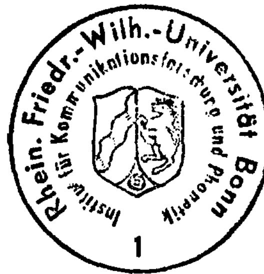
Eing.-Nr.

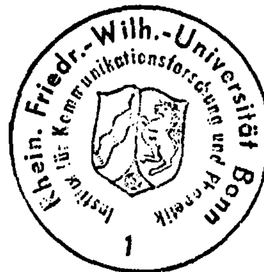
1965 International Conference on Computational Linguistics

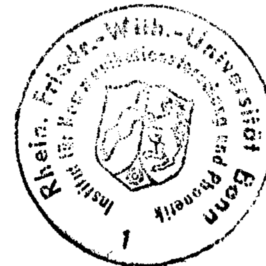
AUTOMATIC DEEP STRUCTURE ANALYSIS USING AN APPROXIMATE FORMALISM

D. Lieberman, D. Lochak and K. Ochel



IBM Research Center
P. O. Box 218
Yorktown Heights, New York
U.S.A.





ABSTRACT

The automatic sentence structure analysis procedure described earlier[(1)] has been programmed. The formalism is not transformational, but is sufficiently general to permit the expression of a wide variety of structural models. It is now being used to obtain automatic structural descriptions which are very close to the "deep structure" structural descriptions of transformational theory.

The formalism will be described, and its use for various structural models will be illustrated. A detailed description of the reformulation of a transformational grammar of English in terms of the approximate formalism will be given.

The salient features of the computer program which operates on the formalism will be described, and samples of output will be presented.

---

(1) D. Lieberman, "A Procedure for Automatic Sentence Structure Analysis". Paper presented at the 1963 Annual Meeting of the AMTCL, Denver, Colorado, August 25-27, 1963.

The objective of the work described herein is the development of a computerized linguistic basis for application to practical and theoretical problems in automatic language processing. The three main parts of work are: 1) a formalism for expressing grammatical information, 2) a sentence analysis procedure based on the formalism, and 3) a grammar of English expressed in terms of the formalism, but motivated by transformational theory.

The Formalism

The formalism is not intended to represent any particular linguistic theory or model, but rather, as the name implies, is simply a vehicle for expressing various models. It is designed to be sufficiently restrictive to permit the development of an associated sentence analysis procedure, and at the same time, to be sufficiently flexible to permit the relatively straightforward representation of most current linguistic models and variants, such as IC analysis, dependency theory, context-free and context-sensitive phrase structure grammars, etc. The formalism is not well adapted to the direct representation of transformational grammars, but, as described below, can be used to obtain structural descriptions very close to the "deep structure" of a transformational description.

In the present system, a structural description is a single labeled tree, with no further inherent restrictions (a sentence with multiple syntactic readings will have multiple trees, but each reading is represented by a single tree). Further restrictions, such as projectivity for example, can be imposed by appropriate use of "condition statements" described below.

The grammatical information is expressed through the formation of a set of category or node types. This collection is called a node dictionary. Each node type in the node dictionary has the following format:

Field A - node type

Field Bi - list of possible immediate ancestors (order irrelevant)

Field Ci - list of possible immediate descendents (order irrelevant)

Field Cij - lists of conditions associated with selection of the items in the corresponding Ci-field. Each condition statement consists of a condition type and an appropriate number of tree addresses. Conditions are used to express restrictions such as word order, government, agreement, etc. The types of conditions presently in the program are described below.

Field D - continuity (do all the lexical items dominated by this node type (Field A) occupy a continuous segment of the input string).

Field E - blockage (list of node types blocked by the current node type). Each of the node types in the list may be accompanied by a Tree Address, with the interpretation that nodes of the given type are blocked by the current node only if they are dominated by the node at the Tree Address.

Fields Gi - subcategorization features.

The following types of conditions are now included in the program:

Type A Format A^{+}_{-} (Tree Address)

Interpretation: The current branch (the Ci under which the condition occurs) requires either the presence (if + is used) or the absence (if - is used) of the node at the Tree Address.

