Report STL-88-11
30 June 1987

The HP-NL System

John Nerbonne
Derek Proudian

Software Technology Laboratory

Hewlett-Packard Laboratories
P.O. Box 10490
Palo Alto, California 94303-0971

The HP-NL System

John Nerbonne and Derek Proudian
Hewlett-Packard Laboratories,
Palo Alto, CA

June 30, 1987

The following individuals have all contributed to the development of the *HP-NL* system in one way or another: Barry Arons, Susan Brennan, Lew Creary, Marilyn Friedman, Dan Flickinger, Mark Gawron, Dave Goddeau, Brett Kessler, John Nerbonne, Katrina Peirce, Carl Pollard, Derek Proudian, Geoff Pullum, Diana Roberts, Ivan Sag, and Tom Wasow.

Abstract

This paper provides an overview of *HP-NL* , a natural language understanding system under development at Hewlett-Packard Laboratories. The underlying linguistic theory of *HPSG* is briefly described together with a brief history of the *HP-NL* project. Finally the archictecture of *HP-NL* is sketched and some example applications are discussed.

1 Overview and Introduction

The *HP-NL* system is a natural language processing program under development at Hewlett-Packard Laboratories in Palo Alto. The development of the system has been informed by the grammatical theory *Head-driven Phrase Structure Grammar*, (or *HPSG*) developed by Carl Pollard, Ivan Sag, and others at Stanford University.

In very rough summary, the system takes as input typewritten English, and produces as output a set of logical expressions, each of which corresponds to one possible interpretation of the English expression. The processing to this point is application-independent. The system then translates the logical expressions into database queries in an application-dependent fashion, queries a database, and then articulates the results using a variety of media. Experimental interfaces to speech input and to command language output (a mailer) also exist.

Section 2 of this technical report describes the goals of the system and its intended applications. Section 3 sketches the underlying theory and provides references to more complete descriptions, while section 4 describes the history of the effort. Section 5 outlines the architecture of the application-independent part of the system, and section 6 describes the applications interface and some database query applications. A final section provides pointers to further information for those interested in applications, source code, or documentation.

The current implementation is written in Common Lisp on a UNIX operating system called HP-UX, running on Hewlett-Packard 9000 series 300 computers, and consists of some 50,000 lines of code.

2 Goals and Intended Applications

The goal of the Natural Language Project at HP Labs has been to demonstrate the feasibility of natural language processing. We have consistently based our processing on the best available theoretical work in Linguistics, and we have built the system with the goal of providing technology to support as wide a range of applications as possible.

We have nonetheless concentrated on application areas where we believe a natural language interface could prove itself best commercially. This should be an application with a large number of users for whom special training is unattractive—either because they are occasional users or because they are computer-shy. It should furthermore be an application where a large variety of information types may have to be accessed or presented. The sense of this requirement is that a natural language interface is overkill where a simple range of information needs to be exchanged. A menu or simple graphic interface is appropriate in such cases. Finally, we prefer application areas with large markets and well-established standards, so that our own work is marketable and easily transportable.

Database query fits these requirements perfectly; databases are widespread and users come to them with a variety of informational needs. There is a large potential market for simple interfaces to databases, and standards (particularly in the querying of databases) are emerging. Database query has therefore been our first choice among applications.

The decision to concentrate on database query doesn't obviate the need for flexible design, however, since database query, even if standardized, is still evolving. We have interfaced both to relational databases (using *SQL*) and to databases written in a frame representation language (using a special query langauge, ***HP-RL***). This flexibility was possible because the system provides an application-independent level of semantic representation.

We have also built the system to support other applications. We have experimentally attempted to improve speech recognition through the use of lexical and syntactic information, which requires points of entry to processing which wouldn't be needed in database query applications. We have also interfaced experimentally to a mailing program.

3 The Theory underlying HP-NL

The theory of Head-driven Phrase Structure Grammar, described in [9], derives its name from the central role played by grammatical heads and their associated complements. Roughly speaking, heads are linguistic elements (words and phrases) that determine syntactic and semantic restrictions on the phrases (complements) that characteristically combine with them to form larger phrases. Verbs are the heads of verb phrases (and of sentences, in the system described here), nouns are the heads of noun phrases, and so forth.

