CMU-CS-81-118

Multi-Strategy Parsing

And Its Role in Robust Man-Machine Communication

Philip J. Hayes and Jaime G. Carbonell
Carnegie-Mellon University
Pittsburgh, PA 15213

13 May 1981

Abstract

Robust natural language interpretation requires strong semantic domain models, "fail-soft" recovery heuristics, flexible control structures, and focused user interaction when automatic correction proves infeasible. Although single-strategy parsers have met with some success, a multi-strategy approach, with strategies selected dynamically according to the type of construction being parsed at any given time, is shown to provide a higher degree of flexibility, redundancy, and ability to bring task-specific domain knowledge (in addition to general linguistic knowledge) to bear on both grammatical and ungrammatical input. This construction-specific, multi-strategy approach can also help provide tightly focused interaction with the user in cases of semantic or structural ambiguity by allowing such ambiguities to be represented without duplication of unambiguous material. The approach also aids in task-specific language development by allowing direct interpretation of languages defined in terms natural to the task domain. A parsing algorithm integrating case-frame instantiation and partial pattern matching strategies is presented. The algorithm can deal with conjunctions, fragmentary input, and ungrammatical structures, as well as less exotic, grammatically correct input.

This research was sponsored in part by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory under contract F33615-78-C-1551, and in part by the Air Force Office of Scientific Research under Contract F49620-79-C-0143. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of DARPA, the Air Force Office of Scientific Research or the US government.

1. Introduction

When people use language spontaneously, they seldom respect grammatical niceties. Instead of producing sequences of complete, grammatically well-formed sentences, they typically omit or repeat words, break off in mid-utterance, interject spurious phrases, speak in fragments, or otherwise use incorrect grammar. Whereas people experience little difficulty comprehending ungrammatical utterances, most natural language computer systems are unable to process errorful input at all. Such inflexibility in parsing is a serious impediment to the use of natural language in interactive computer systems. Accordingly, we [6] and other researchers including Weischedel and Black [14], and Kwasny and Sondheimer [9], have attempted to produce flexible parsers, i.e. parsers that can accept ungrammatical input, correcting the errors when possible, and generating several alternative interpretations for later selection by the user if appropriate.

While different in many ways, all prior approaches to flexible parsing operate by applying a uniform parsing process to a uniformly represented grammar. Because of the linguistic performance problems involved, no uniform procedure can be as simple and elegant as the methods followed by parsers incorporating a pure linguistic competence model, such as Parsifal [10]. Indeed, flexible parsing based on a uniform grammar may involve several strategies applied in a predetermined order when the input deviates from the grammar, but the choice of strategy is never sensitive to the specific type of construction being parsed. In light of experience with FlexP [6], our own flexible parser for limited-domain task-oriented languages, we have come to the conclusion that such uniformity is unnecessary, inefficient, and inferior in functionality to an approach capable of selecting among several parsing strategies, always applying the one most appropriate to the particular type of construction being parsed at the time. A parser using this construction-specific approach might, for instance, use linear pattern matching to parse idiomatic phrases, or specialized noun phrases such as names, dates, or addresses (see also [4]), but switch to a special case-oriented parsing strategy to deal with case constructions, such as noun phrases with trailing prepositional phrases, or imperative phrases. The underlying principle is simple: ***The appropriate knowledge must be brought to bear at the right time -- and it must not interfere at other times.***

The advantages we claim for a multi-strategy construction-specific approach to parsing (and with which the bulk of this paper is concerned) include:

- greater accuracy and efficiency in the flexible parsing of ungrammatical input
- greater efficiency in parsing grammatical input
- an ability to represent ambiguity without duplicating unambiguous parts, making it easier to indicate the exact source and nature of the ambiguity to the user, and thus facilitating interactive correction
- improved task-specific language development by allowing language definitions, natural in terms of the underlying domain, to be used directly as a grammar without prior compilation into a uniform formalism

In addition to explaining the theoretical advantages of a construction-specific approach, this paper presents the design of a flexible parser that integrates several distinct construction-specific parsing strategies. Our objective is not to design a totally general and complete parser applicable to all of English, but rather to develop a flexible and robust task-oriented parser, applicable to a wide range of tasks, where domain knowledge and specialized constructions can be exploited and integrated with more general syntax and semantics. The initial application domain for the parser is in an interface to various computer subsystems (or tools). This interface and, therefore, the parser should be adaptable to new tools by substituting domain-specific data bases (called "tool descriptions") that govern the behavior of the interface, including the invocation of parsing strategies, dictionaries and concepts, rather than requiring any domain adaptations to the interface system itself.

The following sections substantiate the advantages we claim for construction-specific parsing by describing some problems that arose with our original flexible parser, and showing how these problems are ameliorated by a construction-specific approach. First, we describe the structure of FlexP and discuss problems that arose in parsing both grammatical and ungrammatical input using uniform parsing procedures and grammar representations. Secondly, we consider the representation of ambiguities that can arise through flexible parsing, and the ways in which the form of representation can affect interactive resolution of these ambiguities. Thirdly, we discuss the interactions between grammar representations and task-oriented language definition and development. Finally, we make the preceding sections more concrete by presenting the design of two construction-specific parsing strategies and the mechanism that dynamically selects between them while parsing task-oriented natural language imperative constructions; imperatives were chosen because commands and queries given to task-oriented natural language front ends often take that form [6].

