A Phrase-Structured Grammatical Framework for Transportable Natural Language Processing[1]

Bruce W. Ballard

AT & T Bell Laboratories
600 Mountain Avenue
Murray Hill, N.J. 07974

Nancy L. Tinkham

Department of Computer Science
Duke University
Durham, N.C. 27706

We present methods of dealing with the syntactic problems that arise in the construction of natural language processors that seek to allow users, as opposed to computational linguists, to customize an interface to operate with a new domain of data. In particular, we describe a *grammatical formalism,* based on augmented phrase-structure rules, which allows a parser to perform many important domain-specific disambiguations by reference to a pre-defined grammar and a collection of auxiliary files produced during an initial knowledge acquisition session with the user. We illustrate the workings of this formalism with examples from the grammar developed for our *Layered Domain Class* (LDC) system, though similarly motivated systems ought also to benefit from our formalisms. In addition to showing the theoretical advantage of providing many of the fine-tuning capabilities of so-called *semantic grammars* within the context of a domain-independent grammar, we demonstrate several practical benefits to our approach. The results of three experiments with our grammar and parser are also given.

1. Introduction

As a result of advances in natural language processing, programs that provide practical English-language capabilities have begun to rival more conventional means of computer interactions for certain purposes, including database retrieval, online help facilities, and limited forms of office assistance. Although several prototype systems have provided customization facilities that allow users to specify synonyms, syntactic paraphrases, and the like, traditional approaches have resulted in systems wedded to a single domain of data. That is, users are unable to access novel types of data without acquiring a new or modified processor specifically tailored to the new domain by the system designer(s). Not surprisingly, an important trend in natural language system design is in allowing users themselves to adapt an existing processor for a new domain. Accordingly, prototype systems that permit user customizations or rapid customizations by a designer have included REL, POL and ASK (Thompson and Thompson 1975, 1981, 1983), CONSUL (Mark 1981; Wilczynski 1981), IRUS (Bates and Bobrow 1983), KLAUS (Haas and Hendrix 1980), TEAM (Hendrix and Lewis 1981; Grosz 1983), a system developed at Bell Labs (Ginsparg 1983), and our own LDC system (Ballard 1982, 1984; Ballard and

[1] This research was supported in part by the National Science Foundation under Grant Numbers MCS-81-16607 and IST-83-01994 and in part by the Air Force Office of Scientific Research under Grant Number 81-0221.

0362-613X/84/020081-16$03.00

Lusth 1983, 1984; Ballard, Lusth and Tinkham 1984a, 1984b). Since the successful construction of a transportable system requires sound methods of *representing* what is to be learned, the design of formalisms to be used in transportable natural language processors relates to the *scientific,* as well as the *engineering,* aspects of computational linguistics.

In this paper we present methods of dealing with the syntactic problems that have arisen in the construction of our LDC system. In particular, we shall describe a grammatical formalism, based on *augmented phrase-structure rules,* which allows a parser to make domain-specific decisions by referring to a dictionary and other auxiliary files produced during an initial learning session with the user. We illustrate the workings of our grammatical formalism with examples from the existing LDC grammar, but we note that similarly motivated systems ought also to benefit from our formalisms. We will also include the results of some experiments with our existing grammar as applied to several domains.

In addition to showing the theoretical advantage of being able to provide many of the fine-tuning capabilities of so-called *semantic grammars* within the context of a domain-independent grammar, we demonstrate several practical benefits to our approach. For example, the conciseness of our formalism allows shorter grammars than many previous formalisms would allow, at least for the intended class of retrieval applications. This offers not only added perspicuity but other benefits as well. For instance, we have been able to write simple (almost trivial) LISP routines that pre-process a grammar to construct the files used by the parser to increase efficiency and to perform valuable disambiguations.

2. Overview of Syntactic Processing

The primary purpose of this paper is to introduce a *grammatical formalism* that we have developed for use in specifying grammars for a transportable natural language processor. As suggested by the term "framework" in the title of the paper, however, we will touch upon certain related concepts in order to give a complete account of the language constructs that can be dealt with by our formalism. In total, then, we shall discuss

1. a phrase-structured *grammatical formalism;*
2. a required *dictionary* format and associated *compatibility* file; and
3. an implied format for *parse structures.*

We begin with a brief indication of the ways in which these topics tie together in our existing and in other conceivable systems.

2.1 The phrase-structure grammar

The format we have developed to represent our grammars is intended to capture the spirit of phrase-structure specifications such as the following, which specifies simple noun phrases.

the (Ord / Super) (Num) (Super) Adj* Noun=Head PP*

Parentheses denote optionality, * denotes the Kleene-star, and / denotes alternation. To distinguish between terminal symbols, parts-of-speech, and multiple-word grammatical categories, we have used the convention of no caps, initial cap, and all caps, respectively.

In deciding on a precise, internal representation for grammars, we decided to adopt a LISP-like prefix notation. Thus, the actual specification of the expression shown above, assuming it intends to capture the structures of descriptive noun phrases (say, "NPdesrip"), is as follows. Note that we have included a context-sensitive augmentation, of a form to be discussed later, for the second Super.

```
(setq NPdescrip '(Seq
    (Quote the)
    (Opt Alt  (Get Ord)
              (Get Super))
    (Opt Get Num)
    (Opt Get Super = (need not part Super))
    (* Get Adj)
    (Get Noun Head)
    (* Call PP)
))
```

Each of the seven command types illustrated here is described in detail in Section 3, together with examples of their idiomatic usage. The reader may notice that our grammatical formalism resembles both transition network and phrase-structure grammars (Woods 1970, 1980; Heidorn 1975), and our later speaking of grammar rules as "commands" further reveals an affinity with ATNs. For the reasons given in Section 5.2, however, we prefer to view the *grammars* as collections of augmented phrase-structure rules, while certain aspects of the current top-down *parser* act very much like an ATN parser. The reader may also notice that the current utilization of the formalism has resulted in parse structures having the flavor of case grammars, especially the ways in which complex relative clauses and handled. On the other hand, the provision for associating a feature list with each phrase somewhat resembles the systemic structures of Winograd 1972.

2.2 The dictionary and compatibility files

Our grammar rules assume that an input to be parsed will be presented as a sequence of sets of token candidates, where each token candidate corresponds roughly to a word or inflection of a word found in the system dictionary. Each dictionary listing for a word is made up of one or more *meanings,* where each meaning comprises

(a) the *word* itself;
(b) its *part of speech;*
(c) the associated *root* word; and

(d) zero or more associated *features,* each with one or more possible values.

As an example, the entry

```
(offices  Subtype  office  (nt  room)  (sp  plur))
   1        2        3       4          5
```

says that

1. the word "offices" has been found or is being proposed for the current token candidate;
2. the word refers to some of the domain objects of some object type;
3. the root word is "office";
4. the objects being referred to are of type "room"; and
5. the word is a plural noun.

In addition to the "features" found in the dictionary, which provide for simple context dependencies, a *compatibility* file is assumed to be available which contains information on acceptable attachments for such units as prepositional phrases and relative clauses. An example of how this information can be used, together with a simple example of a possible set of prepositional triples, is found in Section 3.3.3.

2.3 Parse structures

Our grammar assumes that, during parsing, each non-atomic syntactic category will have associated with it a "parse structure" consisting of

(a) the *name* of the structure;
(b) a list of *features* giving possible values of various parameters (as illustrated shortly) associated with the phrase; and
(c) a list of labeled *items,* namely words and pointers to nested phrases.

For example, the parse structure

```
(NP  (feats (nt person) (sp plur))
     (Adj . lousy)
     (Head . advisor))
```

might correspond to the noun phrase "lousy advisors", where the features indicate that the phrase refers to domain objects of nountype (nt) person and is plural (plur), and the items indicate that the head noun is "advisor" and the adjective "lousy" is present. For the sake of completeness, and to help prepare the reader for the discussion that follows, Figure 1 gives both complete and "abbreviated" parse structures (i.e. a skeleton structure without nountype and compatibility information) for the sentence

"How many graduate students were failed by the instructor that John took AI from?"

