Cascaded ATN Grammars

William A. Woods

Bolt Beranek and Newman, Inc.
50 Moulton Street
Cambridge, Massachusetts 02138

A generalization of the notion of ATN grammar, called a cascaded ATN (CATN), is presented. CATN's permit a decomposition of complex language understanding behavior into a *sequence* of cooperating ATN's with separate domains of responsibility, where each stage (called an ATN *transducer*) takes its input from the output of the previous stage. The paper includes an extensive discussion of the principle of *factoring* -- *conceptual factoring* reduces the number of places that a given fact needs to be represented in a grammar, and *hypothesis factoring* reduces the number of distinct hypotheses that have to be considered during parsing.

1. Introduction

ATN grammars, as presented in Woods (1970), are a form of augmented pushdown store automata, augmented to carry a set of register contents in addition to state and stack information and to permit arbitrary computational tests and actions associated with the state transitions. Conceptually, an ATN consists of a network of states with connecting arcs between them. Each arc indicates a kind of constituent that can cause a transition between the states it connects. The states in the network can be conceptually divided into "levels" corresponding to the different constituents that can be recognized. Each such level has a start state and one or more final states. Transitions are of three basic types, as indicated by three different types of arc. A WRD (or CAT) transition corresponds to the consumption of a single word from the input string, a JUMP transition corresponds to a transition from one state to another without consuming any of the input string, and a PUSH transition corresponds to the consumption of a phrase parsed by a subordinate invocation of some level of the network to recognize a constituent.

ATN's have the advantage of being a class of automata into which ordinary context-free phrase structure and "augmented" phrase structure grammars have a straightforward embedding, but which permit various transformations to be performed to produce grammars that can be more efficient than the original. Such transformations can reduce the number of states or arcs in the grammar or can reduce the number of alternative hypotheses that need to be explicitly considered during parsing. (Some transformations tend to reduce both, but in general there is a tradeoff between the two). Both kinds of efficiency result from a principle that I have called "factoring", which amounts to merging common parts of alternative paths in order to reduce the number of alternative combinations explicitly enumerated. The former ("conceptual factoring") results from factoring common parts of the grammar to make the grammar as compact as possible, while the latter ("hypothesis factoring") results from arranging the grammar so as to factor common parts of the hypotheses that will be enumerated at parse time.

Conceptual factoring promotes ease of human comprehension of the grammar and should facilitate learning of grammars by machine. Hypothesis factoring promotes efficiency of run time execution. In this paper, I will present a generalization of the notion of ATN grammar, called a cascaded ATN or CATN, that capitalizes further on the principle of factoring in a manner similar to serial decomposition of finite state machines. A CATN consists of a sequence of ATN *transducers* each of which takes its input from the output of the previous. An ATN transducer is an ATN that includes among its actions an output operation ("TRANSMIT") which can be executed on arcs to generate elements of an output sequence. Such an ATN cascade gains a factoring advantage from merging common computations at early stages of the cascade.

0362-613X/80/010001-12$01.00

Cascaded ATN's are analogous to serial decomposition of finite state machines and carry many of the advantages of such decomposition into the domain of more general recognition automata. The normal decomposition of natural language description into levels of phonology, lexicon, syntax, semantics, and pragmatics can be viewed as a cascade of ATN transducers - one for each of the individual levels. Viewing natural language understanding as parsing with such a cascade has computational advantages and also provides an efficient, systematic framework for characterizing the relationships between different levels of analysis due to conceptual factoring. The factoring advantages of cascade decompositions can thus serve as a partial explanation of why such a componential description of natural language understanding has arisen.

2. Factoring in ATN's and Phrase Structure Grammars

As discussed above, the principle of factoring involves the merging of common parts of alternative paths through an ATN or similar structure in order to minimize the number of combinations. This can be done either to reduce the size of the grammar or to reduce the number of alternative hypotheses considered at parse time. *Conceptual* factoring attempts to reduce the size of the grammar by minimizing the number of places in the grammar where the same or similar constituents are recognized. Frequently such factoring results from "hiding" some of the differences between two paths in registers so that the paths are otherwise the same and can be merged. For example, in order to represent number agreement between a subject and a verb, one could have two distinct paths through the grammar - one to pick up a singular subject and correspondingly inflected verb, and one to pick up a plural subject and its verb. By keeping the number of the subject in a register, however, one can merge these two paths so there is only one push to pick up the subject noun phrase and one push to pick up the main verb.

In other cases, conceptual factoring results from merging common initial, final, and/or medial sequences of paths across a constituent that are not the same, but which share subsequences. For example, an interrogative sentence can start with an auxiliary verb followed by the subject noun phrase, while a declarative can start with a noun phrase followed by the auxiliary. In either case, however, the subsequent constituents that can make up the sentence are the same and the grammar paths to recognize them can be merged. Moreover, in either case there can be initial prepositional phrases before either the subject or the auxiliary and again these can be merged. When one begins to represent the details of supporting auxiliaries that are present in interrogatives but not in declaratives, the commonalities these modalities have with imperatives, and the interaction of all three with the various possibilities following the verb (depending on whether it is transitive or intransitive, takes an indirect object or complement, etc.), this kind of factoring becomes increasingly more important.

In ordinary phrase structure grammars (PSG's), the only mechanism for capturing the kinds of merging discussed above is the mechanism of *recursion* or "pushing" for constituent phrases. In order to capture the equivalent of the above merging of commonality between declaratives and interrogatives, one would have to treat the subject-auxiliary pair as a constituent of some kind (an organization that is linguistically counter-intuitive). Alternatively, one can capture such factoring in a PSG by emulating an ATN - e.g., by constructing a phrase structure rule for every arc in the ATN and treating the states at the ends of the arc as constituents. Specifically, an arc from s1 to s2 that picks up a phrase p can be represented by a phrase structure rule s1 --> p s2, and a final state s3 can be expressed by an "e rule" s3 --> e (where e represents the "empty string"). In either case, one is forced to introduce a "push" to a lower level of recursion where it is not necessary for an ATN, and to introduce a kind of "constituent" that is motivated solely by principles of factoring and not necessarily by any linguistic criteria of constituenthood.

