Flexible Parsing[1]

Philip J. Hayes
George V. Mouradian

Computer Science Department
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

When people use natural language in natural settings, they often use it ungrammatically, leaving out or repeating words, breaking off and restarting, speaking in fragments, etc. Their human listeners are usually able to cope with these deviations with little difficulty. If a computer system is to accept natural language input from its users on a routine basis, it should be similarly robust. In this paper, we outline a set of parsing flexibilities that such a system should provide. We go on to describe FlexP, a bottom-up pattern matching parser that we have designed and implemented to provide many of these flexibilities for restricted natural language input to a limited-domain computer system.

1. The Importance of Flexible Parsing

When people use natural language in natural conversation, they often do not respect grammatical niceties. Instead of speaking sequences of grammatically well-formed and complete sentences, people often leave out or repeat words or phrases, break off what they are saying and rephrase or replace it, speak in fragments, or use otherwise incorrect grammar. The following example conversation involves a number of these grammatical deviations:

A: I want ... can you send a memo a message to to Smith

B: Is that John or John Smith or Jim Smith

A: Jim

Instead of being unable or refusing to parse such ungrammatical utterances, human listeners are generally unperturbed by them. Neither participant in the above dialogue, for instance, would have any difficulty.

When computers attempt to interact with people using natural language, they face a very similar situation; the people will still tend to deviate from whatever grammar the computer system is using. The fact that the input is typed rather than spoken makes little difference; grammatical deviations seem to be inherent in spontaneous human use of language whatever the modality. So, if computers are ever to converse naturally with humans, they must be able to parse their inputs as flexibly and robustly as humans do. While considerable advances have been made in recent years in applied natural language processing, few of the systems that have been constructed have paid sufficient attention to the kinds of deviation that will inevitably occur in their input if they are used in a natural environment. In many cases, if the user's input does not conform to the system's grammar, an indication of incomprehension followed by a request to rephrase may be the best he can expect. We believe that such inflexibility in parsing severely limits the practicality of natural language computer interfaces, and is a major reason why natural language has yet to find wide acceptance in such applications as database retrieval or interactive command languages.

In this paper, we report on a flexible parser, called FlexP, suitable for use with a restricted natural language interface to a limited-domain computer system. We describe first the kinds of grammatical deviations we are trying to deal with, then the basic design characteristics of FlexP with justification for them based on the kinds of problem to be solved, and finally more details of our parsing system with worked examples of its operation. These examples, and most of the others in the paper, represent natural language input to an electronic mail system that we and others [2] are constructing as part of our research on user interfaces. This system employs FlexP to parse its input.

[1] This research was sponsored by the Air Force Office of Scientific Research under Contract F49620-79-C-0143.

0362-613X/81/040232-11$01.00

2. Types of Grammatical Deviation

There are a number of distinct types of grammatical deviation, and not all types are found in all types of communication situation. In this section, we first define the restricted type of communication situation that we will be concerned with, that of a limited-domain computer system and its user communicating via a keyboard and display screen. We then present a taxonomy of grammatical deviations common in this context, and by implication a set of parsing flexibilities needed to deal with them.

2.1 Communication with a Limited-Domain System

In the remainder of this paper, we will focus on a restricted type of communication situation, that between a limited-domain system and its user, and on the parsing flexibilities needed by such a system to cope with the user's inevitable grammatical deviations. Examples of the type of system we have in mind are database retrieval systems, electronic mail systems, medical diagnosis systems, or any systems operating in a domain so restricted that they can completely understand any relevant input a user might provide. There are several points to be made.

First, although such systems can be expected to parse and understand anything relevant to their domain, their users cannot be expected to confine themselves to relevant input. As Bobrow et. al. [3] note, users often explain their underlying motivations or otherwise justify their requests in terms quite irrelevant to the domain of the system. The result is that such systems cannot expect to parse all their inputs even with the use of flexible parsing techniques.

Secondly, a flexible parser is just part of the conversational component of such a system, and cannot solve all parsing problems by itself. For example, if a parser can extract two coherent fragments from an otherwise incomprehensible input, the decisions about what the system should do next must be made by another component of the system. A decision on whether to jump to a conclusion about what the user intended, to present him with a set of alternative interpretations, or to profess total confusion, can only be made with information about the history of the conversation, beliefs about the user's goals, and measures of plausibility for any given action by the user. (See [10] for more discussion of this broader view of graceful interaction in man-machine communication.) Suffice it to say that we assume a flexible parser is just one component of a larger system, and that any incomprehensions or ambiguities that it finds are passed on to another component of the system with access to higher-level information, putting it in a better position to decide what to do next.

Finally, we assume that, as is usual for such systems, input is typed, rather than spoken as is normal in human conversations. This simplifies low-level processing tremendously because key-strokes, unlike speech wave-forms, are unambiguous. On the other hand, problems like misspelling arise, and a flexible parser cannot even assume that segmentation into words by spaces and carriage returns will always be correct. However, such input is still one side of a conversation, rather than a polished text in the manner of most written material. As such, it is likely to contain many of the same type of errors normally found in spoken conversations.

2.2 Misspelling

Misspelling is perhaps the most common class of error in written language. Accordingly, it is the form of ungrammaticality that has been dealt with the most by language processing systems. PARRY [14], LIFER [11], and numerous other systems have tried to correct misspelt input from their users.