As shown in Section 6.2.2, our feature lists are reminiscent of the systemic structures given in Winograd (1972). In effect, they constitute a repository of information about words contained within a phrase (possibly nested within it) and allow information to be passed both up and down a parse tree, thus enabling valuable context-sensitivities, both syntactic and semantic.

3. The Grammatical Formalism

Our grammatical formalism is built around seven types of syntax rules which we will often refer to as "command" types. The first three of these ("basic" commands) are used to specify words, parts of speech, and syntactic categories, while the remaining four ("control" commands) provide facilities for optionality, possible repetition, alternation, and sequence. In addition to the primary function of each of the commands, through which the grammar writer (i.e. system designer) can specify any context-free grammar, each of the basic commands may be *augmented* with information that enables the grammar writer to specify certain context-sensitive constraints that (a) enable similar grammatical constructs to be collapsed into what would otherwise be an over-generating unit, and (b) allow the parser to perform useful disambiguations. The latter facility can be especially valuable in a transportable environment, where the system designer is unable to predict many of the syntactic, word sense, and other forms of ambiguity that will arise.

We now discuss each command type, after which we describe each of the augmentations allowed.

3.1 Basic commands

The three basic grammar commands are

Quote	to specify a given word or set of words;
Get	to specify a part of speech; and
Call	to specify a syntactic category.

All grammar "commands" are processed by the parser (a LISP program) and should not be confused with operations in LISP or other programming languages. We will now briefly describe these commands. As described shortly, these commands may contain augmentations to assure proper agreement among syntactic components.

The *Quote* command instructs the parser to find one of a list of words in the next token slot of the input (i.e. as the next token). For example,

```
(Quote (a an))
```

says to pick up the next word if it is "a" or "an", otherwise fail. If the list of words contains just one word, the superfluous parentheses may be dropped, for instance

```
(Quote the) = (Quote (the))
```

The default action of Quote is not to add to the parse structure being built up, but an optional label is provided for. Thus, if the article "an" is seen by the command

```
(Quote (a an the) Art)
```

then the feature (Art . an) will be added to the parse structure. Our current grammar uses Quote sparingly, partly since few words are known in advance in our transportable environment. However, having a Quote facility is

```
(NP (feats (nt student) (head noun) (sp plural) (func count))
    (RelO (feats (nl (Subj Verb Obj Part Prep Arg)
                     (instructor fail student nil nil nil))
                 (sp plural))
          (Subj (feats (nt instructor) (sp sing) (head noun) (Art def))
                (RelA (feats (nl (Subj Verb Obj Part Prep Arg)
                                 (student take course nil from instructor))
                             (sp sing))
                      (Prep . from)
                      (Obj (feats (nt course) (sp sing) (head nounval))
                           (Nounval . AI))
                      (Verb . take)
                      (Subj (feats (nt student) (sp sing) (head nounval))
                            (Nounval . John)))
                (Head . instructor))
          (Verb . fail))
    (Head . student)
    (Nounmod . graduate))
```

(a) Complete Parse Structure

```
(NP (func count)
    (RelO (Subj (RelA (Prep . from)
                      (Obj (Nounval . AI))
                      (Verb . take)
                      (Subj (Nounval . John)))
                (Head . instructor))
          (Verb . fail))
    (Head . student)
    (Nounmod . graduate))
```

(b) Abbreviated Parse Structure

Figure 1. *Complete and Abbreviated Parse Structures for the Sentence "How many graduate students were failed by the instructor that John took AI from?"*

useful in allowing the grammar writer to capture various "noise words" without having to define artificial grammatical category names or resort to a proliferation of features. Thus we feel quite comfortable in writing, at the appropriate place(s) in a grammar, the command

(Quote (whether if))

The *Get* command instructs the parser to find a word having one of a list of parts of speech. As with Quote, an abbreviation is permitted if only one part of speech is to be allowed (which is most often the case). Some examples are:

(Get (Ord Super))
(Get Noun)

When a Get command is processed, the word it picks up is incorporated into the current parse structure and labeled appropriately. The default label is the part-of-speech category but an optional second argument to Get may be used to specify any other label name. For instance,

(Get Noun Head)

says to pick up a noun and label it Head. This would be useful if several nouns occur within a given phrase and need to be distinguished. If it is desired to recognize a given part of speech without adding to the parse structure, the label "nil" may be given, thus

(Get Art nil)

would recognize an article without affecting the parse being built up. Finally, if one of several parts of speech is to be allowed, the dummy label "=" may be used to assure the default action. However, this is necessary only in situations where augmentations (as discussed in Section 3.3) are present. For example,

(Get (Vpres Vpast) = (vtype intrans))

would cause the word being recognized to be labeled according to its dictionary specification.

The *Call* command instructs the parser to process an embedded constituent, such as a noun phrase, prepositional phrase, relative clause, and so forth, and so our Call is analogous to the "push" operation of conventional ATNs (Woods 1970). As with the Get command, whatever is

found will be labeled as specified or by default. Thus, the commands

```
(Call NP Arg)

(Call Relc)
```

call for a noun phrase (NP) to be labeled Arg and a relative clause (Relc) to be labeled Relc.

Normally, each Call-ed routine, and also the top-level constituent S, will have a separate parse structure associated with it. Thus, when a constituent phrase has been parsed using a Call command, it will be pointed to rather than having its components physically included in the parent phrase. In the LISP implementation, the associated structure is simply buried one additional level and assigned a feature list (to be discussed shortly) of its own. For example, if in recognizing the phrase "big houses" the adjective "big" is parsed directly by the grammar routine for noun phrases, it would be placed into the parse structure as indicated by

```
(NP  ((sp plur) (nt building))
     (Head . house)
     (Adj . big))
```

where the first set of nested parentheses give the feature list of the noun phrase as discussed in Sections 3.3. and 6.2.2. If on the other hand our grammar identified some potentially larger unit (say, adjective phrase) as a separate syntactic construct, and the word "big" was recognized as a constituent of it, we would have

```
(NP  (feats (sp plur) (nt ...))
     (Head . house)
       (AdjPh (feats (nt ...))
            (Adj . big)))
```

where the precise "nt" value would be determined by the augmentations present in the Call command.

In some instances, however, either for the sake of perspicuity or to avoid redundancy, it is useful to isolate and name a sequence of commands as a "macro" for which recognized items will be incorporated directly into the parent structure, i.e. the parse structure current when the Call command was encountered. This is handled by giving a label of nil when the macro interpretation is desired. For example, our noun phrase grammar includes the command

```
(Call Ordnum nil)
```

where Ordnum looks for ordinals, superlatives, and numbers.

3.2 Control commands

In addition to the three primitive commands just described, we provide commands to specify optionality, possible repetition, alternatives, and sequence.

The *Opt* command instructs the parser to attempt to process an arbitrary command, but Opt will succeed even if this attempt fails. Some example Opt commands are

```
(Opt Quote of)

(Opt Get Noun Head)

(Opt Call PP)
```

In addition to applying Opt to a basic command, as shown here, Opt may be applied to any of the remaining control commands discussed below. Since the scope of Opt, and also of * as discussed next, consists of a single command, we have avoided introducing a superfluous set of parentheses surrounding its argument.

The *Star* command, or simply *, denotes the Kleene-star and says to perform the embedded command 0 or more times. Some examples are

```
(* Get Adj)

(* Call Relc)
```

During parsing, a * command is treated as an Opt that re-invokes itself upon each success.

