Controlling Search in Flexible Parsing

Steven Minton , Philip J. Hayes, and Jill Fain

Computer Science Department , Carnegie-Mellon University
Pittsburgh, PA 15213, USA

Abstract

Most natural language parsers require their input to be grammatical. This significantly constrains the search space that they must explore during parsing. Parsers which attempt to recover from extragrammatical input contend with a search space that is potentially much larger, since they cannot necessarily prune branches when grammatical expectations are violated. In this paper we discuss the control structure of the experimental MULTIPAR parser, which directs its search by exploring potential parses in order of their degree of grammatical deviation/

1 Introduction

Most natural language processing systems parse their input by searching through a space of partial parses. They operate this way because, even though complete utterances alone or in context may be quite unambiguous, natural language is highly ambiguous locally. For instance, individual words can be ambiguous in their meaning or part of speech (e.g. "bank"), or components of utterances can fit together in more than one way (e.g. "look at the man with the telescope"). A parser's search space for a given input is defined by the relevant set of local ambiguities. A search succeeds if a globally acceptable parse is found that accounts for all the input. There are various techniques to reduce search in parsing, including looking ahead to resolve local ambiguities [5], or ignoring local alternatives that are inconsistent with domain specific semantic constraints |2, 4]. However, no techniques can completely eliminate search from natural language parsing.

The search problem becomes much worse if we require a parser to cope with extragrammatical input. For practical natural language interface systems, this requirement is a real one [1]. Such interfaces must contend with the grammatical errors that inevitably arise when people use natural language interactively. Moreover, they also must cope with input that is correct, but outside their domain restricted grammars.[3] We use the term *flexible parser* for a parser that can handle extragrammatical input.

The major search problem in flexible parsing lies in the criterion for identifying failing branches of the search — normally the violation of some syntactic or semantic expectation. While parsers that require grammatical input can employ this constraint to prune the search tree, flexible parsers must act more cautiously. If a candidate parse violates an expectation, it may mean that the candidate parse is incorrect and should be abandoned, or it may mean that the input really does violate the parser's expectations in the way that has been detected. In the latter case, the parser should not abandon the branch, but should try to recover from the deviation and complete the parse along that branch.

AT&T Bell Laboratories Scholar

This research was sponsored in part by the Air Force Office of Scientific Research under Contract AFOSR 82-0219.

One approach to this problem is to abandon a search branch only when the flexible parser has run out of correction techniques to apply. This, in effect, enlarges the grammar of the parser to cover not only the inputs originally considered grammatical, but also those that can be recognized by any combination of available recovery methods. Although straightforward, this approach is unlikely to produce acceptable results. First, given the range of possible recovery techniques [1], the search space will quickly become unmanageably large. Second, the recovery techniques may generate spuriously corrected" parses of grammatical input. Finally, the described approach provides no way to distinguish between parses that involve widely varying degrees of correction (e.g. simple spelling corrections versus hypothesization of entire phrases).

What is needed, then, is a control structure that allows the normal criterion of extragrammaticality to cut off failing searches, but also accommodates the application of recovery techniques to reactivate failed search branches if no grammatical parse can be found. Moreover, the recovery techniques should be ordered across all search branches according to the degree of ungrammatically their use implies, i.e. the simpler ones (like spelling correction) must be tried in all branches of the parse before the more complex and unlikely ones (like missing word insertion) are tried in any branch.

This paper presents a control structure which satisfies these goals. The next section describes the control structure from the point of view of the programmer constructing a parser that uses it. Section 3 discusses some efficiency issues that arose in implementing the control structure.

2 A Programmer's View of the Control Structure

The control structure described in this paper was developed in the context of a restricted domain parser consisting of a collection of caseframe instantiation strategies. We have previously used the phrases multi strategy [3] and entity oriented [2] to describe this approach. There is no space here to describe this parser, called MULTIPAR, in detail. The most important characteristic of MULTIPAR from the control structure point of view is that its caseframe interpretation strategies are programmed directly, rather than being driven by a declarative formalism such as a transition network. We will refer to the person who writes strategies as the strategy programmer. In some sense, the strategy programmer is the user of the control structure.

Each strategy is an expert at parsing certain types of constructs. Strategies cooperate by calling upon each other to parse sections of the input sentence. When a strategy encounters particular difficulties (violated expectations) while parsing its input, several options are typically available. The options always include simply

[3] We use the term grammar broadly here to cover the semantic expectations used by many restricted domain systems in addition to syntactic ones.

reporting failure, but may also include recovery methods to resolve the violated expectation. The MULTIPAR control structure provides the programmer with a method of specifying the alternative ways of proceeding, and indicating how much of a deviation each option would represent, without requiring him to schedule the investigation of the options explicitly. This scheduling is taken care of by the control structure automatically

The construct provided[4] by the control structure to specify alternative ways of proceeding in the face of violated expectations is the SPLIT statement A SPLIT statement splits the computation into parallel branches — one branch for each option. For each branch, the programmer specifies a *flexibility increment* indicating the degree of grammatical deviation implied by producing a successful parse via that branch. For instance, if a violated expectation could be resolved by a spelling correction or by hypothesizing a missing word, these two options would be specified as different branches of a SPLIT. The control structure would then pursue the two options independently. However, the spelling •correction option would have a lower flexibility increment than the missing word hypothesization, and so it would be pursued first. If it led to a complete parse, the missing word hypothesization would *never* be tried.