A phrase structure grammar emulating an ATN as in the above construction will contain all of the factoring that the ATN contains, but will not make a distinction between the state name and the phrase name. Failure to make this distinction masks the intuitions of state transition that lead to some of the ATN optimization transformations and the conceptual understanding of the operation of ATN's as parsing automata. The difference here is a lot like the difference between the way that LISP implements list structure in terms of an underlying binary branching "cons" cell and the way that it is appropriate to view lists for conceptual reasons. For exactly the same kinds of reasons, it is appropriate to think of certain sequences of constituents that make up a phrase as sequences of immediate constituents rather than as a right-recursive nest of binary branching phrases.

From the perspective of *hypothesis* factoring, the distinction made in an ATN between states that can be recursively pushed to and states that merely mark intermediate stages in the recognition of a constituent sequence permits a distinction between that part of a grammar that is essentially finite state (and hence amenable to certain kinds of optimization) and that which is inherently recursive. This permits such operations as mechanically eliminating unnecessary recursion and performing finite-state optimizations procedures on what remains - see Woods (1969). These transformations can result in significant gains in parsing efficiency by trading recursion for iteration wherever possible and by minimizing the non-determinism (by hypothesis factoring) in the resulting networks.

The construction given above for emulating an ATN with a PSG can, of course, emulate the same hypothesis factoring optimization that an ATN permits, but its ability to do so depends critically on the use of e-rules for the final states. Most parsers for PSG's, on the other hand, do not permit e-rules, probably because they are highly non-deterministic when applied bottom-up. Unfortunately, the construction that transforms a PSG with e-rules into an equivalent PSG with no e-rules would give up some of the factoring achieved in the ATN emulation when applied to final states that are not obligatorily final (a common occurrence in natural language grammars). Every transition coming into such a state, would effectively be duplicated - once leading to an unambiguously final state (s1 --> p), and once forcing subsequent consumption of additional input (s1 --> p s2). It thus appears that as a class of formal automata, ATN's permit a greater flexibility in capturing hypothesis factoring advantages than do conventional PSG's.

As we have discussed them, the principles of conceptual factoring and hypothesis factoring have been motivated by different measures of cost. Nevertheless, many of the factoring transformations that can be applied to ATN's gain a simultaneous efficiency in both dimensions. This is not always the case however. In particular, the transformations that optimally minimize nondeterminism for left-to-right parsing tend to cause an increase in the number of states and arcs in a grammar (unless fortuitous regularity causes a collapsing). Since a major characteristic of the ATN grammar formalism is that it permits the expression of mechanical algorithms for performing hypothesis factoring transformations, it is probably appropriate for grammar writers to devote their attention to conceptual factoring as a grammar writing style, while leaving to various grammar compilation algorithms the task of transforming the grammar into an efficient parsing engine. However, in absence of such compilers, it is always possible within the ATN formalism for a grammar writer to incorporate explicit hypothesis factoring structure into his grammar and to make tradeoffs between the two factoring principles.

3. Notation

ATN's are characterized as automata by specifying their computations in terms of instantaneous configurations and a transition function that computes possible successor configurations. As such, they can admit a variety of superficial syntaxes, without changing the essential nature of the automaton. In this paper, I will use a notation that is somewhat more concise and slightly more convenient than the original ATN syntax specified in Woods (1970). The major change will be a formal distinction between a phrase type and an initial state for recognizing a phrase. (The original ATN specification used the initial state to serve double duty.) Moreover, I will permit a given phrase type to have several distinct initial states and several phrase types to share some initial states. This permits somewhat greater flexibility in factoring and sharing common parts of different phrase types. The pop arcs of these ATN's will indicate the phrase type being popped, and a given state can be a final state for several phrase types. A BNF specification of the syntax I will use is given in Figure 1 on the next page.

A simple example, using the conventions given in the figure, is the following grammar:

```
(m (accepts q)
   (s1 (initial q)
       ('a s2 (setr n 1)))
   (s2
       (q s3 (setr n !(1 + !c)))
       (J s3))
   (s3
       ('b s4))
   (s4
       (pop q !n)))
```

This grammar is equivalent (minus augmentation) to the phrase structure grammar: q-->'a'b, q-->'aq'b. It parses a string of n a's followed by n b's and (through its augments) pops the number n.

4. Cascaded ATN's

The advantages of having semantic and pragmatic information available at early stages of parsing natural language sentences have been demonstrated in a variety of systems.[1] Ways of achieving such close interaction between syntax and semantics have traditionally involved writing semantic interpretation rules in 1-1 correspondence with phrase structure rules (e.g., Thompson, 1963), writing "semantic grammars" that integrate syntactic and semantic constraints in a single grammar (e.g., Burton, 1976), or writing ad hoc programs that combine such information in unformalized ways. The first approach requires as many syntactic rules as semantic rules, and hence is not really much different from the se-