The *Alt* command instructs the parser to perform exactly one of a set of commands, which it tries in the order they are given. Alt will fail if none of its arguments succeeds. Some examples are

```
(Alt  (Get Noun)
      (Get Pron))

(Alt  (Call PP)
      (Call Relc))
```

An interesting and frequent instance arises when one of several constructs is to be optionally recognized. We have found it pragmatically preferable in these instances, both visually and for the sake of efficiency, to code for an Alt of alternatives, and surround this Alt with an Opt, rather than apply Opt to each alternative with an Alt outside. That is, we would normally write

```
(Opt Alt  (Call PP)
          (Call Relc))
```

rather than

```
(Alt  (Opt Call PP)
      (Opt Call Relc))
```

As a second pragmatic remark related to Alt, we note that the command

```
(Alt  (Get Noun)
      (Get Pron))
```

may function differently from

```
(Get (Noun Pron))
```

when the next word to be parsed has several parts of speech associated with it. In the former case, the noun reading will be taken if at all possible, while in the latter either a noun or pronoun reading is equally acceptable to the grammar, so the first grammatical category appearing in the scanner output will determine what is selected.

Since the problem of mis-recognitions is especially troublesome in our projected voice-input environment (Biermann et al. 1983), this distinction can be important.

Finally, the *Seq* command instructs the parser to perform a list of commands in order. For example,

```
(Seq (Get Prep)
     (Call NP))
```

says to recognize a preposition, then call for a noun phrase. If any member of the list supplied to Seq fails, then the entire Seq command will fail. Before doing so, however, attempts will be made via the backtracking mechanisms of the parser to re-interpret what has already been parsed by previous commands of the Seq. Strictly speaking, Seq is redundant since its effect can be obtained by using a macro with a dummy name. However, we find it convenient to be able to code nameless sequences where they are used, somewhat analogous to the use of LAMBDA in LISP, or of BEGIN-END blocks in Algol.

3.3 Augmentations

The seven syntax commands described above provide the grammar writer with convenient means of specifying context-free rules for fragments of natural language (in fact, only four of the seven are required in order to do this). However, the inadequacies of a pure context-free formulation of natural language syntax are well recognized, and various treatments have been used to overcome them (Bobrow and Webber 1980; Heidorn 1975; Colmerauer 1978; Kimball 1972; Marcus 1980; Pereira 1981; Pratt 1975; Rieger and Small 1979; Robinson 1982; Sager and Grishman 1975; and Woods 1970, 1980). Within our grammatical formalism, we have provided means of specifying several forms of useful "compatibilities" among the elements of a phrase or clause. As the reader may observe, most of our provisions for context-sensitive specifications could in theory be done in a strictly context-free fashion, though not conveniently (e.g. a large and potentially exponential increase in the number of parts of speech might be required). A related use of augmentations is to "annotate" the parse structure with information that will be useful in its subsequent semantic processing.

Compatibility checking is done according to *augmentations* which occur as optional parameters of Quote, Get and Call commands. With a few exceptions, augmentations may be used in any combination. Thus, the general form of the three basic commands, which we described in simplified form in Section 3.1, is

(Quote <literal word(s)> {<label> {<p1> ... <pN>}})

(Get <part(s) of speech> {<label> {<p1> ... <pN>}})

(Call <routine name> {<label> {<p1> ... <pN>}})

where braces denote optionality and where each parameter (denoted by pi) has one of the forms we now describe.

3.3.1 Feature-value pairs and a not-local marker

The simplest type of augmentation, which applies to any of the three basic commands (i.e., Quote, Get, Call), consists of a *feature-value pair* that supplies information on, and thus restricts the allowable values for, some "feature" of the current phrase. Since dictionary listings also contain feature-value pairs, and words being incorporated into a phrase must have features compatible with those already in the phrase, a feature-value specification in the grammar may serve to restrict the set of legal words to be processed. That is, the information about a word about to be incorporated creates an inconsistency, thus causing the command being considered to fail. As an example, the command

(Get Noun Head (sp sing))

contains the feature-value augmentation "(sp sing)" which says that the "sp" feature of the current phrase must have the value "sing". In essence, the command calls for a singular noun. The command

(Get Ved Verb (type past part))

gives an example of a feature-value pair that somewhat more liberally requires that the "type" feature of the current phrase be one of two possible values.

Another use of feature-value pairs is to incorporate information into the phrase being processed that will be used to determine the acceptability of subsequent commands, as described in Section 3.3.4. For example, the command

(Opt Quote by nil (byfront))

might be used at the top of a grammar routine for passive relative clauses to indicate that the phrase actually began with the word "by" as opposed to the word being found elsewhere. Note that, in situations such as this, an isolated feature name without associated values is sufficient and thus allowed by the formalism.

Still another use of feature labels is to annotate the parse structure being built up, as in

(Get (Aux = (func yesno))

Naturally, different uses of feature labels may occur in a single command. For instance, the word "many" in the determining phrase "how many" might be coded as

(Quote many nil (func count) (sp plural))

where the "func" feature is an annotation and the "sp" feature assures that the subsequent head noun will be plural.

In some situations, it may be desirable to ignore feature-value pairs in the dictionary listing for a word and, in this event, a *notlocal* augmentation may be used. For example, the command

(Get Prep Part notlocal)

might be used to indicate that the features associated with a preposition are to be ignored when the word is being

used as a particle. By allowing an attachment of "notlocal", we avoid some of the need to associate multiple senses with all the words that some command is interested in recognizing.

3.3.2 Feature labels

The second type of augmentation applies to Call commands and consists of a *feature label* specification similar to the feature-value pairs discussed above. The effect is to require some feature of the current phrase to agree with some possibly different feature of the child phrase about to be created. That is, the values of the two features are to be shared. If the parent and child phrase are to agree on the same feature label, this label is merely included at the end of the Call command. As an example, we might account for the first part of postnominal comparative phrases of the form "(which is) (not) <compar> than", where parentheses denote optionality and "<compar>" denotes a comparative such as "better" or "bigger", by

(Call CompPh Comp nt sp)

which assures that the top-level verb will agree in plurality (sp), and the relative pronoun in nountype (nt), with the head noun to be modified. In the event that every feature of the parent and child phrases should agree, the fictitious label *all* may be used. For example, our Adj routine handles adjective and present participle modifiers, and also negated forms of these (e.g. "non failing"), and the latter are processed by a low-level NegAdj routine invoked by

(Call NegAdj neg all)

This allows the scope of the negation to be retained, since the modifier is nested inside a "neg" label to the noun phrase being built up, yet assures that the modifier is compatible with all features in the noun phrase just as though it were to be processed directly.

When parent and child phrases are to agree on different labels, we include an "agree" triple of the form

(agree <parent feature> <child feature>)

As an example, we might wish to associate two separate nountype features with argument-taking nouns like "classmate", one for the type of world object the word itself refers to and one for the type of world object associated with its argument. In this case, the word "classmates" might receive the dictionary listing

```
(classmates   Argnoun classmate
              (nt student) (ntarg student) (sp plur))
```

and we might recognize the phrase "classmates of John" by

```
(Seq (Get Argnoun Head (head argnoun))
     (Quote of)
     (Call NP NounArg (agree ntarg nt)))
```

since at the time the Call command is encountered the parse structure will be

```
(NP  (feats  (head argnoun) (ntarg student)
             (nt student) (sp plur))
     (Head . classmate))
```

3.3.3 Nonlocal frame parameters

Another form of augmentation associated with Call commands is the *nonlocal frame parameter,* which is similar to, but more general than, what is available using a feature-value pair. It allows the grammar writer to enforce agreements between or among phrases, and has the form

((agreement type) {<labels> <feature>})*

where {...}* indicates repetition of "..." 0 or more times. The required compatibility among the indicated constituents is that an appropriate tuple be found in the set of tuples associated with the specified agreement type. Each label-feature pair tells what elements of the tuple have already been found, and which of their features contains the desired compatibility information. As indicated in Section 6.1, compatibility tuples are created during knowledge acquisition and made available to our parser in parallel with the dictionary listings from the scanner. For example, prepositional compatibilities might be indicated by a set of triples such as

```
((Head Prep Arg)
 (book in table)
 (book on table)
 (book on chair)
 (chair on table))
```

so that the prepositional phrase "in the old table" would be allowed to attach to a phrase whose head noun has the nountype *book* but not to one having the nountype *chair.* As an example, we might use the command

(Call PP PP ((Prepinfo) Head nt))

to require a 3-way agreement among the indicated component (Head) of the current phrase and remaining components (Prep and Arg) of the routine (PP) about to be invoked. The "nt" feature tells where in the current phrase and the remaining phrases the values for agreement are to be found. In our current grammar, the prepositional phrase (PP) routine locates a single word to fill the Prep slot and finds an NP phrase to fill the Arg slot. Thus, the 3-way compatibility in effect is to be found among

1. the nt feature of the current phrase,
2. the single word filler for the Prep slot, and
3. the nt feature for the phrase that fills the Arg slot.