A stylized example of a split statement is:

```
(Split (+0 actionA)
       (+1 actionB)
       (+3 actionC) ....)
```

Execution of this SPLIT statement produces a three-way branch in the search tree. Action A has a zero flexibility increment, implying no grammatical deviation along this branch. Actions B and C have flexibility increments of 1 and 3 respectively. This means that Actions B and C would be scheduled for later investigation, while Action A would be pursued immediately.

The system maintains a global Current Flexibility Level whose value is equal to the flexibility level of the least deviant partial parse that remains to be investigated. In this way, the control structure can guarantee that parses are attempted in strict flexibility order and can generate all anc' only parses at the lowest flexibility level at which a global parse succeeds. In particular, if a grammatical parse can be found, then all and only grammatical parses will be generated.

It is important to note that the flexibility level of a parse is the sum of all flexibility increments of all SPLIT statement branches used to achieve the parse. In terms of the stylized example above, this means that other branches in entirely different parts of the tree may be tried between trying Actions B and C. It also means that Actions B and C may be tried, even if Action A succeeds locally, so long as the parse fragment produced by Action A does not participate in a complete global parse. This global comparison of the sums of the local flexibility increments is crucial in ensuring that recovery techniques are attempted in order of drasticness across the entire search space of a parse. It also ensures that improbable combinations of recovery techniques are not applied if simpler parses can be found.

Another advantage provided by the SPLIT statement is that recovery actions can be closely integrated with the normal parsing process. Instead of having a separate recovery phase that occurs independently of normal parsing, recovery actions occur within the local context of strategies. Therefore, only recovery actions appropriate to the context need be applied. This is important for recovery strategies such as spelling correction, where availability of the local context can provide information that constrains the range of possible corrections.

Let us now look at a less stylized use of SPLIT. The following algorithm is a simplified version of the strategy MULTIPAR uses for parsing imperative sentences.

Imperative Caseframe Strategy

1. Find the head verb of *the sentence.*

2. Retrieve an unmstantiated caseframe for the action associated with this verb.

3 Identify the semantic type of the syntactic direct object. Call the Nounphrase Strategy to find an object of that type at the begining of the unparsed segment.

4. Determine the unnued marked cases and SPLIT

0 Alternative!: Recognize next word as a case-marker for an unfilled marked case; attempt to fill that case with the remaining segment.

+ 5 Alternative 2: Hypothesize that a case marker for an unfilled marked case is missing; attempt to fill the case with the remaining segment.

5. If sentence has not been completely parsed, go to 4.

Let us assume that MULTIPAR is being used as the front end to a mail system, and that the user has just composed a message to be sent. To parse a command such as "Mail message to Paul@CMUA", the strategy would first identify "Mail" as the head verb, and SEND as its corresponding action, and then call the Nounphrase Strategy to recognize a potential MSG-OBJECT as the direct object. Assuming this lower level strategy parses "message" correctly, the imperative strategy then reaches the SPLIT statement. At this point, two branches of the search tree are created with flexibility levels equal to the sum of the Current-Flexibility-Level (which is 0) and the corresponding flexibility increments. The branch corresponding to Alternative2 is scheduled by the control structure at flexibility level 5 (0 plus 5). The branch corresponding to Alternativel still has flexibility level 0 (0 plus 0), and so it continues immediately. Alternativel would successfully recognize "to" as a marker for SEND's destination case, and call a lower-level strategy to parse "Paul@CMUA" as the MSG-DESTINATION. Thus, this branch of the parse succeeds and the other branch spawned by the SPLI[T] is never tried.

A common error in spontaneous input is to omit case markers, so let us suppose now that the input reads "Mail message Paul@CMUA". As before, after "message" is recognized as the direct object, the SPLIT statement is encountered. However, this time Alternativel reports failure. If the control structure finds no other branches of the tree suspended at flexibility level 0 (the Current Flexibility Level), it will look for suspended branches at higher flexibility levels. In our present example, it will find the branch suspended earlier at level 5. The Current Flexibility-Level is set to 5, and computation is restarted at Alternative2. This means that the imperative strategy will now hypothesize that a case-marker has been omitted, and will try to parse "Paul@CMUA" as one of the unfilled cases for SEND. When "Paul@CMUA" is recognized as a possible MSG-DESTINATION, the input will have been completely accounted for, and the parse would be the same as for the first example.[5] Notice that this recovery action is specific to the violated

MULTIPAR is implemented in Common Lisp.