[1] There are some compensating disadvantages if the semantic domain is more complex than the syntactic one, but we will assume here that immediate semantic feedback is desired.

```
<ATN> -> (<machinename> (accepts <phrasetype>*) <statespec>*)
                ;an ATN is a list consisting of a machine name, a
                ;specification of the phrasetypes which it will
                ;accept, and a list of state specifications.
<statespec> -> (<statename> {optional <initialspec>} <arc>*)
<initialspec> -> (initial <phrasetype>*)  ;indicates that this state
                ;is an initial state for the indicated phrasetypes.
<arc> -> (<phrasetype> <nextstate> <act>*)  ;a transition that
                             ;consumes a phrase of indicated type.
      -> (<pattern> <nextstate> <act>*)  ;a transition that consumes
                             ;an input element that matches a pattern.
      -> (J <nextstate> <act>*)  ;a transition that jumps to a new
                             ;state without consuming any input.
      -> (POP <phrasetype> <form>)  ;indicates a final state
                        ;for the indicated phrase type and specifies
                        ;a form to be returned as its structure.
<nextstate> -> <statename>   ;specifies next state for a transition.
<pattern> -> ( <pattern>* )  ;matches a list whose elements match
                             ;the successive specified patterns.
          -> <wordlist>      ;matches any word in the list.
          -> &               ;matches any element.
          -> --              ;matches any subsequence.
          -> <form>          ;matches value of <form>.
          -> <<classname>>   ;matches anything that has or inherits
                             ;the class name as a feature.
<wordlist> -> {'<word> | '<word>, <wordlist>}
<act> -> (transmit <form>)   ;transmit value of form as an output.
      -> (setr <registername> <form>)  ;set register to value of form.
      -> (addr <registername> <form>)  ;add the value of form to the
                 ;end of the list in the indicated register (assumed
                 ;initially NIL when the register has not been set).
      -> (require <proposition>)  ;abort path if proposition is false.
      -> (dec <flaglist>)         ;set indicated flags.
      -> (req <flagproposition>)  ;abort path if proposition is false.
      -> (once <flag>)  ;equivalent to (req (not <flag>)) (dec <flag>).
<flagproposition> -> <boolean combination of flag registers>
<proposition> -> <form>     ;the proposition is false if the value
                            ;of the form is NIL.
<form> -> !<registername>   ;returns contents of the register.
       -> '<liststructure>  ;returns a copy of a list structure
                  ;except that any expressions preceded by ! are
                  ;replaced by their value and any preceded
                  ;by @ have their value inserted as a sublist.
       -> !c  ;contents of the current constituent register.
       -> !<liststructure>  ;returns value of list structure
                            ;interpreted as a functional expression.
```

Figure 1. BNF specification of ATN syntax.

mantic grammar approach (this is the conventional way of defining semantics of programming languages). The second approach has the tendency to miss generalities and its results do not automatically extend to new domains. It misses syntactic generalities, for example, by having to duplicate the syntactic information necessary to characterize the determiner structures of noun phrases for each of the different semantic kinds of noun phrase that can be accepted. Likewise, it tends to miss semantic generalizations by repeating the same semantic tests in various places in the grammar when a given semantic constituent can occur in various places in a sentence. The third approach, of course, may yield some level of operational system, but does not usually shed any light on how such interaction should be organized, and is difficult to extend.

Rusty Bobrow's RUS parser (Bobrow, 1978) is the first parser to my knowledge to make a clean separation between syntactic and semantic specification while gaining the benefit of early and incremental semantic filtering and maintaining the factoring advantages of an ATN. It can be characterized as a cascade of two ATN's - one doing syntactic analysis and one doing semantic interpretation. Such a cascade of ATN's provides a way to reduce having to say the same thing multiple times or in multiple places, while providing efficiency comparable to a "semantic" grammar and at the same time maintaining a clean separation between syntactic and semantic levels of description. It is essentially a mechanism for permitting decomposition of an ATN grammar into an assembly of cooperating ATN's, each with its own characteristic domain of responsibility.

As mentioned previously, a CATN is a sequence of ordinary ATN's that include among the actions on their arcs an operation TRANSMIT, which transmits an element to the next machine in the sequence. The first machine in the cascade takes its input from the input sequence, and subsequent machines take their input from the TRANSMIT commands of the previous ones. The output of the final machine in the cascade is the output of the machine as a whole. The only feedback from later stages to earlier ones is a filtering function that causes paths of the nondeterministic computation to die if a later stage cannot accept the output of an earlier one.

The conception of cascaded ATN's arose from observing the interaction between the lexical retrieval component and the "pragmatic" grammar of the HWIM speech understanding system (Woods et al., 1976). The lexical retrieval component made use of a network that consumed successive phonemes from the output of an acoustic phonetic recognizer and grouped them into words. Because of phonological effects across word boundaries, this network could consume several phonemes that were part of the transition into the next word before determining that a given word was possibly present. At certain points, it would return a found word together with a node in the network at which matching should begin to find the next word (essentially a state remembering how much of the next word has already been consumed due to the phonological word boundary effect). This can be viewed as an ATN that consumes phonemes and transmits words as soon as its has enough evidence that the word is there.

The lexical retrieval component of HWIM can thus be viewed as an ATN whose output drives another ATN. This led to the conception of a complete speech understanding system as a cascade of ATN's, one for acoustic phonetic recognition, one for lexical retrieval (word recognition), one for syntax, one for semantics, and one for subsequent discourse tracking. A predecessor of the RUS parser (Bobrow, 1978) was subsequently perceived to be an instance of a syntax/semantics cascade, since the semantic structures that it was obtaining from the lexicon to filter the paths through the grammar could be viewed as ATN's. Hence, practical solutions to problems of combinatorics in two different problem areas have independently motivated computation structures that can be viewed as cascaded ATN's. It remains to be seen how effectively cascades can be used to model acoustic phonetic recognition or to track discourse structure, but the possibilities are intriguing.

4.1 Specification of a CATN Computation

As with ordinary ATN's and other formal automata, the specification of the computation of a CATN will consist of the specification of an instantaneous "configuration" of the automaton and the specification of a transition function that computes possible successor configurations for any given configuration. Since CATN's are nondeterministic, a given configuration can in general have more than one successor configuration and may occasionally have no successor. One way to implement a parser for CATN's would be to explicitly mimic this formal specification by implementing the configurations as data structures and writing a program to implement the transition function. Just as for ordinary ATN's, however, there are also many other ways to organize a parser, with various efficiency tradeoffs.

