysis," in *Communications of the ACM*, Volume 13, Number 10, 1970.

[Woods, Kaplan, and Nash-Webber, 1972]
Woods,W.A.. R.M.Kaplan, and B.L.Nash-Webber, "The Lunar Science Natural Language Information System: Final Report," BBN Report No. 2378, Bolt Beranek and Newman Inc., Cambridge, MA, 1972.