DOCUMENT RESUME

ED 037 733 AL 002 367

AUTHOR	Woods, W. A.
TITLE	Augmented Transition Networks for Natural Language Analysis.
INSTITUTION	Harvard Univ., Cambridge, Mass. Computation Lab.
SPONS AGENCY	National Science Foundation, Washington, D.C.
PEPORT NO	CS-1
PUB DATE	Dec 69
NOTE	111p.
EDRS PRICE	EDRS Price MF-$0.50 HC-$5.65
DESCRIPTORS	*Algorithms, *Computational Linguistics, *Context Free Grammar, Grammar, *Mathematical Linguistics, Mathematical Models, *Semantics, Transformation Generative Grammar

ABSTRACT

The augmented transition network described in this report was developed in the course of work in semantic interpretation in the context of a computer system which answers English questions. In order to provide mechanical input for the semantic interpreter, a parsing program based on the notion of a "recursive transition network grammar" was developed. The form of presentation of rules made possible by this grammar is called a "recursive transition network," the augmented version of which is presented here. The parsing system has proved to be an extremely powerful system capable of performing the equivalent of transformational analysis in little more time than that customarily required for context free analysis alone. The system is also convenient for the designer of the grammar and facilitates experiments with various types of structural representations and various parsing strategies. This report, the first of several, presents a discussion of the augmented transition network as a grammar model, including a number of theoretical results concerning the efficiency of the model for parsing. (FWB)

THE AIKEN COMPUTATION LABORATORY
Harvard University

AUGMENTED TRANSITION NETWORKS
FOR
NATURAL LANGUAGE ANALYSIS

Report No. CS-1

to the

National Science Foundation

W.A. Woods

Principal Investigator

December 1969

Cambridge, Massachusetts

THE AIKEN COMPUTATION LABORATORY

Harvard University

AUGMENTED TRANSITION NETWORKS FOR NATURAL LANGUAGE ANALYSIS

Report No. CS-1

to

the National Science Foundation

W.A. Woods

Principal Invéstigator

Cambridge, Massachusetts

December 1969

PREFACE

One of the major objectives of NSF grant GS-2301 is the implementation of the semantic interpretation procedure described in Woods (1967) and the investigation of various semantic interpretation problems as they arise in the context of a computer system which answers English questions. Accordingly, one of the first tasks performed under the grant was the implementation of the semantic interpreter and the verification of its performance. Woods (1968) describes the basic semantic interpretation system as implemented and gives a number of examples of its interpretations. The implementation at that time consisted of the semantic interpretation procedure alone--the syntax trees which were input to the procedure were produced by hand, and there was no retrieval component implemented for answering the questions.

Late in the fall of 1968, in order to provide mechanical input for the semantic interpreter, I began constructing a parsing program based on the notion of a recursive transition network grammar, a model very much like a finite state transition graph except for the presence of non-terminal as well as terminal symbols as labels on the arcs. A non-terminal label causes a recursive application of the transition network to recognize a construction of the type indicated by the label before the transition so labeled is permitted. This model, which is weakly equivalent to a non-deterministic pushdown store automaton, occurred to me as a natural representation of the type of grammar that one would get if he carried the use of the Kleene * operator and bracketed alternatives in the right-hand sides of context free grammar rules (a notation used by many linguists) to its logical conclusion by permitting arbitrary regular expressions as the right-hand sides of rules. One could then merge all of the rules with a given non-terminal symbol as their left-hand side and could represent this rule either by its regular expression or alternatively by an equivalent finite state transition graph (over the total vocabulary of terminal and non-terminal symbols). It is this latter form of representation which I have called a recursive transition network. In the course of this implementation, I learned that a similar approach to natural language analysis had been used by Thorne, Bratley, and Dewar (1968) and by Bobrow and Fraser (1969). My approach is in effect a generalization and formalization of these earlier parsers and provides a number of additional capabilities.

In addition to many advantages for efficient context free recognition and improved strong generative power, the transition network model also provides a convenient means for incorporating syntactic and semantic conditions for guiding the parsing and for performing transformations and relocations of constituents. This is done by associating arbitrary conditions and structure building actions with the arcs of the network. This augmented network is a kind of "transducer", whose effects are to make changes in the contents of a set of registers associated with the network and whose transitions can be conditional on the contents of those registers. Registers can be used to hold pieces of syntactic structure whose position and function in the syntactic structure being built might not yet have been determined.

Experience with the parsing system has shown it to be an extremely powerful system--capable of performing the equivalent of transformational analysis in little more time than that customarily required for context free analysis alone. In addition, the system is convenient for the designer of the grammar and facilitates experiments with various types of structural representations and various parsing strategies. By the spring of 1969, an expanded version of the parser was in preparation and an early version was in operation with a basic transition network of about 30 states. During the summer of 1969, with the help of Mrs. Madeleine Bates, a graduate student who did much of the grammar development for the parser, the expanded version was completed and debugged and a number of experiments with various parsing strategies were carried out. A considerably more powerful grammar was also developed during this time, and a second semantic interpreter program, incorporating a number of improvements over the original interpreter was put into operation. Two other graduate students who worked with me during the summer developed programs for use in a retrieval component for use by the system. Mr. Benjamin Brosgol wrote a set of data base functions and semantic rules which enable the system to answer English questions about the transition network that drives the parser, and Miss Nancy Neigus wrote a resolution theorem prover to be used in the execution of "smart quantifiers" in the retrieval component.

This report presents a discussion of the augmented transition network as a grammar model, including a number of theoretical results concerning the efficiency of the model for parsing. A second report will describe the implemented transition network parser and some of the experiments which have been performed with the system. Research dealing with the semantic interpreter and the retrieval component is continuing and will be described in a later report.

W. A. Woods
December, 1969

CONTENTS

LIST OF FIGURES

SECTION 1

TRANSITION NETWORK MODELS

1.1 Motivation

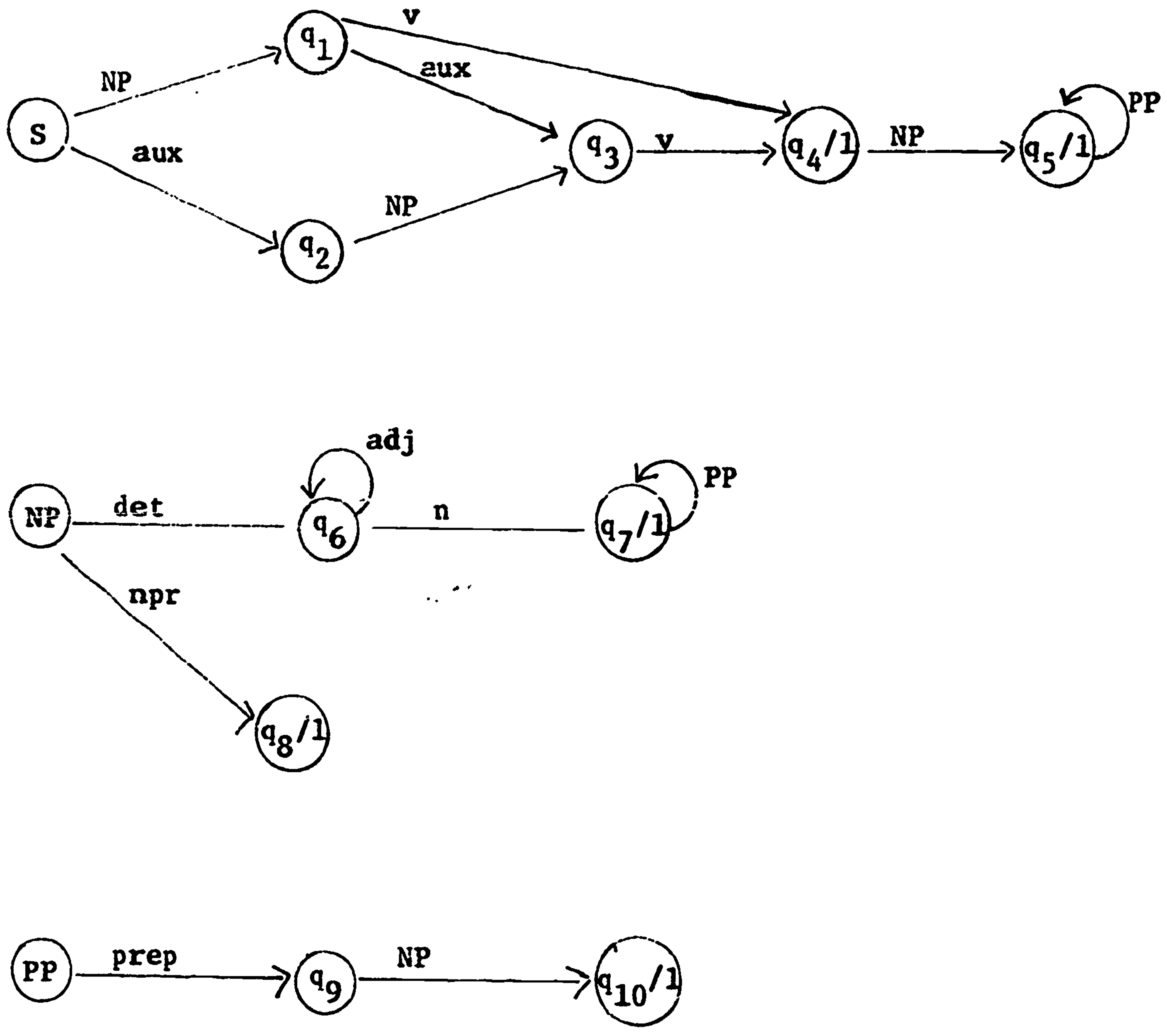
One of the early models for natural language grammars was the finite-state transition graph corresponding to a finite-state machine that accepted (or generated) the sentences of a language. In this model, the grammar is represented by a network of nodes and directed arcs connecting them. The nodes correspond to states in a finite state machine, and the arcs represent transitions from state to state. Each arc is labeled with a symbol whose input can cause a transition from the state at the tail of the arc to the state at its head. This model has the attractive feature that the sequences of words which make up a sentence can be read off directly by following the paths through the grammar from the initial state to some final state. Unfortunately, the model is grossly inadequate for the representation of natural language grammars because of its failure to capture many of the regularities of natural language grammars. The most notable of these is the pushdown mechanism that permits one to suspend the processing of a constituent at a given level while using the same mechanism to process an embedded constituent.

Suppose, however, that one added the mechanism of recursion directly to the transition graph model by fiat. That is, suppose that one took a collection of transition graphs each with a name, and permitted not only terminal symbols to be labels on the arcs but also non-terminal symbols naming constructions which must be present in order for the transition to be followed. The determination of whether such a construction was in fact precent in a sentence would be done by a "subroutine call" to another transition graph (or the same one). The resulting model of grammar, which we will call a recursive transition network, is equivalent in generative power to that of a context-free grammar or pushdown store automaton, but as we will show, allows for greater efficiency of expression, more efficient parsing algorithms, and natural extension by "augmentation" to more powerful models which allow various degrees of context dependence and more flexible structure-building during parsing. We will argue in fact that an "augmented" recursive transition network is capable of performing the equivalent of transformational recognition without the necessity of a separate inverse transformational component, and that this parsing can be done in an amount of time which is comparable to that of ordinary context free recognition.

1.2 Recursive transition networks

A recursive transition network is a directed graph with labelled states and arcs, a distinguished state called the start state, and a distinguished set of states called final states. It looks essentially like a non-deterministic finite state transition diagram except that the labels on the arcs may be state names as well as terminal symbols. The interpretation of an arc with a state name as its label is that the state at the end of the arc will be saved on a pushdown store and the control will jump (without advancing the input tape) to the state that is the arc label. When a final state is encountered then the pushdown store may be "popped" by transferring control to the state which is named on the top of the stack and removing that entry from the stack. An attempt to pop an empty stack when the last input character has just been processed is the criterion for acceptance of an input string. The state names that can appear on arcs in this model are essentially the names of contructions that may be found as "phrases" of the input tape. The effect of a state-labeled arc is that the transition that it represents may take place if a construction of the indicated type is found as a "phrase" of the input at the appropriate point in the input string.

Figure 1 gives an example of a recursive transition nework for a small subset of English. It accepts such sentences as "John washed the car," "Did the red barn collapse?", etc. It is easy to visualize the range of acceptable sentences from inspection of the transition network. To recognize the sentence, "Did the red barn collapse," the network is started in state S. The first transition is the aux transition to state q_2 permitted by the auxilliary "did". From state q_2 we see that we can get to state q_3 if the next "thing" in the input string is a NP. To ascertain if this is the case, we call the state NP. From state NP we can follow the arc labeled det to state q_6 because of the determiner "the". From here, the adjective "red" causes a loop which returns to state q_6, and the subsequent noun "barn" causes a transition to state q_7. Since state q_7 is a final state, it is possible to "pop up" from the NP computation and continue the computation of the top level S beginning in state q_3 which is at the end of the NP arc. From q_3 the verb "collapse" permits a transition to the state q_4, and since this state is final and "collapse" is the last word in the string, the string is accepted as a sentence.



S is the start state

q_4, q_5, q_7, q_8, and q_{10} are the final states

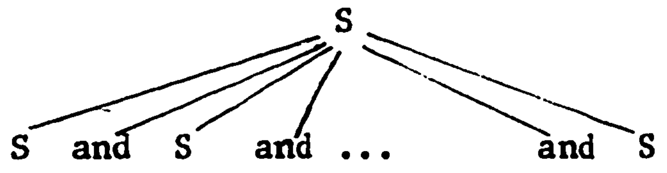
Figure 1: A sample transition network

The fact that the recursive transition network is equivalent to a pushdown store automaton is not difficult to establish. Every recursive transition network is essentially a pushdown store automaton whose stack vocabulary is a subset of its state set. The converse fact that every pushdown store automaton has an equivalent transition net could be established directly, but can be more simply established by noting that every pushdown store automaton has an equivalent context-free grammar which has an equivalent recursive transition net as we will show.

1.3 Augmented transition networks

It is well known (c.f. Chomsky,1964) that the strict context free grammar model is not an adequate mechanism for characterizing the subtleties of natural languages. Many of the conditions which must be satisfied by well-formed English sentences require some degree of agreement between different parts of the sentence which may or may not be adjacent (indeed which may be separated by a theoretically unbounded number of intervening words). Context sensitive grammars could take care of the weak generation of many of these constructions, but only at the cost of losing the linguistic significance of the "phrase structure" assigned by the grammar (c.f. Postal, 1964). Moreover, the unaided context free grammar model is unable to show the systematic relationship that exists between a declarative sentence and its corresponding question form, between an active sentence and its passive, etc. Chomsky's theory of transformational grammar (Chomsky, 1965), with its distinction between the surface structure of a sentence and its deep structure, answers these objections but falls victim of inadequacies of its own (c.f. Schwarcz, 1967, or McCawley, 1968). In this section we will describe a model of grammar based on the notion of a recursive transition network which is capable of performing the equivalent of transformational recognition without the need for a separate transformational component and which meets many of the objections that have been raised against the traditional model of transformational grammar.

The basic recursive transition network model as we have described it is weakly equivalent to the context-free grammar model and differs in strong equivalence only in its ability to characterize unbounded branching, as in structures of the form:



The major features which a transformational grammar adds to those of the context free grammar are the abilities to move fragments of the sentence structure around (so that their positions in the deep structure are different from those in the surface structure), to copy and delete fragments of sentence structure, and to make its actions on constituents generally dependent on the contexts in which those constituents occur. We can add equivalent facilities to the transition network model by adding to each arc of the transition network an arbitrary condition which must be satisfied in order for the arc to be followed, and a set of structure building actions to be executed if the arc is followed. We call this version of the model an augmented transition network.

The augmented transition network builds up a partial structural description of the sentence as it proceeds from state to state through the network. The pieces of this partial description are held in registers which can contain any rooted tree or list of rooted trees and which are automatically pushed down when a recursive application of the transition network is called for, and restored when the lower level (recursive) computation is completed. The structure-building actions on the arcs specify changes in the contents of these registers in terms of their previous contents, the contents of other registers, the current input symbol, and/or the results of lower level computations. In addition to holding pieces of substructure that will eventually be incorporated into a larger structure, the registers may also be used to hold flags or other indicators to be interrogated by conditions on the arcs.

Each final state of the augmented network has associated with it one or more conditions which must be satisfied in order for that state to cause a "pop"--i.e., to return from a lower level computation to the next higher one, or to complete the analysis when the end of the string is encountered. Paired with each of these conditions is a function which computes the value to be returned by the computation. A distinguished register, *, which contains the current input word when a word is being scanned, is set to the result of the lower level computation when the network returns to the arc which called for the recursive computation.