A configuration of a CATN consists of a vector of state configurations of the successive machines, plus a pointer to the input string where the first machine is about to take input. The transition function (nondeterministic) operates as follows:

1. A distinguished register C is set (possibly nondeterministically) to the next input element to be consumed and the pointer in the input string is advanced. Then a stage counter k is set to 1.
2. The state of the kth machine in the sequence is used to determine a set of arcs that may consume the current input (possibly following a sequence of JUMPs, PUSHes, and POPs to reach a consuming transition).
3. Whenever a transmission operation TRANSMIT is executed by the stage k machine, the stage k+1 configuration is activated to process that input, and the stage k+1 component of the configuration vector is updated accordingly. If the k+1 stage cannot accept the transmitted structure, the configuration is aborted.

As for a conventional ATN, the format of the state configurations of the individual machines consist of a state name, a set of registers and contents, and a stack pointer (or its equivalent).[2] Each element of a stack is a pair consisting of a PUSH arc and a set of register contents. Transitions within a single stage are the same as for ordinary ATN's.

4.2 Uses of CATN's

A good illustrative example of the use of cascaded ATN's for natural language understanding would be a three stage machine consisting of a first stage that performs lexical analysis, a second stage for syntactic analysis, and a third stage for semantic analysis. The lexical stage ATN would consume letters from an input sequence and perform word identification, including inflectional analysis, tense extraction (e.g., BEEN => PASTPART BE), decomposition of contractions, and aggregation of compound phrases, producing as its output a sequence of words with syntactic categories and feature values. This machine could also perform certain standard bottom-up, locally determined parsings such as constructing noun phrase structures for proper nouns and pronouns. Ambiguity in syntactic class, in word grouping, and in homographs within a syntactic class can all be taken care of by the nondeterminism of this first stage machine (e.g., "saw" as a past tense of "see" vs present tense of "saw" can be treated by two different alternative outputs of the lexical stage).

This lexical stage machine is not likely to involve any recursion, unlike other stages of the cascade, but does use its registers to perform a certain amount of buffering before deciding what to transmit to the next stage. Because stages such as this one will reach states where they have essentially finished with a particular construction and are ready to begin a new one, a convenient action to have available on their arcs is one to reset all or a specified set of registers to their initial empty values again. Such register clearing is similar to that which happens on a push to a lower level, except that here the previous values need not be saved. The use of a register clearing action thus has the desired effect without the expense of a push.

The second stage machine in our example will perform the normal phrase grouping functions of a syntactic grammar and produce TRANSMIT commands when it has identified constituents that are serving specific syntactic roles. The third stage machine will consume such constituents and incorporate them into an incremental interpretation of the utterance (and may also produce differential likelihoods for alternative interpretations depending on the semantic and pragmatic consistency and plausibility of the partial interpretation).

The advantage of having a separate stage for the semantic interpretation, in addition to providing a clean separation between syntactic and semantic levels of description and a more domain-independent syntactic level, is that during the computation, different partial semantic interpretations that have the same initial syntactic structure share the same syntactic processing. In a single "semantic" ATN, such different semantic interpretation possibilities would have to make their own separate syntactic/semantic predictions with no sharing of the syntactic commonality between those predictions. Cascaded ATN'S avoid this while retaining the benefit of strong semantic constraint.

4.3 Benefits of CATN's

The decomposition of a natural language analyzer into a cascade of ATN's gains a "factoring" advantage similar to that which ATN's themselves provide with respect to ordinary phrase structure grammars. Specifically, the cascading allows alternative configurations in the later stages of the cascade to share common processing in the earlier stages that would otherwise have to be done independently. That is, if several semantic hypotheses can use a certain kind of constituent at a given place, there need be only one syntactic process to recognize it.[3]

Cascades also provide a simpler overall description of the acceptable input sequences than a single monolithic ATN combining all of the information into a single network would give. That is, if any semantic level process can use a certain kind of constituent at a given place, then there need be only one place in the syntactic stage ATN that will recognize it. Conversely, if there are several syntactic contexts in which a constituent filling a given semantic role can be found, there need be only one place in the semantic ATN to receive that role. (A single network covering the same facts would be expected to have a number of states on the order of the product, rather than the sum, of the numbers of states in the individual stages of the cascade.)

[2] For example, Earley's algorithm for context free grammars (Earley, 1968) replaces the stack pointer with a pointer to a place where the configuration(s) that caused the push can be found. A similar technique can be used with ATN grammars.

[3] One might ask at this point whether there are situations in which one cannot tell what is present locally without "top-down" guidance from later stages. In fact, any such later stage guidance can be implemented by semantic filtering of syntactic possibilities. For example, if there is a given semantic context that permits a constituent construction that is otherwise not legal, one can still put the recognition transitions for that construction into the syntactic ATN with an action on the first transition to check compatibility with later stage expectations (e.g., by transmitting a flag indicating that it is about to try to recognize this special construction).

An additional advantage provided by the factoring commonality introduced by the cascade is that the resulting localization of early stage activities in a single place provides a single place for a given linguistic fact to be learned, rather than independent versions of essentially the same fact having to be learned in different semantic contexts. Moreover, the separation of the stages of the cascade provides a decomposition of the overall problem into individually learnable skills. These facts may be significant not only for theories of human language development and use, but also for computer systems that can be easily debugged and can contribute to their own acquisition of improved language skill.

The above facts suggest that the traditional characterization of natural language in terms of phonemes, syllables, words, phrases, sentences, and higher level pragmatic constructs may be more deeply significant than just a convenience for scientific manipulation.

4.4 Parsing with CATN's

Conceptually, each ATN in a cascade produces (nondeterministically) a sequence of inputs for the next stage, which the next stage then parses. One could implement a computer parsing algorithm for a cascade in several ways. For example, the individual components of a configuration could be incremented as described above, with the later stages advanced as soon as the earlier stages transmit something. Alternatively, the later stages could wait until the earlier stages have completed a path through the input sequence before they begin to process the output of the earlier stages. The latter approach has the advantage of not performing second stage analysis on a path that will eventually fail at the first stage. On the other hand, it will result in the first stage occasionally continuing to extend partial paths that could already be rejected at the second stage.