As a second example of compatibility information, we currently maintain 6-tuples for verb phrases, where each position corresponds to a possible clause element as shown in Section 4.1, so that the tuples

```
((Subj Verb Obj Prep Arg Part)
 (student fail course nil nil nil)
 (student take course from instructor nil)
 (student make grade in course nil)
 (instructor fail student nil nil nil)
 (instructor cross student nil nil up))
```

might be used to say that a student may take a course but not an instructor, an instructor may fail a student but not vice versa, an instructor may be said to have crossed up a student, and so forth.

In the implementation of our parser, compatibility information is maintained in only one place and all related structures point to this location. This means that compatibility information will be passed both up and down the parse structure under construction to aid in disambiguation and subsequent interpretation.

In the event that one or more of the clause elements associated with a nonlocal frame parameter is to be considered optional, a nonlocal frame parameter may be accompanied by an *opt* parameter to this effect, so that

```
(Call Passive Relc (Relinfo) Obj nt) (opt Subj))
```

calls for a passive relative clause in which the object (Obj) has been relativized and in which the subject (Subj) may be omitted, as in "the book given to Paul". As an example of when more than one element of a compatibility tuple will already have been found when tuple agreement is requested, suppose we wish to handle "deep" relativizations, as in a "a book Smith wrote the last chapter of". (Although a linguist friend has recently told one of us that this is bad English, it occurs quite often in computational settings and is therefore sensible to inject into a realistic grammar.) Here, the triple chapter-of-book is to be checked, so the preposition "of" is preceded by its argument as well as by the noun it modifies. Thus, assuming "nt!" gives the nountype of "Smith" that has been passed down by the methods of Section 3.3.2, we might write

```
Call Prep PP-hole ((Prepinfo) Head nt Arg nt!))
```

to find a lone preposition with appropriate agreement.

3.3.4 The "need feature" parameter

Another form of augmentation, which enables the grammar writer to deal with situations where the presence of certain information enables rather than blocks another phrase, involves a *need* parameter that may occur with any of the three basic commands or, with a related effect, with an Opt command. The effect of this type of augmentation is to allow the associated grammar command to be used only if one of a specified list of features or items (parts of speech) is already present in the current phrase or, if desired, to enable a command only in the absence of previous features or items. Ordinarily "need" will check for the presence of a feature, but optional "not", "part", and "tok" flags will cause its behavior to be modified appropriately. As an example usage of "need", assume that we want to account for "ordinal modifiers", i.e. phrases such as "from the right", that may occur as postnominal modifiers when a prenominal ordinal has occurred. That is, we wish to allow "second column from the right" yet disallow "columns from the right". This might be accomplished by using the command

```
(Call Ordmod = (need part Ord))
```

As a second example, we might use the command

```
(Get Noun Head (sp plur) (need part Art Num))
```

to indicate that a plural noun is to be accepted only if either an article (Art) or number (Num) has already appeared in the current phrase as a "part" (of speech).

To check for a token feature, we use "tok". For example, to treat "more interesting" but not "more good" as a comparative (thus ruling out the anaphoric comparative reading of a sentence such as "we'll have more good things to talk about tomorrow"), we put a "parap" feature in the dictionary listing for paraphrastic adjectives (i.e., those that take "more" as opposed to "-er") and then write

```
(Get Adj Compar (need tok parap))
```

As an example of the complementary usage of "need", suppose we wish to allow a post-nominal comparative to occur with its associated "than" phrase only if a pre-nominal comparative has not been found. That is, we want to allow "better student than Jack" and "students better than Jack" but not "better students better than Jack". In this situation we might check for a possible post-nominal comparative with the command

```
(Get Compar = (need not part Compar))
```

In the event that a command is to be treated as optional in some cases but is required in others, a "need" augmentation may be used along with an Opt command and, in such situations, the command to be considered optional must be given inside parentheses (otherwise the "need" would be associated with the command itself and not with its optionality). As an example, we might account for elliptical noun phrases such as "the seven" or "the last" via the command

```
(Opt (Get Noun Head) (need part Ord Num))
```

which says that a Noun must appear unless the phrase being processed already has an ordinal (Ord) or a number (Num).

3.3.5 The "next words" parameter

Finally, to allow a fixed grammar to be applicable to domains in which certain unpredicted idiomatic constructions arise, we provide a *next* parameter that indicates that the dictionary word being processed must be literally followed by one or more words. The "next" parameter is available for both Quote and Get commands and, more important, may also be contained within the dictionary listing of a word. As an example, we might

account for the idiom "to pick up on" by including the following as one of the senses for the word "picked"

(picked Ved pick-1 (next up))

and then indicate in the compatibility file that "on" acts as a particle for the verb whose root is "pick-1". To be more extreme, we might have

(kicking Ving kick-2 (next the bucket))

together with an indication that "kick-2" is an intransitive verb, to indicate that the phrase "to kick the bucket" is to be treated idiomatically. The "next" feature might also be used in the grammar as a simplification, allowing us to replace

```
(Seq (Quote at nil (quant leastmany))
     (Quote least)
     (Quote as)
     (Quote many))
```

with the simpler

(Quote at nil (next least as many) (quant leastmany))

4. Current Utilization of the Grammatical Formalism

We have indicated that the phrase-structure grammatical formalism discussed in this paper is being used in the context of LDC, a transportable natural language processor. We now describe briefly the nature of the parser and current grammar associated with this system, then give the results of some experiments with the grammar and parser.

4.1 The current LDC grammar

Most of the present LDC grammar comprises
(a) a fairly elaborate noun phrase grammar, and
(b) a case-like specification of sentence-level and fairly complex relative clauses.

For example, we presently provide for relative clauses of many varieties (e.g. "by whom a book was given to Bill" as well as "who gave a book to Bill") having case frames of the form

Subj Verb {Object} {{Prep} Arg} {Particle}

where braces denote optionality. We also provide for many kinds of pre-nominal modifiers, including ordinals, superlatives, adjectives, and noun modifiers. Many forms of comparative phrases are also provided, including elliptical ones such as "a longer document than xletter" and "students making a better grade than Bill in CPS152". Our initial grammar was approximately six pages in length and yet, due to the consolidations made possible by our phrase-structure rules, contained almost all syntactic structures available in the original 20-page ATN grammar used by NLC from which it was adopted, and many other structures (notably, more elaborate comparative and relative clause forms). However, this grammar provided for only noun phrases, albeit complex ones. Our current grammar is eight pages in length and, due to consolidations made possible by enhancements to the formalism, allows both wh- and yes-no questions, imperatives, passives, deep raisings, several forms of fronting, a few discontinuous constituents, and some exotic modifier types mentioned in Ballard (1984). The syntactic processing of a representative input for LDC was indicated in Figure 1, and information of the scope of the current syntactic and semantic coverage of LDC can be found in Ballard (1984).