As in many other syntactic theories, grammatical categories in HPSG are represented as complexes of feature specifications (or, as they are often called, attribute-value pairs). Feature values in HPSG may themselves be categories, or sequences of categories; features whose values are category sequences are often called stack-valued features. One use of stack-valued features is for handling subcategorization, i.e., the specification of what types of complements a given category may combine with. The formation of a constituent in which the head combines with a complement corresponds to popping the subcategorization stack. This mechanism permits HPSG to dispense with the device of "bar-level" that is widely used in other recent syntactic theories.

The possibility of recursive structure inherent in allowing features to take categories as values is a powerful device. It permits all the phonological, syntactic, and semantic information associated with any word, phrase, or sentence to be encoded in a single formal object, viz., a category.

Categories can combine into larger categories according to certain specified rules. Each rule stipulates how the information contained in the constituent categories is to be combined. A rule may, in principle, make reference to any of the kinds of information represented in a category. In the implementation under discussion here, the rules are context-free phrase structure rules; that is, the phonological information in the constituents is always combined by means of simple concatenation. HPSG theory does not of itself impose this constraint, but allows for the description of non-context-free languages. In our work on English so far, however, no grammatical facts have been observed to necessitate the use of devices going beyond context-free power.

The categories mentioned in rules tend to be highly schematic. That is, they stipulate values only for a very small subset of the possible features. Categories may combine according to a grammar rule just in case they are consistent with it. Formally, this is accomplished through the use of unification: a set of categories may be combined by means of a given rule if each category unifies with the corresponding category in the rule. The resulting category is also constructed using unification, as specified in the rule. Typically, a category serving as a complement to a head is unified with part of the head's category; more specifically, it is unified with a category in the list giving the subcategorization restrictions on the head.

Not all of the information about how categories combine needs to be stipulated in the rules. There are several general principles governing the feature composition of derived categories. The two most important are the *Head Feature Principle* and the *Binding Inheritance Principle.*

The Head Feature Principle says, roughly, that a phrase constructed by means of a grammar rule will have the same values as its head for a certain class of features (known as head features), unless the rule stipulates otherwise. The head features include, inter alia, those specifying major category type (e.g. nounhood, verbhood, etc.), person, and number. Hence, the lexical information about category type, person, and number will be propagated to phrases from their heads. This makes it possible, among many other things, to treat subject-verb agreement as holding between the subject and predicate phrases (which are typically contiguous), rather than between the words on which the agreement markings actually occur (which are often not contiguous).

The Binding Inheritance Principle's intuitive content is that a phrase will acquire a value for a certain feature if any of its constituent categories has that value for that feature. The features involved here (called binding features) are typically those needed for the analysis of dependencies between non-contiguous sentence elements. For example, in a sentence like (1), the information that there is a gap in the position of the object of the preposition *to* is passed up through the verb phrases *assigned to*, *been assigned to*, *has been assigned to*, and the clause *equipment has been assigned to* by means of a (category-valued) binding feature called SLASH.

(1) List consultants whom equipment has been assigned to.

This information is needed in order to license the appearance of the objective case relative pronoun *whom*. The Binding Inheritance Principle guarantees that this information will be propagated through the tree, without requiring any special statement on the grammar rules themselves.

There are other general principles that govern the operation of the grammar rules, but these two will suffice for our brief introduction. The advantages of these principles are that they allow the grammar rules to remain simple, and ensure that additional rules will automatically reflect the properties expressed in the principles.

One further device that should be mentioned is the *lexical rule*, which in HPSG has taken over much work that other theories assign to the grammar. Since most syntactic and semantic information is stored in lexical entries, some mechanisms are needed to minimize redundant stipulations in the lexicon (cf. [6] and [5] for an account of the lexicon). Inheritance is one important mechanism that will be taken up again below; lexical rules also serve this function by creating new lexical entries out of already existing ones. This permits us, for example, to enter only the base form of each (regular) verb into the lexicon, because rules will then create entries for the various inflected forms (past, present-third-singular, progressive, etc.). This process can be automated because the syntactic, semantic, and morphological properties of these inflected forms are (almost) completely predictable from the base forms. More complex lexical rules, like one for handling the passive voice, are also employed.

4 History

HP-NL developed out of the GPSG project, which is described in some detail by [4]. The GPSG system employed a set of grammar rules (largely generated by means

of a metagrammar), a simple bottom-up parser, and semantic representations based on the lambda calculus (requiring several levels of reduction and simplification prior to disambiguation). During the period 1983-1985, these features of the system were replaced by the current modules serving the same functions. At the same time, the system was ported from HP Pascal workstations to HP-UX workstations.