In general, one can envisage an implementation in which the second stage can wait until the first stage has proceeded some distance past the current point before commencing its operations. This could either be done by having a fixed "lookahead" parameter which would always run the first stage some number of transmissions ahead of the second stage, or one could have a command that the first stage could execute when it considered its current path sufficiently likely to make it worthwhile for the second stage to operate on it. In fact, to handle both of these cases, one could simply have the first stage buffer its information in registers until it is ready for the next stage to work on it and only then perform the transmissions. For the remainder of this paper, I will assume that this is done and that the next stage begins to operate as soon as its input is transmitted.

As presented above, an instantaneous configuration of a CATN is essentially a vector of configurations for the individual stages of the cascade. Let us call the individual configurations IC's and the vector as a whole a configuration vector. Since any two configuration vectors having the same IC in some component will perform the same computation for that component and will only differ when they transmit to a subsequent stage, a parsing implementation should merge such common components and only perform their processing once. This can be achieved by representing the set of instantaneous configurations of the CATN not simply as a set of IC vectors, but as a tree structure (TC) that merges the common initial parts of those vectors. That is, each vector representing an instantaneous configuration of the CATN will be represented by a path through the TC from root to leaf, with the successive nodes in the path being the successive IC's of the vector. It is straightforward to transform the transition function that computes successor configuration vectors into a transition function that computes successor TC's from a given TC.

The TC representation has the characteristic that as long as the common left parts of configuration vectors are merged, the computation of a given IC at some level k will be done only once. To fully capitalize on the factoring advantages of this representation, one would like to assure that the common initial parts of alternative configuration vectors remain merged. This happens automatically for alternative stage k+1 computations that stem from a common stage k configuration. However, it is possible for two distinct k stage configurations, which have gone their separate ways and accumulated their own trees of higher level configurations, to come again to essentially the same k-stage configuration via different paths. This can happen especially with lexical stage computations when one word is recognized and the parsing of the next word begins. To provide maximum factoring, it is thus necessary to check for such cases and merge subtrees when the IC's at their heads are found to be equivalent.

When the k-stage network happens to be a finite state machine (i.e., makes no use of registers or recursion) the detection of a duplicate configuration is easy due to the simple equivalence test (i.e., sameness of state). When it is a general ATN, the detection of the conditions for merging are somewhat more involved (due to the register contents), and the likelihood of such merging being possible tends to be less. Hence for such stages the cost of checking for duplication may not be worth the benefit. Interestingly, it appears that the early stages of phonetic, lexical, and simple phrase recognition do have essentially finite state transition networks, while those of the later stages, where such sharing is not as important or as likely, is more apt to require non-finite-state register activities.

4.5 Comparison of Cascading with Recursion

Some interesting questions arise when considering the nature of cascaded ATN's as automata. For example, since a number of activities that are normally done with recursion in ATN's and other phrase structure grammars can be done by separate stages of a cascade, one is led to wonder about the relationship between cascading and recursion. That is, instead of arcs of an ATN pushing for a constituent of a certain kind, occasionally a cascade can be set up to find constituents of that kind and transmit them to a later stage of the cascade as units. A particular example, which has occasionally been proposed informally, would be for an early stage processor to group the input words into basic noun phrases, verb groups, etc. and for a later stage to take such units as input. Clearly this is a task normally performed by recursion. One might then wonder whether cascading was just another form of recursion, or somehow equivalent to it.

It turns out that cascading is in some respects weaker than recursion, and in other respects it is more powerful. In the next section, I will give an example of a context free cascade that can recognize a language that cannot be recognized by a single context free ATN. Hence, cascading clearly increases the power of a basic ATN beyond that provided by recursion alone. On the other hand, one is considerably more constrained in the way he can use cascading when writing a grammar than he is in the use of recursion. For example, indefinitely deep recursion can be used to recognize noun phrases inside prepositional phrases inside noun phrases, etc. When setting up a cascade of two ATN's to perform such grouping, the earlier cascade cannot model this directly, but instead would have to recognize "elementary" noun phrases consisting of, say, determiner, adjectives, and head noun, and would use looping transitions to accept subsequent prepositional phrases and relative clauses. Moreover, this stage of the cascade could not content itself solely with the noun phrases, but would also have to transmit the other elements of the sentence (auxiliaries, verbs, adverbs, particles, etc.) so that the later stages of the cascade will have a chance to see them. That is, a stage of a cascade provides a level of description of the entire input sequence in terms of a sequence of units to be transmitted to a later stage of analysis. Hence it appears that cascading is a fundamentally different operation that interacts with recursion and overlaps some of its functions in interesting ways.

Another interesting comparison arises between cascaded ATN's and the kinds of transformations used in a transformational grammar. If one attempts to use a transformational grammar by successively applying its transformations in reverse to the surface string, one repeatedly performs a partitioning of the input into a sequence of units as described above. That is, in applying a reverse transformation to a syntax tree in the course of a reverse transformational analysis, the operation of matching the pattern description of the transformation to the syntax tree amounts to finding a level at which the syntax tree can be "cut" yielding a sequence of units matching the sequence of elements in the pattern of the rule. This is exactly the kind of partitioning of the input into units that is done by a stage of a cascaded ATN. Moreover, the result of the transformation is expressed by a "right-hand-side" of the transformational rule, which may reorder the input sequence into a slightly modified sequence, and may copy an element several times, modify it in certain restricted ways, or even delete it (under suitable restrictions). In exactly the same way, a stage of a cascade can transmit the units that it has picked up in a different order than it found them, can duplicate a unit, drop a unit, insert a constant, and transmit units that are modified from the form in which they were recognized. In short, a stage of an ATN cascade can mirror the activity of any given transformational rule.

