A multi-processing approach to natural language

by RONALD M. KAPLAN

Harvard University
Cambridge, Massachusetts

INTRODUCTION

Natural languages such as English are exceedingly complicated media for the communication of information, attitudes, beliefs, and feelings. Computer systems that attempt to process natural languages in more than the most trivial ways are correspondingly complex. Not only must they be capable of dealing with elaborate descriptions of how the language is put together (in the form of large dictionaries, grammars, sets of inference strategies, etc.), but they must also be able to coordinate the activities and interactions of the many different components that use these descriptions. For example, speech understanding systems of the sort that are currently being developed under ARPA sponsorship must have procedures for the reception of speech input, phonological segmentation and word recognition, dictionary consultation, and morphological, syntactic, semantic, and pragmatic analyses. The problems of coordination and control are reduced only slightly in less ambitious projects such as question answering, automatic programming, content analysis, and information retrieval. Of course, large-scale software systems in other domains might rival natural language programs in terms of the number and complexity of individual components. The central theme of the present paper, however, is that natural language control problems have a fundamentally different character from those of most other systems and require a somewhat unusual solution: the many natural language procedures should be conceptualized and implemented as a collection of *asynchronous communicating parallel processes.*

A common technique for organizing a large system is to arrange the flow of control in a hierarchy that follows the intuitive "outside-in" flow of data. In language processing, this design calls for a serial invocation of procedures, with the input routines succeeded by segmentation, word recognition, morphological analysis, and dictionary look-up. The output of these procedures would be passed to the syntactic analyzer (parser), which would build syntactic structures and send them on to the semantic and inference-making routines. The difficulty with this straightforward arrangement is that each procedure in the chain must operate in a locally uncertain environment. For example, there might not be enough information in the incoming signal to determine precisely what the string of words is (is it 'nitrite' or 'night rate'?), or dictionary consultation might produce several senses for a single, clearly identified word ('saw' as a noun, a form of the verb 'saw', or a form of the verb 'see'). Later on, the syntactic analyzer might discover several parses, or the semantic procedures might find multiple interpretations. Each level of analysis might be prepared to handle many independent possibilities, some of which are passed from earlier modules, and some of which it generates and passes on. Except for certain unusually well-behaved inputs, this linear control strategy will lead to exponential increases in the amount of computation, and the system will be hopelessly swamped in the combinatorics.

Any realistic system must have ways of selectively ignoring certain implicit possibilities, thereby reducing the effective size of the computation space. After all, most sentences spoken in everyday conversation are not ambiguous, given their total linguistic and pragmatic context. This means that if we search the computation space to exhaustion, we will find that most of the possibilities for most of the inputs lead to dead-ends, and there is only one globally consistent interpretation. The problem is to minimize the number of dead-ends encountered in arriving at this interpretation and to stop computing as soon as it is achieved. Thus if the segmentation and word recognition routines first come up with the 'nitrate' possibility, we want to feed this immediately to the syntactic and semantic routines. If a meaningful interpretation results, the extra work necessary to discover the 'night rate' sequence can be avoided, as can the additional effort that all "later" modules would have to devote to it. But if 'nitrate' is incompatible with the surrounding context, the segmentation routines must be able to resume where they left off and produce 'night rate'. In general, the various modules must be capable of communicating results and intermingling their operations in a "heterarchical"[4] fashion according to some *heuristic* strategies.

The simple hierarchical model must be abandoned, but this does not mean that module interactions can be completely unconstrained. We distinguish between *intrinsic* and *extrinsic* constraints on the order of computation. It is logically impossible for an operation that applies to a datum to be executed before that datum has come into existence. For example, the syntactic analysis of a section of the input cannot begin until at least some of the possi-

ble words in that section have been identified and at least some of their syntactic properties have been retrieved from the dictionary. This is an instance of an intrinsic ordering restriction. On the other hand, it is not logically necessary for the entire set of word or dictionary possibilities to be explicitly available prior to any syntactic operations, although "outside-in" systems impose this kind of extrinsic ordering constraint. Other control regimes might enforce different extrinsic constraints: left-to-right models, for example, require that syntactic processing be completed early in the input before segmentation is even attempted in later sections. Clearly, a model that imposes no extrinsic constraints and ensures that all intrinsic constraints are satisfied will provide the maximum degree of freedom and safety for the exercise of heuristic control strategies, and consequently, should result in the most efficient and effective natural language processors.