For the benefit of those familiar with the GPSG project, we describe the main points of development from the earlier project. The transition from GPSG to *HP-NL* principally involved four components of the system: the grammar, the lexicon, the parser, and the semantics processor.

The modifications to the grammar and lexicon involved a massive relocation of syntactic information from phrase structure rules to lexical entries. Thus, while the GPSG system had around 75 basic rules and 10 metarules, yielding a compiled grammar of about 380 rules that the parser actually accessed, the *HP-NL* system has a total of fewer than 20 grammar rules. The reduction is possible because the rules now employed are highly schematic. The lexicon has correspondingly increased in size, but whereas the GPSG lexicon was an unstructured list of items, each entered by hand in its entirety, the *HP-NL* lexicon is hierarchically structured, making use of inheritance, and employing *lexical rules* to generate entries whose content is predictable from details given in other entries. These changes were motivated by several considerations, the most prominent being: (i) to facilitate the treatment of lexical exceptions to grammatical rules; (ii) to permit the design of a more sophisticated parser; and (iii) to simplify the process of adding new words to the lexicon.

The *HP-NL* parser was developed in conjunction with these changes. Like its predecessor, it operates bottom-up, using the syntactic and semantic information from the lexical entries corresponding to the words in the string to guide its analysis. Unlike its predecessor, it has the property we describe as *head-driven*, finding first the head of a constituent, then using the information found in the head to direct the construction of the other members of that constituent, based on structural information found in the grammar rules. As a result it has the unusual property of working neither left-to-right nor right-to-left, exclusively; it works rather from the head out for each local constituent. The construction of syntactic and semantic representations for phrases and sentences is thus accomplished by means of merging two sets of partial information, with one set stored in the lexical entries, and the other set stored in the rules of the grammar. [11] contains further information on the parser.

The semantic analyses produced by the GPSG parser were highly complex lambda calculus expressions. These required a great deal of simplification before they could be

passed on to the next module. Such simplification operations were required in every sentence analysis, since lambda conversion had to remove *all* lambda expressions from a formula before it could be passed to the transducer that created the database query language expressions. Eliminating the lambda calculus from semantic computations tremendously simplified semantic processing, opening the door to the possibility of using semantic information to prune the search space of the parser. A more complete sketch of the the semantic representation language *NFLT* may be found in [3].

The four principal changes just outlined required numerous auxiliary changes in the system, so that virtually no element of the old GPSG system remains, apart from the overall architecture of the system.

Some novel facilities have also been added, most notably a *Pragmatics* module with the capability of tracking the course of interaction, and thereby allowing the use of pronouns. Before this facility was added, each sentence was processed independently, and no information from previous user questions was retained. With the introduction of the pragmatics module, the user may interact with the natural language system in the more efficient and much more natural dialogue mode. [2] describe the processing of pronouns in discourse in detail.

Finally, while these design changes were underway, significant enhancements were also made in linguistic coverage. Important extensions include coordinate structures, described in [10], reflexive and intrasentential anaphora, some comparative constructions, constructions with missing obejcts ("tough movement"), and some indefinite quantification. .

5 Major Core Components of HP-NL

HP-NL may be divided into an application-independent core and an applications interface. This section sketches the application-independent core, and the following section describes the applications interface.

The application-independent core consists of several distinct processing modules and the data structures they employ. These modules are: (1) a Lexical Analyzer which transforms a sentence represented as a string of characters into a sequence of data structures (i.e. parse-nodes) representing words, (2) a Parser which creates syntactic structures (i.e. parse-trees) from this sequence, (3) a Semantics Processor which assigns an initial semantics (i.e. an application independent logic expression) to each

parse-tree created, (4) a Pragmatics module which uses discourse information to resolve pronoun references, (5) a Disambiguator which maps application *independent* logic expressions into application *specific* ones, (6) a Query Transducer which converts database specific *logic expressions* into sentences of the *application's query language* (e.g. *SQL*), and finally (7) a Control Module which orchestrates the flow of information among the various modules.