However, transformational rules are normally considered to apply in a cycle governed by the number of levels of embedding of clauses in the sentence, so that the number of successive transformations applied can be unbounded. By contrast, in an ATN cascade, there are only a finite number of stages in the cascade. Moreover, successive transformations in a transformational grammar are free to discard everything that was learned about the structure of the input in the matching of the previous transformation and there is no constraint that the manner in which a subsequent transformation analyzes the result of the previous transformation bear any relationship to the level of description imposed on the input by that previous transformation. In an ATN cascade, there is an assumed sequence of *progressive aggregation and higher level of description* implied by the transduction of information to successive stages of the cascade, with each stage perceiving the input in the terms that it was described by the previous. Thus, the ATN cascade seems to impose additional constraints on the process of language recognition that are not imposed by an ordinary transformational grammar.[4]

[4] These constraints tend to promote the efficiency of the processing. See Woods (1970) for a discussion of some of the inherent inefficiencies of an ordinary transformational analysis.

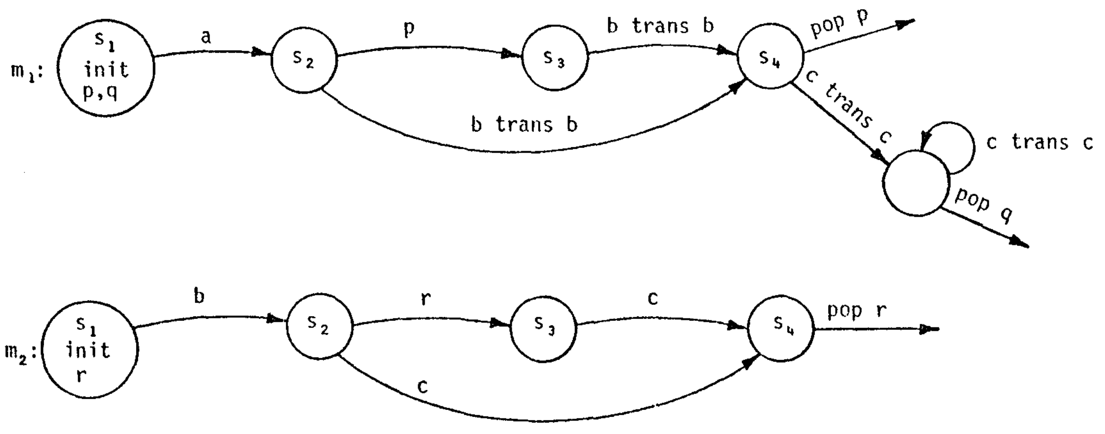

Figure 2. ATN cascade for $\{a^n b^n c^n: n \geq 1\}$

Experience with ATN grammars for natural language indicates that everything that a transformational grammar of natural language does can be done with even a single ATN, so there does not appear to be any need for more than a finite number of stages of a cascade. On the other hand, the arguments presented here indicate that one may be able to obtain a simpler description of an overall set of facts with a cascade than with a single monolithic ATN. It is possible, therefore, that a cascade of ATN's corresponds to a more appropriate formalization of the underlying facts of language that gave rise to the original model of transformational grammar than does the conventional conception.

4.6 A Simple Formal Example

As a simple example of what a cascade of ATN's can do, I will give here a simple ATN cascade that without the use of registers can recognize the set of strings of the form n a's followed by n b's followed by n c's, for arbitrary n. This language is a traditional example of a language that is not context free but is context sensitive. However, it does happen to be specifiable as the intersection of two context free languages. Capitalizing on this fact, it is possible to represent it by a cascade of two "context free" ATN's (i.e., ATN's which do not use registers to check constraints between different constituents). This cascade effectively computes the intersection of two ways of viewing the input. The two ATN's, whose structure is illustrated in Figure 2 above (where "trans" in the figure is short for "transmit"), can be written as follows:

```
(m1 (accepts q)
    (s1 (initial p q)
        ('a s2))
    (s2
        (p s3)
        ('b s4 (transmit 'b)))
    (s3
        ('b s4 (transmit 'b)))
    (s4 (pop p)
        ('c s5 (transmit 'c)))
    (s5 (pop q)
        ('c s5 (transmit 'c))))

(m2 (accepts r)
    (s1 (initial r)
        ('b s2))
    (s2
        (r s3)
        ('c s4))
    (s3
        ('c s4))
    (s4 (pop r)))
```

These two machines correspond to the grammars:

q-->pc*, p-->ab, p-->apb

and

r-->bc, r-->brc

with augmentation such that the b's and c's accepted by the first grammar are passed through to be accepted by the second. The first stage checks that the number of a's and b's agree and accepts any number of c's, while the second stage requires that the b's and c's agree.

4.7 Another Example - Syntax and Semantics

Another, less trivial example is the use of an ATN cascade to represent syntactic and semantic knowledge sources of a language understanding system. We will give here a brief example illustrating a kind of cascading of syntactic and semantic knowledge similar to that done by R. Bobrow in his RUS parser (Bobrow, 1978). A rough characterization of this parser is that as the syntactic component works its way through a noun phrase, it accumulates information about the determiner structure and initial premodifiers of the head noun until it encounters the head noun (i.e., takes a path corresponding to a hypothesis that it has found the head noun). At that point, it begins to transmit information to the semantic stage, starting with the head noun, and followed by the premodifiers of that noun. Then it continues to pick up post modifiers of the noun phrase, transmitting them to the semantic stage as it encounters them, and finally, when it hypothesizes that the noun phrase is completed, it transmits the determiner information.

In a similar way, in the parsing of a clause, the syntactic ATN can wait until it has encountered the main verb before transmitting that verb followed by its subject and any fronted adverbial modifiers. After that it can transmit subsequent post verbal elements as they are encountered, and finally transmit any governing modality information such as tense, aspect, and any governing negations.