An ability to correct spelling implies the existence of a dictionary of correctly spelled words (possibly augmented by a set of morphological rules to produce derived forms). An input word not found in or derivable from the dictionary is assumed to be misspelt and is compared against each of the dictionary words and their derivations. If one of these words comes close enough to the input word according to some criteria of lexical matching, it is used in place of the input word.

Spelling correction may be attempted in or out of context. For instance, there is only one reasonable correction for "relavent" or for "seperate", but for an input like "un" some kind of context is typically necessary as in "I'll see you un April" or "he was shot with the stolen un." In effect, context can be used to reduce the size of the dictionary to be searched for correct words. This both makes the search more efficient and reduces the possibility of multiple matches of the input against the dictionary. The LIFER [11] system uses the strong constraints typically provided by its semantic grammar in this way to reduce the range of possibilities for spelling correction.

A particularly troublesome kind of spelling error results in a valid word different from the one intended, as in "show me on of the messages". Clearly, such an error can only be corrected through comparison against a contextually restricted subset of a system's vocabulary.

2.3 Novel Words

Even accomplished users of a language will sometimes encounter words they do not know. Such situations are a test of their language learning skills. If one did not know the word "fawn", one could at least decide it was a colour from "a fawn coloured sweater". If one just knew the word as referring to a young deer, one might conclude that it was being used to mean the colour of a young deer. In general, beyond making direct inferences about the role of unknown words from their immediate context, vocabulary learning can require arbitrary amounts of real-world knowledge and inference, and this is certainly beyond the capabilities of present day artificial intelligence techniques (though see Carbonell [5] for work in this direction).

There is, however, a very common special subclass of novel words that is well within the capabilities of present day systems: unknown proper names. Given an appropriate context, either sentential or discourse, it is relatively straightforward to classify unknown words as the names of people, places, etc.. Thus in "send copies to Moledeski Chiselov" it is reasonable to conclude from the local context that "Moledeski" is a first name, "Chiselov" is a surname, and together they identify a person (the intended recipient of the copies). Strategies like this were used in the POLITICS [6], FRUMP [8], and PARRY [14] systems.

Since novel words are by definition not in the known vocabulary, how can a parsing system distinguish them from misspellings? In most cases, the novel words will not be close enough to known words to allow successful correction, as in the above example, but this is not always true; an unknown first name of "Al" could easily be corrected to "all". Conversely, it is not safe to assume that unknown words in contexts which allow proper names are really proper names as in: "send copies to al managers". In this example, "al" probably should be corrected to "all". In order to resolve such cases it may be necessary to check against a list of referents for proper names, if this is known, or otherwise to consider such factors as whether the initial letters of the words are capitalized.

As far as we know, no systems yet constructed have integrated their handling of misspelt words and unknown proper names to the degree outlined above. However, several applied natural language processing systems, including the COOP [12] system, allow systematic access to a database containing proper names without the need for inclusion of the words in the system's parsing vocabulary.

2.4 Erroneous segmenting markers

Written text is segmented into words by spaces and new lines, and into higher level units by commas, periods and other punctuation marks. Both classes, especially the second, may be omitted or inserted speciously. Spoken language is also segmented, but by the quite different markers of stress, intonation and noise words and phrases, which we will not consider here.

Incorrect segmentation at the lexical level results in two or more words being run together, as in "runtogether", or a single word being split up into two or more segments, as in "tog ether" or (inconveniently) "to get her", or combinations of these effects as in "runto geth er". In all cases, it seems natural to deal with such errors by extending the spelling correction mechanism to be able to recognize target words as initial segments of unknown words, and vice-versa. As far as we know, no current systems deal with incorrect segmentation into words.

The other type of segmenting error, incorrect punctuation, has a much broader impact on parsing methodology. Current parsers typically work one sentence at a time, and assume that each sentence is terminated by an explicit end-of-sentence marker. A flexible parser must be able to deal with the potential absence of such a marker, and recognize that the sentence is being terminated implicitly by the start of the next sentence. In general, a flexible parser should be able to take advantage of the information provided by punctuation if it is used correctly, and ignore it if it is used incorrectly.

Instead of punctuation, many interactive systems use carriage-return to indicate sentence termination. Missing sentence terminators in this case correspond to two sentences on one line, or to the typing of a sentence without the terminating return, while specious terminators correspond to typing a sentence on more than one line.

2.5 Broken-Off and Restarted Utterances

In spoken language, it is very common to break off and restart all or part of an utterance:

I want to — Could you tell me the name?

Was the man –er– the official here yesterday?

Usually, such restarts are signalled in some way, by "um" or "er", or more explicitly by "let's back up" or some similar phrase.

In written language, such restarts do not normally occur because they are erased by the writer before the reader sees them. Interactive computer systems typically provide facilities for their users to delete the last character, word, or current line as though it had never been typed, for the very purpose of allowing such restarts. Given these signals, the restarts are easy to detect and interpret. However, users sometimes fail to make use of these signals. Input not containing a carriage-return can be spread over several lines by intermixing of input and output, or a user may simply fail to type the "kill" character that deletes the input line he has typed so far, as in:

delete the show me all the messages from Smith where the user probably intended to erase the first two words and start over. A flexible parser should be able to deal with such non-signalled restarts.

2.6 Fragmentary and Otherwise Elliptical Input

Naturally occurring language often involves utterances that are not complete sentences. Often the appropriateness of such fragmentary utterances depends on conversational or physical context as in:

A: Do you mean Jim Smith or Fred Smith?