This division of labor reflects the different sorts of information which must be processed. The Lexical Analyzer, Parser, and Semantics Processor each depend only on syntactic structure, not on application or context; the Pragmatics module is sensitive to the context of utterance, but insensitive to the application; the Disambiguator is sensitive to the fine structure of the database; while the Query Transducer depends only on the gross structure of the application (e.g. *SQL*), not on its fine strucure (i.e. the particular *SQL* database represented). We note that this division of labor provides an intermediate representation of the utterance, at the level of initial semantics, which is completely application independent. It is the existence of this representation which provides the essential portability of the *HP-NL* system.

5.1 Control Module

The *HP-NL* system consists of a number of major components, referred to as *modules*, which must work together in order to produce a reasonable output for a given input. The Control Module of the *HP-NL* system concerns itself with directing and coordinating the flow of information among the various modules of the system. It accomplishes this task by treating the various modules of the *HP-NL* system (e.g. the Parser, the Semantics Processor, etc.) as resources which it may harness to achieve its stated goal. The control module also serves as the surface of the entire *HP-NL* system, thereby encapsulating the functionality provided by *HP-NL* .

Design Issues The Control Module was designed with three major considerations in mind: (1) encapsulation, (2) intelligent processing, and (3) profiling. The encapsulation issue concerns both internal and external information hiding. Internally, the Control Module serves to keep the various modules of the *HP-NL* system ignorant of each other, so that these modules may be independently developed, and yet still work together. Externally, the Control Module serves to hide the details of the *HP-NL* system from the application programs using it, by providing an interface surface to *HP-NL* . This interface was designed to be as simple and stable as possible while still

permitting functional enhancements to *HP-NL* to be easily folded in.

The Control Module also tries to accomplish its stated *goal* as efficiently and intelligently as it can. It considers the problem to be one of finding a successful solution path through its resource space. In other words, given the *resources* it has available (e.g. Lexical Analyzer, Parser, etc.) and knowledge of their various input/output relations, and given *state* information about the current interaction (e.g. the input sentence, parses found so far, focusing information, etc.), the Control Module constructs a *plan* which employs these resources to get from the current state to one in which the desired result has been produced.

Finally, the Control Module gathers statistics about system behavior, collecting both individual module profiles and an overall *HP-NL* system profile.

5.2 Lexical Analyzer

The Lexical Analyzer transforms a sentence represented as a string into a sequence of data structures called *lexical parse-nodes*. Roughly speaking each such parse-node corresponds to word in the sentence. We should point out that these lexical parse-nodes contain both syntactic and semantic information and are the basic elements used by both the Parser and Semantics Processor.

The Lexical Analyzer performs its task in a number of discrete phases. First it identifies substrings of the input sentence which are potential words. Second, each such "word" is morphologically analyzed into root, suffix, prefix, etc. (e.g. *worked* = *work* + *ed*). Third, the lexicon is searched for lexical entries matching the root spellings identified, and copies of these entries are returned. Fourth, relevant lexical rules (e.g. the past-tense rule) are selected based on the suffixes etc. identified, and these lexical rules are applied to the base lexical entries found in the lexicon. Finally, the resultant lex-entry object is transduced into the corresponding lexical *parse-node* for consumption by the Parser.

As an optimization the parse-nodes created via the procedure described above are cached for future use. As a result system performance tends to improve with use. We should mention that in the intended delivery environment (as opposed to the development environment in which the current system exists) all such lexical parse-nodes would be cached, reducing the computational requirements of the Lexical Analyzer to simple dictionary lookup.

We now briefly describe the major data structures used by the Lexical Analyzer.

Lexicon The lexicon is the collection of words known to the system. It's organized hierarchically, so that properties shared by entire classes of lexical items may be specified and described once, thereby reducing redundancy and keeping the size of each lexical entry reasonably small. The total number of lexical entries is kept in check through the use of lexical rules, i.e. rules which produce new lexical items from existing ones based on inflectional information associated with a a particular occurrence of a word. Thus the lexicon consists of relatively sparse descriptions of the base form of words; much of the rich structure of a lexical entry is obtained by inheritance, while the various inflected forms are derived via lexical rules.

Word Classes A word class is a group of words which share syntactic and semantic properties. In *HP-NL* word classes exist independently of the particular words which they comprise, and it is these word-class objects which represent the common properties of the word class. Idiosyncratic properties of individual words are represented directly on the lexical entry itself.

The Word-Class Hierarchy The principle of representing a group of items sharing properties as a class can be applied to word-classes themselves. This gives rise to a word-class hierarchy which groups instances into classes, classes into super-classes, etc. An instance of any class inherits properties from all of the classes above it in the hierarchy (except those overridden by more local information). This hierarchical organization allows the efficient representation of shared information, provides a mechanism for default and idiosyncratic information, and supports the use of lexical rules.