The example presented here, is a constructed one to illustrate the principle, and does not directly represent the analyses by the RUS grammar. The example implements a subset of the semantic rules of the airline flight schedules system of Woods (1967), a predecessor of the LUNAR system (Woods et al.,1972). I will give here only a fragment of the semantic stage ATN that understands designators (i.e., noun phrases). It assumes that the syntactic stage operates as outlined above and, in particular, that it transmits prepositional phrases by transmitting the preposition and then transmitting its object. It also assumes that the syntax stage transmits a signal QUANT when it has hypothesized the end of a noun phrase and is about to transmit the determiner and number information. One could alternatively transmit prepositional phrases as single units to be tested for syntactic and semantic features. I will assume that a pattern such as <flight> on a consuming arc is matched by a constituent that receives the indicated semantic marker (e.g., FLIGHT).

```
(m2 (accepts designators)
 (d1 (initial designator)
  (J d2 (setr vbl (getnewvar)))
 (d2
  ('flight,'plane d/flight (setr head 'FLIGHT))
  ('jet d/flight (setr head 'FLIGHT)
         (addr mods '(JET !vbl)))
  ('airline d/head (setr head 'AIRLINE))
  ('city,'town d/head (setr head 'CITY))
  ('airport,'place d/head (setr head 'AIRPORT))
  ('time d/time)
  ('fare d/fare)
  ('owner,'operator d/owner))
 (d/owner
  ('of d/owner-of))
(d/owner-of
 (<flight> d/head (addr quants (getquant !c))
      (setr head '(OWNER !c))))
(d/fare
 ('(mod first-class),'(mod coach),
          '(mod stand by) d/fare
    (require (not class))
    (setr class !c))
 ('(mod one-way),'(mod round-trip) d/fare
    (require (not type))
    (setr type !c)))
 ('from d/fare-from (require (not from)))
 ('to d/fare-to (require (not to)))
 (J d/head (require class type from to)
    (setr head '(FARE !from !to !type !class)))
(d/fare-from
 (<place> d/fare (addr quants (getquant !c))
                (setr from !c)))
(d/fare-to
 (<place> d/fare (addr quants (getquant !c))
                (setr to !c)))
(d/time
 ('(mod departure) d/time (require (not op))
                (setr op 'DTIME))
 ('(mod arrival) d/time (require (not op))
                (setr op 'ATIME))
 ('of d/time-of (require (not flight)))
 ('in,'at d/time-prep (require (eq op 'ATIME)))
 ('from d/time-prep (require (eq op 'DTIME)))
 (J d/head (require op flight place)
                (setr head '(!op !flight !c))
       (* e.g., (setr head
                '(ATIME AA-57 CHICAGO)))))
(d/time-of
 (<flight> d/time (addr quants (getquant !c))
                (setr flight !c)))
(d/time-prep
 (<place> d/time (addr quants (getquant !c))
                (setr place !c)))
(d/head
 ('QUANT d/quant (setr mod !(packmods))))
(d/flight
 ('from d/flight-from (require (not from)))
 ('to d/flight-to (require (not to)))
 ('(mod first-class),'(mod coach),
        '(mod jet-coach) d/flight (once class)
          (addr mods '(SERVCLASS !vbl !c)))
 ('(mod jet) d/flight (addr mods '(JET !vbl)))
 ('(mod propeller) d/flight (once equip)
          (addr mods '(NOT (JET !vbl))))
 (J d/flight (once connect) (require from to)
   (addr mods '(CONNECT !vbl
                   !(sem from) !(sem to))))
 ('QUANT d/quant (setr mod !(packmods))))
```

```
(d/flight-from
 (<place> d/flight
         (addr quants (getquant !c))
         (setr from !c))
(d/flight-to
 (<place> d/flight
         (addr quants (getquant !c))
         (setr to !c)))
(d/quant
 ('some,'a,'any,'NIL d/some)
 ('each,'every d/each)
 ('all d/all)
 ('not d/not)
 ('the d/the)
 ('this,'that d/this)
 ('which,'what d/what)
 (<integer> d/integer))
(d/some
 ('sg,'pl d/end
    (setr quant '(FOR SOME !vbl /
             !head : !mod ; DLT))))
(d/each
 ('sg d/universal))
(d/all
 ('pl d/universal))
(d/universal
 (J d/end (setr quant '(FOR EVERY !vbl /
             !head : !mod ; DLT))))
(d/not
 ('some d/not-some)
 ('every d/not-every)
 ('all d/not-all))
(d/not-some
 ('sg,'pl d/end
    (setr quant '(NOT (FOR SOME !vbl /
             !head : !mod ; DLT)))))
(d/not-every
 ('sg d/not-universal))
(d/not-all
 ('pl d/not-universal))
(d/not-universal
 (J d/end
    (etr quant '(NOT (FOR EVERY !vbl /
             !head : !mod ; DLT)))))
(d/the
 ('sg d/end (setr quant '(FOR THE !vbl /
             !head : !mod ; DLT)))
 ('pl d/end (setr quant '(FOR EVERY !vbl /
             !head : !mod ; DLT))))
(d/this
 ('sg d/end (setr quant '(FOR THE !vbl /
             !head : !mod ; DLT))))
(d/what
 ('sg d/end (setr quant
    '(FOR THE !vbl / !head :
             (AND !mod DLT) ;
               (PRINTOUT !vbl))))
 ('pl d/end (setr quant
    '(FOR EVERY !vbl / !head :
             (AND !mod DLT) ;
               (PRINTOUT !vbl)))))
(d/integer
 ('sg,'pl d/end (setr quant
    '(FOR !integer MANY !vbl /
             !head : !mod ; DLT))))
(d/end
 (pop <designator> (sem-quant
             !quants !quant !vbl))))
```

In the above fragment grammar, the state d1 gets a variable name to use for the recognized designator, the state d2 dispatches on the head noun of the designator phrase to various states that recognize modifiers that are particular to the head. Eventually the path for each such head will lead to the state d/quant, where the determiner and number information is picked up to build the quantifier that governs this designator. This transition is triggered by the transmission of the flag QUANT from the syntax stage, signaling that the noun phrase is complete and the determiner information is coming. Notice how the quantification information that is common to most designators is shared.