Type B Format 1: B

Interpretation: The current branch is optional

Type B Format 2: B^{+}_{-} (Tree Address)

Interpretation: The current branch is optional only if the node at the Tree Address is present (if + is used) or absent (if - is used).

Type DB Format 1: DB

Interpretation: The current branch is optionally deletable.

Type DB Format 2: DB^{+}_{-} (Tree Address)

Interpretation: The current branch is optionally deletable only if the node at the Tree Address is present (if + is used) or absent (if - is used)

Type DD Format 1: DD

Interpretation: The current branch is obligatorily deletable.

Type DD Format 2: DD^{+}_{-} (Tree Address)

Interpretation: The current branch is obligatorily deletable only if the node at the Tree Address is present (if + is used) or absent (if - is used).

Type G Format: G i/j (Tree Address)(Tree Address)

Interpretation: The presence of the current branch requires identity of the features at subfield i of the G-field of the node at the first Tree Address and subfield j of the G field of the node at the second Tree Address.

Type H Format: **H** (Tree Address) (Tree Address)

Interpretation: The lexical item at the first Tree Address must precede the lexical item at the second Tree Address. If either of the Tree Addresses does not point to a lexical item, the condition statement is in error.

Type S Format: S

Interpretation: The current branch is self satisfying, i.e. it is not a lexical item but has no descendents. This condition is used, for example, with sentence boundary branches.

In addition to the above condition types, a special device is used to indicate that a group of possible descendents of a given node are mutually exclusive. When only two branches are involved, a Type A - condition can be used, but when more than two branches are involved, the use of Type A - conditions becomes awkward.

The Sentence Analysis Procedure

A central feature of any automatic sentence structure analysis procedure is the manner in which syntactic ambiguity is handled during the processing. Even syntactically unique sentences will, during the processing, exhibit multiple potentialities. The formalism for handling such intermediate representations is a very important part of the heuristic capacity of the program. In the present

system, a compact graph-like structure is used for intermediate representation. At the end of the analysis, all syntactic readings of the sentence are represented by a single graph with appropriately marked conditions on its vertices and edges. A special output algorithm is required to extract the syntactically permitted trees from the compact graph representation.

The scanning sequence (single-pass, iterative, multipass, left-to-right, right-to-left, chunk and process, etc.) has been made a semi-independent component of the procedure in order to permit flexibility in the future application of theoretically or heuristically motivated approaches to search strategy. In the present version of the system, a left-to-right single-pass search strategy is used. This very simple search strategy was chosen as a start in order to permit concentration on the complexities in the other parts of the procedure.

The sentence is analyzed one item at a time from left to right. The items on which the analysis procedure operates are not the orthographic words, but rather, the result of a dictionary lookup step which includes some morphophonemic analysis. Thus, _went_ would be analyzed into _go_ + _past_, _painted_ would be analyzed into _paint_ + _past_ or _en_, etc. The dictionary lookup step would also yield, for each item, the node type or types (A-field) of which the item is a descendent, and subcategorization features to be placed in the corresponding G-fields. Thus, after dictionary lookup, the input sentence would be replaced by a string of nodes. These nodes are the items which are processed one at a time, left-to-right.

The following sketch of the analysis procedure is intended to indicate current status; numerous details are omitted.

Suppose dictionary lookup yielded the following string of nodes:

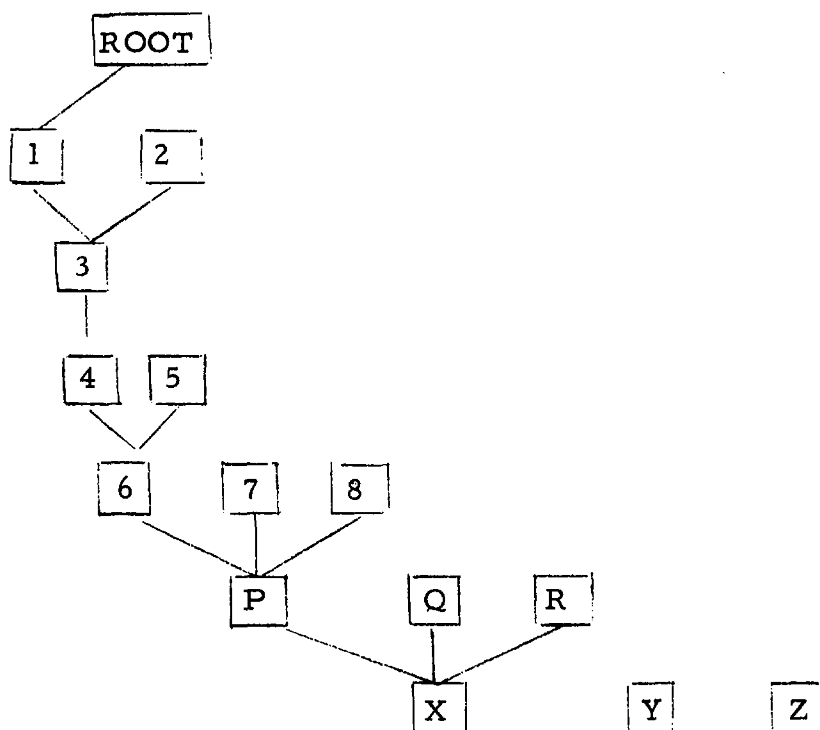
All possible immediate ancestors of X (listed in its B-field) are built up. Thus, if the B-field of node X contained

B1	P
B2	Q
B3	R

the following structure would result

and branches PX, QX and RX would be marked as mutually exclusive in any one reading of the sentence. Then, in the same manner, all possible ancestors of node P would be formed, and the building upward process would continue until a special node type without ancestors (the root of the tree) was reached. If the grammar permits recursion, some method for preventing infinite depth must be introduced. At present, we use an input parameter n which limits the number of new nodes of any one type on a single ancestor string to n. As will become clearer below, this does not limit the total recursion in a sentence to n.

At this point, we have a structure of the form:

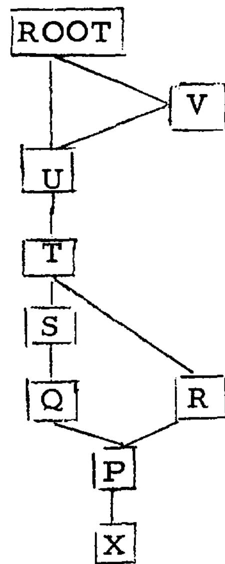


where the nodes have been assigned numbers for convenience of reference herein. In the actual process, they would, of course, be specific node types.

Next, the new nodes without ancestors are built upward systematically. The end result (syntactic readings of the sentence) should not depend on the sequence used. The sequence we are now using was chosen for programming convenience. It is not perfectly clear that the end result is in fact independent of the sequence, but there are, as yet, no indications to the contrary. In the sequence we are now using, node 2 would be built upward next. If new nodes without ancestors were formed, they would be processed next. When processing of node 2 and its ancestors is completed, node 5 is built upward similarly, then node 7, 8, Q and R in that order.

During the building upward of the very first ancestor string X-P-6-4-3-1-ROOT, there was no choice but to continually create new nodes to serve as the required ancestors. However, in subsequent build-ups, a required ancestor node type may already exist, in which case it is used, providing that it doesn't result in a node dominating itself. If two or more nodes share a common ancestor, the corresponding branches are marked as mutually exclusive even if the descendent node types are different and compatible from the point of view of the ancestor node (i.e. they attach to different parts of the ancestor's C-field), because we are still considering the buildup of a single item, node X, which could not serve as two constituents simultaneously in any given syntactic reading of the sentence.

Following completion of the building up of node X, the resultant structure is scanned for closed branches. A branch is called closed if all readings of a sentence require the presence of that branch. The test for whether a branch is closed is whether removal of the branch would completely disconnect node X from the root. For example, in the following structure:



branches U-T and P-X are closed. Closed branches are marked accordingly.