1.3.1 Representation of augmented networks

To make the discussion of augmented transition networks more concrete, we give in figure 2 a specification of a language in which an augmented transition network can be represented. The specification is given in the form of an extended context free grammar in which a vertical bar separates alternative ways of forming a construction and the Kleene star operator (*) is used as a superscript to indicate arbitrarily repeatable constituents. The non-terminal symbols of the grammar consist of English descriptions enclosed in angle brackets, and all other symbols except the vertical bar and the superscript * are terminal symbols (including the parentheses, which indicate list structure). The * which occurs as an alternative right-hand side for the rule for the construction <form>, however, is a terminal symbol and is not to be confused with the superscript *'s which indicate repeatable constituents. The first line of the figure says that a transition network is represented by a left parenthesis, follwed by an arc set, followed by any number of arc sets (zero or more), followed by a right parenthesis. An arc set in turn consists of a left parenthesis, followed by a state name, followed by any number of arcs, followed by a right parenthesis, and an arc can be any one of the four forms indicated in the third rule of the grammar. The remaining rules are interpreted in a similar fashion.

```
<transition network> → (<arc set> <arc set>*)

<arc set> → (<state> <arc>*)

<arc> → (CAT <category name> <test> <action>* <term act>)  |

        (PUSH <state> <test> <action>* <term act>)  |

        (TST <arbitrary label> <test> <action>* <term act>)  |

        (POP <form> <test>)

<action> → (SETR <register> <form>)  |

           (SENDR <register> <form>)  |

           (LIFTR <register> <form>)

<term act> → (TO <state>)  |

             (JUMP <state>)

<form> → (GETR <register>)  |

          *  |

         (GETF  <feature>)  |

         (BUILDQ <fragment> <register>*)  |

         (LIST <form>*)  |

         (APPEND <form> <form>)  |

         (QUOTE <arbitrary structure>)
```

Figure 2: Specification of a language for representing augmented transition networks.

The expressions generated as transition networks by the grammar of figure 2 are in the form of parenthesized list structures, where a list of the elements A, B, C, and D is represented by the expression (A B C D). The transition network is represented as a list of arc sets, each of which is itself a list whose first element is a state name and whose remaining elements are arcs leaving that state. The arcs also are represented as lists, possible forms of which are indicated in the figure. (The conditions and functions associated with final states are represented as "arcs" with no actions or terminal action.) The first element of each of these arcs is a word which names the type of the arc, and the third element is the arbitrary test which must be satisfied in order for the arc to be followed. The CAT arc is an arc which can be followed if the current input symbol is a member of the lexical category named in the list (and the test is satisfied), while the PUSH arc is an arc which causes a pushdown to the state indicated. The TST arc is an arc which permits an arbitrary test to determine whether an arc is to be followed. In all three of these arcs, the actions on the arc are the structure-building actions, and the terminal action specifies the state to which control is passed as a result of the transition. The two possible terminal actions, TO and JUMP, indicate whether the input pointer is to be advanced or not advanced, respectively--that is, whether the next state is to scan the next input word or whether it is to continue to scan the same word. The POP arc is a dummy arc which indicates under what conditions the state is to be considered ~~a final state, and the~~ form to be returned as the value of the computation if the POP alternative is chosen. (One advantage of representing this information as a dummy arc is the ability to order the choice of popping with respect to the other arcs which leave the state.)

The actions and the forms which occur in the network are represented in "Cambridge Polish" notation, a notation in which a function call is represented as a parenthesized list whose first element is the name of the function and whose remaining elements are its arguments. The three actions indicated in figure 2 cause the contents of the indicated register to be set equal to the value of the indicated form. SETR causes this to be done at the current level of computation in the network, while SENDR causes it to be done at the next lower level of embedding (used to send information down to a lower level computation) and LIFTR causes it to be done at the next higher level computation (used to return additional information to higher-level computations).

The forms as well as the conditions (tests) of the transition network may be arbitrary functions of the register contents, represented in some functional specification language such as LISP (McCarthy et al., 1962), a list processing programming language based on Church's lambda calculus and written in Cambridge Polish notation. The seven types of forms listed in the figure are a basic set which is sufficient to illustrate the major features of the augmented transition network model. GETR is a function whose value is the contents of the indicate register, * is a form whose value is usually the current input word, and GETF is a function which determines the value of the specified feature for the current input word. (In the actions which occur on a PUSH arc, * has the value of the lower level computation which permitted the PUSH transition.)

BUILDQ is a useful structure-building form which takes a list structure representing a fragment of a parse tree with specially marked nodes and returns as its value the result of resplacing those specially marked nodes with the contents of the indicated registers.† Specifically, for each occurrence of the symbol + in the list structure given as its first argument, BUILDQ substitutes the contents of one of the listed registers (the first listed register replacing the first + sign, the

† The BUILDQ function which is implemented in the experimental parsing system (to be described in a later report) is considerably more versatile than the version described here. Likewise, the implemented parser contains additional formats for arcs as well as other extensions to the language specified here. There has been no attempt to define a basic irredundant set of primitive conditions, actions, and forms, but rather an effort has been made to allow flexibility for adding "natural" primitives which facilitate the writing of compact grammars. For this reason, the set of possible conditions, actions, and forms has been left open-ended to allow for experimental determination of useful primitives. However, the arc formats and actions described here, together with arbitrary LISP expressions for conditions and forms, provides a model which is equivalent in power to a Turing machine and therefore complete in a theoretical sense.

second register the second +, etc.). In addition, BUILDQ replaces occurrences of the symbol * in the fragment with the value of the form *.

The remaining three forms are basic structure-building forms (out of which any BUILDQ can be duplicated) which respectively make a list of the values of the listed arguments, append two lists together to make a single list, and produce as value the (unevaluated) argument form. An illustrative fragment of an augmented transition network is given in figure 3. In the next section we will describe the operation of this network and discuss some of the features of the augmented transition network model.

1.3.2 An illustrative example

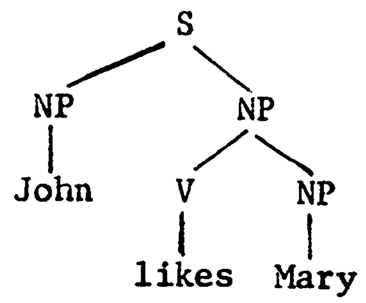
Figure 3 gives a fragment of an augmented transition network represented in the language of figure 2. This fragment is an augmentation of the portion of the transition network of figure 1 which consists of the states S/, Q1, Q2, Q3, Q4, and Q5. The augmented network builds a structural representation in which the first constituent of a sentence is a _type_ (either DCL or Q) which indicates whether the sentence is declarative or interrogative, the second constituent is the subject noun phrase, the third is an auxilliary (or NIL if there is no auxilliary), and the fourth is the verb phrase constituent. This representation is produced regardless of the order in which the subject noun phrase and the auxilliary occur in the sentence. The network also produces a representation of a verb phrase constituent even though there is no pushdown in the network corresponding to a verb phrase. It will be helpful, both for the understanding of the notation and for the understanding of the operation of the augmented network, to follow through an example at this point using the network fragment of figure 3.

Before proceeding to work an example it is necessary to explain the representation of the parse trees which is used by the network fragment. The parse trees are represented in a parenthesized notation in which the representation of a node consists of a list whose first element is the name of the node and whose remaining elements are the representations of the constituents of that node.

```
(S/  (PUSH  NP/  T
          (SETR SUBJ *)
          (SETR TYPE (QUOTE DCL))
          (TO Q1))
     (CAT  AUX  T
          (SETR AUX *)
          (SETR TYPE (QUOTE Q))
          (TO Q2)))
(Q1  (CAT  V  T
          (SETR AUX NIL)
          (SETR V *)
          (TO Q4))
     (CAT  AUX  T
          (SETR AUX *)
          (TO Q3)))
(Q2  (PUSH  NP/  T
          (SETR SUBJ *)
          (TO Q3)))
(Q3  (CAT  V  T
          (SETR V *)
          (TO Q4)))
(Q4  (POP (BUILDQ (S + + + (VP +)) TYPE SUBJ AUX V) T)
     (PUSH NP/  T
          (SETR VP (BUILDQ (VP (V +) *) V))
          (TO Q5)))
(Q5  (POP (BUILDQ (S + + + +) TYPE SUBJ AUX VP) T)
     (PUSH  PP/  T
          (SETR VP (APPEND (GETR VP) (LIST *)))
          (TO Q5)))
```

Figure 3: An illustrative fragment of an augmented transition network.

For example, the parse tree:



would be represented in this notation by the expression:

(S (NP John) (VP (V likes) (NP Mary))).

This representation can also be viewed as a labelled bracketing of the sentence in which a left bracket for a phrase of type X is represented by a left parenthesis followed by an X, and the matching right bracket is simply a right parenthesis.

Let us now consider the operation of the augmented network fragment of figure 3 for the input sentence "Does John like Mary?".

1. We begin the process in state S/ scanning the first word of the sentence, "does". Since this word is an auxilliary, its dictionary entry would mark it as a member of the category AUX and therefore (since its arbitrary condition T is the universally true condition) the arc (CAT AUX T ...) can be followed. (The other arc which pushed down to look for a noun phrase will not be successful.) In following this arc, we execute the actions: (SETR AUX *), which puts the current word "does" into a register named AUX, (SETR TYP (QUOTE Q)), which puts the symbol "Q" into a register named TYPE, and

(TO Q2), which causes the network to enter state Q2 scanning the next word of the sentence "John".

2. State Q2 has only one arc leaving it, which is a push to state NP/. The push will be successful and will return a representation of the structure of the noun phrase which will then become the value of the special register *. We will assume that the representation returned is the expression "(NP John)". Now, having recognized a construction of type NP, we proceed to perform the actions on the arc. The action (SETR SUBJ *) causes the value "(NP John)" to be placed in the register SUBJ, and the action (TO Q3) causes us to enter the state Q3 scanning the next word "like". The register contents at this point are:

TYPE	:	Q
AUX	:	does
SUBJ	:	(NP John).

3. From state Q3, the verb "like" allows a transition to state Q4, setting the contents of a register V to the value "like" in the process, and the input pointer is advanced to scan the word "Mary".

4. Q4, being a final state could choose to "POP", indicating that the string that has been processed so far is a complete sentence (according to the grammar of figure 1); however, since this is

not the end of the sentence, this alternative is not successful. However, the state also has an arc which pushes down to state NP/, and this alternative will succeed, returning the value "(NP Mary)". The action (SETR VP (BUILDQ (VP (V +) *) V)) will now take the structure fragment "(VP (V +) *)" and substitute the current value of * for the occurrence of * in the fragment and replace the occurrence of + with the contents of the indicated register V. The resulting structure, "(VP (V like) (NP Mary))" will be placed in the register VP, and the action (TO Q5) causes a transition to state Q5 scanning beyond the end of the input string. The register contents at this point are:

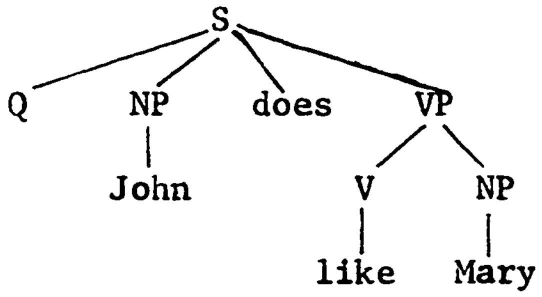
TYPE	:	Q
AUX	:	does
SUBJ	:	(NP John)
V	:	like
VP	:	(VP (V like) (NP Mary))

5. We are now scanning the end of the sentence, and since Q5 is a final state (i.e., it has a "POP" arc), and the condition T is satisfied, the sentence is accepted. The form "(BUILDQ (S + + + +) TYPE SUBJ AUX VP)" specifies the value to be returned as the analysis of the sentence. The value is obtained by substituting the contents of the registers TYPE,

SUBJ, AUX, and VP for the successive instances of the symbol "+" in the fragment "(S + + + +)" to give the final sentence analysis:

(S Q (NP John) does (VP (V like) (NP Mary)))

which represents the parse tree:



In ordinary context free recognition, the structural descriptions of sentences are more or less direct representations of the flow of control of the parser as it analyzes the sentence. The structural descriptions assigned by the structure building rules of an augmented transition network, as we can see from the example, are comparatively independent of the flow of control of the algorithm. This is not to say that they are not determined by the flow of control of the parser, for this they surely are; rather we mean to point out that they are not isomorphic to the flow of control as in the usual context free recognition algorithms. It is possible for a constituent that is found in the course of analysis to appear in the final structural description several times or not at all, and its location may be entirely different from that in which it was found in the surface structure.

In addition, the structural description assigned to a constituent at one point during the analysis may be changed or transformed before that structure is incorporated into the final structural description of the sentence as a whole. These facilities plus the ability to test arbitrary conditions allow the equivalent of a transformational deep structure to be constructed while the parser is performing transitions that are isomorphic to the surface structure of a sentence.

1.4 Transformational recognition

The usual model of transformational grammar is a generative model consisting of a context free (base) grammar and a set of transformational rules which map syntax trees into new (derived) syntax trees. The generation of a sentence with such a grammar consists of first constructing a deep structure using the base component grammar and then transforming this deep structure into a surface structure by successive applications of transformations. The terminal nodes (or leaves) of the surface structure tree give the final form of the sentence. This model of transformational grammar is totally oriented toward the generation of sentences rather than their analysis, and although there is clearly an algorithm for the use of such a grammar to analyze a sentence--namely the procedure of "analysis by synthesis" (Matthews, 1962)--this algorithm is so inefficient as to be out of the question for any practical application. (The analysis by synthesis method consists of applying the rules in the "forward" (generative) direction in all possible ways to generate all of the possible sentences of the language while looking to see if the sentence which you are trying to analyze turns up in the list.)

Two attempts to formulate more practical algorithms for transformational recognition (Petrick, 1965, and MITRE, 1964) resulted in algorithms which were still too time consuming to be practical for the analysis of more than a few test sentences with small sample grammars. Both of these algorithms attempt to analyze sentences by applying the transformations in reverse, a procedure which is far less straightforward than it sounds. The difficulty with simply performing the transformations in reverse is twofold. First, the transformations operate on tree structures and produce tree structures as their values. In the forward direction, they begin with the deep structure tree and end with the surface structure tree. In order to reverse this process, one needs first to obtain a surface structure tree for the input sentence. However, there is no component in the transformational model which characterizes the possible surface structures (their only characterization is implicit in the changes which can be made in the deep structures by means of the transformations). Both the MITRE and the Petrick analysis procedures solve this problem by constructing an "augmented grammar" which consists of the rules of the original base component grammar plus additional rules which characterize the structures which can be added by transformations.

In the MITRE procedure this "surface grammar" is constructed by hand and no formal procedure is available for constructing it from the original transformational grammar. In the Petrick procedure, there is a formal procedure for obtaining an augmented grammar but it will not necessarily terminate unless the length of the possible input sentences is first circumscribed (which unfortunately reduces the class of sentences that can be accepted to a finite set--theoretically analyzable by table lookup).

In the MITRE procedure, the augmented grammar is used to assign a complete "tentative" surface structure which is then subjected to inverse transformations. In the Petrick procedure inverse transformations are applied to partially built up surface structures and the processes of applying transformations and building structure are interwoven. In both systems, the inverse transformations may or may not produce a legitimate deep structure. If they do, then the sentence is accepted, but if they do not, then the tentative surface structure was spurious and is rejected. There is no way to construct a context free surface grammar which will assign all and only legitimate surface structures. One must settle for one which will assign all legitimate surface structures plus additional spurious ones. Moreover, the only way to tell the two apart is to perform the inverse transformations and check the resulting "tentative" deep structures.

The second difficulty in this method of analysis is the combinatorial explosion of the number of possible inverse transformation sequences that can be applied to a given surface structure tree. Although many of the transformations when applied in the forward direction are obligatory so that only one possible action can be taken, almost all of the inverse transformations are optional. The reason for this is that even though a given structure looks like it could have been produced by a given forward transformation so that the inverse transformation can be performed, there is no guarantee that the same structure could not have arisen in a transformational derivation in some other way. Therefore both the alternative of applying the inverse transformation and that of not applying it must both be tried whenever an inverse transformation can apply. The number of active paths can grow exponentially with the number of transformations applied. Moreover, the forward transformations usually don't specify much information about the structure which results from applying the transformation (even though the linguist may know a good deal about what the resulting structure must be like). For this reason the inverse transformations are not as selective as their forward counterparts and many more spurious applications of transformations are allowed. That is, whereas most forward sequences of transformations will lead to successful surface structures, most inverse sequences will not lead to legitimate deep structures, and a large amount of unnecessary wasted effort is therefore expended on dead end paths. To make matters worse, it is not always clear what the stopping conditions on the inverse transformational process should be. Some inverse transformational sequences could go on forever and it is not clear what set of conditions is sufficient to guarantee that a given sequence will not eventually lead to a legitimate deep structure. In short, the inverse transformational process is an extremely complicated one and is impractically inefficient to implement.