The multi-processing control model described below constitutes a framework in which these ideal conditions can be met. It has evolved from earlier work on a "General Syntactic Processor" (GSP), which has been discussed in some detail in another paper (Kaplan, in press[2]). The essential concepts of GSP, insofar as they are relevant to multi-processing, are presented in the following section. A later section describes the advantages of using asynchronous processes within the syntactic component of a natural language system.

AN OVERVIEW OF GSP

GSP is a relatively simple and compact syntactic processor that can emulate the operation of several other powerful algorithms, including Woods' augmented transition network (ATN) parser,[5] Kay's "powerful parser,"[2,3] and transformational generators.[1] This is possible because GSP incorporates in a general way some of the basic facilities that the other algorithms have in common. Most important for the present discussion, it gives explicit recognition to the fact that syntactic strategies are inherently non-deterministic, consisting of many alternatives that must be computed independently. Within this non-deterministic organization, GSP grammars are represented as arbitrary procedures as Winograd[5] has recently advocated. GSP also provides a small set of primitive operations that can be invoked by grammatical procedures and which seem sufficient for most ordinary syntactic strategies.

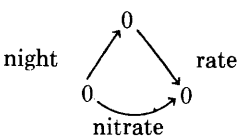
There are two distinct sources of ambiguity in syntactic processing: *grammatical alternatives* and *structural alternatives*. Grammatical alternatives occur because natural language sentences and phrases can be realized in many different ways. For example, sentences in English can be either transitive ('Mother cooked the roast.') or intransitive ('Mother cooked.'), while noun phrases might or might not begin with determiners ('the men' versus 'men'). A grammar must somehow describe the myriad patterns that the language allows, and one of the most elegant ways to express such possibilities is in the form of a (suitably augmented) finite-state transition network. (See Woods[6] for a fuller discussion of these issues). Thus GSP grammars are transition networks consisting of sets of states connected by labeled directed arcs. Paths through the grammar following arcs from state to state denote the sequences of formatives that may occur in valid phrases, while multiple arcs emanating from a single state indicate grammatical alternatives that correspond to non-deterministic choices. In actual practice, a GSP grammar is a collection of such networks, each one characterizing some type of syntactic constituent (e.g., sentence, noun phrase, verb phrase, prepositional phrase). These sub-grammars are tantamount to the multiple levels of an ATN grammar or the collection of rules interpreted by the Kay parser.

GSP also gives a formal account of the second source of ambiguity, structural alternatives. As noted above, the input to a syntactic analyzer might contain multiple possibilities, because of uncertainties at the input and dictionary look-up stages (the nitrate/night-rate example). Other alternatives not foreshadowed in the input might arise from the operation of various parts of the grammar. For example, the noun phrase sub-grammar must indicate that the string following 'saw' in sentence (1a) can be parsed as a noun phrase in at least three ways, corresponding to the interpretations (1b-d):

(1) (a) I saw the man in the park with the telescope.
    (b) The man had the telescope.
    (c) The park had the telescope.
    (d) I used the telescope to see the man.

The three noun phrase possibilities must be processed by the sentence level network to produce, in the absence of semantic constraints, two distinct parses for the entire sentence, thus signifying that it is globally ambiguous. The point is that a syntactic processor must be capable of handling structural alternatives in its input and intermediate results, as well as producing them as output.

This being the case, GSP provides a single data organization to represent constituents at all stages of processing. This data structure, called a *chart*, is also a transition network with states and arcs (which are called *vertexes* and *edges* to avoid confusion with the grammar networks). Edges correspond to words and constituents, while vertexes correspond to junctures between constituents. A sequence of edges through the chart therefore represents a single interpretation of the input, and structural alternatives are specified by multiple edges leaving a vertex, as portrayed in (2):



In this simple figure, the arrows indicate the temporal order of constituents, with an edge pointing to the set of

its possible successors. In fact, a temporal successor is just one of many properties that an edge can have. As a bare minimum, an edge that corresponds to a parent node in a linguistic tree must have a pointer to its daughter vertex, and Kaplan[2] (in press) has described the chart as a compact representation for a "family of strings of trees." An edge may have a variety of other syntactic and semantic properties, and the chart is in fact a very general, very complicated kind of syntactic graph.

The basic function of GSP is to apply a grammar network to a chart. Starting at some state and some vertex, GSP compares every arc with every edge. Unlike the arcs of an ordinary finite-state grammar, the labels on GSP arcs are interpreted as sequences of operations to be performed. Some of these operations are predicates which determine the admissibility of a transition by examining not only properties of the current edge but also any information that might have been saved away in named "registers" by previous transitions along this path. If a transition is permitted, other operations on the arc will be executed. These operations can build structures, add them to registers for use on later transitions, or insert them into the chart as structural alternatives at some vertex. Finally, the operations can cause GSP's attention to shift (non-deterministically) to a new state and a new vertex, usually one of the successors of the current edge, where the edge-arc comparison procedure is repeated. Notice that the programs on the arcs give each GSP subgrammar the power of a Turing machine, since registers can store an arbitrary amount of information which can be used to affect transitions on later arcs.