B: Jim

A: Send a message to Smith

B: OK

A: with copies to Jones

A flexible parser must be able to parse such fragments given the appropriate context.

There is a question here of what such fragments should be parsed into. Parsing systems that have dealt with the problem have typically assumed that such inputs are ellipses of complete sentences, and that their parsing involves finding that complete sentence, and parsing it. Thus the sentence corresponding to "Jim" in the example above would be "I mean Jim". Essentially this view has been taken by the LIFER [11] and GUS [3] systems. An alternative view is that such fragments are not ellipses of more complete sentences, but are themselves complete utterances given the context in which they occur, and should be parsed as such. Certain speech parsers, including HEARSAY-II [9] and HWIM [20], are oriented towards this more bottom-up view of fragments, and this view is also the basis of our approach to fragmentary input, as we will explain more fully below. Carbonell [personal communication] suggests a third view appropriate for some fragments: that of an extended case frame. In the second example above, for instance, A's "with copies to Jones" forms a natural part of the case frame established by "send a message to Smith". Yet another approach to fragment parsing is taken in the PLANES system [15] which always parses in terms of major fragments rather than complete utterances. This technique relies on there being only one way to combine the fragments thus obtained, which may be a reasonable assumption for many limited domain systems.

Ellipses can also occur without regard to context. A type that interactive systems are particularly likely to face is crypticness in which articles and other non-essential words are omitted as in "show messages after June 17" instead of the more complete "show me all messages dated after June 17". Again, there is a question of whether to consider the cryptic input complete, which would mean modifying the system's grammar, or to consider it elliptical, and complete it by using flexible techniques to parse it against the complete version as it exists in the standard grammar.

Other common forms of ellipsis are associated with conjunction as in:

John got up and [John] brushed his teeth.

Mary saw Bill and Bill [saw] Mary.

Fred recognized [the building] and [Fred] walked towards the building.

Since conjunctions can support such a wide range of ellipsis, it is generally impractical to recognize such utterances by appropriate grammar extensions. Efforts to deal with conjunction have therefore depended on general mechanisms that supplement the basic parsing strategy, as in the LUNAR system [19], or that modify the grammar temporarily, as in the work of Kwasny and Sondheimer [13]. We have not attempted to deal with this type of ellipsis in our parsing system, and will not discuss further the type of flexibility it requires.

2.7 Interjected Phrases, Omission, and Substitution

Sometimes people interject noise or other qualifying phrases into what is otherwise a normal grammatical flow as in:

I want the message dated I think June 17

Such interjections can be inserted at almost any point in an utterance, and so must be dealt with as they arise by flexible techniques.

It is relatively straightforward for a system of limited comprehension to screen out and ignore standard noise phrases such as "I think" or "as far as I can tell". More troublesome are interjections that cannot be recognized by the system, as might for instance be the case in

Display [just to refresh my memory] the message dated June 17.

I want to see the message [as I forgot what it said] dated June 17.

where the unrecognized interjections are bracketed. A flexible parser should be able to ignore such interjections. There is always the chance that the unrecognized part was an important part of what the user was trying to say, but clearly, the problems that arise from this cannot be handled by a parser.

Omissions of words (or phrases) from the input are closely related to cryptic input as discussed above, and one way of dealing with cryptic input is to treat it as a set of omissions. However, in cryptic input only inessential information is left out, while it is conceivable that one could also omit essential information as in:

Display the message June 17

Here it is unclear whether the speaker means a message dated on June 17 or before June 17 or after June

17. (We assume that the system addressed can display things immediately, or not at all.) If an omission can be narrowed down in this way, the parser should be able to generate all the alternatives (for contextual resolution of the ambiguity or for the basis of a question to the user). If the omission can be narrowed to one alternative then the input was merely cryptic.

Besides omitting words and phrases, people sometimes substitute incorrect or unintended ones. Often such substitutions are spelling errors and should be caught by the spelling correction mechanism, but sometimes they are inadvertent substitutions or uses of equivalent vocabulary not known to the system. This type of substitution is just like an omission except that there is an unrecognized word or phrase in the place where the omitted input should have been. For instance, in "the message over June 17", "over" takes the place of "dated" or "sent after" or whatever else is appropriate at that point. If the substitution is of vocabulary that is appropriate but unknown to the system, parsing of substituted words can provide the basis of vocabulary extension. Wilks [17] has developed techniques for relaxing semantic constraints when they are apparently violated by relations implied by the input, as in the previous example, where "June 17" and "the message" do not fit the normal semantic constraints of the "over" relation.

2.8 Agreement Failure

It is not uncommon for people to fail to make the appropriate agreement between the various parts of a noun or verb phrase as in :

I wants to send a messages to Jim Smith.

The appropriate action is to ignore the lack of agreement, and Weischedel and Black [16] describe a method for relaxing the predicates in an ATN grammar which typically check for such agreements. However, it is generally not possible to conclude locally which value of the marker (number or person) for which the clash occurs is actually intended.

2.9 Idioms

Idioms are phrases whose interpretation is not what would be obtained by parsing and interpreting them constructively in the normal way. They may also not adhere to the standard syntactic rules. Idioms must thus be parsed as a whole in a pattern matching kind of mode. Parsers based purely on pattern matching, like that of PARRY [14], are able to parse idioms naturally, while others must either add a preprocessing phase of pattern matching as in the LUNAR system [19], or mix specific patterns in with more general rules, as in the work of Kwasny and Sondheimer [13]. Semantic grammars [4, 11] provide a relatively natural way of mixing idiomatic and more general patterns.