1.5 Augmented transition networks for transformational recognition

Kuno (1965) suggested that it should be possible to augment the surface structure grammar of a transformational grammar in such a way that it "remembered" the equivalent deep structure constructions and could build the deep structure of the sentence while doing the surface structure parsing, without the necessity of a separate inverse transformational component. The model which he proposed at that time, however, was not adequate to deal with some of the more powerful transformational mechanisms such as the extraposition of a constituent from an arbitrarily deep embedding. The augmented transition network, on the other hand, provides a model which is capable of doing everything that a transformational grammar can do and is therefore a realization of part of the Kuno prediction. It remains to be seen whether a completely mechanical procedure can be developed to take a transformational grammar in the usual formalism and translate it into an equivalent augmented transition network, but even if such a procedure is available, it may still be more appropriate to use the transition network model directly for the original linguistic research and grammar development. The reasons for this are several: First, the transition network that could be developed by a mechanical procedure from a traditional transformational grammar could not be expected to be as efficient as that which could be designed by hand. Moreover, the transition network model provides a mechanism which satisfies many of the objections which have been raised by linguistis against the transformational grammar as a linguistic model (such as its incompatibility with many psycholinguistic facts which we know to characterize human language performance).

A third reason for preferring the transition network model to the usual formulation of transformational grammar is the power which it contains in its arbitrary conditions and its structure building actions. The model is equivalent to a Turing machine in power and yet the actions which it performs are "natural" ones for the analysis of language. Most linguistic research in the structure of language and mechanisms of grammar has attempted deliberately to build models which do not have the power of a Turing machine but which make the strongest possible hypotheses about language mechanisms by proposing the least powerful mechanism that can do the job. As a result of this approach many variations of the transformational grammar model have been proposed with different basic repertories of transformational mechanisms. Some have cyclic transformation rules, others do not; Some have a distinct "post cycle" that operates in a different mode after all of the cyclic rules have been applied. There are various types of conditions that may be asked, some models have double structural descriptions, some have ordered rules, some have obligatory rules, some have blocking rules, etc. In short there is not a single transformational grammar model, there are myriad. Moreover these models are more or less incomparable. They do not fall within a single general framework so that their relative merits can be compared. If one such model can handle some features of language and another can handle different features, there is no systematic procedure for incorporating them both into a single model. In the augmented transition network model, the possibility exists to add to the model whatever facility is needed and seems natural to do the job. One can add a new mechanism by simply inventing a new basic predicate to use in conditions or a new function to use in the structure building rules. It is still possible to make strong hypotheses about the types of conditions and actions that are required, but when one finds that he needs to accomplish a given task for which his basic model has no "natural" mechanism, there is no problem extending the augmented transition network model to include it. This requires only the relaxation of the restrictions on the types of conditions and actions, and no reformulation of the basic model.

1.6 Previous transition network models

Two previous parsing systems based on a form of augmented transition network have been described in the literature. Thorne, Bratley, and Dewar (1968) describe a procedure for natural language analysis based on a "finite state transition network" (which is applied recursively), and Bobrow and Fraser (1969) describe a system which is "an elaboration of the procedure described by Thorne, Bratley, and Dewar." Although these systems bear considerable similarity to the one we have described, they differ from it in a number of important respects which we will describe shortly. Let us first however, briefly describe the two systems.

1.6.1 The Thorne system

The Thorne system assigns a representation of syntactic structure which attempts to simultaneously represent the deep structure and the surface structure of a sentence. Constructions are listed in the order in which they are found in the surface structure, with their deep structure functions indicated by labelling. Inversions in word order are indicated by marking the structures which are found "out of place" (i.e., in positions other than their deep structure positions) without moving them from their surface structure positions, and later in the string the position where they would have occurred in the deep structure is indicated by the appropriate deep structure function label followed by an asterisk. (They do not describe a procedure for constituents which are found in the surface structure to the right of their deep structure positions. Apparently their grammar does not deal with such constructions.)

Thorne views his grammar as a form of transformational grammar whose base component is a finite-state grammar and permits recursion to take place only via transformations. According to Thorne, the majority of transformation rules can be viewed as "meta rules" in the sense that "they operate on other rules to produce derived rules rather than operating on structural descriptions to produce new structural descriptions." He uses an augmented transition network containing both the original deep structure rules plus these derived rules as the grammar table to drive his parsing algorithm, but is not able to handle the word order inversion transformations and the conjunction transformations in this way. Instead, he implements these features as exceptions embedded in his parsing program.

1.6.2 The system of Bobrow and Fraser

Bobrow and Fraser (1969) describe a parsing system which is an elaboration of the Thorne parser. Like the Thorne parsings, the general form of their analysis "resembles the surface structure analysis of the sentence, with added indications of moved constituents and where they are located in deep structure." This grammar model is also a form of augmented transition networl . whose actions include setting flags and function labels and whose conditions include testing previously set flags. Unlike the Thorne system, however, Bobrow's system provides a facility for transferring information back to some previously analyzed constituent. In general the conditions on an arc can be arbitrary LISP functions (the system is programmed in LISP), and the actions for transferring information can be arbirary LISP functions. The conditions and actions actually implemented in the system, however, are limited to flag testing and to the transferring back into previously recognized structures new deep structure function labels.

According to Bobrow (personal communication) the major differences between his system and that of Thorne is the use of symbolic flag names (instead of bit positions), a facility for mnemonic state names, the ability to transfer information back to previously analyzed constituents, and a facility for active feature values in the dictionary (these are actually routines which are stored in the dictionary entry for the word rather than merely activated by features stored in the dictionary.)

1.6.3 Comparison with the present system

In comparing the augmented transition network system described in this paper with the systems of Bobrow and Fraser and of Thorne et al., there are two domains of comparison which must be distinguished--the formal description of the model and the implementation of the parsing system. One of the major differences between this parsing system and those of Bobrow and Thorne is the degree to which such a distinction is made. The Thorne paper does not describe the augmented transition network model which they use except to point out that the grammar table used by the parsing program "has the form of a finite-state network or directed graph--a form appropriate for the representation of a regular grammar." The transition network model is apparently formalized only in the form in which it actually occurs in the parsing program (which is not described). The conditions on the arcs seem to be limited to tests of agreement of features associated with lexical items and constituents, and the actions are limited to recording the current constituent in the output representation, labeling constituents, or inserting dummy nodes and markers. The mechanisms for word order inversion and conjunction are not represented in the network but are "incorporated into the program."

The Bobrow and Fraser paper improves considerably on the power of the basic transition network model used by Thorne et al. It adds the facility for arbitrary conditions and actions on the arcs thus increasing the power of the model to that of a Turing machine. In this system as in Thorne's, however, there is no distinction between the model and the implementation. Although the conditions and actions are arbitrary as far as the implementation is concerned, there is no separate formal model which characterizes the data structures on which they operate.

That is, in order to add such an arbitrary condition, one would have to know how the LISP implementation of the parsing algorithm works, and where and how its intermediate results are stored. The range of conditions and actions available without such information--i.e., the condition and action subroutines actually provided in the implementation--consist of setting and testing flags and transmitting function labels back into previously analyzed constituents. Both in Bobrow's system and in Thorne's the actual representation of constituent structure is isomorphic to the recursive structure of the analysis as determined by the history of recursive applications of the transition network, and it is produced automatically by the parsing algorithm.

The augmented transition network as we have defined it provides a formalized transition network model with the power of a Turing machine independent of the implementation. The model explicitly provides for the isolation of various partial results in named registers and allows arbitrary conditions and actions <u>which apply to the contents of these registers</u>. Thus it is not necessary for a grammar writer to know details of the actual implementation of the parsing algorithm in order to take advantage of the facility for arbitrary conditions and actions.[†]

[†] In the experimental parsing system there is sometimes an advantage to using conditions or actions which apply to features of the implementation that are not in the formal model. Actions of this sort are considered to be <u>extensions</u> to the basic model, and the features of the implementation which allow them to be added easily are largely features of the BBN LISP system (Bobrow, Murphy, and Teitelman, 1969) in which it is written.

The building of the constituent structure is not performed automatically by the parsing algorithm in this model, but must instead be specified by explicit structure building rules. The result of this feature is that the structures assigned to the sentence no longer need to be isomorphic to the recursion history of the analysis, but are free to move constituents around in the representation. Thus, the representation produced by the parser may be a true deep structure representation of the type assigned by the more customary transformational grammar models (or it could also be a surface structure representation, a dual purpose representation as in the Thorne and Bobrow systems, or any of a number of other representations such as deoendencv representations). The explicit structure-building actions on the arcs together with the use of registers to hold pieces of sentence structure (whose function and location may not yet have been determined) provides an extremely flexible and efficient facility for moving constituents around in their deep structure representations and changing the interpretation of constituents as the course of an analysis proceeds. It is even possible to build structures with several levels of nesting while remaining at a single level of the transition network, and conversely to go through several levels of recursion of the network while building a structure which has only one level. No facility like this is present in either the Thorne or the Bobrow systems.

Another feature of the augmented transition network parsing system presented here which distinguishes it from the Thorne and Bobrow systems is the effort that went into designing a language for the specification of the augmented transition network that would be convenient and natural to the grammar designer rather than to the machine or to a computer programmer. It is possible in a few pages to completely specify the possible syntactic forms for the representation of an augmented transition network to be input to this parsing system. Each arc is represented by a mnemonic name of the type of arc, the arc label, an arbitrary condition, and a list of actions to be executed if the arc is followed. The condition and actions are represented as expressions in Cambridge Polish notation with mnemonic function names, and care has been exercised to provide a basic repertoire of such functions which is "natural" to the task of natural language analysis. One of the goals of the experimental parsing system is to evolve such a set of natural operations through experience writing grammars for it, and many of the basic operations described in this paper are the result of such evolution. One of the unique characteristics of the augmented transition network model is the facility to allow for evolution of this type.

Other distinguishing features between this system and the systems of Bobrow and of Thorne lie in the method of implementation and the goals of the system. For example, Thorne was interested in characterizing certain psychological features of the ways in which humans parse sentences whereas this is not the major concern of the system of Bobrow and Fraser nor of the system described here. Both Bobrow and Thorne, however, were concerned with producing all of the analyses of a sentence without performing repetitive analyses of the same constituent for different overall analyses in which it appears, and both use a parallel parsing strategy to accomplish this. Neither of them perform any semantic analysis of the sentences. The present parser was designed to be used with a semantic interpreter in a system whose objective is to select the "most likely" syntactic analysis which "makes sense" to the semantic interpreter. In this system we may require the capability for enumerating all of the analyses of the sentence in some cases. However, if it is possible to select the "most likely" parsing which "makes sense" without exhaustively processing all of the ambiguous syntactic structures which could be assigned to the sentence, then this system attempts to do so. For this reason and other reasons having to do with the flexibility of the experimental system for the investigation of different basic operations, the experimental parsing system which we have implemented pursues the individual parsings independently rather than in parallel. In practice, the result is simply avoiding much of the processing which would have to be done otherwise in most of the sentences which are encountered. We show in a later section that the transition network model that we have presented is amenable to parsing by a modified form of the Early recognition algorithm--a parallel-type context free recognition algorithm which operates within a general time bound proportional to the cube of the length of the input string and is one of the most efficient context free parsing algorithms yet devised. The general n^3 time bound still applies to the augmented transition networks provided that the conditions and actions on the arcs take a bounded amount of time.

1.7 Advantages of the augmented transition network model

The augmented transition network model of grammar has many advantages as a model for natural language, some of which carry over to models of programming languages as well. In this section we will review and summarize some of the major features of the transition network model which make it an attractive model for natural language.

1.7.1 Perspicuity

Context free grammars have been immensely successful (or at least popular) as models for natural language in spite of formal inadequacies of the model for handling some of the features that occur in existing natural languages. They maintain a degree of "perspicuousness" since the constituents which make up a construction of a given type can be read off directly from the context free rule. That is, by looking at a rule of a context free grammar, the consequences of that rule for the types of constructions that are permitted are immediately apparent. The pushdown store automaton, on the other hand, although equivalent to the context free grammar in generative power does not maintain this perspicuousness. It is not surprising therefore that linguists in the process of constructing grammars for natural language have worked with the context free grammar formalism and not directly with pushdown store automata, even though the pushdown store automaton through its finite state control mechanism allows for some economies of representation and for greater efficiency in resulting parsing algorithms.

The theory of transformational grammar proposed by Chomsky is one of the most powerful tools for describing the sentences that are possible in a natural language and the relationships that hold among them, but this theory as it is currently formalized (to the limited extent to which it is formalized) loses the perspicuousness of the context free grammar. It is not possible in this model to look at a single rule and be immediately aware of its consequences for the types of construction that are possible. The effect of a given rule is intimately bound up with its interrelation to other rules, and in fragments of transformational grammars for real languages, it may require an extremely complex analysis to determine the effect and purpose of any given rule. The augmented transition network provides the power of a transformational grammar but maintains much of the perspicuousness of the context free grammar model. If the transition network model were implemented on a computer with a graphics facility for displaying the network, then it would be one of the most perspicuous (as well as powerful) grammar models available.

1.7.2 Generative power

Even without the conditions and actions on the arcs, the recursive transition network model has greater strong generative power than the ordinary context free grammar. This is due to its ability to characterize constructions which have an unbounded number of immediate constituents. Ordinary context free grammars cannot characterize trees with unbounded branching without assuming an infinite set or rules. (Another way of looking at the recursive transition network model is that it is a finite representation of a context free grammar with an infinite set of rules.) When the conditions and actions are added, the model attains the power of a Turing machine, although the basic operations which it performs are "natural" ones for language analysis. Using these conditions and actions, the model is capable of performing the equivalent of transformational analysis without the need for a separate transformational component.

Another attractive feature of the augmented transition network grammar model is the fact that one doesn't seem to have to sacrifice efficiency to obtain power. In the progression from context free grammars to context sensitive grammars, to transformational grammars, the time required for the corresponding recognition algorithms increases enormously. The transition network model, however, while achieving all of the power of a transformational grammar, does so without apparently requiring much more time than is required for ordinary context free recognition. (This will be illustrated to some extent by the example in section 1.7.6.)

An additional advantage of the augmented transition network model over the transformational grammar model is that it is much closer to a dual model than the transformational grammar. That is, although we have described it as a recognition or analysis model which analyzes sentences, there is no real restriction against running the model in a generative mode to produce or generate sentences. The only change in operation that would be required is that conditions which look ahead in the sentence would have to be interpreted in the generation algorithm as decisions to be made which if chosen will impose constraints on the generation of subsequent portions of the sentence. The transformational grammar model on the other hand is almost exclusively a generative model. The analysis problem for the transformational grammar is so extremely complicated that no completely satisfactory recognition algorithm for transformational grammar has yet been found. The only existing algorithms are prohibitively time consuming and expensive.

1.7.3 Efficiency of representation

A major advantage of the transition network model over the usual context free grammar model is the ability to merge the common parts of many context free rules thus allowing greater efficiency of representation. For example, the single regular-expression rule S → (Q) (NEG) NP VP replaces the four rules:

$S \rightarrow NP \quad VP$

$S \rightarrow Q \quad NP \quad VP$

$S \rightarrow NEG \quad NP \quad VP$

$S \rightarrow Q \quad NEG \quad NP \quad VP$

in the usual context free notation. The transition network model can frequently achieve even greater efficiency through merging because of the absence of the linearity constraints that are present in the regular expression notation.

The merging of redundant parts of rules not only permits a more compact representation but also eliminates the necessity of redundant processing when doing the parsing. That is, by reducing the size of the grammar representation, one also reduces the number of tests which need to be performed during the parsing. In effect, one is taking advantage of the fact that whether or not a rule is successful in the ordinary context free grammar model, information is frequently gained in the process of matching it (or attempting to match it) which has implications for the success or failure of later rules. Thus, when two rules have common parts, the matching of the first one has already performed some of the tests required for the matching of the second one. By merging the common parts, one is able to take advantage of this information to eliminate the redundant processing in the matching of the second rule.

In addition to the direct merging of common parts of different rules when constructing a transition network model, the augmented transition network through its use of flags allows for the merging of similar parts of the network by recording information in registers and interrogating it with conditions on the arcs. Thus it is possible to store in registers some of the information that would otherwise be implicitly remembered by the state of the network and to merge states whose transitions are similar except for conditions on the contents of registers. For example, consider two states whose transitions are alike except that one is "remembering" that a negative particle has already been found in the sentence, while the other permits a transition which will accept a negative particle. These two states can be merged by setting a flag to indicate the presence of a prior negative particle and placing a condition on the arc which accepts the negative particle. to block it if the negative flag is set.

The process of merging similar parts of the network through the use of flags, while producing a more compact representation, does not result in an improvement in processing time and usually requires slightly more time. The reason for this is the increased time required to test the conditions and the presence of additional arcs which must be processed even though the conditions will prevent them from being followed. In the absurd extreme, it is possible to reduce any transition network to a one-state network by using a flag for each arc and placing conditions on the arcs which forbid them to be followed unless one of the flags for a possible immediately preceeding arc has been set. The obvious inefficiency here is that at every step it would be necessary to consider each arc of the network and apply a complicated test to determine whether the arc can be followed. There is thus a trade off between the compactness of the representation which can be gained by the use of flags and the increase in processing time which may result. This seems to be just one more example of the ubiquitous space-time trade off that occurs for almost any computer programming problem.