This brief discussion should convey a general feeling for how GSP is organized and what it does. In terms of multiprocessing, the most important points to remember are the following: syntactic processing is conceptualized as the interplay between two transition networks, a grammar and a chart. The grammar is stable and is interpreted as a collection of active procedures that operate on the chart. On the other hand, the chart is both dynamic and passive: it is constantly being augmented with new edges and vertexes as the grammar recognizes new structural possibilities, but it does not usually call for the execution of any actions.

MULTI-PROCESSING IN SYNTAX

Both the augmented transition network and Kay parsers guarantee that the intrinsic restrictions on the order of computation will be met, but they also impose certain extrinsic constraints. The popularity of these algorithms is due in part to the fact that their extrinsic constraints have the desirable effect of cutting down on the size of the computation space, but this is not always done in the most advantageous way. Certain benefits achieved by the ATN parser, which is essentially a top-down algorithm, are lost by the bottom-up Kay procedure, and vice versa. In this section we briefly outline these parsing strategies and point out some of their weaknesses and disadvantages. We then introduce the basic elements of the parallel processing model and show how this control regime can eliminate extrinsic constraints in syntax and lead to a mixture of bottom-up and top-down benefits.

The ATN parser begins processing at the highest syntactic level (the start of the sentence sub-grammar) and at the earliest part of the input, it being assumed that the beginning of the input corresponds to the left boundary of a well-formed sentence. Control passes from left to right through the grammar and input until an arc is reached that calls upon some other sub-grammar to compute a lower-level constituent (e.g. a noun phrase). At this point, the status of computation in the sentence network is stored away on a stack, computation is suspended, and control passes to the beginning of the lower network. If a noun phrase is identifiable at that position in the input, it is constructed and returned to the suspended sentence level, which resumes its left to right scan. Notice that the lower level computations are initiated only when their results will fit into some globally consistent pattern, so that much unnecessary computation can be avoided. For example, in the sentence 'The man left' this strategy will never look for a noun phrase beginning at 'left', since the sentence level has no need for it. Furthermore, the ATN parser can easily stop when the first complete sentence is found; if it is semantically interpretable, the additional effort to discover other parses can be eliminated.

Unfortunately, this strategy is unrealistic in several respects. Very few of the utterances in ordinary conversation are complete, well-formed sentences in which the left boundary is clearly discernible. Instead, conversation consists of a sequence of fragments and partial constituents which cannot be handled in a top-down way, even though they are perfectly understandable to human observers. A noun phrase response to a direct question should be recognized as a noun phrase even though it is not embedded in a sentential context. This is a case in which the extrinsic constraints have mistakenly ruled out some meaningful structural alternatives.

A second shortcoming of the top down strategy is that in its most naive form, it can lead to the repetition of certain computations. Consider the garden path sentence (3):

(3) The cherry blossoms in the orchard are beautiful.

If the parser first assumes that 'blossoms' is the main verb, it will expect the sentence to end after 'orchard' and be quite surprised to find the trailing verb phrase. To obtain the correct analysis, it must back up and change its hypothesis about the syntactic category of 'blossoms,' and then rescan the rest of the string. Unless special care is taken, the computation in which 'in the orchard' was recognized as a prepositional phrase will be re-executed. The top-down extrinsic constraints are not sufficient to forestall this useless effort. To avoid this, the previous structure must be retained and made available to subsequent analysis paths.

The chart provides a convenient repository for such well-formed substrings (WFSS): we can simply associate an extra vertex with the 'in' edge to record the prepositional phrases that have been identified at that position. Then the grammar operations must be modified so that they consult the chart to see if such a vertex exists before invoking the prepositional phrase sub-grammar again. Notice the control problems that arise with this solution: in general there may be several ways of realizing a particular phrase at a particular edge (e.g., 'old men and women' might or might not mean that the women are old). If all the possibilities are computed and inserted at the WFSS vertex together, then some of the work necessary to do this might turn out to be superfluous when we stop after finding the first parse. However, if the alternative phrases are constructed and inserted one at a time, there is the risk of backing up, rescanning, and encountering a WFSS vertex that is only partially complete. Thus there appears to be a trade-off between searching for the first parse and utilizing the simple WFSS mechanism to full advantage.