As indicated below, some of the first domains that we used to test the modules of LDC were
(a) a *final grades* domain,
(b) a *document preparation* domain, and
(c) the original *matrix* domain of NLC.

In the interest of assuring domain independence, only those constructs that arise in two or more of these domains (or others we have tested Prep with) appeared in the initial LDC grammar. This has caused us to temporarily discount certain features, especially (a) low-level syntax for domain-specific noun phrases, such as the optional indefinite article in "a B+" for the final grades domain and floating-point numbers for the matrix domain, and (b) certain unusual or idiomatic verb phrase forms.

4.2 The initial to-down parser

When designing a syntax processor for a class of formal or natural language grammars, one often begins with a *top-down* implementation, then uses this system to test and refine the formalism and/or grammar(s) of interest, and only later contemplates a more efficient *bottom-up,* possibly table-driven, implementation. We have followed this practice in designing an initial top-down parser for LDC. Backtracking is used when a command fails because either the wrong type of input word has been encountered or an inconsistency has been created by compatibility information. Since semantic information is being used for disambiguation during parsing, our parser returns the first (and only) fully acceptable structure if finds.

Our *kernel* parser, which deals with the seven command types but ignores augmentations, was designed, coded and tested by one person in fewer than 10 days and occupies less than four pages of lightly commented LISP code. The full repertoire of augmentations, on the other hand, required roughly six man-months of effort to design and implement and occupies about eight additional pages of code. This is partly due to the additions and refinements being made to the formalism as the parser developed. Detailed information on the construction of the initial LDC parser is given in Ballard and Tinkham (1983).

4.3. Experiments with the LDC Grammar and Parser

Having completed the parser and constructed a suitably broad grammar for use in several layered domains, we have begun to experiment with our parser in the manner of Slocum (1981) to see how significantly certain of our

pruning methods can reduce the time complexity of parsing. Some of the pruning techniques we have studied are

1. the use of a *Start* file that tells what grammatical categories may begin each syntax routine,
2. the use of *local compatibility checking* Section 3.3.1), and
3. the use of *non-local compatibility checking* (Sections 3.3.2 and 3.3.3).

In addition to efficiency interests, the latter two techniques influence what parse structure is found for spurious or ambiguous inputs, which is also a concern worthy of study. However, we have postponed an investigation of parsing "accuracy" until we have begun to elicit actual inputs from realistic users. We now describe several experiments with the LDC grammar and parser.

4.3.1 Usefulness of the "Start" file

As an initial experiment with the LDC grammar and parser, we constructed representative inputs of varying lengths from each of the three domains mentioned above and ran the parser both with and without the pruning capability provided by the *Start* file. This file gives information analogous to that provided by the *First* relation of conventional LL(1) compiler theory, and is created automatically by an off-line process that traverses a grammar (see Section 6.1). The actual inputs that were used follow.

Document domain:

"the first two sentences"
"the longest sentence in the first paragraph"
"the shortest sentence in the first paragraph containing a misspelled word"

Final Grades domain:

"the best student"
"the best undergraduate Ballard taught in AI"
"the student with the highest grade in the course Mary made a B+ in"

Matrix domain:

"the first two rows"
"the entries added up by command 7"
"the second entry the last four commands added five to that is positive in matrix 2"

Statistics were gathered for both the number of grammar commands executed, and the actual running time. In particular, we had the parser keep track of the number of times an attempt was made to execute a Call command, and also how many times the test of whether to invoke the body of the Call succeeded. At the system level, we had UNIX[2] report the amount of time (in seconds) spent by the parser for each of the 18 runs (9 sentences each run with and without Start information). The results of this study are given in Table 1. As can be seen, the use of the *Start* file led to a significant reduction both in the number of Call statements entered or executed and in the actual parsing time. In fact, for 8 of the 9 inputs, parse time was reduced by more than half. An explanation for the fairly long parse times is given in Section 5.3.

4.3.2 Expense of macro-style SEQ commands

After studying the results given above, we wondered how greatly the observed improvements were due to the presence of unnecessary Call commands. In the grammar being tested we had, for the sake of clarity, included many Call commands that were invoked at only one place in the grammar, and we wondered what the results would be if these "superfluous" Call commands were replaced by their associated bodies. Since only Call commands have "Start" lists associated with them, we conjectured that it would actually be an *advantage* to have many Call commands, especially for a lengthy Alt (alternative) command that would do lots of superfluous work. To test this hypothesis we ran an experiment to compare the original grammar against a modification of it in which the bodies of several of the superfluous Call commands were instantiated at the point of call. For reasons not relevant to this discussion, it was first necessary to re-order some of the grammar rules in order to make a fair comparison. Thus, there were three grammars to be tested. Each grammar was run both with and without the Start information for several of the sample inputs shown above. For the most complicated input, "the second entry the last four commands added five to that is positive in matrix 2", the results are given in Table 2, where the statistics for comparison are found in the second and third lines. Although the full significance of these preliminary results is not clear, it is apparent that the instantiations had

(a) a small positive effect when Start information was not being used, and
(b) a small negative effect when Start information was used.

These results lend credence to the hypothesis that introducing convenient Call commands will not lead to an increase in parsing efficiency in the presence of the Start file.

4.3.3 Expense of nonlocal compatibility checking

Upon noting the long parse times in the preceding experiments, we wondered how much of this time was being spent in nonlocal compatibility checking and whether this checking increased parse time, due to the extra work involved, or decreased it, by early pruning of erroneous potential parses. Accordingly, we considered two grammars, one with augmentations calling for nonlocal checking to be done, the other without such parameters, but identical otherwise, and compared the parsing times for each grammar for each of the four noun phrases

"the first class"
"the best undergraduate student Ballard taught in AI"
"the name of the class the student the professor taught liked"

[2]UNIX is a trademark of AT&T Bell Laboratories.

Table 1. *Reduction in Parsing Times by the Use of the "Start" File*

Domain	Sentence Length	Without Start Considered	Without Start Time	With Start Considered	With Start Entered	With Start Time	Percent Savings
Document	short	21	8.5	6	4	2.9	66
	medium	49	18.9	24	12	8.5	55
	long	80	32.3	29	16	12.2	62
Grades	short	21	8.2	6	4	2.7	67
	medium	36	14.3	16	10	6.5	55
	long	94	37.i	42	22	13.8	64
Matrix	short	21	7.4	6	4	2.4	68
	medium	29	12.0	17	8	5.5	54
Average	short		8.0			2.7	66
	medium		15.1			6.8	55
	long		36.7			16.7	55

* "the four tallest classes"

where "*" denotes an unacceptable sentence. In each case both local compatibility checking and checking of Start information were done. The results appear in Table 3. As with the experiments discussed above, these measurements have limited statistical significance due to the small sample sizes of grammars and of inputs. However, the preliminary indication is that checking for nonlocal compatibility adds more time to parsing than it saves. This suggests that the benefits of such nonlocal checking will be primarily in improved accuracy of disambiguations, rather than speed of parsing.

5. Discussion

We now summarize what we believe to be the most significant aspects of our formalism, briefly comment on the relation of our grammars to conventional ATN grammars, and finally mention some of the drawbacks to our approach.