In many cases, the use of registers to hold pieces of an analysis provide automatic flags, so that it is not necessary to set up special registers to remember such information. For example, the presence of a previous negative particle in a sentence can be indicated by the non-emptiness of a NEG register which contains the particle. Similarly the presence of an auxilliary verb is indicated by the non-emptiness of an AUX register which contains the auxilliary verb.

1.7.4 Capturing regularities

One of the linguistic goals of a grammar for a natural language is that the grammar capture the regularities of the language. That is, if there is a regular process that operates in a number of environments, then the grammar should embody that process in a single mechanism or rule, and not in a number of independent copies of the same process for each of the different contexts in which it occurs. A simple example of this principle is the representation of the prepositional phrase as a constituent of a sentence because the construction consisting of a preposition followed by a noun phrase occurs often in English sentences in many different environments. Thus the model which did not treat prepositional phrases as constituents would be failing to capture a generality. This principle is a variation of the economy principle, which says that the best grammar is that which can characterize the language in the fewest number of symbols. A grammar which made essentially independent copies of the same information would be wasting symbols in its description of the language, and that model which merged these multiple copies into a single one would be a better grammar because it used fewer symbols. Thus the economy principle tends to favor grammars which capture regularities.

The transition network model with the augmentation of arbitrary conditions on the arcs and the use of registers to contain flags and partial constructions provides a mechanism for recognizing and capturing regularities. Whenever the grammar contains two or more subgraphs of any size which are essentially copies of each other, then it is a symptom of a regularity that is being missed. That is, there are two essentially identical parts of the grammar which differ only in that the finite state control part of the machine is remembering some piece of information, but otherwise the operation of the two parts of the graph are identical. To capture this generality, it is sufficient to explicitly store the distinguishing piece of information in a register (e.g., by a flag) and use only a single copy of the subgraph.

1.7.5 Efficiency of operation

In addition to the efficiency of operation which results from the merging of common parts of different rules, the transition network model provides a number of other advantages for efficient operation. One of these is the ability to postpone decisions by reworking the network. A great inefficiency of many grammars for natural language is the procedure whereby the grammar "guesses" some basic feature of a construction too early in the process of recognizing it. For example, guessing whether a sentence is active or passive before the processing of the sentence has begun. This results in the parser having to follow several alternatives until that point in the sentence where enough information is present to rule out the erroneous guesses. A much more desirable approach is to leave the decision unmade until a point in the construction is reached where the necessary information is present to make the decision. The transition network model allows one to take this approach.

As we will show in section 2, the transition network model allows one to "optimize" the network by making it deterministic (except for recursion). If several arcs with the same label, leave a given state, then a modified network can be constructed which has at most one arc with a given label leaving any given state. This results in an improvement in operation efficiency because of the reduced number of active configurations which need to be followed during the parsing. The deterministic network keeps identical looking analyses merged until that point at which they are no longer identical, thus postponing the decision as to which path it is on until the first point where the two paths differ, at which point the input symbol usually determines the correct path. The augmented transition network may not permit the completely automatic optimization which the unaugmented model permits, but it is still possible to adopt the general approach of minimizing the number of active configurations by reducing the non-determinism of the network, thus postponing decisions until the point in the input string where they make a difference. The holding of pieces of the analysis in registers until their appropriate function is determined allows one to wait until such decisions have been made before building the syntactic representation, which may depend on the decision. This facility allows one to postpone decisions even when building deep structure representations of the type assigned by a transformational grammar.

The necessity of following several active configurations during parsing is a result of the potential ambiguity of natural language. The source of this ambiguity lies in the recursion operation of the network, since without recursion the network would be a finite state machine which can be made completely deterministic. As we will show in the next chapter, it is possible to eliminate much of the recursion from a transition network (in fact we can eliminate all of the recursion except for that induced by self embedding symbols), thus reducing still further the number of active configurations which need to be followed. In the augmented network model, one seems in practice to be able to use conditions on the arcs to determine uniquely when to push down for a recursion, leaving only the action of popping up as the source of ambiguity and the cause for multiple active configurations. The use of appropriate conditions (including semantic ones) on the POP arcs of the network allows one to reduce this ambiguity still further.

One of the most interesting features of the use of registers in the augmented transition network is the ability to make tentative decisions about the sentence structure and then change your mind later in the sentence without backtracking. For example, when one is at the point in parsing a sentence where he is expecting a verb and he encounters the verb "be", he can tentatively assign it as the main verb by putting it in the main verb register. If he then encounters a second verb indicating that the "be" was not the main verb but an auxilliary helping verb, then the verb "be" can be moved from the main verb register into an auxilliary verb register and the new main verb put in its place. This technique, like the others, tends to reduce the number of active configurations which need to be followed during the parsing. In the next section we give an example which provides a number of illustrations of this technique of making tentative decisions and then changing them.

1.7.6 A second example

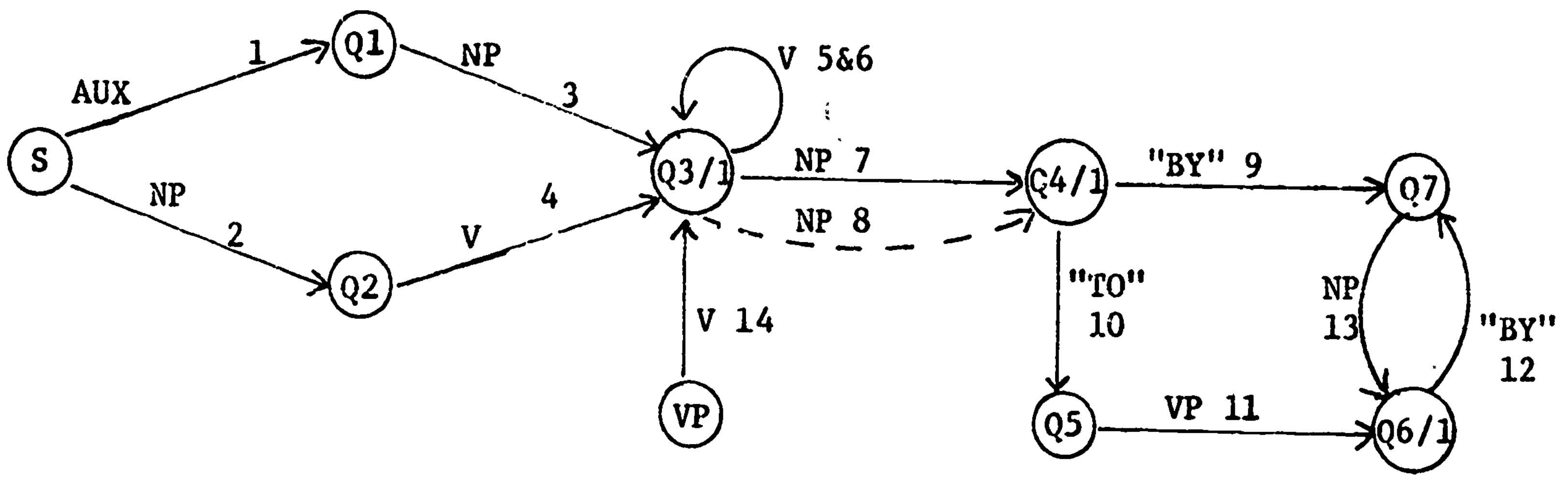
In this section we give an example that illustrates some of the advantages of the augmented transition network which we have been discussing--especially the facilities for making tentative decisions that are changed as the parsing proceeds. Figure 4 gives a fragment of a transition network which characterizes the behavior of the auxilliary verbs "be" and "have" in indicating the passive construction and the perfect tense. We will consider the analysis provided by this sample network for the sentence "John was believed to have been shot," a sentence with a fairly complex syntactic structure. In doing so, we will see that the augmented transition network clearly characterizes the changing expectations as it proceeds through the analysis, and that it does do without the necessity of backtracking or pursuing different alternatives.

Figure 4 is divided into three parts--a pictorial representation of the network with numbered arcs, a description of the conditions and forms associated with the final states, and a list of the conditions and actions associated with the arcs of the network. In the pictorial representation, S, NP, and VP are non-terminal symbols, AUX and V are lexical category names, and the arcs labeled "TO" and "RY" are to be followed only if the input word is "to" or "by" respectively. The dotted arc with label NP is a special kind of "virtual" arc which can be followed if a noun phrase has been placed on a special "hold list" by a previous HOLD command. It removes the item from the hold list when it uses it. The hold list is a feature which provides a natural facility for dealing with constituents which are found out of place and which must be inserted in their proper location before the analysis can be complete. The items placed on the hold list are marked with the level at which they were placed on the list, and the algorithm is prevented from popping up from that level until the item has been "used" by a virtual transition at that level or some other level.

Final states are represented in the pictorial representation by the diagonal slash and the subscript 1, a notation which is common in the representation of finite state automata. The conditions necessary for popping up from a final state and the expression which determines the value to be returned are indicated in part (b) of the figure. The parenthesized representation of tree structure is the same as that used in section 1.3.2. Conditions TRANS and INTRANS test whether a verb is transitive or intransitive, respectively, and the condition S-TRANS tests for verbs like "believe" and "want", which can take an embedded nominalized sentence as their "object". Features PPRT and UNTENSED mark respectively the past participial form and the standard untensed form of a verb.

dotted arc with label NP is a special kind of "virtual" arc which can be followed if a noun phrase has been placed on a special "hold list" by a previous HOLD command. It removes the item from the hold list when it uses it. The hold list is a feature which provides a natural facility for dealing with constituents which are found out of place and which must be inserted in their proper location before the analysis can be complete. The items placed on the hold list are marked with the level at which they were placed on the list, and the algorithm is prevented from popping up from that level until the item has been "used" by a virtual transition at that level or some other level.

Final states are represented in the pictorial representation by the diagonal slash and the subscript 1, a notation which is common in the representation of finite state automata. The conditions necessary for popping up from a final state and the expression which determines the value to be returned are indicated in part (b) of the figure. The parenthesized representation of tree structure is the same as that used in section 1.3.2. Conditions TRANS and INTRANS test whether a verb is transitive or intransitive, respectively, and the condition S-TRANS tests for verbs like "believe" and "want", which can take an embedded nominalized sentence as their "object". Features PPRT and UNTENSED mark respectively the past participial form and the standard untensed form of a verb.



(a) Pictorial representation with numbered arcs.

Q3:

Condition: (INTRANS (GETR V))

Form:

```
(BUILDQ (S + + (TNS +)(VP (V +))) TYPE SUBJ TNS V)
```

Q4 and Q6

Condition: T

Form:

```
(BUILDQ (S + + (TNS +)(VP (V +) +)) TYPE SUBJ TNS V OBJ)
```

(b) Conditions and forms for final states

Figure 4: A partial transition network

(continued on next page)

	Conditions	Actions
1.	T	(SETR V *) (SETR TNS (GETF TENSE)) (SETR TYPE (QUOTE Q))
2.	T	(SETR SUBJ *) (SETR TYPE (QUOTE DCL))
3.	T	(SETR SUBJ *)
4.	T	(SETR V *) (SETR TNS (GETF TENSE))
5.	(AND (GETF PPRT) (EQ (GETR V) (QUOTE BE)))	(HOLD (GETR SUBJ)) (SETR SUBJ (BUILDQ (NP (PRO SOMEONE)))) (SETR AGFLAG T) (SETR V *)
6.	(AND (GETF PPRT) (EQ (GETR V) (QUOTE HAVE)))	(SETR TNS (APPEND (GETR TNS) (QUOTE PERFECT))) (SETR V *)
7.	(TRANS (GETR V))	(SETR OBJ *)
8.	(TRANS (GETR V))	(SETR OBJ *)
9.	(GETR AGFLAG)	(SETR AGFLAG NIL)
10.	(S-TRANS (GETR V))	(SENDR SUBJ (GETR OBJ)) (SENDR TNS (GETR TNS)) (SENDR TYPE (QUOTE DCL))
11.	T	(SETR OBJ *)
12.	(GETR AGFLAG)	(SETR AGFLAG NIL)
13.	T	(SETR SUBJ *)
14.	(GETF UNTENSED)	(SETR V *)

(c) Conditions and actions on arcs.

Figure 4: A partial transition network (concluded)

We begin the analysis of the sentence, "John was believed to have been shot," in state S, scanning the first word of the sentence, "John." Since "John" is a proper noun, the pushdown for a noun phrase on arc 2 will be successful, and the actions for that arc will be executed placing the noun phrase (NP (NPR JOHN)) in the subject register SUBJ and recording the fact that the sentence is declarative by placing DCL in the TYPE register. The second word of the sentence, "was", allows the transition of arc 4 to be followed, setting the verb register V to the standard form of the verb "BE" and recording the tense of the sentence in the register TNS. The register contents at this point correspond to the tentative decision that "be" is the main verb of the sentence, and a subsequent noun phrase or adjective (not shown in the sample network) would continue this decision unchanged.

In state Q3, the input of the past participle "believed" tells us that the sentence is in the passive and that the verb "be" is merely an auxilliary verb indicating the passive. Specifically, arc 5 is followed because the input word is a past participle form of a verb and the current content of the verb register is the verb "be". This arc revises the tentative decisions by holding the old tentative subject on the special hold list, setting up a new tentative subject (the indefinite someone), and setting the flag AGFLAG which indicates that a subsequent agent introduced by the preposition "by" may specify the subject. The main verb is now changed from "be" to "believe" and the network returns to state Q3 scanning the word "to". The register contents at this point are:

SUBJ	(NP (PRO SOMEONE))
TYPE	DCL
V	BELIEVE
TNS	PAST
AGFLAG	T

and the noun phrase (NP (NPR JOHN)) is being held on the hold list.

None of the arcs leaving state Q3 are satisfied by the input word "to". However, the presence of the noun phrase "John" on the hold list allows the virtual transition of arc 8 to take place just as if this noun phrase had been found at this point in the sentence. (The transition is permitted because the verb "believe" is marked as being transitive.) The effect is to tentatively assign the noun phrase (NP (NPR JOHN)) as the object of the verb believe. If this were the end of the sentence and we chose to pop up from the resulting state Q4, then we would have the correct analysis "someone believed John."

The input of the word "to" to state Q4 tells us that the "object" of the verb "believe" is not merely the noun phrase "John", but is a nominalized sentence with "John" as its tentative subject. The effect of arcs 10 and 11 is to send down the necessary information to an embedded calculation which will complete the embedded clause and return the result as the object of the verb "believe". Arc 10 prepares to send down the noun phrase (NP (NPR JOHN)) as the embedded subject, the tense PAST, and the type DCL. Arc 11 then pushes down to state VP scanning the word "have".

At this point, we find ourselves in an embedded computation with the register contents:

SUBJ	(NP (NPR JOHN))
TYPE	DCL
TNS	PAST

The arc 14 permits a transition if the current input is a verb in its standard untensed, undeclined form (i.e., one cannot say: "John was believed to has been shot"). Since "have" is such a form, the transition is permitted and the main verb of the embedded sentence is tentatively set to "have" as would befit the sentence "John was believed to have money."