A node with a closed branch is called definite. If a node is not definite, it is called potential. This nomenclature will be used below.

The next step is the testing of the conditions associated with the various branches established during the buildup. The conditions to be tested for a given branch are listed in the corresponding part of the C-field of the node from which the branch descends. Since the results of condition testing depend on blockage, which has not yet been discussed, a detailed description of condition testing will be postponed until processing of the next item (one which is not the leftmost item) is considered.

After condition testing, blockages are applied. There are two types of blockage - permanent and temporary, and each type can be definite or potential. The information causing blockage is in the D-field and the E-field of each node type. The D-field is used to indicate whether or not the node is continuous, i. e. whether the lexical items dominated by the node occupy a continuous segment of the input string. Each of the continuous nodes created during the building up process causes temporary blockage to be applied to every node which it does not dominate. If the node causing blockage is definite, the blockage is definite; if the node causing blockage is potential, the blockage is potential. When a node causes temporary definite blockage a list is kept, at the node causing blockage, of all the temporary blockages caused by that node. Later, when the node causing blockage has been filled with the required constituents, the blockage it caused

is removed. In the case of temporary potential blockage, a similar list is created and, in addition (for reasons described below), a record of the node causing blockage is made at each blocked node.

The next blockage action involves permanent blockage (both definite and potential) and is guided by information in the E-fields of the various nodes created during the building up process. However, we are still discussing the processing of the very first (leftmost) item in the input string, and in this case, permanent blockage does not apply. Permanent blockage action will be described below when processing of the next input item is considered.

This completes processing of the first item, and we proceed to the next input item. The same building up process is carried out, except that now, a node (other than the root of the tree) required as an ancestor may already exist. In this case, the existing node is used as the ancestor, providing that: 1) it is not definitely blocked, and 2) the required branch is not closed. If the existing node is potentially blocked, a connection is made, but the connection and the node which caused the potential blockage are marked as mutually exclusive in any one reading of the sentence. This is why (as described above) a record of the node causing potential blockage is kept at the potentially blocked node. If a connection is made to a potentially blocked node, alternative ancestors are also created since (as described below) the potential blockage may later be changed, retroactively, to definite blockage and the previously made connection would be erased.

It can now be seen why (as mentioned above) the recursion parameter n does not limit the total recursion in a sentence to n.

The recursion parameter limits the number of new nodes of a given type which can be created along an ancestor string during the building up of a given input item. Thus, in the building up of item 2, for example, n nodes of a given type may be created along an ancestor string and the string may then be connected to an already existing node which in turn may have had n nodes of the given type created along its ancestor string.

After the building up of item 2 is completed, closed branches are marked on the newly created structure as was done in the processing of item 1. However, in the case of item 2 and subsequent items, further action is taken. If a branch marked closed had also been previously (during the building up) marked as mutually exclusive with some other branch, the other branch is erased.

Whenever a branch is erased, a routine called CLEANUP is brought into action. This routine follows up all the consequences of erasing a branch, and cleans up the structure accordingly. For example, if the branch being erased is marked as a necessary co-occurrence of some other branch, the other branch is also erased. If the branch being erased is the sole ancestor of some node, that node is erased, i.e. all of its branches are erased. If a branch being erased is an obligatory constituent of some permanently definitely blocked node, and there are no competing branches representing that constituent, the node is erased. During cleanup, a branch which was not previously closed may become closed, thus making some previously potential node definite. If the node in question caused blockage at the time it was created, the blockage would have been potential. The blockage is now made definite retroactively.

This is the purpose of the bookkeeping described above regarding connections made into potentially blocked nodes. A record was kept of all branches connected into potentially blocked nodes. When a particular potential blockage becomes definite, the corresponding marked branches are erased. Or, cleanup may operate in the reverse direction. If a branch which had been connected into a potentially blocked node becomes closed during cleanup, the node which caused the blockage is erased.