5.1 Some advantages of our formalism

Our goals in developing the grammatical formalism discussed in this paper were, first, to prepare for transportability and, second, to be able to specify grammars in a simple, understandable, and succinct fashion. Concerning the first goal, we have seen how nountype, plurality, and various forms of compatibility information can be conveniently passed up and down a parse structure under construction. This information can prove useful in disambiguations and in subsequent semantic processing. In particular, our formalism allows us to state many restrictions of semantic grammars within a general grammar, independent of the domain(s) at hand. This is possible because the files created during knowledge acquisition, namely the dictionary and associated compatibility file, contain many types of domain-specific information that is useful in making syntactic and semantic decisions during parsing. For instance, we can write a simple and relatively short set of syntax routines for relative clauses that overgenerates (i.e. allows many spurious structures) since adequate restriction information is available in the preprocessed files. This is similar in spirit to the isolation of restriction information described in Sager and Grishman (1975).

Some of the features of our formalism which we consider desirable but not related directly to the goal of transportability are the following.

1. Our Quote and Get commands represent a *consolidation over lower-level command types* of the ATN form adopted for our previous NLC system. For instance, we use a single Get command to take the place of what would require three separate commands in the NLC grammar.
2. We have provided for *default labels* in the parse structures being built, which grammar writers can override if they choose.
3. We often allow for *lists of items where only one item would be expected.* For instance, we may ask for one of several words or parts or speech, as in (Quote (that which who)) or (Get (Num Super)), or specify several compatibility-checking augmentations in a single command.
4. We provide for *arbitrary embeddings of commands* within any of the composite commands, similar to the nestings provided in LISP and a number of modern programming languages.

Table 2. *Effect of In-line Instantiation of Superfluous Call Commands*

Grammar	Without Start Considered	Without Start Time	With Start Considered	With Start Entered	With Start Time
Original Grammar	84	40.2	54	28	25.1
Re-ordered Grammar	94	46.7	59	30	27.0
Re-ordered Grammar with Instantiated Calls	69	45.5	52	27	31.1

5. Our grammars do not require *dummy node names,* such as those occurring in typical ATN grammars.
6. We handle most restriction specifications by a small number of augmentations which cling to the seven command types. This implies the presence within our grammars of an *easily visible context-free skeleton grammar,* which one can detect without having to trace through and ignore various testing commands.
7. Due to the crispness of our grammar commands, presence of consolidations with appropriate defaults, and manner of embedding augmentations within commands, we are able to work with *relatively compact grammars,* which aids in their comprehension and manipulation.
8. The crispness of our seven commands has also allowed for *useful preprocessing of the grammar* to create the Start and Adjacency files. This form of information has been found useful by several researchers, yet is often supplied directly from the human author of the grammar, and must be updated when the grammar changes. Our files are created automatically.

5.2 Comparisons with Augmented Transition Network Formalisms

As remarked earlier, our grammatical formalism, which is based on what we regard as phrase-structure rules, bears some resemblance to ATN grammars (Woods 1970, 1980), but there are important differences. First, the notion of a network *node* is almost entirely absent from our formalism. For example, in the following typical ATN node

```
(Q2  (PUSH NP / T
     (SETR Subj *)
     (TO Q3)))
```

we note the presence of both (a) the label Q2, and (b) the reference to the successor node Q3, neither of which has a counterpart in our grammars. By eliminating such node names, we reduce the space needed to specify grammar rules, which helps make our grammars more readable. We also reduce the redundancy of the grammar representation, which makes updates easier and less error-prone. Actually, node names for ATN grammars need not be given explicitly, but their elimination requires that networks be stored as linked structures that resist convenient manipulations with standard text editors. This was the alternative representation chosen for the ATN grammars of NLC. As a final point concerning readability of grammars, we note the common practice of conveying ATN grammars by giving an actual network diagram, which by its non-linear nature cannot be so conveyed to the computer system.

A second departure of our formalism from conventional ATNs is that we have systematically avoided the temptation to introduce opportunities for making *arbitrary tests* (a provision shared, evidently, with even more glee, in various logic grammars, e.g., Pereira 1981). Such opportunities are provided for in ATN grammars by the TST node type, and also by allowing arbitrary LISP predicates at various points. As indicated in Section 4.3, we have taken pains to restrict the repertoire of conditions that may be checked, hoping that such restrictions will better direct the grammar writer to the salient features that need to be considered. Our interest is more to provide a formalism that can be used easily and profitably than to provide something with formal Turing ability. Thus, we have sought to identify a small but adequate number of easily understandable conditions to be checked when doing a Get, Call, and so forth. Another departure is that many of our augmentations, most conspicuously that described in Section 3.3.3, amounts to setting up *demons* to monitor the process of a routine about to be called so that appropriate information passing will occur. This allows a routine to be written just once, yet lead to different sorts of compatibility checking based upon the context in which it is invoked. This philosophy appears to differ from that upon which the essentially bottom-up SEND mechanism of ATNs is based.

5.3 Limitations of our approach

We now mention some of the difficulties we have encountered with our approach to transportable parsing. Before doing so, we note that the excessive parse times suggested in Tables 1-3 are no longer a problem, since our parser now runs on a VAX in compiled LISP (and the tables were

Table 3. *Time Spent in Nonlocal Compatibility Checking*

Sentence	Without Checking	With Checking	Percent Overhead
Short	4.6	5.2	13.0
Medium	10.2	13.7	34.3
Long	18.8	25.7	36.7
Incorrect	8.7	9.0	3.4

created on a 16-bit PDP-11/70 running fully interpreted LISP without hash tables). At present, parse times are hovering at around one second.

First, we note that at present the noun positions in case frames and prepositional triples consist of *object types* of the domain at hand. Although this level of abstraction is somewhat more general than specific surface words, we are considering means whereby the user can make use of the hierarchical relationships among domain entities to reduce the redundancy of some case frame specifications. For example, "person" might be used to mean "either student or instructor", and in fact entire taxonomies might be introduced.

Second, we have observed a few places in our grammar, especially the treatment of quantifiers, determiners, and their allowable co-occurrences, where the old ATN formalism we used for the NLC grammar would result in more concise and not necessarily less perspicuous grammars. For instance, "all/each of the" is okay but not "every of the"; "every/each one of" is okay but not "all one of the"; and "all the" is okay but not "each/every the". In fact, we at times sketch out a new class of constructs to be incorporated into our grammar in a pidgin transition network form, then seek ways to linearize.

Finally, we note that several existing grammatical formalisms, such as string grammar and ATN grammar, have enjoyed more than a decade of refinement and have led to quite efficient parsing mechanisms. We expect the principles we have developed to undergo similar improvements during the course of our research. Though much of the current LDC grammar was derived from corresponding parts of the ATN grammar of NLC, there is no reason to doubt that we will be able to adopt relevant portions of other large grammars (e.g. Robinson 1982; Sager 1981), thus taking advantage of previous research directed more toward the linguistic versus domain-modeling aspects of natural language syntax.

6. A Glimpse at the Overall LDC Environment

The design of LDC began in 1981 with the goal of allowing a system designer to quickly create interfaces to new domains by supplying vocabulary and domain structure information to a customizing module. However, LDC soon developed into a system where all information about a new domain is acquired from prospective users, as had been done for REL (Thompson and Thompson 1975) and KLAUS (Haas and Hendrix 1980). The system is loosely based upon strategies developed for our English-language programming system NLC (Ballard and Biermann 1979; Biermann and Ballard 1980; Sigmon 1981; Fink 1982; Geist, Kraines and Fink 1982; and Biermann, Ballard and Sigmon 1983), and has been designed to provide a natural language query capability for office domains whose data are stored on the computer as informally structured text-edited data files (Ballard and Lusth 1984). The system comprises

(a) a knowledge acquisition module called "Prep",
(b) a highly-parameterized English-Language processor, and
(c) a knowledge-based Retrieval module.

We now summarize the operation of the knowledge acquisition and English processing components, referring the reader to Ballard and Lusth (1984) for details concerning retrieval processing.

6.1 Knowledge acquisition

The initial interaction between a user and LDC, which involves telling the system about a new domain, consists of a knowledge-acquisition session with the preprocessor, which we call "Prep". In particular Prep asks for

(1) the names of each type of "entity" (object) of the domain;
(2) the nature of the relationships among entities;
(3) the English words that will be used as nouns, verbs, and modifiers; and
(4) morphological and semantic properties of these new words.