2. Construction-Specific Flexible Parsing

Our present flexible parser, which we call FlexP, is intended to parse correctly input that corresponds to a fixed grammar, and also to deal with input that deviates from that grammar by erring along certain classes of common ungrammaticalities. Because of these goals, the parser is based on the combination of two uniform parsing strategies: bottom-up parsing and pattern-matching. The choice of a bottom-up rather than a top-down strategy was derived from our need to recognize isolated sentence fragments, rather than complete sentences, and to detect restarts and continuations after interjections. However, since completely bottom-up strategies lead to the consideration of an unnecessary number of potentially spurious alternatives in correct input, the algorithm used allowed some of the economies of top-down parsing for non-deviant input. Technically speaking, this made the parser *left-corner* rather than bottom-up. We chose to use a grammar of linear patterns rather than, say, a transition network for three reasons: 1) Pattern-matching meshes well with bottom-up parsing by allowing lookup of a pattern from the presence in the input of any of its constituents. 2) Pattern-matching facilitates recognition of utterances with

omissions and substitutions when patterns are recognized on the basis of partial matches. 3) Pattern-matching is necessary for the recognition of idiomatic phrases. More details of the justifications for these choices can be found in [6].

FlexP has been tested extensively in conjunction with a gracefully interacting interface to an electronic mail system [1]. "Gracefully interacting" means that the interface appears friendly, supportive, and robust to its user. In particular, graceful interaction requires the system to tolerate minor input errors and typos, so a flexible parser is an important component of such an interface. While FlexP performed this task adequately, the experience turned up some problems related to the major theme of this paper. These problems are all derived from the incompatibility between the uniform nature of the grammar representation and the kinds of flexible parsing strategies required to deal with the inherently non-uniform nature of some language constructions. In particular:

- Different elements in the pattern of a single grammar rule can serve radically different functions and/or exhibit different ease of recognition. Hence, an efficient parsing strategy should react to their apparent absence, for instance, in quite different ways.

- The representation of a single unified construction at the language level may require several linear patterns at the grammar level, making it impossible to treat that construction with the integrity required for adequate flexible parsing.

The second problem is directly related to the use of a pattern-matching grammar, but the first would arise with any uniformly represented grammar applied by a uniform parsing strategy.

For our application, these problems manifested themselves most markedly by the presence of case constructions in the input language. Thus, our examples and solution methods will be in terms of integrating case-frame instantiation with other parsing strategies. Consider, for example, the following noun phrase with a typical postnominal case frame:

"the messages from Smith about ADA pragmas dated later than Saturday".

The phrase has three cases marked by "from", "about", and "dated later than". This type of phrase is actually used in FlexP's current grammar, and the basic pattern used to recognize descriptions of messages is:

```
<?determiner *MessageAdj MessageHead *MessageCase>
```

which says that a message description is an optional (?) determiner, followed by an arbitrary number (*) of message adjectives followed by a message head word (i.e. a word meaning "message"), followed by an arbitrary number of message cases. In the example, "the" is the determiner, there are no message adjectives, "messages" is the message head word, and there are three message cases: "from Smith", "about ADA pragmas", and "dated later than". Because each case has more than one component, each must be recognized by a separate pattern:

```
<%from Person>
<%about Subject>
<%since Date>
```

Here % means anything in the same word class, "dated later than", for instance, is equivalent to

"since" for this purpose.

These patterns for message descriptions illustrate the two problems mentioned above: the elements of the case patterns have radically different functions - the first elements are case markers, and the second elements are the actual subconcepts for the case. Also, a single construction at the language level is spread over several patterns in the grammar. What consequences does this have for the parsing process? Because the parser has no information about the relationship between the cases and the top-level pattern (other than that the results of the case patterns match the last element in the top-level pattern), several powerful, but specialized, strategies for dealing with (regular or irregular) case constructions cannot be employed. For instance, since case indicators are typically much more restricted in range of expression, and therefore much easier to recognize than their corresponding subconcepts, a plausible strategy for a parser that "knows" about case constructions is to scan input for the case indicators, and then parse the associated subconcepts top-down. This strategy is particularly valuable if one of the subconcepts is malformed or of uncertain form, such as the subject case in our example. Neither "ADA" nor "pragmas" is likely to be in the vocabulary of our system, so the only way the end of the subject field can be detected is by the presence of the case indicator "from" which follows it. However, the present parser cannot distinguish case indicators from case fillers - both are just elements in a pattern with exactly the same computational status, and hence it cannot use this strategy.

Section 5 describes an algorithm for parsing case constructions flexibly. At the moment, the algorithm works only on a mixture of case constructions and linear patterns, but eventually we envisage a number of specific parsing algorithms, one for each of a number of construction types, all working together to provide a more complete flexible parser. Section 5 also points out a number of efficiencies that its construction-specific approach makes possible in the parsing of grammatical input.