The word-class hierarchy employed by *HP-NL* isn't a simple tree. Both instances (words) and classes may inherit from multiple parents. Ultimately lexical entries inherit information from one or more of the following four overlapping hierarchies:

- The *CONTROL* hierarchy which encodes subcategorization information.
- The *AUXITY* hierarchy which distinguishes main verbs and from auxiliaries.
- The *WORD* hierarchy which provides part of speech information.

- The *NUMBER-LEX* hierarchy which assigns number (i.e. singular, mass, or plural).

Lexical Rules As discussed above *HP-NL* make use of lexical rules to capture regularity in inflectional morphology. Their use simplifies the task of adding lexical items and reduces redundancy. For more thorough discussion of the use of lexical rules, and on hierarchical lexicons, see [6] and [5].

5.3 Grammar

The grammar contains two kinds of objects: *initial symbols* and *grammar rules.* Together with the information stored in lexical entries, the grammar provides the parser with the information needed to analyze the structure and meaning of phrases and sentences. Both the grammar rules and the initial symbols are quite general, relying on the lexicon to provide most of the information needed by the parser. This generality is what enables us to keep the grammar small (about 20 rules at present) while still providing substantial grammatical coverage.

Initial Symbols Initial symbols characterize the types of phrases that will be accepted. This is the same notion as that of a *start symbol* in conventional context-free grammars. In other words, once a phrase structure tree which covers the input string has been built using the grammar rules, we check to see whether the type of the root node of this tree is compatible with at least one member of our set of initial symbols.

The present *HP-NL* system has initial symbols for five sentence types: *Declarative, Yes-No-Question, Subject-Wh-Question, Non-Subject-Wh-Question* and *Imperitive.* However, there is nothing built into *HP-NL* which forces it to be an *S* parser. We could make the system into an *NP* parser by simply replacing the sentential initial symbols mentioned above with some noun phrase ones.

Grammar Rules The grammar rules used by the *HP-NL* system are schematic context-free phrase structure rules. However, instead of using *symbols* as the rule elements we use constellations of syntactic features called *feature structures.* Consequently, the match criterion used when applying these rules is *syntactic unification* as rather than *symbol identity.*

Conceptually, a grammar rule consists of a left-hand side (LHS) which is a feature structure and a right-hand side (RHS) which is a sequence of feature structures. Also associated with each grammar rule is a *semantic operation* and a *processing agenda.* The semantic operation specifies how to construct a meaning for the LHS based on the meanings of the elements of the RHS. The processing agenda specifies for the parser the order in which the elements of the RHS should be constructed. This processing order is constrained to form a contiguous constituent (or island) during construction. Note that for many rules, the agenda and semantics may use default values, and thus need not be specified explicitly. The following example is an actual rule of the system.

Example:(A Grammar Rule)

```
(define-grammar-rule AUX-NEGATION
   Schema    = (X -> H X1)
   X         = (Head-features = ((HFL PLUS)))
   H         = (Head-features = ((MAJ V) (FORM FIN) (AUX PLUS)
                                 (COORD MINUS MID)(INV MINUS)))
   X1        = (Head-features = ((MAJ NOT))
                Bindings      = ((QUE NONE) (REL NONE) (SLASH NONE)))
   Agenda    = (X1 H)
   Semantics = (Negate-Auxiliary H X1)
   )
```

The *schema* determines the type and number of elements in the rule, as well as their left-to-right surface order. These symbols are effectively placeholders for feature structures that we would like to occupy these positions. Following the schema we assign particular feature structures to each placeholder element defined. This is accomplished, as in the example, by using the symbol as key and a keyworded list of the desired feature structures as value. Finally, default values for semantic operation or processing agenda are overridden using the keys *semantics* and *agenda* respectively.

5.4 Feature System

The *HP-NL* feature system defines the set of syntactic *features* allowable in specifying the syntactic shape of words and phrases. We resolve traditionally complex syntactic categories such as *VP* or *NP* into constellations of primitive features such as *MAJ*, each of which has a fixed range of possible values such as *N* or *V*. Doing this allows us

to define a rich set of feature structures from a relatively constrained set of primitive features and feature values.