For example, in describing a building organization domain, a user might supply to Prep the structural and language-related information that "room" is a primitive entity type; "large", "small", and "vacant" are adjective modifiers; "conference" is a noun modifier; "office" is a noun referring to some objects of type "room"; and that rooms have "wing" as a higher-level domain entity. At

this point semantic specifications for the modifiers mentioned above are given, and morphological variants are supplied, e'g. "rooms" is the plural of "room", "larger" is the comparative form of "large", "vacant" has no associated comparative, and so forth.

Having completed a session with the user, Prep digests its newly acquired information to produce various files to be used during subsequent processing of English inputs by the English-Language processor. The first file created by Prep is the *Dictonary,* which is used as input by the scanner, and whose format was suggested in Section 2.2. Some dictionary listings are provided for common, domain-independent terms such as articles, ordinals, and certain verbs.

The second file created by Prep gives *Compatibility* information on (a) verb case frames and (b) expected prepositional attachments. Verb case frames are passed along almost directly from the user's specifications, whereas prepositional attachments are determined by heuristics related to layered domains. Case frames are specified by subject, verb, optional particle, optional object, and optional preposition-argument pair, while legitimate prepositional attachments are specified by entity-preposition-entity triples.

The third file created by Prep tells the parser what dictionary words can *Start* each grammatical unit (syntax routine) of the grammar. This information is roughly equivalent to that provided by the LL(1) tables of compiler theory. However, Prep takes the attached features into account, and must also account for multiple word meanings. Several existing natural language processors have found such Start information useful, though to our knowledge most implementations have had the information supplied by hand from the system designer, rather than automatically constructed from a novel dictionary file and for an evolving grammar, as we provide for. The entire code needed to create the Start file is about 30 LISP lines. Its brevity and conceptual simplicity are due to the crispness of our phrase-structure formalism.

The fourth file created by Prep is a *Adjacency* file that tells the scanner which dictionary words a given word may be followed by in a legal input. This form of information, which is being used by the current "voice scanner" of our related NLC system (Biermann 1981), will likely prove useful when we introduce voice input to LDC. To our knowledge the Adjacency file is without counterpart in conventional compiler design. Although the recursive routines responsible for creating the Adjacency file resemble those related to the Start file, they involve combinatoric considerations and are much more complicated.

Finally, two additional files created by Prep, which are not involved in parsing, supply *domain structure* information for semantic processing and *adjective and verb semantics* for retrieval.

6.2 English-language processing

The organization of the natural language processing portion of LDC resembles that of interpreters for conventional programming languages by exhibiting a linear sequence of modules without complex interaction among them. A pictorial overview of the English-language processor and retrieval module is given in Figure 2, which also gives an idea of how the domain-specific files produced during the user's interactive session with Prep are used. As suggested there, the scanner and parser supply the semantics module with an internal rendering of the user's input, whereupon semantics appeals to a retrieval component, and then sends the top-level response to the output module to be printed in user-readable form. We now comment briefly upon each module involved in English-language processing.

6.2.1 Scanning

The role of the *scanner* is to identify each word of the typed or spoken input and retrieve information about it from the Dictionary file, which will have been created by Prep as described in Section 2.1. For words having more than one dictionary listing, all possible meanings (readings) are sent to the parser, where context will be used to select one of them. For the sake of run-time efficiency, morphological variants (inflections) of domain-specific terms will already have been stored in the dictionary, so run-time stemming is not needed.

The existing LDC scanner assumes typed input but, as described in Biermann et al. (1983), we have used a Nippon DP-200 voice recognition unit, and more recently a continuous-speech Verbex device, with our previous NLC system, and its introduction into LDC is being contemplated. When this occurs, some of the word meanings will have been taken from a "synophone" list

6.2.2 Parsing

As an example of how parse structures are built up, consider the noun phrase

"the largest white house in Ohio"

for which the scanner will have supplied information such as

(the Art the)
(largest Super large (nt building parcel))
(white Adj white (nt building))
(house Subtype house (sp sing) (nt building))
(in Prep in)
(Ohio Nounval Ohio (sp sing) (nt state))

For simplicity we have given just one reading for each word, but in general each word may have several.

When the NP routine is entered, an initial structure with a label of NP but null feature and item lists is created.

(NP (feats))

Since our present grammar checks for the word "the" using Quote, rather than Get, no parser output occurs when "the" is seen, although optional output is allowed by

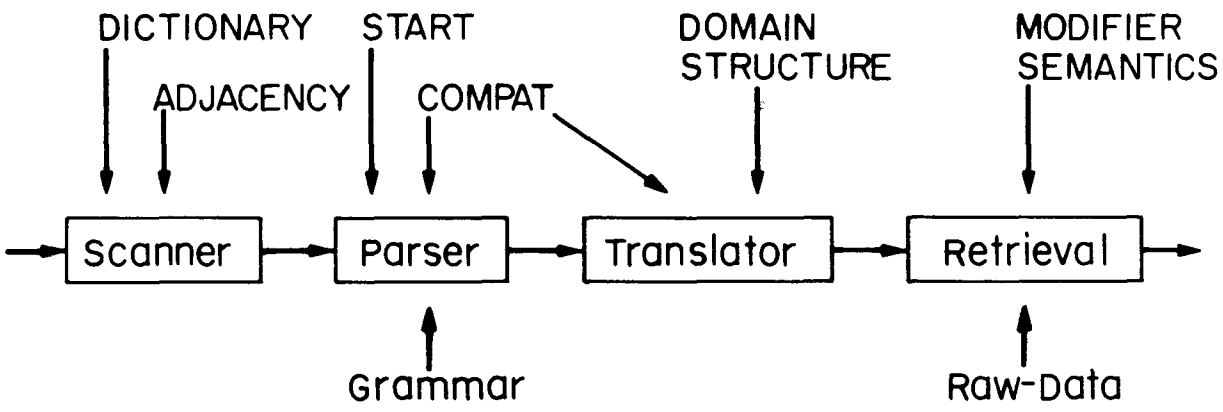

Figure 2. *Pictorial Overview of the English-Language Processor and Retrieval Module*

the Quote command. Next, since the dictionary listing for "largest" indicates that it can modify entities of type building and parcel, the parse structure upon parsing the word "largest" will become

```
(NP  (feats (nt building parcel))
     (Super . large))
```

Now since "white" is marked as an adjective that may only modify entities of type building, its incorporation will lead to an updated parse structure of

```
(NP  (feats (nt building))
     (Adj . white)
     (Super . large))
```

Here the fact that more recent phrase elements appear to the left of previous ones is an artifact of the LISP implementation, and the order is basically ignored during post-parser processing. Next, the word "house" will be processed by the grammar command

```
(Get Subtype Head (head subtype))
```

giving rise to

```
(NP  ((head subtype) (sp sing) (nt building))
     (Head . house)
     (Adj . white)
     (Super . large))
```

Next, the post-modifier "in Ohio" will be processed and, upon returning from a recursive call to the noun phrase grammar, the new parse structure will be

```
(NP  (feats (head subtype) (sp sing) (nt building))
     (PrepPh (feats (nl    (Head Prep Arg)
                           (building in state)))
             (NP  (feats   (head nounval)
                           (nt state) (sp sing))
                  (Head . Ohio))
             (Prep . in))
     (Head . house)
     (Adj . white)
     (Super . large))
```

The interested reader may wish to reconsider Figure 1 for an example of a complete parse structure for a more complicated input. Further details on the mechanisms of parsing are given in Ballard and Tinkham (1983).