Below, we list a number of the parsing strategies that we envisage might be used. Most of these strategies exploit the constrained task-oriented nature of the input language:

- **Case-Frame Instantiation** is necessary to parse general imperative constructs and noun phrases with postnominal modifiers. This method has been applied before with some success to linguistic or conceptual cases [12] in more general parsing tasks. However, it becomes much more powerful and robust if domain-dependent constraints among the cases can be exploited. For instance, in a file-management system, the command "Transfer UPDATE.FOR to the accounts directory" can be easily parsed if the information in the unmarked case of *transfer* ("update.for" in our example) is parsed by a file-name expert, and the destination case (flagged by "to") is parsed not as a physical location, but a logical entity inside a machine. The latter constraint enables one to interpret "directory" not as a phone book or bureaucratic agency, but as a reasonable destination for a file in a computer.

- **Semantic Grammars** [7] prove useful when there are ways of hierarchically clustering domain concepts into functionally useful categories for user interaction. Semantic

grammars, like case systems, can bring domain knowledge to bear in disambiguating word meanings. However, the central problem of semantic grammars is non-transferability to other domains, stemming from the specificity of the semantic categorization hierarchy built into the grammar rules. This problem is somewhat ameliorated if this technique is applied only to parsing selected individual phrases [13], rather than being responsible for the entire parse. Individual constituents, such as those recognizing the initial segment of factual queries, apply in may domains, whereas a constituent recognizing a clause about file transfer is totally domain specific. Of course, this restriction calls for a different parsing strategy at the clause and sentence level.

- **(Partial) Pattern Matching** on strings, using non-terminal semantic-grammar constituents in the patterns, proves to be an interesting generalization of semantic grammars. This method is particularly useful when the patterns and semantic grammar non-terminal nodes interleave in a hierarchical fashion.

- **Transformations to Canonical Form** prove useful both for domain-dependent and domain-independent constructs. For instance, the following rule transforms possessives into "of" phrases, which we chose as canonical:

```
[<ATTRIBUTE> in possessive form, <VALUE> legitimate for attribute]
                    =>
[<VALUE> "OF" <ATTRIBUTE> in simple form]
```

  Hence, the parser need only consider "of" constructions ("file's destination" => "destination of file"). These transforms simplify the pattern matcher and semantic grammar application process, especially when transformed constructions occur in many different contexts. A simple form of string transformation was present in PARRY [11].

- **Target-specific methods** may be invoked to parse portions of sentences not easily handled by the more general methods. For instance, if a case-grammar determines that the case just signaled is a proper name, a special name-expert strategy may be called. This expert knows that names can contain unknown words (e.g., Mr. Joe Gallen D'Aguila is obviously a name with D'Aguila as the surname) but subject to ordering constraints and morphological preferences. When unknown words are encountered in other positions in a sentence, the parser may try morphological decomposition, spelling correction, querying the user, or more complex processes to induce the probable meaning of unknown words, such as the project-and-integrate technique described in [2]. Clearly these unknown-word strategies ought to be suppressed in parsing person names.

3. The Representation of Ambiguity and Focused Interaction

If a flexible parser being used as part of an interactive system cannot correct ungrammatical input with reasonable certainty, then the system user must be involved in the resolution of the difficulty or the confirmation of the parser's correction. The approach taken by Weischedel and Black [14] in such situations is to inform the user about the nature of the difficulty, in the expectation that he will be able to use this information to produce a more acceptable input next time, but this can involve the user in substantial retyping. A related technique, adopted by the COOP system [8], is to paraphrase back to the user the one or more parses that the system has produced from the user's input, and to

allow the user to confirm the parse or select one of the ambiguous alternatives. This approach still means a certain amount of work for the user. He must check the paraphrase to see if the system has interpreted what he said correctly and without omission, and in the case of ambiguity, he must compare the several paraphrases to see which most closely corresponds to what he meant, a non-trivial task if the input is lengthy and the differences small.

Experience with our own flexible parser, FlexP, suggests that the way requests for clarification in such situations are phrased makes a big difference in the ease and accuracy with which the user can correct his errors, and that the user is helped most by a request focusing as tightly as possible on the exact source and nature of the difficulty. Accordingly, we have adopted the following simple principle for our new flexible parser: ***when the parser cannot uniquely resolve a problem in its input, it should ask the user for a correction in as direct and focused a manner as possible.*** Moreover, this request for clarification should not prejudice the processing of the rest of the input, either before or after the problem occurs. In other words, if the system cannot parse one segment of the input, it should be able to bypass it, parse the remainder, and then ask the user to restate that *and only that* segment of the input. Similarly, if a small part of the input is missing or garbled and there are a limited number of possibilities for what ought to be there, the parser should be able to indicate the list of possibilities together with the context from which the information is missing rather than making the user compare several complete paraphrases of the input that differ only slightly.

In the remainder of this section, we show how a construction-specific approach to parsing can contribute to focused interaction in cases of error. We restrict our attention to situations in which a flexible parser can correct an input error or ungrammaticality, but only to within a constrained set of alternatives. We show why it is difficult for a flexible parser based on uniform methods to produce a focused ambiguity resolution request for the user to distinguish between such a set of corrections, and how a construction-specific parser can produce one much more easily. Realizing the advantages afforded by the construction-specific approach requires that special representations be devised for all the structural ambiguities that each construction type can give rise to. We illustrate these arguments with examples involving case constructions. The example ambiguity representations for case constructions are designed for use with the parser described in Section 5.