Feature Structures and Unification Formally a *feature structure* is defined as follows:

$$
\begin{array}{lcl}
\langle featurestructure \rangle & \longrightarrow & (\langle headmatrix \rangle \ . \ \langle bindingmatrix \rangle) \\
\langle headmatrix \rangle & \longrightarrow & (\langle featspec \rangle^*) \\
\langle bindingmatrix \rangle & \longrightarrow & (\langle bindspec \rangle^*) \\
\langle featspec \rangle & \longrightarrow & (\langle headfeat \rangle \ \langle featval \rangle^*) \\
\langle bindspec \rangle & \longrightarrow & (\langle bindfeat \rangle \ . \ \langle featurestructure \rangle) \\
\langle headfeat \rangle & \longrightarrow & \text{Atom} \\
\langle featval \rangle & \longrightarrow & \text{Atom} \\
\langle bindfeat \rangle & \longrightarrow & \text{Atom}
\end{array}
$$

The unification of two feature structures is obtained by intersecting the values of like features. By convention we consider the absense of a specification for a feature in a feature structure to indicate the disjunction of all possible values for that feature (essentially a don't care condition). Unification fails if the intersection of any like features is empty.

This treatment of feature structures and unification, together with the stipulation that all primitive features and values must be predeclared, permits an extremely efficient implementation of the unification operation. This is fortuitous since unification makes up the backbone of the syntactic components of *HP-NL* , putting its efficient computation at a premium. See [11] for details.

5.5 Parser

The job of the parser is to map well-formed input into corresponding syntactic structures. Because of the ambiguities inherent in natural language, this mapping is not necessarily one-to-one. There may be more than one syntactic structure (i.e. *parse tree*) which can be produced from a given input, or there may be none at all. The parser works by applying *grammar rules* to the given lexical input (i.e."sentence"), thereby constructing well-formed phrases (i.e. sub-trees) in the usual way. If the parser succeeds, it returns a *parse*; a copy of the initial symbol which licensed the structure together with the structure produced. Otherwise it returns NIL.

The *HP-NL* parser is basically a *chart parser* [8,7,12] modified to handle the complexities introduced by the nature of *HPSG* grammar rules, lexical items, and feature passing principles. A discussion of issues involved in parsing HPSG grammars may be found in [11].

5.6 The Semantic Representation Language NFLT

The translation of input sentences into a logical formalism is a common feature of computer systems for natural-language understanding, and one which is shared by the *HPSG* system. A *distinctive* feature of this system, however, is the particular logical formalism involved, which is called *NFLT* (Neo-Fregean Language of Thought). [3] sketch the semantic representation language *NFLT* in more detail.

The formalism is called "neo-Fregean" because it incorporates many of the semantic ideas of Gottlob Frege, inventor of the predicate calculus; it is called a "language of thought" because unlike English, which is first and foremost a medium of *communication*, *NFLT* is designed to serve as a medium of *reasoning* in computer problem-solving systems. The language is the result of augmenting and partially reinterpreting the standard predicate calculus formalism in several ways:

- Predicates of Variable Arity. Since argument roles are explicitly marked, predicates can combine with a variable number of arguments without explicit existential quantification over "missing arguments".
- Sortally Restricted Quantification. Quantifiers include a sortal restrictor clause in addition to the usual scope clause, which permits representation of non-first-order quantifiers, such as *most.*
- Non-Extensionality. Predicate symbols denote relations (rather than extensions), and sentential formulas denote situations (rather than truth values). This permits representation of causal relations.
- Intentionality and Conceptual Raising. The use of specialized terms denoting concepts and propositions provides a principled framework for representing beliefs, intentions, etc.

While *NFLT* is much closer semantically to natural language than is the standard predicate calculus, and is to some extent inspired by psychological considerations,

it is nevertheless a formal logic admitting a mathematically precise semantics. The intended semantics incorporates a Fregean distinction between sense and denotation, associated principles of compositionality, and a somewhat non-Fregean theory of situations or situation-types as the denotations of sentential formulas.

5.7 Semantics Processor

The Semantics Processor transduces syntactic parse trees into trees of semantics nodes. Essentially its output is a logic expression (i.e. an *NFLT* formula), but it includes additional information—pronouns, reference markers, and indefinites—which are needed for the discourse processing done by the Pragmatics module.