2.10 User-Supplied Changes

In normal human conversation, once something is said, it cannot be changed, except indirectly by more words that refer back to the original ones. In interactively typed input, there is always the possibility that a user may notice an error he has made and go back and correct it himself, without waiting for the system to pursue its own, possibly slow and ineffective, methods of correction. With appropriate editing facilities, the user may do this without erasing intervening words, and, if the system is processing his input on a word-by-word basis, may thus alter a word that the system has already processed. A flexible parser must be able to take advantage of such user-provided corrections to unknown words, and to prefer them over its own corrections. It must also be prepared to change its parse if the user changes a valid word to another different but equally valid word.

3. An Approach to Flexible Parsing

Most current parsing systems are unable to cope with most of the kinds of grammatical deviation outlined above. This is because typical parsing systems attempt to apply their grammars to their inputs in a rigid way, and since deviant input, by definition, does not conform to the grammar, they are unable to produce any kind of parse for it at all. Attempts to parse more flexibly have typically involved parsing strategies to be used after a top-down parse using an ATN [18] or similar transition net has failed. Such efforts include the ellipsis and paraphrase mechanisms of LIFER [11], the predicate relaxation techniques of Weischedel and Black [16], and several of the devices for extending ATN's proposed by Kwasny and Sondheimer [13]. An important exception to this observation is the case of parsers, including HEARSAY-II [9] and HWIM [20], designed for spoken input with its inherent low-level uncertainty. HWIM, in particular, incorporates techniques to apply an ATN in a bottom-up style, and thus can capitalize on whatever points of certainty it can find in the input. Fragmentary input, however, is typically the only type of ungrammaticality dealt with by speech parsers.

In the remainder of this paper, we outline an approach to parsing that was designed with ungrammatical input specifically in mind. We have embodied this approach in a working parser, called FlexP, which can apply its grammar to its input flexibly enough to deal with most of the grammatical deviations discussed in the previous section. We should emphasize, however, that FlexP is designed to be used in the interface to a restricted-domain system. As such, it is intended to work from a domain-specific semantic grammar, rather than one suitable for broader classes of input. FlexP thus does not embody a solution for flexible parsing of natural language in general. In describing FlexP, we will note those of its techniques that seem unlikely to scale up to use with more complex grammars with wider coverage.

We have adopted in FlexP an approach to flexible parsing based not on ATN's, but closer to the pattern matching techniques used in the PARRY system [14], possibly the most robust natural language processing system yet constructed. At the highest level, the design of FlexP can be characterized by the following three features:

- *pattern matching:* This provides a convenient way to recognize idioms, and also aids in the detection of omissions and substitutions in non-idiomatic phrases.
- *bottom-up rather than top-down parsing:* This aids in the parsing of fragmentary utterances, and in the recognition of interjections and restarts.
- *parse suspension and continuation:* This is important for dealing with interjections, restarts, and implicit terminations.

In the rest of this section, we examine and justify these design characteristics in more detail, and in the process, give an outline of FlexP's parsing algorithm.

3.1 Pattern Matching

We have chosen to use a grammar of linear patterns rather than a transition network because pattern matching meshes well with bottom-up parsing, because it facilitates recognition of utterances with omissions and substitutions, and because it is ideal for the recognition of idiomatic phrases.

The grammar of the parser is a set of rewrite or production rules; the left-hand side of one of these rules is a linear pattern of constituents (lexical or higher level) and the right-hand side defines a result constituent. Elements of the pattern may be labelled optional or allow for repeated matches. We make the assumption, certainly true for the grammar we are presently working with, that the grammar will be semantic rather than syntactic, with patterns corresponding to idiomatic phrases or to object and event descriptions meaningful in some limited domain, rather than to general syntactic structures.

Linear patterns fit well with bottom-up parsing because they can be indexed by any of their components, and because, once indexed, it is straightforward to confirm whether a pattern matches input already processed in a way consistent with the way the pattern was indexed.

Patterns help with the detection of omissions and substitutions because in either case the relevant pattern can still be indexed by the remaining elements that appear correctly in the input, and thus the pattern as a whole can be recognized even if some of its elements are missing or incorrect. In the case of substitutions, such a technique can actually help focus the spelling-correction, proper-name-recognition, or vocabulary-learning techniques, whichever is appropriate, by isolating the substituted input and the pattern constituent that it should have matched. The (often highly selective) restrictions on what can match the constituent can then be used to reduce the number of possibilities considered by these relatively expensive lexical correction techniques.

3.2 Bottom-Up Parsing

Our choice of a bottom-up strategy is based on our need to recognize isolated sentence fragments. If an utterance that would normally be considered only a fragment of a complete sentence is to be recognized top-down, there are two straightforward approaches to take. First, the grammar can be altered so that the fragment is recognized as a complete utterance in its own right. This is undesirable because it can cause enormous expansion of the grammar, and because it becomes difficult to decide whether a fragment appears in isolation or as part of a larger utterance, especially if there is the possibility of missing end-of-sentence markers. The second option is for the parser to infer from the conversational context what grammatical category (or sequence of sub-categories) the fragment might fit into, and then to do a top-down parse from that sub-category. This essentially is the tactic used in the GUS [3] and LIFER [11] systems. This strategy is clearly better than the first one, but has two problems: first, of predicting all possible sub-categories which might come next, and secondly, of inefficiency if a large number are predicted. Kwasny and Sondheimer [13] use a combination of the two strategies by temporarily modifying an ATN grammar to accept fragment categories as complete utterances at the times they are contextually predicted.