The following input is typical for an electronic mail system interface [1] with which FlexP was extensively used:

the messages from Fred Smith that arrived after Jon 5

The fact that this is not a complete sentence in FlexP's grammar causes no problem. The only real difficulty comes from "Jon", which should presumably be either "Jun" or "Jan". FlexP's spelling corrector can come to the same conclusion, so the output contains two complete parses which are passed onto the next stage of the mail system interface. The first of these parses looks like:

```
[DescriptionOf: Message
 Sender: [DescriptionOf: Person
          FirstName: fred
          Surname: smith
         ]
 AfterDate: [DescriptionOf: Date
             Month: january
             DayOfMonth: 5
            ]
]
```

This schematized property list style of representation should be interpreted in the obvious way. Since FlexP operates by bottom-up pattern matching of a semantic grammar of rewrite rules, it can parse directly into this form of representation, which is the form required by the next phase of the interface.

If the next stage has access to other contextual information which allows it conclude that one or other of these parses was what was intended, then it can proceed to fulfill the user's request. Otherwise it has little choice but to ask a question involving paraphrases of each of the ambiguous interpretations, such as:

Do you mean:
1. the messages from Fred Smith that arrived after January 5
2. the messages from Fred Smith that arrived after June 5

Because it is not focused on the source of the error, this question gives the user very little help in seeing where the problem with his input actually lies. Furthermore, the system's representation of the ambiguity as several complete parses gives it very little help in understanding a response of "June" from the user, a very natural and likely one in the circumstances. In essence, the parser has thrown away the information on the specific source of the ambiguity that it once had, and would again need to deal adequately with that response from the user. The recovery of this lost information would require a complicated (if done in a general manner) comparison between the two complete parses.

One straightforward solution to the problem is to augment the output language with a special ambiguity representation. The output from our example might look like:

```
[DescriptionOf: Message
 Sender: [DescriptionOf: Person
          FirstName: fred
          Surname: smith
         ]
 AfterDate: [DescriptionOf: Date
             Month: [DescriptionOf: AmbiguitySet
                     Choices: (january june)
                    ]
             DayOfMonth: 5
            ]
]
```

This representation is exactly like the one above except that the Month slot is filled by an AmbiguitySet record. This record allows the ambiguity between january and june to be confined to

the month slot where it belongs rather than expanding to an ambiguity of the entire input as in the first approach we discussed. By expressing the ambiguity set as a disjunction, it would be straightforward to generate from this representation a much more focused request for clarification such as:

Do you mean the messages from Fred Smith that arrived after **January** or **June** 5?

A reply of "June" would also be much easier to deal with.

However, this approach only works if the ambiguity corresponds to an entire slot filler. Suppose, for example, that instead of mistyping the month, the user omitted or so completely garbled the preposition "from" that the parser effectively saw:

the messages Fred Smith that arrived after Jan 5

In the grammar used by FlexP for this particular application, the connection between Fred Smith and the message could have been expressed only by "from", "to", or "copied to" (or synonyms thereof). To represent the ambiguity, FlexP generates three complete parses isomorphic to the first output example above, except that Sender is replaced by Recipient and CC in the second and third parses respectively. Again, this form of representation does not allow the system to ask a focused question about the source of the ambiguity or interpret naturally elliptical replies to a request to for a distinction among the three alternatives. The previous solution is not applicable because *the ambiguity lies in the structure of the parser output rather than at one of its terminal nodes.* Using a case notation, it is not permissible to put an "AmbiguitySet" in place of one of the deep case markers.[1] To localize such ambiguities and avoid duplicate representation of unambiguous parts of the input, it is necessary to employ a representation like the one designed for use by the new flexible parser described in Section 5:

```
[DescriptionOf: Message
 AmbiguousSlots: (
                  [PossibleSlots: (Sender Recipient CC)
                   SlotFiller: [DescriptionOf: Person
                                FirstName: fred
                                Surname: smith
                               ]
                  ]
                 )
 AfterDate: [DescriptionOf: Date
             Month: january
             DayOfMonth: 5
            ]
]
```

This example parser output is similar to the two given previously, but instead of having a Sender slot, it has an AmbiguousSlots slot. The filler of this slot is a list of records, each of which specifies a SlotFiller and a list of PossibleSlots. The SlotFiller is a structure that would normally be the filler of a

---

[1] Nor is this problem merely an artifact of case notation, it would arise in exactly the same way for a standard syntactic parse of a sentence such as the well known "I saw the Grand Canyon flying to New York." The difficulty arises because the ambiguity is structural, and structural ambiguities can occur no matter what form of structure is chosen.

slot in the top-level description (of a message in this case), but the parser has been unable to determine exactly which higher-level slot it should fit into; the possibilities are given in PossibleSlots. With this representation, it is now straightforward to construct a directed question such as:

Do you mean the messages **from**, **to**, or **copied to** Fred Smith that arrived after January 5?