The leaf nodes of the input parse tree correspond to lexical items whose semantic translations are given. The semantics processor works bottom-up, computing semantic translations as it traverses the parse tree. We emphasize that semantics may be computed for any parse tree, including trees of subsentential constituents, therby allowing semantics computations to be performed either in lock step with syntactic analysis, or as a post process.

5.8 Pragmatics Processing

The *Pragmatics Processor* accepts an input from the Semantics Processor consisting of a logical expression (in *NFLT*), together with a list of pronouns and a list of available antecedents. Each pronoun is equipped with a list of syntactically determined *contraindices*, with which it may be assumed not to corefer. From this input, pragmatics constructs a model of the discourse which it maintains and updates after each user query. The model is designed according to *Centering* theory, whose use in our system has been described in [2].

The major task of pragmatics processing is the determination of likely antecedents for pronouns. To this end, it keeps a list of discourse entities under discussion. The list is ordered by discourse prominence, and the most prominent element on the list is called the *preferred center* of the discourse. The basic idea is that one is likely to refer to preferrred centers using pronouns, but not using nonpronominal means, and *vice versa* for noncenters. Using this principle of Centering theory, contraindices, and morphologically marked agreement information, a most likely antecedent may be

assigned to each pronoun.

5.9 Disambiguator

The Disambiguator solves a problem which arises because of the use of application-independent predicates in the initial semantics produced by *HP-NL* . This problem occurs when a generic *NFLT* predicate is ambiguous with respect to the set of possible application specific relations it can be mapped onto. Consider, for example, the sentences:

(1) "Does Dan manage Lyn?" $\Longrightarrow$ `(manages agt:Dan ptnt:Lyn)`
(2) "Does Dan manage Nat-Lang?" $\Longrightarrow$ `(manages agt:Dan ptnt:Nat-Lang)`

with the initial semantics given. As far as the parser and semantics processor are concerned, except for the particular constants chosen to fill the *:ptnt* role, these two questions are identical. This is because linguistically there is no difference between "Lyn" and "Nat-Lang", both are proper noun phrases, and for all these modules know "Nat-Lang" could be the name of an employee. Consequently the *NFLT* predicate chosen is the same in both cases.

Nonetheless, from the applcations's point of view, these two expressions pick out two distinct relations: one which holds between a supervisor and an employee, and another which holds between a supervisor and an organization. It is the disambiguator's job to capture this distinction, and to substitute appropriate application-specific predicates for the generic ones introduced by the semantics processor.

The disambiguator accomplishes its task using a specialized kind of sortal reasoning. The system maintains a table of predicate/argument pairs (called a *dis-data* file) which tells the disambiguator what specific predicate to substitute for a generic predicate based on the semantic sort of the generic predicate's arguments. A sortal hierarchy is also maintained by the system, allowing the disambiguator to quickly infer subsumption relations among sorts. Using this information the disambiguator systematically makes the relevant substitutions, producing what is referred to as a *disambiguated semantics* (in contradistinction to the initial semantics it started with). We should point out that constructing an appropriate *dis-data* file is one of the major tasks involved in porting *HP-NL* to a new domain.

5.10 Query Transducer

The *Query Transducer* converts a logical expression into the query language of the targetted application. This subsystem is not application-independent in the same sense as the other modules insofar as there needs to be a distinct transducer for each query language *HP-NL* targets. However, each of these transducers is by and large independent of the details of the application (e.g. the particular database represented). We briefly describe two transducers which have been written.

The HP-RL Transducer This transducer takes as input a disambiguated logical formula of *NFLT* together with an associated literal pragmatic force, and produces as output an equivalent database query in the query language provided by the frame-based representation language *HP-RL* . At present, the *HP-NL* core system distinguishes the pragmatic forces of assertion, question, and imperative, while the *HP-RL* Transducer deals only with questions. (A limited class of imperatives is presently transformed, prior to transduction, into a logical form that will yield an appropriate query. Extension of the transducer to deal with assertions and commands would be a relatively straightforward, though substantial, task.) The transducer works in an essentially compositional fashion, first computing *HP-RL* query fragments for logical subformulas, and then composing the fragments into larger ones in a way dependent on the particular logical superformulas involved. In some cases, the transduction process involves a certain amount of domain-specific inference, as, for example, when an existential quantifier over positive quantities is replaced by a query fragment that tests the results of subtracting one given quantity from another in an appropriate way.