A top-down strategy builds a constituent only on demand, when it is needed to fulfill a higher-level phrase. The bottom-up Kay algorithm builds all types of phrases wherever they can be recognized in the chart, independent of any larger context. This is done in such a way that when needed by a higher-level computation, a phrase will have already been entered in the chart if it can be recognized at all. The chart becomes a collection of WFSS tables, one for each identified constituent at each edge, and all sub-grammars operate as though they were rescanning previously analyzed sections of the chart. In terms of the description above, the main function of the Kay parser's extrinsic ordering constraints is to preclude encounters with incomplete WFSS vertexes. The essential restriction is that examination of a vertex must be postponed until every vertex to its right has been exhaustively processed by every sub-grammar. In effect, an external controller must simulate a garden-path back up, invoking the sub-grammar at each step (see Kaplan[2] (in press) for a discussion of one way in which this constraint may be implemented).

The Kay algorithm overcomes some of the difficulties of the ATN parser. The total reliance on the well-formed substring machinery ensures that no computation will ever be repeated, so that analyses of sentences that require back up should be more efficient. In addition, the Kay strategy has no trouble identifying the fragments prevalent in natural discourse. These advantages are purchased at some cost, however. The Kay parser can handle fragments because it compulsively searches for every type of constituent in every position, but this can involve a considerable amount of wasted effort when the input is in fact well-formed. Furthermore, although it is possible to stop after the first parse has been discovered, this does not really help to reduce the amount of computation as it does for the ATN. The first parse will not emerge until the back up has reached the beginning of the chart, at which point the processing of the entire computation space is virtually complete. Thus the Kay algorithm is even less amenable to heuristic guidance than the top-down ATN parser.

For both syntactic algorithms, the extrinsic constraints are enforced to govern the circulation of information between the different sub-grammar networks, to make sure that the results of one computation are available when needed by another. The simple top-down approach is to invoke a sub-grammar each time its results are needed by another; the bottom-up approach computes results *before* they are needed and saves them in the chart, where they can be accessed on demand. Of course, the order in which alternatives are considered *within* a single sub-grammar can freely vary without serious global consequences, and this is one area where heuristic strategies can be very helpful. If processing will stop after the first parse, heuristics can direct the parser to find the most likely analysis with the minimum computation, and the effort to obtain a meaningful interpretation can be drastically reduced.

What happens if heuristics are used to change the interactions *between* the independent sub-grammars? In many cases the violation of an extrinsic constraint will have no deleterious effects (which is why it is extrinsic). Often, significant reductions in the amount of computation can be achieved. However, if the violation causes premature scanning of an incomplete vertex, some meaningful analyses may never be discovered. The multi-processing framework to which we now turn provides a general solution to the incomplete vertex problem, thereby eliminating the necessity for extrinsic ordering constraints.

Basically, we conceive of the various sub-networks of a GSP grammar as a collection of asynchronous processes. They operate on overlapping chart sections and use the chart to communicate with one another. For example, a noun phrase process can produce a structure and place it in the chart so that it will be found by any process that can make use of it (such as those corresponding to the prepositional phrase or sentence sub-grammars). Syntactic processes can cause other processes to be initiated, but once created they are, in principle, independent entities and can exert no direct control over each other's activities. Coordination between processes is entirely intrinsic, determined solely by their internal schedules of information production and consumption.

When a process is created at an edge in the chart, a vertex is established to serve as a communication *port* between the process and the global environment. Each edge has a process table in which the port and *type* (e.g., noun phrase) of the new process are recorded. A process is created only when some other entity wants to examine the information it produces. Since identical information will come through the ports of two processes of the same type at the same edge, an edge can only have one process of a given type. When a new consumer of information appears and tries to start a process, the edge's table is consulted to see if an appropriate process already exists. If so, the

previously established port is made available to the new consumer. In general, a process has no knowledge of how many or which entities are using the information it sends to its port, and a consumer has no idea of when or why the producer was created.

A process scans the chart, and whenever it recognizes and constructs a constituent, it sends it to its port. There the constituent gets incorporated as a new edge, and a port is thus a dynamic vertex. It has no edges when it is first established, but edges can appear at any time, as its process sends them back. If no constituents can be found, then its process will eventually exhaust itself without returning any results, and the port's edgeset will always be empty. This indicates to all consumers that the process' type of constituent does not exist at this position in the chart.

Notice that as long as a process is active, its port is essentially an incomplete vertex as described above. The intrinsic coordination of processes is accomplished in the following way: besides containing the results of its process, a port must indicate whether or not its process has terminated. When a port is created, it is given the initial marking "active," which persists until its process sends a termination signal. This must be the process' last action before it goes into dormancy.