Bottom-up parsing avoids the problem of trying to predict what grammatical sub-category a sentence fragment should be parsed into. The data-driven nature of bottom-up parsing means that any sub-category can be parsed as an isolated unit in exactly the same way as a complete sentence, so that no prediction about what sub-category to expect is necessary. On the other hand, if a given input can be parsed as more than one sub-category, a bottom-up approach has no good way of distinguishing between them, even if only one would be predicted top-down.

In a system of limited comprehension, fragmentary recognition is sometimes necessary because not all of an input can be recognized, rather than because of intentional ellipsis. Here, it may not be possible to make predictions, and bottom-up parsing has a clear advantage. As described below, bottom-up strategies, coupled with suspended parses, are also helpful in recognizing interjections and restarts.

While well suited to parsing fragmentary input, pure bottom-up parsing (as, for instance, described by Chester [7]) can result in the generation of an unnecessarily large number of intermediate structures that cannot form part of a completed parse. To reduce these inefficiencies, FlexP does not require all elements of a pattern to be present before including the pattern in the parse structure, and is thus similar to the left-corner algorithm also described by Chester. (For greater detail see Aho and Ullman [1].) In fact, FlexP is more restrictive than the left-corner algorithm. In normal left-to-right processing, if a new word is accounted for by a pattern that has already been partially matched by previous input, other patterns indexed by the new word are not considered as possible matches for that word. This is a heuristic designed to limit the number of partial parses that need to be investigated, and could lead to missed parses, but in practical experience, we have not found this to be a problem. Exactly how this heuristic operates will be made clear by the description of the parsing algorithm in the following section.

3.3 Parse Suspension and Continuation

FlexP employs the technique of suspending a parse with the possibility of later continuation to help with the recognition of interjections, restarts, and implicit terminations. To make clear what this means, it is first necessary to sketch the operation of the parsing algorithm as a whole. This will also serve to clarify the discussion of bottom-up parsing in the previous section. Examples of the algorithm in action are given in Section 4.

FlexP's parsing algorithm maintains a set of partial parses, each of which accounts for the input already processed but not yet accounted for by a completed parse. The parser attempts to incorporate each new input word into each of the partial parses by one of the following methods:

1. fitting the word directly into one of the pattern slots available for matching at the right-hand edge of the partial parse;
2. finding a chain of non-terminal grammar sub-categories that allow the word to fit indirectly at the right-hand edge of the partial parse;
3. finding a common super-category of the input word and the sub-category at the top of the partial parse, so that the partial parse can be extended upwards and the input word will then fit into it by either method 1 or 2 above;
4. same as 1, but based on flexible pattern matching;
5. same as 2, but based on flexible pattern matching;
6. same as 3, but based on flexible pattern matching.

Flexible pattern matching is explained in Section 4. As the description of method 3 implies, FlexP does not build the partial parses any higher than is necessary to account for the input already processed. In particular, FlexP does not try to build each partial parse up to a complete utterance unless a complete utterance is needed to account for the input seen so far.

Which of the six methods is used to incorporate the new word into the existing set of partial parses is determined in the following way. The parser first tries method 1 on each of the partial parses. For each of them, it may succeed in one or more than one way or it may fail. If it succeeds on any of them, the ones on which it fails are discarded, the ones on which it succeeds are extended in all the ways that are possible, the extensions become the new set of partial parses for possible extension by the next input word, and the other five methods are not tried. If method 1 fails for all partial parses, the same procedure is repeated for method 2, and so on. If no partial parse can be extended by any of the six methods, the entire set of partial parses is saved as a suspended parse, and the input is used either to start a completely new set of partial parses, or to extend a previously suspended parse. Clearly, the policy of not attempting the more complicated methods if the simpler methods succeed can result in some parses being missed. However, in practice, we have found it heuristically adequate for the small domain-specific grammar we have been using, and much more efficient than trying all methods regardless of the outcomes of the others. On the other hand, if completeness became important, it would be simple to change FlexP always to try all methods.

There are several possible explanations for input mismatch, i.e. the failure of an input to extend the currently active set of partial parses.

- The input could be an implicit termination, i.e. the start of a new top-level utterance, in which case the previous utterance should be assumed complete.
- The input could be a restart, in which case the active parse should be abandoned and a new parse started from that point.
- The input could be the start of an interjection, in which case the active parse should be temporarily suspended, and a new parse started for the interjection.

It is not possible, in general, to distinguish between these cases at the time the mismatch occurs. If the active parse is not at a possible termination point, then input mismatch cannot indicate implicit termination, but may indicate either restart or interjection. It is necessary to suspend the active parse and wait to see if it is continued at the next input mismatch. On the other hand, if the active parse is at a possible termination point, input mismatch does not rule out interjection or even restart. In this situation, our algorithm tentatively assumes that there has been an implicit termination, but suspends the active parse anyway for subsequent potential continuation.

