A Construction-Specific Approach to Focused Interaction in Flexible Parsing

Philip J. Hayes
Carnegie-Mellon University
Pittsburgh, PA 15213

Abstract[1]

A flexible parser can deal with input that deviates from its grammar, in addition to input that conforms to it. Ideally, such a parser will correct the deviant input; sometimes, it will be unable to correct it at all; at other times, correction will be possible, but only to within a range of ambiguous possibilities. This paper is concerned with such ambiguous situations, and with making it as easy as possible for the ambiguity to be resolved through consultation with the user of the parser - we presume interactive use. We show the importance of asking the user for clarification in as focused a way as possible. Focused interaction of this kind is facilitated by a construction-specific approach to flexible parsing, with specialized parsing techniques for each type of construction, and specialized ambiguity representations for each type of ambiguity that a particular construction can give rise to. A construction-specific approach also aids in task-specific language development by allowing a language definition that is natural in terms of the task domain to be interpreted directly without compilation into a uniform grammar formalism, thus greatly speeding the testing of changes to the language definition.

1. Introduction

There has been considerable interest recently in the topic of flexible parsing, i.e. the parsing of input that deviates to a greater or lesser extent from the grammar expected by the parsing system. This interest springs from very practical concerns with the increasing use of natural language in computer interfaces. When people attempt to use such interfaces, they cannot be expected always to conform strictly to the interface's grammar, no matter how loose and accomodating that grammar may be. Whenever people spontaneously use a language, whether natural or artificial, it is inevitable that they will make errors of performance. Accordingly, we [3] and other researchers including Weischedel and Black [6], and Kwasny and Sondheimer [5], have constructed flexible parsers which accept ungrammatical input, correcting the errors whenever possible, generating several alternative interpretations if more than one correction is plausible, and in cases where the input cannot be massaged into full grammaticality, producing as complete a partial parse as possible.

If a flexible parser being used as part of an interactive system cannot correct ungrammatical input with total 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 [6] 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 [4], 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 suggests that the way requests for clarification in such situations are phrased makes a big difference to the ease and accuracy with which the user can correct his errors, and that the user is most helped by a request which focuses as tightly as possible on the exact source and nature of the difficulty. Accordingly, we have adopted the following simple principle for the new flexible parser we are presently constructing: ***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.*** Furthermore, 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. Or again, 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 what follows, we examine some of the implications of these ideas. We restrict our attention to cases in which a flexible parser can correct an input error or ungrammaticality, but only to within a constrained set of alternatives. We consider how to produce a focused ambiguity resolution request for the user to distinguish between such a set of corrections. We conclude that:

- the problem must be tackled on a construction-specific basis,
- and special representations must be devised for all the structural ambiguities that each construction type can give rise to.

We illustrate these arguments with examples involving case constructions. There are additional independent reasons for adopting a construction-specific approach to flexible parsing, including increased efficiency and accuracy in correcting ungrammaticality, increased efficiency in parsing grammatical input, and ease of task-specific language definition. The first two of these are discussed in [2], and this paper gives details of the third.

2. Construction-Specific Ambiguity Representations

In this section we report on experience with our earlier flexible parser, FlexP [3], and show why it is ill-suited to the generation of focused requests to its user for the resolution of input ambiguities. We propose solutions to the problems with FlexP. We have already incorporated these improvements into an initial version of a new flexible parser [2].

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

[1] This research was sponsored 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 author and should not be interpreted as representing the official policies, either expressed or implied, of the Air Force Office of Scientific Research or the US Government.

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. FlexP operates by bottom-up pattern matching of a semantic grammar of rewrite rules which allows it to 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 procede 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 connexion between Fred Smith and the message could have been expressed (to within synonyms) only by "from", "to", or "copied to". FlexP can deal with this input, and correct it to within this three way ambiguity. To represent the ambiguity, it 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 distinguish between 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.[2] To localize such ambiguities and avoid duplicate representation of unambiguous parts of the input, it is necessary to employ a representation like the one used by our new flexible parser:

```
[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 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 AmbiguousSlot records as the disjunction (in boldface) 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 which 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 construction, conjoined list, linear fixed-order pattern, 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. We have chosen to achieve this by designing a number of different parsing strategies, one for each type of construction that will be encountered, and making the parser switch

[2]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.

between these strategies dynamically. Each such construction-specific parsing strategy encodes detailed information about the types of structural ambiguity possible with that construction and incorporates the specific information necessary to detect and represent these ambiguities.

3. Other Reasons for a Construction-Specific Approach

There are additional independent reasons for adopting a construction-specific approach to flexible parsing. Our initially motivating reason was that dynamically selected construction-specific parsing strategies can make corrections to erroneous input more accurately and efficiently than a uniform parsing procedure. It also turned out that such an approach provided significant advantages in the parsing of correct input as well. These points are covered in detail in [2].

A further advantage is related to language definition. 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. The patterns actually used in FlexP look like:

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

The first top-level pattern says that a message description is an optional (?) determiner, followed by an arbitrary number (*) of message adjectives followed by a message head word (one meaning "message"), followed by an arbitrary number of message cases. Because each case has more than one component, each must be recognized by a separate pattern like the second and third above. Here % means anything in the same word class, "that arrived after", for instance, is equivalent to "since" for this purpose.

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 for a flexible parser, 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 such a system 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>)
        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 usefulness of FlexP.

The construction-specific approach offers a way round this problem. Since the parsing strategies and ambiguity representations are specific to particular constructions, it is possible to represent each different type of construction differently - there is no need to translate the language into a uniformly represented grammar. In addition, the constructions in terms of which it is natural to define a language 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>)
        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 enough accessible from this representation that an intermediate compilation phase should not be required, with all the ensuing benefits mentioned above for language development.

4. Conclusion

There will be many occasions, even for a flexible parser, when complete, unambiguous parsing of the input to an interactive system is impossible. In such circumstances, the parser should interact with the system user to resolve the problem. Moreover, to make things as easy as possible for the user, the system should phrase its request for clarification in terms that focus as tightly as possible on the real source and nature of the difficulty. In the case of ambiguity resolution, this means that the parser must produce a representation of the ambiguity that does not duplicate unambiguous material. This implies specific ambiguity representations for each type of construction recognized by the parser, and corresponding specific parsing strategies to generate such representations. There are other advantages to a construction-specific approach including more accurate and efficient correction of ungrammaticality, more efficient parsing of grammatical input, and easier task-specific language development. This final benefit arises because a construction-specific approach allows a language definition that is natural in terms of the task domain to be interpreted directly without compilation into a uniform grammar formalism, thus greatly speeding the testing of changes to the language definition.

Acknowledgement

Jaime Carbonell provided valuable comments on earlier drafts of this paper.

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, American Assoc. for Artificial Intelligence, Stanford University, August, 1980, pp. 116-120.

2. Carbonell, J. G. and Hayes, P. J. Dynamic Strategy Selection in Flexible Parsing. Carnegie-Mellon University Computer Science Department, 1981.

3. 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.

4. Kaplan, S. J. *Cooperative Responses from a Portable Natural Language Data Base Query System*. Ph.D. Th., Dept. of Computer and Information Science, University of Pennsylvania, Philadelphia, 1979.

5. 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.

6. Weischedel, R. M. and Black, J. Responding to Potentially Unparseable Sentences. Tech. Rept. 79/3, Dept. of Computer and Information Sciences, University of Delaware, 1979.