6.2.3 The translator

Traditional approaches to natural language database interface seek to provide access to a user's data by constructing a formal query to an existing retrieval system that arose in the database community without the intention of an eventual English-language interface. Our approach differs in that we have built our own retrieval processor and endowed it with the ability to act as a *knowledge-base* by directly processing complex semantics of English modifiers (Ballard 1984). In this manner, we avoid many of the awkward requirements of the typical "translation" process from formatted English (e.g. parse structures) to a formal query. Thus, our translation process involves traversing the nodes of a parse structure in a prescribed order (e.g. relative clauses are processed before adjectives, which are processed before ordinals and numbers). Translation is carried out recursively, starting with a top-level noun phrase, and proceeding to embedded phrases. A complete discussion of the query language produced can be found in Ballard and Lusth 1984.

6.3 Current status of the LDC system

The initial version of the knowledge acquisition module of LDC was completed in the fall of 1982, and the separately tested modules of the English-language processor and the retrieval module were integrated in May 1983 to form a complete system. Since that time, the system has been run by the system authors and gives a real-time response to each input in under a minute while time-sharing on a heavily-loaded 16-bit PDP-11/70 minicomputer running UNIX. All coding of LDC has been done in a local dialect of LISP, except for the retrieval module which was first written in Pascal and later re-written in C. Some of the domains which we have been using to test the modules of LDC, along with sample noun phrase inputs for these domains, were mentioned in Section 4.3.1.

At present, most work with LDC will is being carried out by the first author at AT&T Bell Laboratories, where a conversion from Duke LISP to Franz LISP has enabled the system to run on a VAX computer. Both Duke and Bell Labs have acquired Symbolics 3670 LISP machines, and a Lisp Machine version of the system, possibly resulting from redesign as well as recoding, is likely. In any event, the system is expected to undergo substantial enhancements in both syntactic and semantic coverage during the coming months.

References

Ballard, B. 1982 A "domain class" approach to transportable natural language processing. *Cognition and Brain Theory,* 5 (3): 269-287.

Ballard, B. 1984 The syntax and semantics of user-defined modifiers in a transportable natural language processor. *Proceedings of Coling84.* Stanford University (July): 52-56.

Ballard, B. and Biermann, A. 1979 Programming in natural language: NLC as a prototype. *ACM National Conference,* Detroit, Mich.: 228-237.

Ballard, B. and Lusth, J. 1983 An English-language processing system that "learns" about new domains. *National Computer Conference:* 1-46.

Ballard, B. and Lusth, J. 1984 The design of DOMINO: a knowledge-based retrieval module for transportable natural language access to personal databases. *Proceedings of the Workshop on Expert Database Systems,* Kiawah Island, South Carolina.

Ballard, B.; Lusth, J.; and Tinkham, N. 1984 LDC-1: a transportable, knowledge-based natural language processor for office environments. *ACM Transactions on Office Information Systems,* 2(1): 1-25.

Ballard, B.; Lusth, J.; and Tinkham, N. 1984 Transportable English language processing for office environments. *National Computer Conference.*

Ballard, B. and Tinkham, N. 1983 A phrase-structured grammatical formalism for transportable natural language processing. Technical Report CS-1983-4, Department of Computer Science, Duke University (April).

Bates, M. and Bobrow, R. 1983 A transportable natural language interface for information retrieval. *Sixth Annual International ACM SIGIR Conference,* Washington, D.C.

Biermann, A. 1981 Natural Language Programming. *Nato Advanced Study Institute on Automatic Program Construction,* Bonas, France (September 28 to October 10).

Biermann, A. and Ballard, B. 1980 Toward natural language computation. *American Journal of Computational Linguistics,* 6(2): 71-86.

Biermann, A.; Ballard, B.; and Sigmon, A. 1983 An experimental study of natural language programming. *International Journal of Man-Machine Studies* 18 (1): 71-87.

Biermann, A.; Rodman, R.; Ballard, B.; Betancourt, T.; Bilbro, G.; Deas, H.; Fineman, L.; Fink, P.; Gilbert, K.; and Heidlage, F. 1983 Interactive natural language processing: a pragmatic approach. *Conference on Applied Natural Language Processing,* Santa Monica, Ca.: 180-191.

Bobrow, R. and Webber, B. 1980 Knowledge representation for syntactic/semantic processing. *First Annual Conference on Artificial Intelligence,* Stanford University.

Colmerauer, A. 1978 Metamorphosis grammars. In Bolc, Ed., *Natural Language Communication with Computers.* Springer-Verlag.

Fink, P. 1982 Conditionals in a natural language system. Master's thesis, Department of Computer Science, Duke University.,

Geist, R.; Kraines, D.; and Fink, P. 1982 Natural language computation in a linear algebra course. *National Educational Computer Conference:* 203-208.

Ginsparg, J. 1983 A robust portable natural language data base interface. *Conference on Applied Natural Language Processing,* Santa Monica, Ca.: 25-30.

Grosz, B. 1983 TEAM: A transportable natural language interface system. *Conference on Applied Natural Language Processing,* Santa Monica, Ca.: 39-45.

Haas, N. and Hendrix, G. 1980 An approach to acquiring and applying knowledge. *First National Conference on Artificial Intelligence,* Stanford University: 235-239.

Heidorn, G. 1975 Augmented phrase-structure grammars. In Webber, B. and Schank, R., Eds., *Theoretical Issues in Natural Language Processing:* 1-5.

Hendrix, G. and Lewis, W. 1981 Transportable natural-language interfaces to databases. *Proceedings of the 19th Annual Meeting of the ACL,* Stanford University: 159-165.

Kimball, J. 1972 Seven principles of surface structure parsing in natural language. *Cognition* 2 (1): 15-47.

Marcus, M. 1980 *A Theory of Syntactic Recognition for Natural Language.* MIT Press, Cambridge, MA.

Mark, W. 1981 Representation and inference in the Consul system. *International Joint Conference on Artificial Intelligence.*

Pereira, F. 1981 Extraposition grammars. *Americal Journal of Computational Linguistics* 7(4): 243-256.

Pratt, V. 1975 LINGOL - A progress report. *International Joint Conference on Artificial Intelligence:* 422-428.

Rieger, C. and Small, S. 1979 Word expert parsing. *International Joint Conference on Artificial Intelligence:* 723-728.

Robinson, J. 1982 DIAGRAM: A grammar for dialogues. *Communications of the ACM* 25(1): 27-47.

Sager, N. 1981 *Natural Language Information Processing: A Computer Grammar of English and Its Applications.* Addison-Wesley.

Sager, N. and Grishman, R. 1975 The restriction language for computer grammars of natural language. *Communications of the ACM* 18: 390-400.

Sigmon, A. 1981 The semantics of looping structures in natural language computation. Master's thesis, Department of Computer Science, Duke University.

Slocum, J. 1981 A practical comparison of parsing strategies. *Annual Meeting of the Assoc. for Computational Linguistics,* 1-6.

Thompson, B. and Thompson, F. 1981 Shifting to a higher gear in a natural language system. *National Computer Conference:* 657-662.

Thompson, B. and Thompson, F. 1983 Introducing ASK, a simple knowledgeable system. *Conference on Applied Natural Language Processing,* Santa Monica, CA: 17-24.

Thompson, F. and Thompson, B. 1975 Practical natural language processing: the REL system as prototype. In Rubinoff, M. and Yovits, M., Eds., *Advances in Computers,* Vol. 3, Academic Press.

Wilczynski, D. 1981 Knowledge acquisition in the Consul system. *International Joint Conference on Artificial Intelligence.*

Winograd, T. 1972 *Understanding Natural Language.* Academic Press.

Woods, W. 1970 Transition network grammars for natural language analysis. *Communications of the ACM,* 13(10): 591-606.

Woods, W. 1980 Cascaded ATN grammars. *American Journal of Computational Linguistics* 6(1): 1-12.