Such questions can be generated by outputting and highlighting AmbiguousSlot records as the disjunction of the normal case markers for each of the PossibleSlots followed by the normal translation of the SlotFiller. The main point here, however, does not concern the question generation mechanism, nor the exact details of the formalism for representing ambiguity, it is, rather, that a radical revision of the initial formalism was necessary in order to represent structural ambiguities without duplication of non-ambiguous material.

The adoption of such representations for ambiguity has profound implications for the parsing strategies employed by any parser that tries to produce them. For each type of construction that such a parser can encounter, and here we mean construction types at the level of case constructions, conjoined lists or linear fixed-order patterns, the parser must "know" about all the structural ambiguities that the construction can give rise to, and must be prepared to detect and encode appropriately such ambiguities when they arise. Construction-specific parsing techniques fit this requirement perfectly. Each construction-specific parsing strategy can encode detailed information about the types of structural ambiguity possible with that construction and incorporate the specific information necessary to detect and represent these ambiguities.

We must reiterate, however, that when possible we advocate the use of domain semantics to resolve selectional and structural ambiguities. Only when semantic disambiguation proves unequal to the task, a not uncommon situation with ill-formed input, do we suggest interaction with the user; but when this interaction occurs it must be as tightly focused as possible on the source of the ambiguity so that both the cognitive and the mechanical (typing) demands on the user are kept to a minimum.

4. Language Definition

Yet another advantage of a construction-specific approach to parsing is related to the definition of limited-domain task-oriented languages. As we will show in this section, the uniform grammar representation required by a uniform parsing strategy makes it difficult to define task-oriented languages in terms natural for the task domain. An alternative is to define the language in a formalism well suited to the domain and then compile this formalism into rules in the uniform grammar representation that the parser actually requires. However, this considerably inhibits the process of language development because it requires an often time-consuming phase of compilation into the uniform grammar after each change to the language definition. On the other hand, as we shall see, a construction-specific approach allows a grammar representation well-suited to the domain to be interpreted directly without any compilation phase.

Since our initial flexible parser, FlexP, applied its uniform parsing strategy to a uniform grammar of pattern-matching rewrite rules, it was not possible to cover constructions like the one used in the examples above in a single grammar rule. A postnominal case frame such as the one that covers the message descriptions used as examples above must be spread over several rewrite rules. Recall that the patterns actually used in FlexP look like:

```
<?determiner *MessageAdj MessageHead *MessageCase>
<%from Person>
<%since Date>
<%to Person>
  ...
```

The point here is not the details of the pattern notation, but the fact that this is a very unnatural way of representing a postnominal case construction. Not only does it cause problems in the process of flexible parsing, as explained in 2, but it is also quite inconvenient to create in the first place. Essentially, one has to know the specific trick of creating intermediate (and from the language point of view, superfluous) categories like **MessageCase** in the example above. Since we designed FlexP as a tool for use in natural language interfaces, we considered it unreasonable to expect the designer of a task-oriented language to have the specialized knowledge to create such obscure rules. Accordingly, we designed a language definition formalism that enabled a grammar to be specified in terms much more natural to the system being interfaced to. The above construction for the description of a message, for instance, could be defined as a single unified construction without specifying any artificial intermediate constituents, as follows:

```
[
StructureType: Object
ObjectName: Message
Schema: [
        Sender: [FillerType: &Person]
        Recipient: [FillerType: &Person Number: OneOrMore]
        Date: [FillerType: &Date]
        After: [FillerType: &Date UseRestriction: DescriptionOnly]
        ]
Syntax: [
        SynType: NounPhrase
        Head: (message note <?piece ?of mail> letter)
        Case: (
              <%from ↑Sender>
              <%to ↑Recipient>
              <%dated ↑Date>
              <%since ↑After>
              )
        ]
]
```

In addition to the syntax of a message description, this piece of formalism also describes the internal structure of a message, and is intended for use with a larger interface system [1] of which FlexP is a part. The larger system provides an interface to a functional subsystem or tool, and is tool-independent in the sense that it is driven by a declarative data base in which the objects and

operations of the tool currently being interfaced to are defined in the formalism shown. The example is, in fact, an abbreviated version of the definition of a message from the declarative tool description for an electronic mail system tool with which the interface was actually used.

In the example, the Syntax slot defines the input syntax for a message; it is used to generate rules for FlexP, which are in turn used to parse input descriptions of messages from a user. FlexP's grammar to parse input for the mail system tool is the union of all the rules compiled in this way from the Syntax fields of all the objects and operations in the tool description. The Syntax field of the example says that the syntax for a message is that of a noun phrase, i.e. any of the given head nouns (angle brackets indicate patterns of words), followed by any of the given postnominal Cases, preceded by any adjectives - none are given here, which can in turn be preceded by a determiner. The up-arrows in the Case patterns refer back to slots of a message, as specified in the Schema slot of the example - the information in the Schema slot is also used by other parts of the interface. The actual grammar rules needed by FlexP are generated by first filling in a pre-stored skeleton pattern for NounPhrase, resulting in:

```
<?determiner *MessageAdj MessageHead *MessageCase>
```

and then generating patterns for each of the Cases, substituting the appropriate FillerTypes for the slot names that appear in the patterns used to define the Cases, thus generating the subpatterns:

```
<%from Person>
<%to Person>
<%dated Date>
<%since Date>
```

The slot names are not discarded but used in the results of the subrules to ensure that the objects which match the substituted FillerTypes end up in the correct slot of the result produced by the top-level message rule. This compilation procedure must be performed in its entirety before any input parsing can be undertaken.