The SQL Transducer This transducer is similar to the one above, but produces output in the database query language *SQL* . The basic architecture and method of the module are the same as for the *HP-RL* transducer, with the compositional construction of embedded *SELECT* statements of *SQL* replacing construction of embedded quantifiers in the *HP-RL* query language. Here again, the transduction process sometimes involves a limited amount of domain-specific inference.

6 Accommodating Applications

The process of interfacing *HP-NL* to a new database query application requires that application-specific vocabulary be added to the lexicon, and that atomic semantic representations be given application-specific interpretations. The process of adding to the lexicon is partially automated by a facility that scans the database for objects, classes etc. that might be mentioned in users' queries. Providing application-specific interpretations for semantic representations amounts to providing a "disambiguation-data" file described in the section above on disambiguation.

Articulation The *Articulation* module of *HP-NL* displays system responses to user queries, and its level of sophistication depends on some application support. The databases currently accessed allow answers in the form of English replies, tables, videodisc images, and synthesized speech.

6.1 Examples of Applications

The Organization Database A small database constructed in the frame representation language *HP-RL* exists solely to demonstrate the natural language system. It contains information about the organizational structure of the Knowledge Technology Lab at Hewlett-Packard. The lexicon, consequently, contains language appropriate to describe the relationships among the people who work in the Lab where job titles include: consultants, managers, technical staff, and clerical staff. The organization is subdivided into divisions, centers, labs, departments, and projects. Managers are specified at all levels. Technical staff is not further subdivided. Clerical staff currently only includes secretaries. Places such as offices, conference rooms, and bathrooms exist. People have telephone numbers, and equipment items such as computers and displays.

As the database is loaded, procedures execute to extract vocabulary items. Class names correspond to common nouns, and instance names to proper names.

The Van Gogh Database A somewhat larger Oracle relational database containing information about portraits by Vincent Van Gogh was constructed especially for the purpose of demonstrating a natural language front end to a relational database

via the *SQL* database query language. The database contains information about the paintings, especially their media, date of creation, present place of exhibition, influence, and subject matter.

7 For Further Information

The *HP-NL* natural language system is under development at

The Natural Language Project
Hewlett-Packard Laboratories
1501 Page Mill Road
Palo Alto CA, 94303

We are also accessible via email: hplnls@hplabs.hp.com.

The software and a complete documentation packet has been made available to selected university affiliates of Hewlett-Packard Laboratories.

Acknowledgements We would like to acknowledge the valuable assistance of all the members of the *HP-NL* project, each of whom, in one way or another, has contributed to the preparation of this document. Special thanks go to Thomas Wasow and Lew Creary for their careful proof reading and suggestions for improvement.

References

[1] *HP-NL User's Guide.* Hewlett-Packard Laboratories, 1986.

[2] Susan E. Brennan, Marilyn W. Friedman, and Carl J. Pollard. A centering approach to pronouns. In *Proceedings of the 25th Annual Meeting of the Association for Computational Linguistics*, 1987.

[3] Lewis G. Creary and Carl J. Pollard. A computational semantics for natural language. In *Proceedings of the 25th Annual Meeting of the Association for Computational Linguistics*, 1985.

[4] Mark Gawron et al. *The GPSG Demo System.* Technical Report, Hewlett-Packard Laboratories, 1982.

[5] Daniel Flickinger. *Lexical Rules in the Hierarchical Lexicon.* PhD thesis, Stanford University, 1987.

[6] Daniel Flickinger, Carl Pollard, and Thomas Wasow. Structure-sharing in lexical representation. In *Proceedings of the 25th Annual Meeting of the Association for Computational Linguistics*, 1985.

[7] Ron Kaplan. A general syntactic processor. In Rustin, editor, *Natural Language Processing*, Algorithmics Press, 1973.

[8] Martin Kay. The mind system. In Rustin, editor, *Natural Language Processing*, Algorithmics Press, 1973.

[9] Carl Pollard and Ivan Sag. *An Information-Based Theory of Syntax and Semantics.* University of Chicago Press, 1987.

[10] Derek Proudian and David Goddeau. *Constituent Coordination in HPSG.* Technical Report 97, Center for the Study of Language and Information, Stanford University, 1987.

[11] Derek Proudian and Carl Pollard. Parsing head-driven phrase structure grammar. In *Proceedings of the 25th Annual Meeting of the Association for Computational Linguistics*, 1985.

[12] Terry Winograd. *Language as a Cognitive Process.* Volume 1:Syntax, Addison-Wesley, 1980.