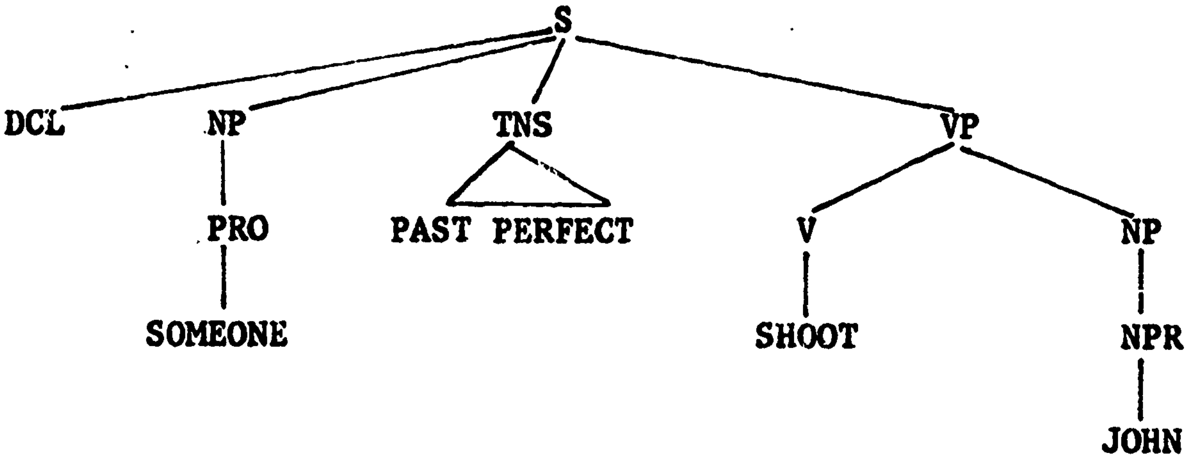
The subsequent past participle "been" following the verb "have" causes transition 6, which detects the fact that the embedded sentence is in the perfect tense (the effect of the auxilliary "have") and adopts the new tentative verb "be" as would befit the sentence, "John was believed to have been a druggist." The register contents for the embedded computation at this point are:

SUBJ	(NP (NPR JOHN))
TYPE	DCL
TNS	PAST PERFECT
V	BE

Once again in state Q3, the input of the past participle "shot" with a tentative verb "be" in the verb register indicates that the sentence is in the passive, and transition 5 puts the noun phrase (NP (NPR JOHN)) on the hold list and sets up the indefinite subject (NP (PRO SOMEONE)). Although we are now at the end of the sentence, both the presence of the noun phrase on the hold list and the fact that the verb "shoot" is transitive prevent the algorithm from popping up. Instead, the virtual transition of arc 8 is followed, assigning the noun phrase "John" as the object of the verb "shoot". The register contents for the embedded computation at this point are:

SUBJ	(NP (PRO SOMEONE))
TYPE	DCL
TNS	PAST PERFECT
V	SHOOT
AGFLAG	T
OBJ	(NP (NPR JOHN))

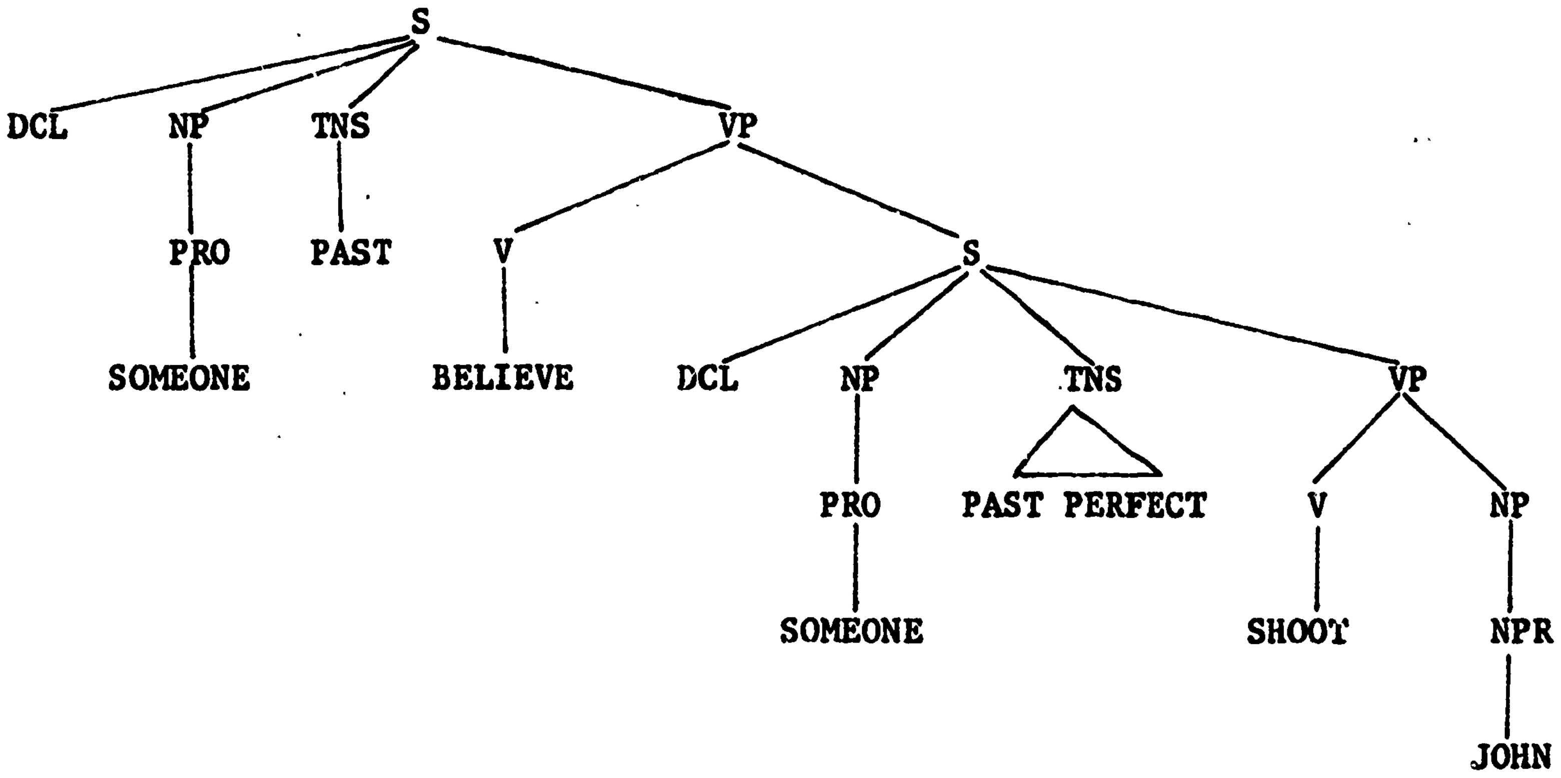
At this point, we are at the end of the sentence in the final state Q4, with an empty hold list so that the embedded computation can return control to the higher level computation which called it. The value returned, as specified by the form associated with the state Q4, is (S DCL (NP (PRO SOMEONE))(TNS PAST PERFECT)(VP (V SHOOT)(NP (NPR JOHN)))) corresponding to the tree:



The higher level computation continues with the actions on arc 11, setting the OBJ register to the result of the embedded computation. Since the higher level computation is also in a final state, Q6, the sentence is accepted and the structure assigned to it (as specified by the form associated with state Q6) is:

(S DCL (NP (PRO SOMEONE))(TNS PAST)(VP (V BELIEVE)(S DCL (NP (PRO SOMEONE))(TNS PAST PERFECT)(VP (V SHOOT)(NP (NPR JOHN))))))

which in tree form is represented as:



This structure can be paraphrased "Someone believed that someone had shot John." If the sentence had been followed by the phrase, "by Harry," there would have been two possible interpretations depending on whether the additional phrase were accepted by the embedded computation or the top level computation. Either case would have resulted in replacing one of the indefinite subjects SOMEONE with the definite subject "Harry." The structure produced in one case would be paraphrased "Someone believed that Harry had shot John," while the other would be "Harry believed that someone had shot John."

1.7.7 Flexibility for experimentation

Perhaps one of the most important advantages of the augmented transition network model is the flexibility that the model provides for experimental linguistic research. The open ended set of basic operations which can be used on the arcs allows for the development of a fundamental set of "natural" operations (for natural language analysis) through experience obtained while writing grammars. The powerful BUILDQ function was developed in this way and has proven extremely useful in practice. The use of the hold list and the virtual transitions are another example of the evolution of a special "natural" operation to meet a need.

A second area of experimentation that is facilitated by the transition network model is the investigation of different types of structural representations. The explicit structure building actions on the arcs of the network allow one to experiment with representations such as dependency grammars, tagmemic representations, which explicitly label the functions of the constituents as well as their types, and various combinations of these. It should even be possible to produce some types of semantic representation by means of the structure building actions on the arcs.

Finally, it is possible to use the conditions on the arcs to experiment with various types of semantic conditions for guiding the parsing and reducing the number of "meaningless" syntactic analyses that are produced. Within the framework of the augmented transition network one can try to take advantage of much of the information which human beings seem to have available during parsing. Many good ideas in this area have gone untried for want of a formalism which could accomodate them.

The experimental parsing system which has been implemented on the time sharing system at Harvard has been constructed in a modular fashion which lends itself to evolution and extension without major changes to the overall system structure. Much of this flexibility is due to the convenience of the LISP programming language in which it is implemented. The system has already undergone several cycles of evolution and a number of new features have been developed in this way, many of which together with the experiments which have been performed with the system are described in a subsequent report. In the next two sections, we will give detailed proofs and constructions for the optimization of the basic transition network model and for its recognition by a modified form of the Early context free recognition algorithm.

SECTION 2

REGULAR EXPRESSION GRAMMARS AND GRAMMAR OPTIMIZATION

Although variations of context-free grammars which allow the right-hand sides of rules to contain optional, repeatable, and alternative constituents have been used for some time both by linguistis and by designers of artifical programming languages[†], many of the potential advantages of this form of grammar have not been exploited. We will show in this section that a formalization of such grammars (called regular expression grammars) is closely related to the recursive transition network model of grammar and permits one to "optimize" a context free grammar to allow for more efficient parsing. This is done by "factoring out" the part of the grammar which is essentially finite state (or "regular") from that part which inherently requires the use of the pushdown store, thus permitting finite state optimization techniques to apply wherever possible.

2.1 Introduction

The context-free grammars contain as a subclass the class of finite-state grammars, which unlike the class as a whole permit transformations that take a given grammar into an equivalent grammar that is "more efficient" in a specialized sense. Using well-known techniques for finite state machine optimizatio , it is possible to construct an unambiguous (deterministic) parsing algorithm which will recognize an input string of length n in n steps, and it is possible to algorithmically construct such a machine with

[†] See for example, MITRE (1964) and Cheatham (1964).

the minimal number of states. Finally it is possible to use techniques for machine decomposition to obtain a componential representation of these states which simplifies the logic of the parsing algorithm. Since a number of results (e.g., the equivalence between context-free grammars and pushdown store automata) suggest that a context free grammar consists of a "finite state part" plus some more powerful mechanism, it would be desirable if there were a way to "factor out" the finite state part of a context free grammar in such a way that the above techniques could be applied to the finite state part of the grammar to yield an improved parsing algorithm. We will present here one method for realizing such a factoring, and show that the recursive transition network is the "natural" parsing algorithm to take advantage of it.

2.2 Regular expression grammars

Define a regular expression grammar to be a quadruple (V_N, V_T, S, P) where V_N is a vocabulary of non-terminal symbols, V_T is a vocabulary of terminal symbols, S is a distinguished symbol in V_N called the initial symbol, and P is a set of productions of the form:

$$X \rightarrow R$$

where X is a symbol in V_N and R is a regular expression[†] over $V = V_T \cup V_N$. The interpretation of a rule $X \rightarrow R$ is that the symbol X in a derivation may be replaced by any string of symbols in the regular set denoted by R. That is, we say that a string ϕ directly produces a string ψ (written $\phi \rightarrow \psi$) if $\phi = w_1 X w_2$, $\psi = w_1 \gamma w_2$, $X \rightarrow R$ is a rule in P, and γ is a string in the regular set denoted by R (written $\gamma \varepsilon R$).

A regular expression grammar is clearly equivalent in weak generative power to an ordinary context free grammar, since every regular set has a finite state grammar and consequently the rule $X \rightarrow R$ could be replaced by the set of rules of the finite state grammar for the set R (with X as the initial symbol) to give an equivalent context free grammar. (The converse is immediate since any context free grammar is a special case of a

[†] A regular expression over a vocabulary V can be defined recursively as follows:

(1) If x is a string in V^* then x is a regular expression denoting the set $\{x\}$

(2) If x is a regular expression over V denoting the set X, then x^* (or $(x)^*$ if parentheses are required for grouping) is a regular expression denoting the set X^*, the set of all concatenations of instances of strings in X.

(3) If x and y are regular expressions over V denoting the sets X and Y, respectively, then xy or $x \cdot y$ (the concatenation of x and y) is a regular expression denoting the set $XY = \{uv\colon\ u \varepsilon X \text{ and } v \varepsilon V\}$. Also $(x + y)$ is a regular expression denoting $X \cup Y = \{u\colon\ u \varepsilon X \text{ or } u \varepsilon Y\}$.

regular expression grammar.) The major difference between the ordinary context free grammar model and the regular expression grammar lies in the strong generative power and the ability to construct an equivalent "reduced" regular expression grammar which factors out the finite state part of the grammar from the "essentially recursive part". The regular expression grammar also allows an efficiency of expression over that of the ordinary context-free grammar in that common parts of different rules can be combined thus eliminating both redundant symbols in the representation and also redundant processing during the parsing. By constructing an equivalent reduced grammar, one can also reduce the number of non-terminal symbols in the grammar and obtain a parsing program that minimizes the use of recursion and makes use of the advantages of finite state parsing wherever it can.

2.3 Recognition automata for regular expression grammars

It is well known that the "natural" recognition automaton for parsing context free grammars is the pushdown store automaton. That is, the class of languages that can be accepted by non-deterministic pushdown store automata is the same as the class of languages that can be generated by context free grammars. However, although the construction which gives a pushdown store automaton that is equivalent to a given context free grammar is quite straightforward, the inverse problem is considerably more difficult. The reason for this is that the usual construction that takes a context-free grammar into a pushdown store automaton[†] results in a one state pushdown store automaton--i.e. does not take advantage of the finite state control of the pushdown store automaton. The inverse problem is complicated because there is no analog of the finite state control mechanism in the ordinary context free grammar model, and hence the usual construction involves first constructing a pushdown store automaton which does not use any of its finite state control but carries all of its information in the pushdown store. The regular expression grammar, however, has an analog of the finite state control mechanism in the regular expressions in the right-hand sides of the rules, and this permits a more natural correspondence between the regular expression grammar and the pushdown store automaton. In fact, we will argue that the type of pushdown store automaton which is "most natural" for the recognition or regular expression grammars is the recursive transition network

The equivalence between finite state automata (as represented for example by finite state transition graphs) and regular expressions is also well known (See for example, McNaughton and Yamada, 1960, Ott and Feinstein, 1961). Book et al. (1969) present a construction which shows that it is possible to preserve ambiguity of representation under these constructions (and that consequently every regular expression has an equivalent unambiguous regular expression). We can make use of these results to construct a recursive

[†] See Ginsburg (1966) for a presentation of the contructions which take context-free grammars into equivalent pushdown store automata and vice versa.

transition network equivalent to a given regular expression grammar and vice versa as follows:

1. To construct a transition network equivalent to a given regular expression grammar, first transform the grammar by grouping all of the rules according to the symbol on the left-hand side and replacing each group by a single rule whose right-hand side is the union (+) of the right-hand sides of all of the rules in the group. We now have one regular expression R_X for each of the non-terminal symbols of the grammar. Now for each non-terminal symbol X, construct the finite state transition graph equivalent to the regular expression R_X and name the start state of this graph X. The collection of transition graphs that result will have both terminal and non-terminal labels, and taken as a whole they will constitute a recursive transition network equivalent to the original regular expression grammar.

2. To construct a regular expression grammar equivalent to a given recursive transition network, construct a rule $X \rightarrow R_X$ for each non-terminal symbol X, where R_X is the regular expression equivalent to the portion of the transition network accessible from the state X (viewed as a finite state transition graph over the terminal and non terminal vocabulary ignoring its interpretation as a recursive transition network). The resulting set of rules constitutes a regular expression grammar equivalent to the original recursive transition network.

We see then that the correspondence between the regular expression grammars and recursive transition networks is a direct extension of the equivalence between regular expressions and finite state automata, and that this correspondence is much more "natural" than the usual correspondence between context free grammars and pushdown store automata. In fact, as we will show, the correspondence is so close that the finite state factoring transformations that can be performed on the regular expression grammar are preserved when transformed into an equivalent recursive transition network. That is the recursion (pushdown) operation of the recursive transtition net corresponds exactly to the rewriting operation of the regular expression grammar, and the finite state control of the network corresponds exactly to the regular expression in the right-hand sides of the rules.

2.4 Reduced regular expression grammars

The theorem that a context free language is essentially context free (i.e., not regular) if and only if all of its context free grammars have self embedding symbols (c.f. Chomsky, 1963) clearly suggests that the only part of a context free grammar that is not finite state is the self embedding of symbols. Since the recursion operation of the recursive transition network and the rewrite operation of the regular expression grammar are the non-finite state parts of these two models of grammar, one may ask whether it is possible to "optimize" the grammar so that these operations apply only to the self embedding of symbols. We will show that such is indeed the case and that it can be done algorithmically. That is, we will describe a "reduction algorithm" which will reduce any regular expression grammar to an equivalent one in which the only non-terminal symbols other than the initial symbol are self embedding symbols. The implication of this result for the design of parsing algorithms is that it is possible to optimize any context free grammar so that the rewrite operation is confined exclusively to self embedding symbols, and all other parts of the grammar may be optimized by finite state techniques. If we take the regular expressions in a reduced regular expression grammar and write minimal finite state machines for recognizing the strings which they denote (representing these machines in the form of state transition diagrams), then the resulting graph is a recursive transtion network which recognizes the strings of the original regular expression grammar.

2.5 The reduction algorithm

We will give here a series of constructions which establishes the following theorem:

> For every regular expression grammar $G=(V_N, V_T, S, P)$, there is an equivalent regular expression grammar $G' = (V'_N, V_T, S, P')$, where V'_N is a subset of V_N consisting only of self-embedding symbols plus the initial symbol S. We call such a grammar reduced.