Finally, it may be worthwhile to note why we implemented FlexP to operate in a breadth-first mode, carrying ambiguous alternative parses along in parallel, rather than investigating them individually depth-first. This choice follows naturally from a decision to parse each input token immediately after it is typed, which in turn follows from our desire to deal with implicit termination of input strings (see Section 2.4). A breadth-first approach allows the parser to make best use of the time during which the user is typing. A depth-first implementation could postpone consideration of some alternatives until the input had been terminated by the user. In such cases, unacceptably long delays might result. Note that the possibility of implicit termination also provides justification for the strategy of parsing each input word immediately after it is typed. If the input signals an implicit termination, then the user may well expect the system to respond immediately to the input thus terminated.

4. FlexP in Operation

This section describes through a series of examples of gradually increasing complexity how FlexP's parsing algorithm operates, and how it achieves the flexibilities discussed earlier. The implementation used to run the examples has been used as the parser for an intelligent interface to an electronic mail system [2]. The intelligence in this interface is concentrated in a *User Agent* that mediates between the user and the underlying mail system to ensure that the interaction goes smoothly. The Agent does this by, among other things, checking that the user specifies the operations he wants performed and their parameters correctly and unambiguously, conducting a dialogue with the user if errors or ambiguities arise. The role of FlexP as the Agent's parser is to transform the user's input into the internal representations employed by the Agent, resolving as many of the errors or potential ambiguities that it can, so as to minimize the amount of interaction between Agent and user necessary to arrive at a correct and unambiguous version of the input. Usually the user's input is a request for action by the mail system or a description of objects known to the mail system. Our examples are drawn from that context.

4.1 Preliminary Example

Suppose the user types

display new messages

Parsing begins as soon as any input is available. The first word is used as an index into the store of *rewrite rules*. Each rule gives a pattern and a structure to be produced when the pattern is matched. The components of the structure are built from the structures or words that match the elements of the pattern. The word "display" indexes the rule:

```
(pattern: (Display   MessageDescription)
 result:  (StructureType: OperationRequest
           Operation:     Display
           Message:       (Filler MessageDescription)))
```

Note that the non-terminals in the pattern of this and subsequent rules are specific to the message system domain, so that the grammar being used is semantic rather than syntactic. Using this rule the parser constructs the partial parse tree

```
(Display    MessageDescription)
   |
   |
 display
```

We call the partially instantiated pattern that labels the upper node a *hypothesis*. It represents a possible interpretation for a segment of input.

The next word "new" does not directly match the hypothesis, but since "new" is a MsgAdj (an adjective that can modify a description of a message), it indexes the rule:

```
(pattern:  (?Det  *MsgAdj  MsgHead  *MsgCase)
 result:   (StructureType: MessageDescription
            Components: ------------))
```

Here, "?" means optional, and "*" means repeatable. For the sake of clarity, we have omitted other prefixes that distinguish between terminal and non-terminal pattern elements. The result of this rule is a structure of type MessageDescription that fits the current hypothesis, and so extends the parse as follows:

```
(Display    MessageDescription)
   |              |
   |              |
   |       (?Det *MsgAdj MsgHead *MsgCase)
   |                |
   |                |
display            new
```

The top-level hypothesis is not yet fully confirmed even though all of its own elements are matched. Its second element matches another lower level hypothesis that is only incompletely matched. This lower pattern becomes the *current hypothesis* because it predicts what should come next in the input stream.

The third input matches the category MsgHead (head noun of a message description) and so fits the current hypothesis. This match fills the last non-optional slot in that pattern. By doing so it makes the current hypothesis and its parent pattern *potentially complete*. When the parser finds a potentially complete phrase whose result is of interest to the Agent (and the parent phrase in this example is in that category), the result is constructed and sent to the Agent.[2] However, since the parser has not seen a termination signal, this parse is kept active. The input seen so far may be only a prefix for some longer utterance such as "display new messages about ADA". In this case "about ADA" would be recognized as a match for MsgCase (a prepositional phrase that can be part of a message description), the parse would be extended, and a revision of the previous structure sent to the Agent.

4.2 Unrecognized Words

When an input word cannot be found in the dictionary, FlexP tries to spelling-correct the input word against a list of possibilities derived from the current hypothesis. For example:

display the new messaegs

produces the partial parse

```
(Display   MessageDescription)
  |            |
  |            |
  |       (?Det  *MsgAdj  MsgHead  *MsgCase)
  |          |      |
  |          |      |
display     the    new
```

The lower pattern is the current hypothesis and has two elements eligible to match the next input. Another MsgAdj could be matched. A match for MsgHead would also fit. Both elements have associated lists of words that match them or occur in phrases that match them. The one for MsgHead includes the word "messages", and the spelling corrector passes this back to the main part of the parser as the most likely interpretation.

In some cases the spelling corrector produces several likely alternatives. The parser handles such ambiguous words using the same mechanisms that accommodate phrases with ambiguous interpretations; that is, alternative interpretations are carried along until there is enough input to discriminate those that are plausible from those that are not. The details are given in the next section.

[2] What happens to the result when the Agent receives it is beyond the scope of this paper. However, we should note that the Agent is not obliged to act on the result right away. One strategy is for the Agent to perform immediately "safe" actions, such as the identification or display of a set of messages, but to wait for explicit termination of "unsafe" commands, such as those to send or delete messages.

The user may also correct the input text himself. These changes are handled in much the same way as those proposed by the spelling corrector. Of course, these user-supplied changes are given priority, and parses built using the former version must be modified or discarded.