While this approach to language definition was successful in freeing the language designer from having to know details of the parser essentially irrelevant to him, it also made the process of language development very much slower. Every time the designer wished to make the smallest change to the grammar, it was necessary to go through the time-consuming compilation procedure. Since the development of a task-specific language typically involves many small changes, this has proved a significant impediment to the utility of FlexP.

The construction-specific approach offers a method of circumventing this problem. Since the parsing strategies and ambiguity representations are specific to particular constructions, it is possible to represent each type of construction differently - there is no need to translate the language into a uniformly represented grammar. In addition, the constructions are exactly those for which there will be specific parsing strategies, and grammar representations. It therefore becomes possible to dispense with the compilation step required for FlexP, and instead interpret the language definition directly. This drastically cuts the time needed to make changes to the grammar, and so makes the parsing system much more useful. For example, the Syntax slot of the previous example formalism

might become:

```
Syntax: [
        SynType: NounPhrase
        Head: (message note <?piece ?of mail> letter)
        Cases: (
               [Marker: %from Slot: Sender]
               [Marker: %to Slot: Recipient]
               [Marker: %dated Slot: Date]
               [Marker: %since Slot: After]
               )
        ]
```

This grammar representation, equally convenient from a user's point of view, should be directly interpretable by a parser specific to the NounPhrase case type of construction. All the information needed by such a parser, including a list of all the case markers, and the type of object that fills each case slot, is directly accessible from this representation eliminating the need for an intermediate, cumbersome compilation phase.

5. A Simple Multi-Strategy Parser

As part of our investigations in task-oriented parsing, we have implemented (in addition to FlexP) a pure case-frame parser exploiting domain-specific case constraints stored in a declarative data structure, and a combination pattern-match, semantic grammar, canonical-transform parser. All three parsers have exhibited a measure of success, but more interestingly, the strengths of one method appear to overlap with the weaknesses of a different method. Hence, we are working towards a single parser that dynamically selects its parsing strategy to suit the task demands.

Our new parser is designed primarily for task domains where the prevalent forms of user input are commands and queries, both expressed in imperative or pseudo-imperative constructs. Since in imperative constructs the initial word (or phrase), establishes the case-frame for the entire utterance, we chose the case-frame parsing strategy as primary. In order to recognize an imperative command, and to instantiate each case, other parsing strategies are invoked. Since the parser knows what can fill a particular case, it can choose the parsing strategy best suited for linguistic constructions expressing that type of information. Moreover, it can pass any global constraints from the case frame or from other instantiated cases to the subsidiary parsers - thus reducing potential ambiguity, speeding the parse, and enhancing robustness.

Consider our multi-strategy parsing algorithm as described below. Input is assumed to be in the imperative form:

1. *Apply string PATTERN-MATCH procedure to the initial segment of the input using only the patterns previously indexed as corresponding to command words/phrases in imperative constructions.* Patterns contain both optional constituents and non-terminal symbols that expand according to a semantic grammar. (E.g., "copy" and "do a file transfer" are

synonyms for the same command in a file management system.)

2. *Access the CASE-FRAME associated with the command just recognized, and push it onto the context stack.* In the above example, the case-frame is indexed under the token <COPY>, which was output by the pattern matcher. The case frame consists of list of pairs ([case-marker] [case-filler-information], ...).

3. *Match the input with the case markers using the same PATTERN-MATCH procedure mentioned above.* If no match occurs, assume the input corresponds to the unmarked case (or the first unmarked case, if more than one is present), and proceed to the next step.

4. *Apply the parsing strategy indicated by the type of construct expected as a case filler. Pass any available case constraints to the sub-parser.* A partial list of parsing strategies indicated by expected fillers is:

    - **Sub-imperative** -- Case-frame parser, starting with the command-identification pattern match above.

    - **Structured-object** (e.g., a concept with subattributes) -- Case-frame parser, starting with the pattern-matcher invoked on the list of patterns corresponding to the names (or compound names) of the semantically permissible structured objects, followed by case-frame parsing of any present subattributes.

    - **Simple Object** -- Apply the pattern matcher, using only the patterns indexed as relevant in the case-filler-information field.

    - **Special Object** -- Apply the parsing strategy applicable to that type of special object (e.g., proper names, dates, quoted strings, stylized technical jargon, etc...)

    - **None of the above** -- (Errorful input or parser deficiency) Apply the graceful recovery techniques discussed below.

5. *If an embedded case frame is activated, push it onto the context stack.*

6. *When a case filler is instantiated, remove the <case-marker>, <case-filler-information> pair from the list of active cases in the appropriate case frame, proceed to the next case-marker, and repeat the process above until the input terminates.*

7. *If all the cases in a case frame have been instantiated, pop the context stack until that case frame is no longer in it.* (Completed frames typically reside at the top of the stack.)