Proof:

First, it is clearly possible to obtain a regular expression grammar equivalent to G which has only one rule for each non-terminal symbol in V_N--e.g. by replacing all of the rules:

$$Z \rightarrow R_1,\ Z \rightarrow R_2,\ \ldots\ ,\ Z \rightarrow R_n$$

for a particular non-terminal Z by the single rule $Z \rightarrow R_Z$, where $R_Z = (R_1 + R_2 + \ldots + R_n)$. Assume that G is in such a form. We can now begin to construct a reduced grammar equivalent to G as follows:

1. Pick a non-terminal symbol Z other than S which does not occur in the right-hand side of the rule $Z \rightarrow R_Z$. If there are no such symbols, then halt.

2. Replace every occurrence of Z in all of the other rules of the grammar with the regular expression R_Z (this replacement preserves regular expressions).

3. Delete the rule $Z \rightarrow R_Z$ and delete the symbol Z from the non-terminal vocabulary.

4. Repeat steps 1 through 3 until there are no more non-terminal symbols (except possibly S) which do not occur in the right-hand sides of their rules. (The algorithm will converge because each interaction eliminates one symbol from V_N and there are only finitely many to start with.)

We now have a grammar in which the only non-terminal symbols other than S are recursive symbols, and we now proceed to give constructions for eliminating those which are not self-embedding--i.e., the left- and right-recursive symbols.

2.5.1 Elimination of left and right recursion

Let $L(Z) = \{X \varepsilon V_N\colon\ Xw \varepsilon R_Z$ for some $w \varepsilon V^*\}$. Then $L(Z)$ is the set of non-terminal symbols which can be accepted as the first symbol of a string in R_Z. Let $L^*(Z)$ be the "closure" of $L(Z)$ in the sense that $L^*(Z)$ is the smallest subset of V_N such that $L(Z) \subseteq L^*(Z)$ and $X \varepsilon L^*(Z) \Rightarrow L(X) \subseteq L^*(Z)$ (i.e., $L^*(Z)$ is the closure of $\{Z\}$ under the operation L).

Similarly, let $R(Z) = \{X \varepsilon V_N\colon\ wX \varepsilon R_Z$ for some $w \varepsilon V^*\}$, let $S(Z) = \{X \varepsilon V_N\colon\ w_1Xw_2 \varepsilon R_Z$ for some $w_1,\ w_2 \varepsilon VV^*\}$, and let $R^*(Z)$ and $S^*(Z)$ be the closures of $R(Z)$ and $S(Z)$, respectively. Then

Z is left recursive if $Z \varepsilon L^*(Z)$,

Z is right recursive if $Z \varepsilon R^*(Z)$,

and Z is self-embedding if $Z \varepsilon S^*(Z)$.

A symbol X can be left (right) recursive for one of two reasons--either it is in $L(X)$ ($R(X)$) (i.e., it is a permissible initial (final) symbol in the right-hand side of the rule $X \rightarrow R_X$) or it is in $L(Y)$ ($R(Y)$) for some Y in $L^*(X)$ ($R^*(X)$). We will show in the next two sections that it is possible to eliminate the first type of left and right recursion (which we will call <u>direct</u> left and right recursion) by constructing an equivalent rule that has no direct left (or right) recursion but which accepts the same set of terminal strings when used in conjunction with the rest of the grammar. In this section we will assume these results and show how to eliminate the second type of left and right recursion. We will describe the algorithm for left recursion only, since the algorithm for right recursion will be exactly analogous.

The existence of left recursion of the second type is due to the existence of <u>left recursion chains</u> $Y_1, Y_2, \ldots, Y_n$, where $Y_1 \varepsilon L(X)$, $Y_2 \varepsilon L(Y_1), \ldots, Y_n \varepsilon L(Y_{n-1})$, and $X \varepsilon L(Y_n)$. We will call such a chain <u>simple</u> if none of the intermediate Y_i's are X's and none of them are repeated. The algorithm for eliminating left recursion from the grammar will consist of the successive shortening of all of the simple left recursion chains by substituting the expression R_{Y_1} for Y_1 in the right-hand side of the rule $X \rightarrow R_X$ until there are no more left recursive chains (and hence no more left recursion of the second type).

The argument is more difficult than it might seem to be at first because of the flexibility of the regular expression grammar. It is possible that a single rule may have several non-terminal symbols as initial symbols, and in particular it is possible to have two initial symbols--one identical to the left-hand side of the rule and the other involved in a left recursion chain that goes through the first. (For example a rule $Y \rightarrow Yc + Xd + f$ when there exists another rule $X \rightarrow Ya + b$.) Consequently, a simple repeated expansion of the initial symbols of a rule which are involved in a left recursion chain may not terminate. (In the above example the repeated expansion of Y in the second rule would never end.) It is necessary therefore to first eliminate direct left recursion from all of the rules before expanding. Each expansion, however, may reintroduce direct left recursion (indeed this is the reason for doing it) and therefore it is necessary to repeat the algorithm for eliminating direct left recursion before each expansion.

It remains only to show that the repeated alternation of expansion and elimination of direct recursion will converge in a finite number of cycles. We can assure ourselves that it does by noting that each cycle of expansion reduces the length of each simple left recursion chain by 1. This follows directly from our method of expansion--we replace each symbol Y in L(X) with the set of sumbols L(Y) (after first making sure that Y is not a member of L(Y) by eliminating direct left recursion) thus permitting only those left recursion chains in the new grammar which can be obtained from left recursion chains in the old one by the deletion of the first element of the chain. Since the longest simple recursion chain orginally can be no longer than the number of non-terminal symbols of the grammar and since each cycle of expansion and elimination of left recursive symbols reduces this length by 1, the algorithm will converge in a finite number of steps to a grammar in which there is no left recursion of the second type. A final application of the direct left recursion elimination algorithm removes all left recursion from the grammar.

2.5.2 Elimination of direct left recursion

To eliminate direct left recursion from the grammar, we construct for each left-recursive symbol X a new rule $X \rightarrow \overline{R}_X$ which does not permit X as an initial symbol, but which produces the same set of terminal strings as the original rule when used in conjunction with the rest of the grammar. We do this by first constructing a finite state transition graph D_X equivalent to R_X and then transforming it as follows:

1. Let X be the start state of the graph, and let an arc from state x to state z with label y be represented by the triple [x, y, z]. If there is more than one arc with label X leaving the start state X, construct an equivalent graph D'_X in which there is only one such arc as follows: Let Q be the set of all states that are accessible from the start state via a single arc labeled X. Add a new state q', and for every arc [x, y, z] leaving a state x in Q, add a new arc [q', y, z] leaving state q'. Now delete all of the arcs [X, X, z] for z in Q and add the single arc [X, X, q']. Finally, delete any states in Q which now have no arcs entering them. The resulting graph D'_X has a single arc leaving state X with label X, namely the arc [X, X, q'] and it is equivalent to the original graph D_X.

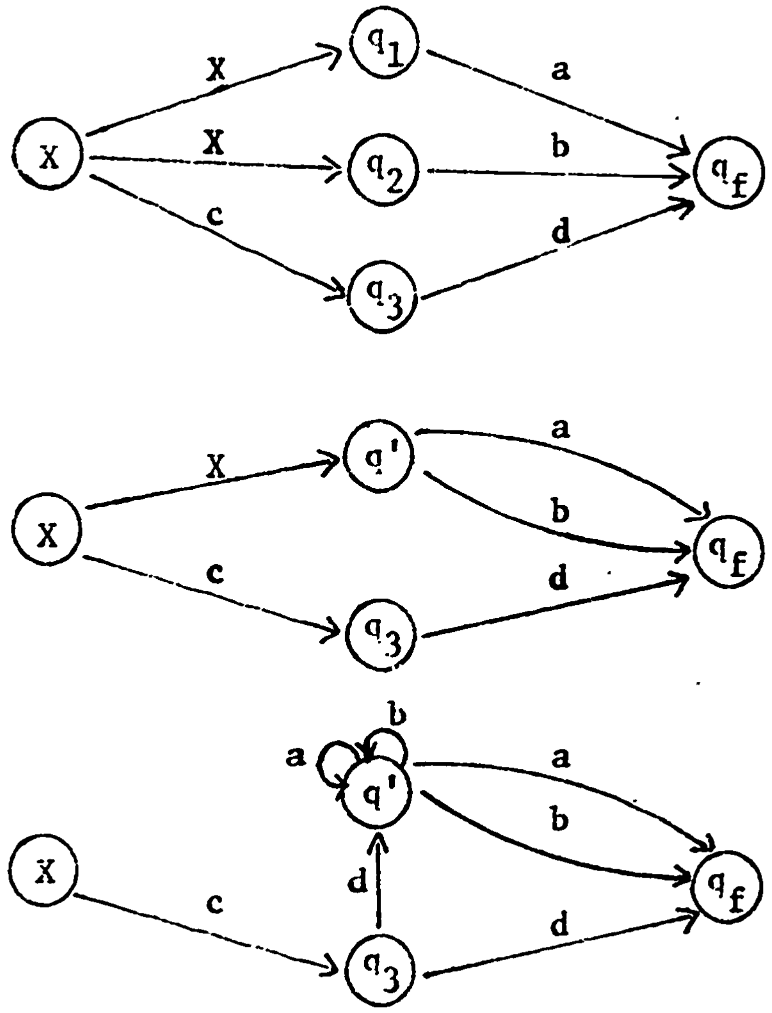
2. Eliminate the direct left recursion of the symbol X by deleting the arc [X, X, q'], and for every arc [x, y, z] which enters a final state z, adding a new arc [x, y, q'] (this is equivalent to adding an e-transition from z to q'). The resulting graph $\overline{D}_X$ will accept the same set of terminal strings as the original when used in conjunction with the rest of the grammar, but without direct left recursion of the symbol X.

3. Construct the regular expression $\overline{R}_X$ from the transiton graph $\overline{D}_X$.

We can show that the grammar which results from the above algorithm will accept the same set of terminal strings as the original grammar by a straightforward recursion on the depth of the parse trees. By construction of the new rule $X \rightarrow \overline{R}_X$, all of the rewritings of the original grammar are permitted by the new grammar except those which rewrite the symbol X as a string beginning with the symbol X. Therefore, any parse tree making use only of these rewritings will still be accepted by the new grammar. Let w be any string recognizable as a construction of any type Y in the original grammar, and let T be any parse tree for the string w analyzed as a Y. If the depth of this tree is 1, then it must result from a single rewriting the right-hand side of which is the terminal string w (which cannot start with the non-terminal X), and hence the same rewriting is possible in the new grammar. Now suppose it is true for all parse trees of the original grammar of depth less than n that the terminal string of that tree is also acceptable to the new grammar as a construction of the same type. Suppose T has depth n. Then let $Z \rightarrow Y_1Y_2 \ldots Y_n$ be the topmost rewriting of the parse tree and let $w_1, w_2, \ldots, w_n$ be the segments of the terminal string dominated by $Y_1, Y_2, \ldots, Y_n$, respectively. By the inductive hypothesis, each of the strings w_i is recognizable as type Y_i by the new grammar and unless $Y_1 = X$ the topmost rewriting is also permitted by the new grammar. Hence the only case of interest is when $Y_1 = X$. In this case we know that the string w_1 is accepted as an X both by the old and new grammars and hence by the machine $\overline{D}_X$. However, by the construction of step 2, every arc that enters a final state of $\overline{D}_X$ also has a copy which enters state q'. Hence w_1 will take machine $\overline{D}_X$ from state X to state q'. Now also, by construction, if the machine D_X accepts the string $Y_1Y_2 \ldots Y_n$ where $Y_1 = X$ then the machine $\overline{D}_X$ when started in state q' will accept the string $Y_2 \ldots Y_n$. Thus the total sequence $w_1w_2 \ldots w_n$ will take $\overline{D}_X$ from state X to a final state and hence w is recognizable as an X by the new grammar. This completes the induction proof.

Example 1 shows the application of this algorithm for the rule $X \rightarrow Xa + Xb + cd$.

R_X: $Xa + Xb + cd$



$\bar{R}_X$: $c(d + d\ (a + b)^*(a + b)) = cd\ (a + b)^*$

Example 1: Elimination of Direct Left Recursion

2.5.3 Elimination of direct right recursion

To eliminate right-recursion from the grammar, we again construct for every right-recursive symbol X a new rule $X \rightarrow \bar{R}_X$ which does not permit X as a final symbol, but which will produce the same set of terminal strings when used with the rest of the grammar. Again, we do this by first constructing a finite state transition graph D_X equivalent to R_X and transforming it as follows:

1. If there is more than one final state and it is not a dead end state, then construct an equivalent graph D'_X in which there is only one final state (and it is a dead end state) as follows: Add a new state q' and for every arc [x, y, z] entering a final state z, add a new arc [x, y, q'] entering state q'. Now delete any of the old final states which are dead end, and let q' be the sole final state in the new graph. The resulting graph D'_X has only one final state (namely q') and it is a dead end state.

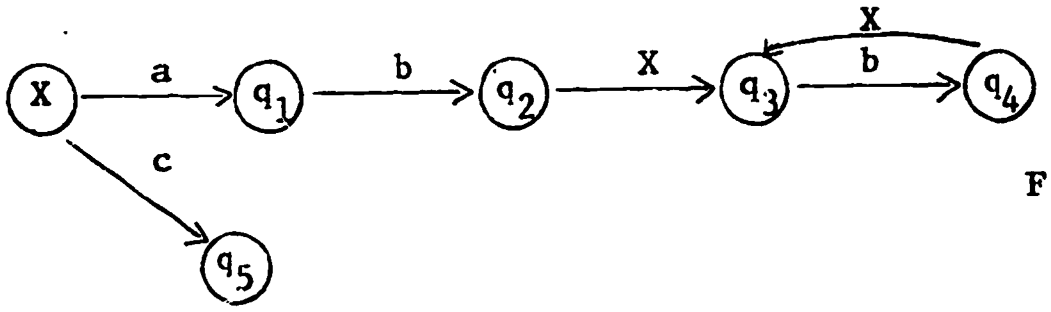
2. Eliminate the right recursion of the symbol X as follows: Let Q be the set of all states from which there is an arc labeled X which goes to the final state, and for every arc [x, y, z] entering a state z in Q, add a new arc [x, y, X] returing to the start state. Now delete all of the arcs [x, X, q'] for states x in Q and delete any states in Q which now have no arcs leaving them. The resulting graph $\bar{D}_X$ will accept the same terminal strings as D_X when used in conjunction with the rest of the grammar, but permits no final symbols X.

3. Construct the expression $\bar{R}_X$ from the graph $\bar{D}_X$.

To show that the new grammar accepts the same set of strings as the original grammar we again proceed by induction on the depth of the parse tree. Let w be a string recognizable by the original grammar as a construction of type Y and let T be any parse tree for the string w analyzed as a Y. As before, if the depth of the parse tree is 1, then the rewriting cannot involve recursion of any type and hence the string is accepted by the new grammar. Assume the result is true for all parse trees of depth less than n and consider T of depth n. Let $Z \rightarrow Y_1 Y_2 \ldots Y_n$ be the topmost rewriting of the tree and let $w_1, w_2, \ldots, w_n$ be the corresponding segmentation of the terminal string. The only case of interest is when Y_n equals X, in which case the sequence $Y_1, Y_2, \ldots, Y_{n-1}$ will take machine D_X (and also $\overline{D}_X$) into a state in Q. Therefore by the construction of $\overline{D}_X$ the same sequence will also take $\overline{D}_X$ back to the start state X, and since w_n is recognizable as an X by the new grammar (by the inductive hypothesis), the entire string w will also be recognizable as an X. This completes the proof. Example 2 shows the application of the algorithm for the rule $X \rightarrow (abX\ (bX)^* + c)$.