4.3 Ambiguous Input

In the first example there was only one hypothesis about the structure of the input. More generally, there may be several hypotheses that provide competing interpretations about what has already been seen and what will appear next. Until these partial parses are found to be inconsistent with the actual input, they are carried along as part of the active parse. Therefore the active parse is a set of partial parse trees each with a top-level hypothesis about the overall structure of the input so far and a current hypothesis concerning the next input. The actual implementation allows sharing of common structure among competing hypotheses and so is more efficient than this description suggests.

The input

were there any messages on.......

could be completed by giving a date ("...on Tuesday") or a topic ("...on ADA"). Consequently, the subphrase "any messages on" results in two partial parses:

```
(?Det  *MsgAdj  MsgHead   *MsgCase)
  |               |          |
  |               |          |
 any           messages   (On Date)
                            |
                            |
                            on

(?Det  *MsgAdj  MsgHead   *MsgCase)
  |               |          |
  |               |          |
 any           messages   (On Topic)
                            |
                            |
                            on
```

If the next input were "Tuesday" it would be consistent with the first parse, but not the second. Since one of the alternatives does account for the input, those that do not may be discarded. On the other hand, if all the partial parses fail to match the input, other action is taken. We consider such situations in the section on suspended parses.

4.4 Flexible Matching

The only flexibility described so far is that allowed by the optional elements of patterns. If omissions can be anticipated, allowances may be built into the grammar. In this section we show how other omissions may be handled and other flexibilities achieved by allowing additional freedom in the way an item is allowed to match a pattern. There are two ways in which the matching criteria may be relaxed, namely

- relax consistency constraints, e.g. number agreement
- allow out-of-order matches

Consistency constraints are predicates that are attached to rules. They assert relationships that must hold among the items which fill the pattern, e.g. number agreement. Although such relationships can usually (it depends on the particular relation) also be expressed through context-free rewrite rules, they can be expressed much more compactly through consistency constraints. The compactness that can be achieved in this way has often been exploited by augmenting context-free parsers to deal with consistency constraints on their context-free rules. The flexibility achieved by relaxing such constraints in ATN parsers [18] has been explored by Weischedel and Black [16] and by Kwasny and Sondheimer [13]. This technique would fit smoothly into FlexP but has not actually been needed or used in our current application.

On the other hand, out-of-order matching is essential for the parser's approach to errors of omission, transposition, and substitution. Even when strictly interpreted, several elements of a pattern may be eligible to match the next input item. For example, in the pattern for a MessageDescription

```
(?Det *MsgAdj MsgHead *MsgCase)
```

each of the first three elements is initially eligible but the last is not. On the other hand, once MsgHead has been matched, only the last element is eligible under the strict interpretation of the pattern.

Consider the input

display new about ADA

The first two words parse normally to produce

```
(Display   MessageDescription)
    |              |
    |              |
    |    (?Det  *MsgAdj  MsgHead  *MsgCase)
    |              |
    |              |
 display          new
```

The next word does not fit that hypothesis. The two eligible elements predict either another message adjective or a MsgHead. The word "about" does not match either of these, nor can the parser construct any path to them using intermediate hypotheses. Since there are no other partial parses available to account for this input, and since normal matching fails, flexible matching is tried.

First, previously skipped elements are compared to the input. In this example, the element ?Det is considered but does not match. Next, elements to the right of the eligible elements are considered. Thus MsgCase is considered even though the non-optional element MsgHead has not been matched. This succeeds and allows the partial parse to be extended to

```
(Display   MessageDescription)
    |              |
    |              |
    |    (?Det  *MsgAdj  MsgHead  *MsgCase)
    |              |                  |
    |              |             (About topic)
    |              |                 |
 display          new              about
```

which correctly predicts the final input item.[3] As already described, flexible matching is applied using the same three methods used for the initial non-flexible matching, i.e. first for direct matches with pattern elements in the current partial parses, then for indirect matches, and then for matches that involve extending the partial parses upwards.

Unrecognizable substitutions are also handled by this flexible matching mechanism. In the phrase

display the new stuff about ADA

the word "stuff" is not found in the dictionary, so spelling correction is tried but does not produce any plausible alternatives. However, the remaining inputs can be parsed by simply omitting "stuff" and using the flexible matching procedure. Transpositions are handled through one application of flexible matching if the first element of the transposed pair is optional, two applications if not.

4.5 Suspended Parses

Interjections are more common in spoken than in written language but do occur in typed input sometimes. To deal with such input, our design allows for blocked parses to be suspended rather than merely discarded.

Users, especially novices, may embellish their input with words and phrases that do not provide essential information and cannot be specifically anticipated. Consider two examples:

display please messages dated June 17

display for me messages dated June 17

In the first case, the interjected word "please" could be recognized as a common noise phrase that means nothing to the Agent except possibly to suggest that the user is a novice. The second example is more difficult. Both words of the interjected phrase can appear in a number of legitimate and meaningful constructions; they cannot be ignored so easily.

[3] This technique can, of course, produce parses in which required pattern elements have no match. Whether these match failures are important enough to warrant interrogation of the user is determined in our example system by the intelligent User Agent which interprets the input. In the case above, failure to match the MsgHead element would not require further interaction because the meaning is completely determined by the non-terminal element, but interaction would be required if, for instance, the topic element failed to match.