8. *If there is more than one case frame on the stack when trying to parse additional input, apply the following procedure:*

    - If the input only matches a case marker in one frame, proceed to instantiate the corresponding case-filler as outlined above. Also, if the matched case marker is not on the most embedded case frame (i.e., not at the top of the context stack), pop the stack until the frame whose case marker was matched appears at the top of the stack.

- If no case markers are matched, attempt to parse unmarked cases, starting with the most deeply embedded case frame (the top of the context stack) and proceeding outwards (i.e., downwards through the context stack). If an unmarked case is matched, pop the context stack until the corresponding case frame is at the top. Then, instantiate the case filler, remove the case from the active case frame, and proceed to parse additional input. If more than one unmarked case matches the input, choose the most embedded one (i.e., the most recent context) and save the state of the parse on the global history stack. (This suggests an ambiguity that cannot be resolved with the information at hand.)

- If the input matches more than one case marker in the context stack, try to parse the case filler via the indexed parsing strategy for each filler-information slot corresponding to a matched case marker. If more than one case filler parses (this is somewhat rare situation - indicating underconstrained case frames or truly ambiguous input) save the state in the global history stack and pursue the parse assuming the most deeply embedded constituent. [Our case-frame attachment heuristic favors the most local attachment permitted by semantic case constraints.]

9. *If a conjunction or disjunction occurs in the input, cycle through the context stack trying to parse the right-hand side of the conjunction as filling the same case as the left hand side. If no such parse is feasible, interpret the conjunction as top-level, e.g, as two instances of the same imperative, or two different imperatives. If more than one parse results, interact with the user to disambiguate.* To illustrate this simple process, consider:

   "Transfer the programs written by Smith and Jones to ..."

   "Transfer the programs written in Fortran and the census data files to ..."

   "Transfer the programs written in Fortran and delete ..."

   The scope of the first conjunction is the "author" subattribute of program, whereas the scope of the second conjunction is the unmarked "object" case of the transfer action. Domain knowledge in the case-filler information of the "object" case in the "transfer" imperative inhibits "Jones" from matching a potential object for electronic file transfer. Similarly "Census data files" are inhibited from matching the "author" subattribute of a program. Thus conjunctions in the two syntactically comparable examples are scoped differently by our semantic-scoping rule relying on domain-specific case information. "Delete" matches no active case filler, and hence it is parsed as the initial segment of a second conjoined utterance. Since "delete" is a known imperative, this parse succeeds.

10. *If the parser fails to parse additional input, pop the global history stack and pursue an alternate parse. If the stack is empty, invoke the graceful recovery heuristics.* Here the DELTA-MIN method [3] can be applied to improve upon depth-first unwinding of the stack in the backtracking process.

11. *If the end of the input is reached, and the global history stack is not empty, pursue the alternate parses. If any survive to the end of the input (this should not be the case unless true ambiguity exists), interact with the user to select the appropriate parse (see [5).]*

The need for embedded case structures and ambiguity resolution based on domain-dependent semantic expectations of the case fillers is illustrated by the following pair of sentences:

"Edit the programs in Fortran"
"Edit the programs in Teco"

"Fortran" fills the *language* attribute of "program", but cannot fill either the *location* or *instrument* case of Edit (both of which can be signaled by "in"). In the second sentence, however, "Teco" fills the *instrument* case of the verb "edit" and none of the attributes of "program". This disambiguation is significant because in the first example the user specified ***which*** programs (s)he wants to edit, whereas in the second example (s)he specified ***how*** (s)he wants to edit them.

The algorithm presented is sufficient to parse grammatical input. In addition, since it operates in a manner specifically tailored to case constructions, it is easy to add modifications dealing with deviant input. Currently, the algorithm includes the following steps that deal with ungrammaticality:

12. *If step 4 fails, i.e. a filler of appropriate type cannot be parsed at that position in the input, then repeat step 3 at successive points in the input until it produces a match, and continue the regular algorithm from there. Save all words not matched on a SKIPPED list.* This step takes advantage of the fact that case markers are often much easier to recognize than case fillers to realign the parser if it gets out of step with the input (because of unexpected interjections, or other spurious or missing words).

13. *If words are on SKIPPED at the end of the parse, and cases remain unfilled in the case frames that were on the context stack at the time the words were skipped, then try to parse each of the case fillers against successive positions of the skipped sequences.* This step picks up cases for which the marker was incorrect or garbled.

14. *If words are still on SKIPPED attempt the same matches, but relax the pattern matching procedures involved.*

15. *If this still does not account for all the input, interact with the user by asking questions focused on the uninterpreted part of the input.* The focused interaction technique discussed earlier is used to resolve semantic ambiguities in the input.

16. *If user interaction proves impractical, apply the project-and-integrate method [2] to narrow down the meanings of unknown words by exploiting syntactic, semantic and contextual cues.*

These flexible parsing steps rely on the construction-specific aspects of the basic algorithm, and would not be easy to emulate in either a syntactic ATN parser or one based on a pure semantic grammar.