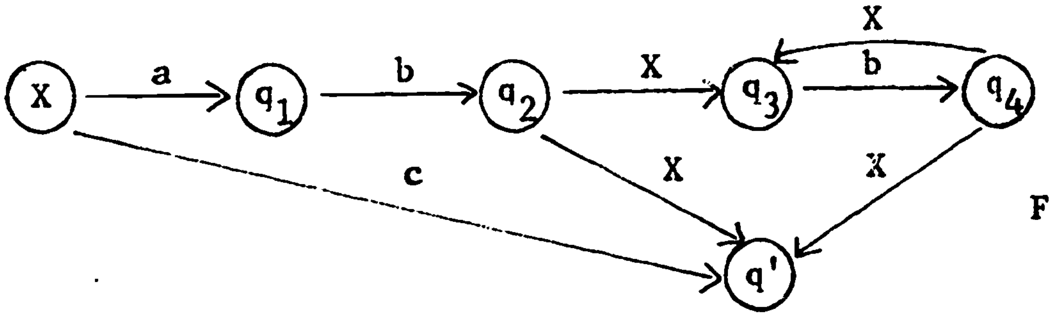
R_X: $(ab\ X\ (bX)^* + c)$

D_X:



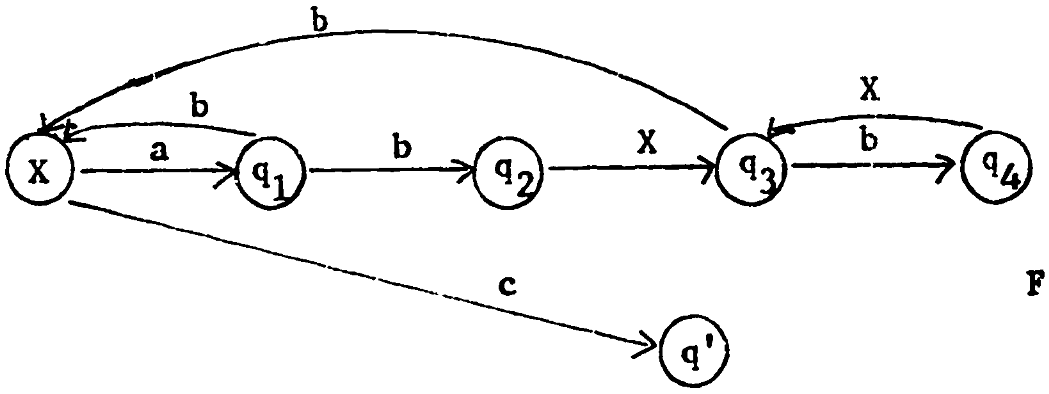
$F = \{q_3, q_5\}$

D'_X:



$F = \{q'\}$

$\bar{D}_X$:



$F = \{q'\}$

$\bar{R}_X$: $(a\ (b + bX(bX)^*\ b))^*\ c = (a\ (bX)^*\ b)^*\ c$

Example 2: Elimination of Right Recursion

SECTION 3

RECURSIVE TRANSITION NETWORKS AND THE EARLY RECOGNITION ALGORITHM

3.1 Introduction

Currently available recognition algorithms for context free grammars fall into two categories according to the bound that can be placed on the amount of time required to parse a string of length n. The straightforward parsing algorithms such as the Harvard Predicitive Analyzer (Kuno and Oettinger, 1963) which simulates the alternative computations of a non-deterministic pushdown store automaton or the immediate constituent analyzer (Herringer et al., 1966) which enumerates all of the reductions that can be performed on a given string by a given context free grammar require an amount of time which is an exponential function of n for some grammars. This is inevitably true for any "straightforward" parsing algorithm because of the existence of context free grammars which are exponentially ambiguous. Recently however, several recognition algorithms have been discovered which have a general time bound proportional to the cube of the length of the input string (e.g., Kasami, 1965, Younger, 1966, and Early, 1968). In addition, certain subclasses of the context free grammars have been shown to be recognizable with smaller time bounds--e.g., linear grammars in time n^2 by Younger (1966) and by Kasami (1967) and LR(k) grammars in time n by Knuth (1965). These results are based on different algorithms for each of the special cases. The Early algorithm, however, matches or surpasses all of these results with a single algorithm which does not need to be "told" the class of grammar on which it is operating. The Early algorithm works within the general n^3 bound for any context free grammar. However, when the given grammar is unambiguous or linear (and in may other cases) the bound is only n^2, and for an LR(k) grammar using look ahead of k symbols, as well as for many other grammars, the Early algorithm has a bound proportional to n. In addition, the Early algorithm operates on the grammar as it is given, whereas the Younger result depends on the construction of an equivalent normal form grammar and the Kasami result requires a standard 2-form grammar.

3.2 Time bounds

Time bounds of the sort described above have a number of limitations in their ability to characterize the "goodness" of an algorithm for practical applications and may tend to be misleading if not carefully analyzed. First, they tend to be pessimistic in that they characterize the behavior of the algorithms for the worst case grammars and the worst case strings. Typically the grammars which one needs to parse in practice will not be the worst case but some intermediate case, and the real figure of concern is the number of operations required to parse a "typical" string for a "typical" grammar. An algorithm such as the Younger algorithm which always realizes its worst case bound for every grammar and every string is clearly not as practical as one which has the same bound in the worst case but generally does much better. The other factor that needs to be taken into account in the evaluation of a time bound of this sort is the size of the constant of proportionality. It is true that the differences that arise between two algorithms due only to the size of the proportionality constant will eventually be swamped by the growth of the factor of n, but this assumes that one will actually parse strings of indefinitely increasing length. Many of the algorithms that achieve the n^3 bound do so at the cost of an immense constant factor, and the length of the input may not be long enough in the typical case to make such an algorithm preferable to one say with an n^4 bound and a much smaller constant. (In comparing the constants of proportionality for two rival algorithms, it is of course necessary to be careful that the definitions of the basic operation (or "step") used for computing the bounds in the two cases are comparable in the amount of time that they would require on some machine.)

The Early algorithm and the bounds for it suffer very little from the first limitation since the algorithm seems to do the best that can be done with any particular grammar and string that it is given. Even when the grammar as a whole is not unambiguous so that the n^2 result holds, the algorithm may still require no more than n^2 time to recognize those strings of the grammar which are not ambiguous. Similarly the algorithm may operate in time n on a large class of strings for a grammar which is not recognizable in time n in general. With respect to the second limitation, the Early algorithm is no worse than the other n^3 algorithms which have been devised (and considerably better than some), but it still suffers from an excessively large constant in some cases--especially when it is using lookahead. We will now show that the transition network model of grammar can be used by a slightly modified version of the Early algorithm to recognize strings within the same time bounds, and that the finite state optimization of the network can provide a reduction in the constant of proportionality. The recursion elimination operation may also move a grammar from the n^3 domain to n^2 or even n, as when the elimination results in a linear grammar or even a finite state grammar. Before we proceed however, we will present a formal definition of a "transition network machine" which will provide the terminology for the description of the recognition algorithm.

3.3 Formal definitions

A *transition network machine* is a quintuple (V_N, V_T, S, M, I), where V_N is a vocabulary of *terminal symbols*, V_T is a vocabulary of *non-terminal symbols*, $S \varepsilon V_N$ is a distinquished *initial symbol*. M is a set of disjoint[†] finite-state automata with input vocabulary $V = V_N \cup V_T$, and I is an indexing function which assigns to each non-terminal in V_N a unique machine in M.

[†] By *disjoint* we mean that no two machines in M have any state names in common. Thus, given the state name alone it is possible to determine which machine we are referring to.

For the sake of simplicity we will assume that the start state of the machine I(X) is named X. Let Q be the total set of states of all the machines in M, F the total set of final states, and δ the transition function which is the union of the individual transition functions of the machines in M (viewing these functions as sets of ordered pairs). The fact that the machines in M are disjoint means that their structure is preserved in this single resulting network $N = (V_N, V_T, Q, \delta, S, F)$. It also means that each state in the set Q uniquely determines the particular automaton to which it belongs and hence the non-terminal symbol which it is trying to recognize. Let h(q) be the function which gives for any state q the non-terminal symbol which that state is trying to build. We will make use of this function as well as the transition function δ in the description of the recognition algorithm. We will also make use of the function $L^*(q)$ defined in section 2.5.1 which gives the set of all non-terminals which can be pushed down for from state q (perhaps via a succession of pushes through intermediate states). All of these functions can be represented in the computer by storing them in the form of tables.

We describe a computation of a transition network machine as follows. A machine configuration consists of a triple (q, w, s) where q is a state in Q, w is the string (in V_T^*) which remains to be scanned, and s is a string of states in Q which keep track of the recursion.

For $q, q', \varepsilon Q$, $w, w' \varepsilon V_T^*$, and $s, s' \varepsilon Q^*$ we write

$$(q, w, s) \vdash\!\!\!-\!\!- (q', w', s')$$

if either

1. $w = aw'$, $s = s'$, and $q' \varepsilon \delta(q, a)$
2. $s' = \overline{q}s$, $q' \varepsilon V_N$, and $\overline{q} \varepsilon \delta(q, q')$
3. $s = q's'$, $w = w'$, and $q \varepsilon F$

The first case is a normal transition, the s\ :ond is a pushdown operation (or recursion), and the third is the pop operation which returns from a lower level of recursion. We define the transitive closure $\overset{*}{\vdash\!\!\!-\!\!-}$ of the relation $\vdash\!\!\!-\!\!-$ by the recursive definition:

$$c_1 \overset{*}{\vdash\!\!\!-\!\!-} c_2 \text{ iff}$$
$$\text{either } c_1 = c_2 \text{ or}$$
$$\exists c_3 \ni c_1 \vdash\!\!\!-\!\!- c_3 \text{ and } c_3 \overset{*}{\vdash\!\!\!-\!\!-} c_2.$$

A terminal string w is accepted by the network if $(S, w, e) \overset{*}{\vdash\!\!\!-\!\!-} (q, e, e)$ for some $q \varepsilon F$ (where e denotes the empty string consisting of no symbols).

3.4 The Early algorithm

The Early recognition algorithm parses an input string $x_1x_2x_3 \ldots x_n$ by constructing for each position i in the string a "state set" S_i' which contains all of the states in which a non-deterministic pushdown store automaton could be at that point in the string. Instead of carrying a pushdown store along with each state, the algorithm carries a pointer to the position in the string where the last pushdown preceeding each state occurred. The n^3 bound on the algorithm depends on the ability to follow any such pointer back to the appropriate position in the string in a fixed amount of time (independent of n)--i.e., it requires the use of a random access store of unlimited size for storing the state sets. This of course is only approximated by the core storage of a real computer, but for reasonable length strings it is a useful approximation. As long as the computation can be performed within the random access memory of the computer the approximation holds.

In this section we will describe a slightly modified form of the Early recognition algorithm which will recognize the strings accepted by a recursive transition network in time proportional to the cube of the length of the string. Early's result that the time is bounded by n^2 in the case of unambiguous strings and that the time is proportional to n for strings for which the size of the active state sets at any point is bounded will also hold.

The algorithm:

Given the input string $x_1x_2x_3 \ldots x_n$ to be parsed, the algorithm proceeds as follows:

1. Construct state set $S_0 = \{[S, 0]\}$, and set the closure $S_0' = S_0 \cup \{[q, 0] : q \varepsilon L^*(S)\}$. This represents the set of all things which we can be looking for at the beginning of a sentence.

2. For $i = 1, 2, \ldots, n$ construct sets S_i and S'_i as follows:

   2.1 (transitions)

   For each $[q, j]$ in S'_{i-1} for which $\delta(q, x_i) \neq \emptyset$, add $[q', j]$ to S_i for each q' in $\delta(q, x_i)$. (That is, $S_i = \{[q', j] : q' \varepsilon \delta(q, x_i) \,\&\, [q, j] \varepsilon S'_{i-1}\}$.)

   2.2 (closure operations)

   Set S'_i initially equal to S_i and scan the states in S'_i in order performing the following operations on each state $[q, j]$:

   2.2a (pushing down)

   For each q' in $L^*(q)$ add $[q', i]$ to the end of S'_i (so that it will be scanned) unless it is already a member.

   2.2b (popping up)

   If q is a final state, then scan the states $[q', j']$ in S'_j for which $\delta(q', h(q)) \neq \emptyset$ (i.e., states which can push down for the symbol $h(q)$). For each such q' and for each q'' in $\delta(q', h(q))$, add $[q'', j']$ to the end of S'_i (so that it will be scanned) unless it is already a member.

   When the last state of S'_i has been scanned and no new states have been added, then S'_i is complete.

3. The string is accepted if the state set S'_n contains a state $[q, 0]$ for some final state q. It is rejected before it reaches the end if any S_i is empty.

To illustrate the operation of this algorithm, we will work through the example "Did the red barn collapse?" using the transition network of Figure 1 of Section 1. We assume that the words, "did", "the", "red", "barn", and "collapse" are marked in a dictionary as members of the lexical classes aux, det, adj, n, and v, respectively. We proceed as follows:

0: $S_0 = \{[S, 0]\}$ by step 1

$S'_0 = \{[S, 0], [NP, 0]\}$

1: $\delta(NP, did) = \emptyset$ and

$\delta(S, did) = \{q_2\}$ since "did" is an aux.

Hence $S_1 = \{[q_2, 0]\}$ by step 2.1.

$\mathcal{L}^*(q_2) = \{NP\}$ (i.e., q_2 can push for a noun phrase), and therefore $S'_1 = \{[q_2, 0], [NP, 1]\}$ by step 2.2a

2: $\delta(q_2, the) = \emptyset$ and

$\delta(NP, the) = \{q_6\}$ since "the" is a det.

Hence $S_2 = \{[q_6, 1]\}$ by step 2.1.

$\mathcal{L}^*(q_6) = \emptyset$; hence $S'_2 = S_2$.

3: $\delta(q_6, red) = \{q_6\}$ since "red" is an adj.

Hence $S_3 = \{[q_6, 1]\}$ by step 2.1, and again $S'_3 = S_3$.

4: $\delta(q_6, \text{barn}) = \{q_7\}$ since "barn" is a noun.

Hence $S_4 = \{[q_7, 1]\}$

$\mathcal{L}^*(q_7) = \{PP\}$ and hence $[PP, 4]$ is added to S'_4.

Also, q_7 is a final state, which means that we have found a complete construction. Its type is $h(q_7) = NP$. We now refer to the state set S'_1 which caused the pushdown to look for this NP (as indicated by the pointer 1 which we have carried along with the state $[q_7, 1]$). Of the two states (q_2 and NP) in the state set S'_1, the state q_2 can take an NP transition to state q_3 (i.e., $\delta(q_2, NP) = \{q_3\}$), and hence we add the pair $[q_3, 0]$ to S'_4 by step 2.2b. The set S'_4 now is equal to $\{[q_7, 1], [PP, 4], [q_3, 0]\}$.

5: $\delta(q_7, \text{collapse}) = \emptyset$

$\delta(PP, \text{collapse}) = \emptyset$

$\delta(q_3, \text{collapse}) = \{q_4\}$ since "collapse" is a verb.

Hence $S_5 = \{[q_4, 0]\}$

$\mathcal{L}^*(q_4) = \{NP\}$, and therefore $[NP, 5]$ is added to S'_5.

Since q_4 is a final state and $h(q_4) = S$, we check the state set S'_0 to see if anything there can take an S transition (this would handle left recursion if there were any). Since there are no such arcs from either of the states S and NP, the final value of S'_5 is $\{[q_4, 0], [NP, 5]\}$. Since this is the end of the string and the state set contair $[q_4, 0]$ and q_4 is a final state, the sentence is accepted by the algorithm.

3.5 A comparative example

Early's algorithm as originally described (Early, 1968) is essentially a special case of the algorithm we have described here in which the states of the transition net are pairs of integers $p \cdot k$ where p is the number of a rule in the context free grammar and k is a count of the number of symbols in the right-hand side of the rule which have been recognized.[†] Instead of a single start state named X to begin a pushdown for the symbol X, the original Early algorithm has a start state $p \cdot 0$ for each rule $D_p \rightarrow C_{p1} \ldots C_{pn_p}$ whose left-hand side is equal to X. The final states are the pairs $p \cdot n_p$ (where n_p indicates that all of the right-hand side of the rule has been recognized). We will give here a brief illustrative example that will indicate the advantages which can be gained by using the transition network version over the unmodified Early recognition algorithm.

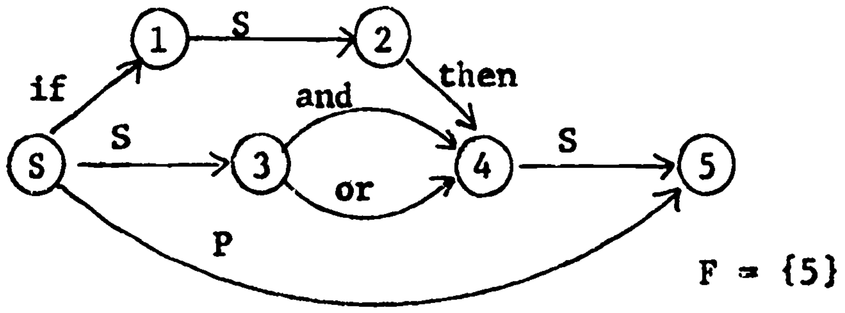
Figure 1a shows a context free grammar for a class of propositional calculus expressions involving the connectives "and, "or" and "if...then", where P is the only primitive proposition. Figure 1b shows an equivalent recursive transition network, and figure 1c shows the transition network that is derived from it when left and right recursion are eliminated and the network is minimized. Figure 2a shows the computation of the modified

[†] When lookahead is involved, the state includes a k-tuple of symbols which are expected values of $x_{i+1}x_{i+2} \cdots x_{i+k}$.