For the latter example, parse suspension works as follows. After the first word, the active parse contains a single partial parse:

```
(Display     MessageDescription)
   |
   |
display
```

The next word does not fit this hypothesis, so it is suspended. In its place, a new active parse is constructed. It contains several partial parses including

```
(For Person)   and   (For TimeInterval)
  |                    |
  |                    |
 for                  for
```

The next word confirms the first of these, but the fourth word "messages" does not. When the parser finds that it cannot extend the active parse, it considers the suspended parse. Since "messages" fits, the active and suspended parses are exchanged and the remainder of the input processed normally, so that the parser recognizes "display messages dated June 17" as if it did not contain "for me".

5. Conclusion

When people use language naturally, they make mistakes and employ economies of expression that often result in language that is ungrammatical by strict standards. In particular, such grammatical deviations will inevitably occur in the input of a computer system that allows its users to employ natural language. Such a computer system must, therefore, be prepared to parse its input flexibly, if it is to avoid frustration for its users.

In this paper, we have outlined the main kinds of flexibility that should be provided by a natural language parser intended for natural use. We have also described a bottom-up pattern matching parser, FlexP, which exhibits many of these flexibilities, and which is suitable for restricted natural language input to a limited-domain system.

Acknowledgements

The authors thank the referees for their extremely detailed comments and Michael McCord and George Heidorn for their helpful editing suggestions.

References

1. Aho, A. V. and Ullman, J. D. *The Theory of Parsing, Translation, and Compiling.* Prentice-Hall, Englewood Cliffs, 1972.
2. Ball, J. E. and Hayes, P. J. Representation of Task-Independent Knowledge in a Gracefully Interacting User Interface. Proc. 1st Ann. Mtg. of the AAAI, Stanford University, August, 1980, pp. 116-120.
3. Bobrow, D. G., Kaplan, R. M., Kay, M., Norman D. A., Thompson, H., and Winograd, T. "GUS: a Frame-Driven Dialogue System." *Artif. Intell. 8* (1977), 155-173.
4. Burton, R. R. Semantic Grammar: An Engineering Technique for Constructing Natural Language Understanding Systems. TR 3453, Bolt Beranek and Newman, Inc., Cambridge, Mass., December, 1976.
5. Carbonell, J. G. Towards a Self-Extending Parser. Proc. 17th Ann. Mtg. of the ACL, La Jolla, Ca., August, 1979, pp. 3-7.
6. Carbonell, J. G. *Subjective Understanding: Computer Models of Belief Systems.* Ph.D. Thesis., Yale University, 1979.
7. Chester, D. "A Parsing Algorithm That Extends Phrases." *Am. J. Comp. Ling. 6,* 2 (1980), 87-96.
8. DeJong, G. *Skimming Stories in Real-Time.* Ph.D. Thesis., Computer Science Dept., Yale University, 1979.
9. Erman, L. D., and Lesser, V. R. HEARSAY-II: Tutorial Introduction and Retrospective View. Tech. Report, Computer Science Department, Carnegie-Mellon University, 1978.
10. Hayes, P. J., and Reddy, R. Graceful Interaction in Man-Machine Communication. Proc. 6th IJCAI, Tokyo, 1979, pp. 372-374.
11. Hendrix, G. G. Human Engineering for Applied Natural Language Processing. Proc. 5th IJCAI, MIT, 1977, pp. 183-191.
12. Kaplan, S. J. *Cooperative Responses from a Portable Natural Language Data Base Query System.* Ph.D. Thesis, Dept. of Comp. and Inform. Science, University of Pennsylvania, 1979.
13. Kwasny, S. C. and Sondheimer, N. K. "Relaxation Techniques for Parsing Grammatically Ill-Formed Input in Natural Language Understanding Systems." *Am. J. Comp. Ling. 7,* 2 (1981), 99-108.
14. Parkison, R. C., Colby, K. M., and Faught, W. S. "Conversational Language Comprehension Using Integrated Pattern-Matching and Parsing." *Artif. Intell. 9* (1977), 111-134.
15. Waltz, D. L. "An English Language Question Answering System for a Large Relational Data Base." *Comm. ACM 21,* 7 (1978), 526-539.
16. Weischedel, R. M. and Black, J. "Responding Intelligently to Unparsable Inputs." *Am. J. Comp. Ling. 6,* 2, (1980), 97-109.
17. Wilks, Y. A. Preference Semantics. In *Formal Semantics of Natural Language,* Keenan, Ed., Cambridge Univ. Press, 1975.
18. Woods, W. A. "Transition Network Grammars for Natural Language Analysis." *Comm. ACM 13,* 10 (Oct. 1970), 591-606.
19. Woods, W. A., Kaplan, R. M., and Nash-Webber, B. The Lunar Sciences Language System: Final Report. TR 2378, Bolt Beranek and Newman, Inc., Cambridge, Mass., 1972.
20. Woods, W. A., Bates, M., Brown, G., Bruce, B., Cook, C., Klovstad, J., Makhoul, J., Nash-Webber, B., Schwartz, R., Wolf, J., and Zue, V. Speech Understanding Systems - Final Technical Report. TR 3438, Bolt Beranek and Newman, Inc., Cambridge, Mass., 1976.

Philip J. Hayes is a research computer scientist in the Department of Computer Science at Carnegie-Mellon University. He received the D.Sc. degree in computer science from the Ecole polytechnique federale de Lausanne in 1977.

George V. Mouradian is a research programmer in the Department of Computer Science at Carnegie-Mellon University. He received the M.Ph. degree in computer science from Syracuse University in 1978.