When the process becomes inactive, any consumer coming along may scan the edges at the port as if it were an ordinary vertex. However, if the process is active, the consumer must take special precautions: if no edges have yet appeared at the port, the consumer must *wait* at the port, either temporarily suspending all computation or else shifting attention for the moment to some other analysis path. If some edges have already been returned, they may be scanned in the normal way, after which the consumer must again go into a wait state. A port may not be entirely abandoned until its process has become inactive. With these facilities for guaranteeing that intrinsic constraints are maintained, any heuristic strategies that seem promising can be used to guide the course of syntactic analysis. No matter what happens, implicit possibilities will never be lost.

To get a feeling for how the performance of the syntactic component can benefit from these facilities, consider how the ATN and Kay strategies could be implemented. The top-down approach still starts with a single sentence process at the beginning of the chart, but it no longer has to suspend itself when lower-level constituents are needed. To obtain a noun phrase, it spawns an asynchronous noun phrase process and then focuses on the structures being returned at the new port. Noun phrases will appear one by one, and each one can be examined as it arrives. If one of them looks particularly promising, the sentence process can continue scanning past it to see if it will fit into a valid parse. Up to this point, the only difference between this arrangement and the implementation described earlier is that the lower process can be executed in parallel (given an appropriately constructed computer). However, if there is a garden path and the sentence level must back up, rescan earlier context, and move forward again to look for a noun phrase in the same position, the port automatically behaves as a WFSS vertex. Furthermore, the intrinsic waiting constraints will keep track of all possibilities, even if the vertex is still incomplete when it is re-entered. Thus, without adding any new heuristics, the multi-processing framework allows all repetitive computations to be safely avoided within a basically top-down strategy.

This procedure still would not be able to handle fragments, since only the sentence process is initially activated. We still need a bottom-up approach to deal with ordinary discourse. For the Kay algorithm, the external controller will start up every type of process at every edge. If all processes compute to exhaustion, the effect will be the same as the previous Kay implementations. But suppose we introduce the simple heuristic that a process' allocation of real computing resources diminishes in proportion to the number of edges it has produced. If a noun phrase is found at a given position, resources will be shifted away from that process for awhile. Therefore, instead of focusing completely on one end of the chart and slowly backing up, computation will be more or less evenly distributed throughout the sentence. This makes it very likely that the first parse will be found relatively early in the analysis. If it is interpretable, the remainder of the computation space can be ignored, and we will have gained a top-down advantage in our bottom-up approach. However, if no complete parse is found, the bottom-up capability of recognizing fragments and partial constituents will be realized.

CONCLUSION

The multi-processing framework allows us to combine the advantages of the top-down and bottom-up approaches in a straightforward way. Even without intelligent heuristics, we have achieved a much improved syntactic analyzer, and as we learn more about how to make successful syntactic guesses, its efficiency and effectiveness should increase even more. This is also a convenient framework in which to test the heuristics that we might devise, and should be of great use in future investigations.

We are also currently investigating the ways in which this framework can be extended to other components of a natural language system. It appears that alternatives in the segmentation, word recognition, and dictionary look-up modules can all be efficiently represented in the chart, and that these modules can themselves be conceived of as collections of asynchronous processes. As in the syntactic component, these processes will sequence themselves automatically, according to their intrinsic constraints. In fact, it should be possible to turn all modules loose on the same chart at the same time, apply any kind of heuristics, without danger of forgetting alternatives. This would mean that natural language processors could be constructed with almost all extrinsic ordering constraints removed. We are currently investigating this possibility, as well as the possibility of treating a semantic network

like a chart and rules of inference as independent processes. We will report on these investigations in the near future.

REFERENCES

1. Friedman, J., "A computer system for transformational grammar," *Communications of the ACM 12,* pp. 341-348, 1969.
2. Kaplan, R., "A general syntactic processor," in Rustin, R., (ed.), *Natural Language Processing.* Algorithmics Press, New York 1973.
3. Kay, M., *Experiments with a powerful parser,* Santa Monica, The RAND Corporation, RM-5452-PR, 1967.
4. Minsky, M., Papert, S., *Research at the Laboratory Envision, Language and Other Problems of Intelligence,* Cambridge, Massachusetts Institute of Technology, Artificial Intelligence Memo No. 252, 1972.
5. Winograd, T., *Understanding Natural Language,* New York, Academic Press, 1972.
6. Woods, W., Transition network grammars for natural language analysis. *Communications of the ACM 13,* pp. 591-606.