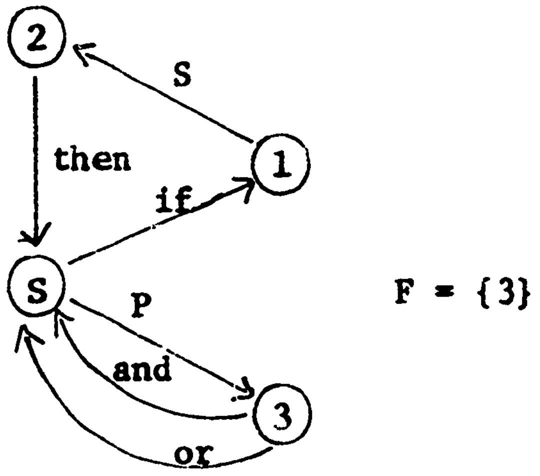
Early algorithm applied to the string "if P and P then P or P" using the optimized transition network of figure 1c, while figure 2b shows the computation of the original Early algorithm using the grammar of figure 1a. The improvement in the number of states that have to be processed is apparent.

1. S → if S then S
2. S → S and S
3. S → S or S
4. S → P

(a) sample context free grammar



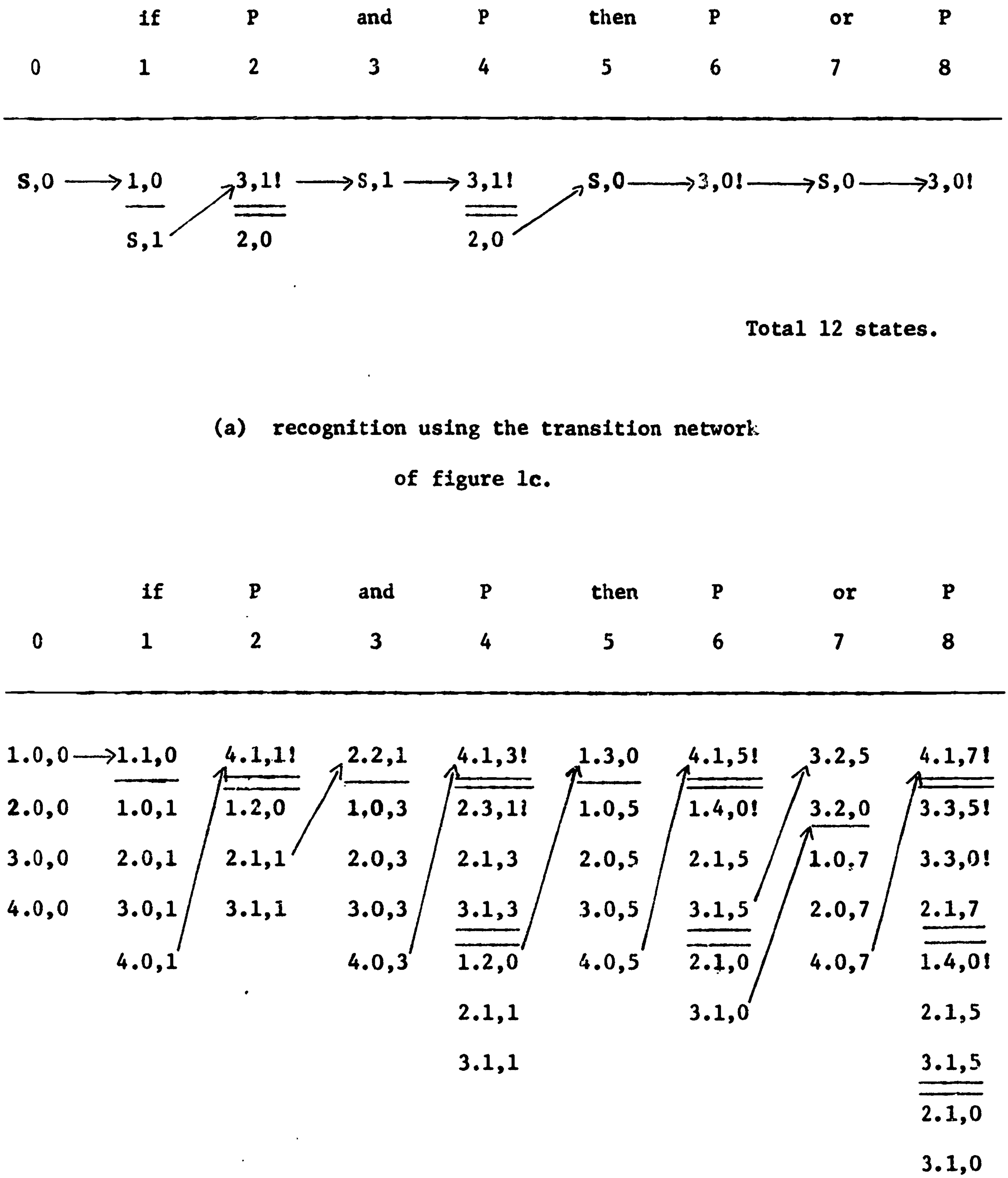
(b) an equivalent transition network



(c) an optimized transition network

Figure 1: An optimized transition network

for a context free grammar



Total 12 states.

(a) recognition using the transition network

of figure 1c.

Total 50 states.

(b) recognition using the original grammar

Figure 2: Comparison of the Early algorithm using on optimized transition network versus the original context-free grammar.

3.6 Time bounds for the Early algorithm

The proof that the modified version of the Early algorithm that we have presented requires no more than n^3 time to parse a string of length n parallels almost exactly the original proof given by Early. The proof relies on the assumption of a random access memory for the storage of the intermediate results that arise during the computation, and the achievement of the n^3 bound (or the n^2 bound for unambiguous grammars) requires careful use of this memory. For example, the qualifications "unless it is already a member" of steps 2.2a and 2.2b require special treatment in order to achieve the n^3 bound. If this were done by sequentially scanning the list S_i', then an extra power of n would be required in time bound because the size of the set S_i' can grow proportional to the length of the string. On the other hand, if a random access array indexed by q' and j' is used to store the state set S_i', then the presence or absence of $[q', j']$ from the state set can be determined by directly interrogating a single bit. (It is sufficient to index the set S_i' by just the back pointers j' and to scan the $j^{\underline{th.}}$ subset for the value of q' since there is a fixed bound on the number of states which can be in such a subset.)

In the computation of the time bound, we will count the number of "operations", where an operation may be taken to be any computation which can be done within a fixed time bound that is independent of the length of the input string. In particular, for the appropriate organization of the random access memory, determining whether a state set S'_j contains a pair [q', j'] will be an operation. Likewise determining whether S'_j contains states which can push for a given non-terminal symbol X and obtaining a pointer to a list of such states will be an operation.

If N is the total number of states in the network, then there are at most N(i + 1) states in the state set S_i. Let R be the maximum number of arcs which leave any state, and let M be the maximum size of the set $L^*(q)$ for any state q. A bound on the number of operations required to parse a string of length n can be computed as follows:

1. It requires at most M + 1 operations to construct the initial set S'_0 since there are at most M states in $L^*(S)$.
2. At a given position i in the string the following bound can be placed on the number of operations:

   2.1 Transitions require at most R operations for each of the states in S_{i-1}, or no more than NRi operations in all.

   2.2 For each state in the set S'_i the number of operations required for the closure operations can be bounded as follows:

   2.2a It requires at most M operations to add all of the states to which the current state can push down since there are at most M states in $L^*(q)$.

   2.2b When q is a final state, with a pointer to some position j in the preceeding string, it requires at most $NR(j + 1)$ operations to scan the state set S'_j and resume any of the computations which pushed down for the current construction.

2.3 Since the operations of steps 2.2a and 2.2b are performed on each of the states in S'_i, the total number of operations for step 2.2 is bounded by

$$N(i + 1)[M + NR(J + 1)] \leq NM(i + 1) + N^2R(i + 1)^2 .$$

3. Since the operations of step 2 are performed at each position of the input string for i=1, ... , n, the total number of operations for this step can be bounded by

$$\sum_{i=1}^{n} [NM(i + 1) + N^2R(i + 1)^2] \leq N^2Rn^3 + O(n^2) .$$

Hence the total number of steps required by the algorithm is bounded by $N^2Rn^3 + O(n^2)$.

The achievement of the n^2 bound for unambiguous grammars requires a little more care in the implementation of the algorithm. It is necessary to get the number of steps required in step 2.2b down to a fixed amount for each state $[q, j]$ in S'_i . To do this we cannot afford to scan the entire state set S'_j to look for states which can push down for the symbol $h(q)$. It is necessary instead to be able to enumerate the class of such states using no more than a fixed amount of time for each one. This can be accomplished by keeping an array S''_j of all the entries in S'_j indexed by the non-terminal symbols pushed for. With such an array it is possible to instantly determine for any specified value of $h(q)$ and position j

a pointer to the list of those states $[q', j']$ in S'_j for which $\delta(q', h(q)) \neq \emptyset$. This list can be constructed and maintained by implementing the algorithm so that when we add $[q', i]$ to S'_i in step 2.2a, we also add $[q, j]$ to the list $S''_i(q')$. Then when we refer to a previous position in the input string in step 2.2b to see whether any states in S' could have pushed for the symbol $h(q)$, we need only consult the appropriate entry $S''_j(h(q))$. (Since the only steps of the algorithm which refer to an earlier point in the string are those in step 2.2b, the only information that needs to be kept about the state sets prior to positions i-1 and i are the sets S''_j. The sets S'_j (and S_j) are only used when i is equal to j and j+1, and they may be discarded thereafter. Hence it is not necessary to keep multiple copies of the entire computation, but only of the active part of the computation S'_{i-1} and S'_i.)

Using this further clarification of the algorithm, the proof that the bound is n^2 for unambiguous grammars goes as follows: Since there can be at most $N(i + 1)$ states in state set S'_i there will be at most a number of operations involved in step 2.2 proportional to $N(i + 1)$ unless some state is added in more than one way. This is true because we have organized the stored information in S''_i so that it takes only one operation for states $[q, j]$ for which $S''_j(h(q))$ is $\emptyset$, and only one operation for each state to be added otherwise. Hence the total number of operations in step 2.2 has a bound proportional to i unless some state is added in more than one way. On the other hand, if some state is added in more than one way, then the grammar will accept ambiguous strings (not necessarily the one that is currently being parsed however) unless the predicted state is a dead end that cannot be completed for any string (in which case it should have been removed from the network). Hence if the grammar was unambiguous (and contained no misleading, ambiguous-looking but dead end predictions) then the number of operations required by step 2.2 is at most proportional to i, and the total number of operations to recognize a string is proportional to n^2

The time n bound on the number of operations for recognizing a "bounded direct ambiguity grammar" (Early, 1968), which includes the class of LR(k) grammars, is achieved, because the total number of states in any state set has a fixed bound for such grammars.

Although we have described here the time bounds for a recognizer only, it is possible as Early shows to use the algorithm as a parser (a routine which not only determines whether a string is a sentence but also builds a representation of all of the structural descriptions of the sentence) within the same time bounds. This requires the use of a representation of structural descriptions which merges the common parts of different descriptions of the string (since some sentences can be exponentially ambiguous even though they require only n^3 steps to recognize them and build the structural descriptions). It is also not difficult to add conditions to the arcs of the transition network (which must be met in order for the arcs to be followed) and still recognize the strings within the same time bounds provided that the conditions have a fixed time bound. For example, the lookahead feature for recognizing LR(k) grammars in time n could be added as a condition on the arc.

4. Conclusion

We have presented a model of grammar based on the notion of a transition network similar to a finite state transition network applied recursively, and have shown it to be a very promising model for natural language analysis. It is capable of building deep structure representations while doing a surface structure analysis of a sentence without a separate explicit reverse transformational component. Also it is capable of considering semantic selectional restrictions while parsing, and it may provide the basis for a harmonious interaction between syntactic and semantic analyses. In addition to having a number of theoretical advantages for efficient parsing, the model is convenient for a human grammar designer to work with and answers a number of objections which linguists have raised against the transformational grammar model.

A transition network parser along the line presented in this report has been implemented in BBN LISP on the SDS 940 time sharing system at Harvard, and a number of experiments have been carried out exploring various parsing strategies and special parsing techniques. Particular attention has been devoted to exploring the interaction between the parser and the semantic interpreter and using semantic information to guide the parsing. The details of the parser implementation and the experiments which have been conducted will be described in a forthcoming report. Experimental evidence, as well as the theoretical arguments presented in this report, indicate that this model will permit the mechanical analysis of natural language to a much greater depth than has been possible with other grammar models and that it will not be necessary to pay an exorbitant penalty in processing inefficiency in order to do this.

BIBLIOGRAPHY

Bobrow, D.G. and Fraser, J.B. 1969. "An Augmented State Transition Network Analysis Procedure," Proceedings of the International Joint Conference on Artificial Intelligence, May 7-9, Washington, D.C.

Bobrow, D.G., Murphy, D., and Teitelman, W. 1968. "BBN LISP System," Bolt, Beranek and Newman Inc., Cambridge, Mass.

Book, R., Even, S., Greibach, S., and Ott, G. 1969. "Ambiguity in Graphs and Expressions," (mimeographed report), Aiken Computation Laboratory, Harvard University, Cambridge, Mass.

Cheatham, T.E. and Sattley, K. 1964. "Syntax-Directed Compiling," AFIPS Conference Proceedings, Vol. 25, (1964 Fall Joint Computer Conference).

Chomsky, N. 1963. "Formal Properties of Grammars," in Handbook of Mathematical Psychology, Vol. 2, Wiley, New York, (Luce, R.D., Bush, R.R., and Galanter, E., Eds.).

Chomsky, N. 1964. "A Transformational Approach to Syntax," in The Structure of Language, Prentice-Hall, Englewood Cliffs, New Jersey, (Fodor, J.A. and Katz, J.J., Eds.).

Chomsky, N. 1965. Aspects of the Theory of Syntax, MIT Press, Cambridge, Mass.

Early, J. 1968. "An Efficient Context-Free Parsing Algorithm," Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania, (Ph.D. thesis).

Ginsburg, S. 1966. The Mathematical Theory of Context-Free Languages, McGraw-Hill, New York.

Herringer, J., Weiler, M., and Hurd, E. 1966. "The Immediate Constituent Analyzer," in Report No. NSF-17, The Aiken Computation Laboratory, Harvard University, Cambridge, Mass.

Kasami, T. 1965. "An Efficient Recognition and Syntax-Analysis Algorithm for Context-Free Languages," AFCRL-65-558, (also presented at Summer School on Combinatorial Methods in Coding and Information Theory, Royan France).

Kasami, T. 1967. "A Note on Computing Time for Recognition of Languages Generated by Linear Grammars," Information and Control, Vol. 10.

Knuth, D.E. 1965. "On the Translation of Languages from Left to Right," Information and Control, Vol. 8.

Kuno, S. 1965. "A system for Transformational Analysis," in Report No. NSF-15, The Computation Laboratory, Harvard University, Cambridge, Mass.

Kuno, S. and Oettinger, A.G. 1963. "Multiple Path Syntactic Analyzer," in Information Processing 1962, North-Holland Publishing Co., Amsterdam.

McCarthy, J. et al. 1962. LISP 1.5 Programmer's Manual, MIT Computation Center, Cambridge, Mass.

McCawley, J.D. 1968. "Meaning and the Description of Languages," Kotoba No Ucho, TEC Company Ltd., Tokyo.

McNaughton, R.F. and Yamada, H. 1960. "Regular Expressions and State Graphs for Automata," IRE Transactions on Electronic Computers, Vol. EC-9.

Matthews, G.H. 1962. "Analysis by Synthesis of Natural Languages," Proceedings of the 1961 International Conference on Machine Translation and Applied Language Analysis, Her Majesty's Stationery Office, London.

MITRE 1964. English Preprocessor Manual, Report SR-132, The MITRE Corporation, Bedford, Mass.

Ott, G. and Feinstein, N.H. 1961. "Design of Sequential Machines from their Regular Expressions," Journal of the ACM, Vol. 8, No. 4.

Petrick, S.R. 1965. "A Recognition Procedure for Transformational Grammars," (Ph.D. thesis), MIT Department of Modern Languages, Cambridge, Mass.

Postal, P.M. 1964. "Limitations of Phrase Structure Grammars," in The Structure of Language, Prentice-Hall, Englewood Cliffs, New Jersey, (Fodor, J.A. and Katz, J.J., Eds.).

Schwarcz, R.M. 1967. "Steps Toward a Model of Linguistic Performance: A Preliminary Sketch," Mechanical Translation, Vol. 10.

Thorne, J., Bratley, P., and Dewar, H. 1968. "The Syntactic Analysis of English by Machine," in Machine Intelligence 3, American Elsevier, New York, (Michie, D., Ed.).

Woods, W.A. 1967. "Semantics for a Question-Answering System," Report No. NSF-19, The Aiken Computation Laboratory, Harvard University, Cambridge, Mass. (Ph.D. thesis).

Woods, W.A. 1968. "Procedural Semantics for a Question-Answering Machine," AFIPS Conference Proceedings, Vol. 33, (1968 Fall Joint Computer Conference).

Younger, D.H. 1966. "Context Free Language Processing in Time n^3," G.E. Research and Development Center, Schenectady, New York.