A further advantage of our mixed-strategy approach is that the top-level case structure, in essence, partitions the semantic world dynamically into categories according to the semantic constraints on the active case fillers. Thus, when a pattern matcher is invoked to parse the recipient case of a file-transfer case frame, it need only consider patterns (and semantic-grammar constructs) that correspond to logical locations inside a computer. This form of expectation-driven parsing in restricted domains adds a two-fold effect to its robustness:

- Many spurious parses are never generated (because patterns yielding potentially

spurious matches are never tried in inappropriate contexts.)

- Additional knowledge (such as additional patterns, grammar rules, etc.) can be added without a corresponding linear increase in parse time since the case-frames focus only upon the relevant subset of patterns and rules. Thus, the efficiency of the system may actually increase with the addition of more domain knowledge (in effect enabling the case frames to restrict context further). This behavior makes it possible to incrementally build the parser without the ever-present fear that a new extension may cause the entire parser to fail due to an unexpected application of that extension in the wrong context.

In closing, we note that the algorithm described above does not mention interaction with morphological decomposition or spelling correction. Lexical processing is particularly important for robust parsing; indeed, based on our limited experience, lexical-level errors are a significant source of deviant input. The recognition and handling of lexical-deviation phenomena, such as unrecognized abbreviations and misspellings, must be integrated with the more usual morphological analysis. Some of these topics are discussed independently in [6]. However, integrating resilient morphological analysis with the algorithm we have outlined is a problem we consider very important and urgent if we are to construct a practical flexible parser.

6. Conclusion

This paper has proposed an approach to parsing limited domain task-oriented languages based on dynamic selection among a number of different parsing strategies, one for each type of construction covered by the language. This approach was shown to have a number of advantages over more traditional uniform parsing strategies, including:

- Grammatical input can be parsed efficiently by the strategy most appropriate to each type of expected construction, bringing exactly the appropriate domain knowledge to bear.

- Ungrammatical input can be analyzed by first isolating the troublesome portions of the input, and subsequently applying our "fail-soft" recovery heuristics (exploiting constraints from other, better understood, constituents in the input), or engaging in focused interaction with the system user.

- Semantic and structural ambiguities can be represented straightforwardly without duplication of unambiguous material, facilitating the construction of focused queries to the user concerning the ambiguity. Moreover, focused interaction facilitates interpretation of elliptical user replies to clarification queries.

- The task-specific language development process can be simplified and made more efficient since the interface designer can define the language in terms natural to the task domain. In addition, the construction-specific approach enables the resulting language definition to be interpreted directly without a time-consuming compilation into a uniform grammar formalism, thus allowing interactive incremental testing when new constructs are added to the task-oriented data base.

References

1. Ball, J. E. and Hayes, P. J., "Representation of Task-Independent Knowledge in a Gracefully Interacting User Interface," *Proc. 1st Annual Meeting of the American Association for Artificial Intelligence,* Stanford University, August 1980 , pp. 116-120.

2. Carbonell, J. G., "Towards a Self-Extending Parser," *Proceedings of the 17th Meeting of the Association for Computational Linguistics,* 1979 , pp. 3-7.

3. Carbonell, J. G., "Δ-MIN: A Search-Control Method for Information-Gathering Problems," *Proceedings of the First AAAI Conference,* August 1980 .

4. Gershman, A. V., *Knowledge-Based Parsing,* PhD dissertation, Yale University, April 1979, Computer Sci. Dept. report # 156.

5. Hayes P. J., "Focused Interaction in Flexible Parsing," *Proc. of 19th Annual Meeting of the Assoc. for Comput. Ling.,* Stanford University, June 1981 .

6. Hayes, P. J. and Mouradian, G. V., "Flexible Parsing," *Proc. of 18th Annual Meeting of the Assoc. for Comput. Ling.,* Philadelphia, June 1980 , pp. 97-103.

7. Hendrix, G. G., Sacerdoti, E. D. and Slocum, J., "Developing a Natural Language Interface to Complex Data," Tech. report Artificial Intelligence Center., SRI International, 1976.

8. Kaplan, S. J., *Cooperative Responses from a Portable Natural Language Data Base Query System,* PhD dissertation, Dept. of Computer and Information Science, University of Pennsylvania, Philadelphia, 1979.

9. Kwasny, S. C. and Sondheimer, N. K., "Ungrammaticality and Extra-Grammaticality in Natural Language Understanding Systems," *Proc. of 17th Annual Meeting of the Assoc. for Comput. Ling.,* La Jolla, Ca., August 1979 , pp. 19-23.

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

11. Parkison, R. C., Colby, K. M., and Faught, W. S., "Conversational Language Comprehension Using Integrated Pattern-Matching and Parsing," *Artificial Intelligence,* Vol. 9, 1977 , pp. 111-134.

12. Riesbeck, C. and Schank, R. C., "Comprehension by Computer: Expectation-Based Analysis of Sentences in Context," Tech. report 78, Computer Science Department, Yale University, 1976.

Waltz, D. L. and Goodman, A. B., "Writing a Natural Language Data Base System," *Proceedings of the Fifth International Joint Conference on Artificial Intelligence,* 1977 , pp. 144-150.

'eischedel, R. M. and Black, J., "Responding to Potentially Unparseable Sentences," Tech. ort 79/3, Dept. of Computer and Information Sciences, University of Delaware, 1979.