A complete interface system might want to confirm its interpreptation with the user.

expectation of finding a case marker. Because the action is context-dependent, it would have been more difficult to achieve in a completely separate recovery phase

Even with this simple example, it will be clear that the size of the search tree can grow rapidly when recovery is attempted. If "Paul@CMUA" qualified as both a MSG-SOURCE and a MSG-DESTINATION, Alternative2 above would have to split again, and two alternative corrected parses would be produced. Then too, Paul@CMUA might be the name of a misspelled message-header. Exploring this alternative would be the responsibility of one of the strategies called while parsing the direct object. Note that spelling correction can potentially generate many alternatives, especially if words in the parser's lexicon can be considered as potential misspellings of other words in the lexicon (perhaps the user intended "Make" instead of "Mail").

These examples may make clearer the importance of exploring all potential parses at lower flexibility levels before any of those at higher levels. Witness the computational expense inherent in recovering a missing case marker, i.e. trying all unfilled cases. If there is still a possibility that branches of the search requiring less drastic recovery techniques might yet succeed, they must be attempted first. For example, the sentence "Mail message should be saved" will be recognized by a strategy for declarative sentences that is invoked in a branch parallel to the imperative strategy. Since this branch succeeds at level 0, it should be examined in its entirety before the imperative strategy attempts to hypothesize a missing case marker

This best-first order of exploring the search tree implies that grammatical parses will be discovered relatively quickly. (A disadvantage, of course, is that ungrammatical, but recoverable parses may be produced significantly more slowly.) Equally important, the use of flexibility levels imposes a partial order on deviant parses, so parses that are highly undesirable will never be discovered if better alternatives exist. For example, a parse with two spelling corrections will not be generated if a parse with a single spelling correction can be found. At times, the ordering may be rather arbitrary (e.g. is a missing case marker worse than a single spelling mistake?). However, such arbitrary judgments tend to overconstrain the search rather than underconstrain it, which seems appropriate.

3 Implementing the Multipar Control Structure Efficiently

In order for the control structure outlined in the previous sections to be of practical use, it must implement the best-first search in an efficient manner, and it must be convenient for the strategy programmer to use. In this section we outline some of the engineering considerations that proved to be crucial in achieving these goals.

- *Usability:* MULTIPAR consists of many communicating strategies, each of which may involve a complex computation. The control structure provides a standard interface for one strategy to call upon another and controls the pseudo parallel exploration of the search tree. An important attribute of the control structure is its unobtrusiveness; the strategy writer is provided with a small set of facilities for executing strategy calls and parallel actions

- *Efficient Context Re-creation:* To return to an alternative on the agenda, the local context at the SPLIT statement must be re-created. Rather than saving the complete state of the computation, context recreation is effected by re-executing the local strategy from its inception. This seemingly inefficient mechanism is quite practical due to two factors: most scheduled alternatives are never attempted during a typical parse, and a caching mechanism is used to store substrategy results.

- Sharing *Strategy Results*: It is often the case that parallel branches will duplicate each other's work, since they may differ only in a few respects. This is especially true when recovery actions are initiated, since the number of branches tends to grow dramatically as higher flexibility levels are reached. Because of this, the mechanism for caching substrategy results has a dual purpose. In addition to enabling rapid context re-creation, it makes the overall operation of the parser more efficient by allowing strategies to share results.[6] For example, to recover from a missing case marker, the lower-level case filler strategy has to be called once for each case that could possibly be filled. Each time it is called it may have to operate somewhat differently depending on the constraints for that case (e.g. call a name-recognizing sub strategy or check to see whether the input can be found among current message headers). However, much of the work may be identical in each instance, and so caching produces considerable savings.

4 Conclusion

All natural language parsers must perform some search, but when a parser is intended to handle ungrammatical as well as grammatical input, its search space becomes very large. The control structure described in this paper allows a large, complex search space of this kind to be explored in an orderly manner. Efficiency is improved by a caching mechanism that takes advantage of the significant amount of redundancy present in the search space. The control structure provides convenient facilities for specifying the search space, while automatically performing the bookkeeping necessary for an efficient search.

We have built a version of MULTIPAR that parses natural language commands to an operating system. Experience with both grammatical and deviant sentences in this domain suggests that the control structure adequately fulfills the requirements for a flexible parser outlined earlier.

5 Acknowledgements

We thank Jaime Carbonell for his help and participation in all phases of this project.

References

1. Carbonell, J. G. and Hayes, P. J. "Recovery Strategies for Parsing Extragrammatical Language." *Computational Linguistics* 70(1984).

2. Hayes, P. J. Entity Oriented Parsing. COLING84, Stanford University, July, 1984.

3. Hayes, P. J. and Carbonell, J. G. Multi Strategy Parsing and its Role in Robust Man Machine Communication. Carnegie-Mellon University Computer Science Department, May, 1981.

4. Hendrix, G. G. Human Engineering for Applied Natural Language Processing. Proc. Fifth Int. Jl. Conf. on Artificial Intelligence, MIT, 1977, pp. 183-191.

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

A relatively sophisticated caching mechanism is required to support this functionality. Due to the parallelism introduced by the SPLIT statement, a strategy may return different results at different flexibility levels, or even at the same flexibility level. The caching mechanism must not only record the various results, but also keep track of which strategies access the cache. This allows the control structure to create appropriate additional branches of the search tree if the cached set of results for a strategy is updated (with a new value at a higher flexibility level) after it has been previously accessed by one or more other strategies.