The transitions that follow d/quant implement most of the d-rules in Woods (1967), which is itself a subset of the d-rules of the LUNAR system (Woods, et al,. 1972; Woods, 1978b). The function sem-quant is a function that performs the sem-quant pair manipulations described in Woods (1978b). These manipulations usually embed the quantifier just constructed (!quant) into the quantifier nest accumulated from below (!quants) to form a quantifier nest to be passed up to a higher clause. They then return the variable name (!vbl) as the "sem" to be inserted into an argument position in the higher structure. The function getquant, here, is a function that extracts the quant from a structure that has been passed up from below and is used to accumulate the quantifier nest (quants) from subordinate designators that should dominate the quantifier of the designator being interpreted. The function packmods examines the contents of the register mods and returns an AND of the mods if there are several, a single mod if there is only one, and T if there are none.

5. Conclusions

In Woods (1977, 1978a; Woods & Brachman, 1978), I discussed the general principle of hypothesis "factoring" - i.e., the coalescing of common parts of alternative hypotheses in such a way that an incremental hypothesis development and search algorithm does not need to individuate and consider separate hypotheses until sufficient information is present to make different predictions in the different cases. The most common example of factoring is the well-known device called "decision trees" in which a cascade of questions at nodes of a tree leads eventually to selection of a particular "leaf" of the tree without explicit comparison to each of the individual leaves. If the tree is balanced, then this leads to the selection of the desired individual leaf in log(n) tests rather than n tests, where n is the number of leaves of the tree. Another example of factoring is the mechanism in ATN grammars whereby common parts of different phrase structure rules are merged, thereby saving the redundant processing of common parts of alternative hypotheses.

One can think of an ATN as a generalization of the notion of decision tree to permit recursion, looping, register augmentation, and recombination of paths. In this paper, I have discussed a generalization of ATN's, called cascaded ATN's (CATN's), which provides additional factoring capabilities. A CATN consists of a sequence of ATN *transducers* the later stages of which take input from the output of the previous stage. ATN cascades permit a decomposition of complex language understanding behavior into a sequence of cooperating ATN's with separate domains of responsibility.

Of specific interest are two distinct notions of the concept of factoring that are beginning to emerge from such considerations. One, which I have called *hypothesis factoring*, provides a reduction through sharing in the number of distinct hypotheses that have to be explicitly considered during parsing. The other, which I will call *conceptual factoring*, provides a reduction through sharing in the number of times or places that a given fact or rule needs to be represented in a long-term conceptual structure (e.g., the grammar). The former promotes efficiency of "run-time" parsing, while the latter promotes efficiency of grammar maintenance and learning. In many cases conceptual factoring promotes hypothesis factoring, but this is not necessarily always the case.

References

Bobrow, R.J. (1978). "The RUS System", in B.L. Webber and R.J. Bobrow, Research in Natural Language Understanding, Quarterly Technical Progress Report No. 3. BBN Report No. 3878, Bolt Beranek and Newman Inc., Cambridge, MA. July.

Burton, R.R. (1976). "Semantic Grammar: An Engineering Technique for Constructing Natural Language Understanding Systems." BBN Report No. 3453, Bolt Beranek and Newman Inc., Cambridge, MA, December.

Earley, J. (1968). "An Efficient Context-free Parsing Algorithm." Ph.D. thesis, Dept. of Computer Science, Carnegie-Mellon University, Pittsburgh, PA.

Thompson, F.B. (1963). "The Semantic Interface in Man-Machine Communication," Report No. RM 63TMP-35, Tempo, General Electric Co., Santa Barbara, CA, September.

Woods, W.A. (1967). "Semantics for a Question-Answering System," Ph.D. thesis, Division of Engineering and Applied Physics, Harvard University. Also Report NSF-19, Harvard Computation Laboratory, September. (Available from NTIS as PB-176-548, and from Garland Publishing, Inc. as a volume in a new series: *Outstanding Dissertations in the Computer Sciences.*)

Woods, W.A. (1969). "Augmented Transition Networks for Natural Language Analysis," Report No. CS-1, Aiken Computation Laboratory, Harvard University, December. (Available from ERIC as ED-037-733; also from NTIS as Microfiche PB-203-527.)

Woods, W.A. (1970). "Transition Network Grammars for Natural Language Analysis," *CACM*, Vol. 13, No. 10, October.

Woods, W.A. (1977). "Spinoffs From Speech Understanding Research," in Panel Session on Speech Understanding and AI, Proceedings of the 5th Int. Joint Conference on Artificial Intelligence, August 22-25, p. 972.

Woods, W.A. (1978a). "Taxonomic Lattice Structures for Situation Recognition," in TINLAP-2, Conference on Theoretical Issues in Natural Language Processing-2, University of Illinois at Urbana-Champaign, July 25-27. (Also in AJCL, Mf. 78, 1978:3).

Woods, W.A. (1978b). "Semantics and Quantification in Natural Language Question Answering," in *Advances in Computers*, Vol. 17. New York: Academic Press. (Also Report No. 3687, Bolt Beranek and Newman Inc.)

Woods, W.A., R.M. Kaplan, and B.L. Nash-Webber (1972). "The Lunar Sciences Natural Language Information System: Final Report," BBN Report No. 2378, Bolt Beranek and Newman Inc., Cambridge, MA, June. (Available from NTIS as N72-28984.)

Woods, W.A., M. Bates, G. Brown, B. Bruce, C. Cook, J. Klovstad, J. Makhoul, B. Nash-Webber, R. Schwartz, J. Wolf, V. Zue (1976). Speech Understanding Systems - Final Report, 30 October 1974 to 29 October 1976, BBN Report No. 3438, Vols. I-V, Bolt Beranek and Newman Inc., Cambridge, MA.

Woods, W.A. and Brachman, R.J. (1978). Research in Natural Language Understanding, Quarterly Technical Progress Report No. 1, 1 September 1977 to 30 November 1977, BBN Report No. 3742, Bolt Beranek and Newman Inc., Cambridge, MA, January. (Now available from NTIS as AD No. AO53958).