The above account of cleanup is not meant to be exhaustive, but simply to describe the main features of the CLEANUP routine.

The cleanup routine may run into a contradiction. For example, a definite node or a closed branch might be marked for erasure. When this happens, the analysis is terminated, and the sentence is labeled "non-grammatical".

Returning to the main routine (CLEANUP is a subroutine used repeatedly during the main routine), the next step is condition testing. Each of the conditions listed in the various newly created node Ci-fields which received candidates are tested. The result of a condition test is YES, NO, or UNTESTABLE. YES means the condition is satisfied, NO means the condition is violated, and UNTESTABLE means that the condition was untestable because one or more of the nodes involved in the condition did not exist at the time the condition was tested.

If the result of a condition test is YES, no further action is taken. If the result is NO, the subsequent action depends on the status of the various branches involved in the test. If none of the branches (including the one under which the test is listed) are

closed, they are marked as mutually exclusive. If the branch under which the test is listed is closed and the other branches involved in the test are not, the latter are erased. If both are closed, the analysis is terminated and the sentence is labeled "non-grammatical". The erasures are carried out by the CLEANUP routine and all consequences of each erasure are followed up as described above.

A few UNTESTABLE results can be acted upon. For example, if a condition on X is that it follow Y, and Y does not exist, the result is equivalent to a NO result. However, with most condition tests resulting in UNTESTABLE, no immediate action is taken. A list of such events is kept and the tests are reapplied after the last item in the sentence is processed. It is, of course, highly desirable to reapply previously untestable conditions the moment they become testable. Methods for accomplishing this, without paying so high a price in machine time and/or space that the advantages are nullified, are being considered, but at this point in the work are not of the highest priority, because the end result (the structural description) should depend only on the collection of conditions and not on the order in which they are applied. This and other problems concerned primarily with machine running time will receive increased emphasis in the future.

Continuing with the main cycle, the next step is application of blockage. First, temporary blockage, as indicated by the D-fields of the newly created nodes, is applied to all relevant nodes, both newly created and previously existing. The procedure is the same as in the case of the first input item.

Next, permanent blockage, as indicated by the E-fields of the newly created nodes, is applied. The procedure is similar to that used for applying temporary blockage, the essential difference being that blockage is applied only to previously existing nodes, and not to newly created nodes. This explains why permanent blockage did not apply during the processing of the first item -- there were no previously existing nodes, only newly created nodes. In the actual program, it was convenient to allow permanent blockage to be applied during the processing of the first item, but the result is, of course, vacuous.

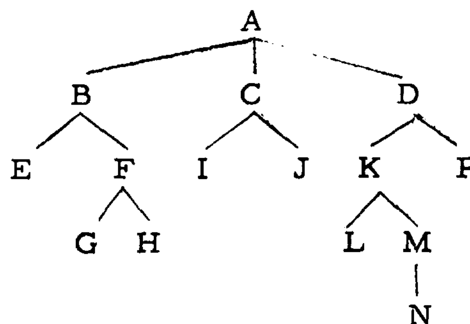
Application of permanent blockage provides possibilities for cleanup which do not occur in the case of temporary blockage. If a node is marked permanently definitely blocked, it is checked to see whether it contains all of its non-deletable obligatory branches. If it does not, it is marked for erasure and the CLEANUP subroutine goes into action.

This completes the processing of item 2. The next and subsequent input items are processed similarly, until the final item, representing end of sentence, is reached. This item has only one possible ancestor (the special node at the root of the tree) and its E-field lists all node types. Thus, it permanently and definitely blocks all nodes. When all of these blockages have been applied and the resulting cleanups have been carried out, the analysis is completed, and we proceed to the output routine.

The purpose of the output routine is to print out, explicitly, each of the possible syntactic readings of the sentence assigned by the analysis procedure. The individual readings are not explicit at

the end of the analysis routine because they are all represented by a single compact graph-like structure with dependencies (required co-occurrence and mutual exclusion) marked on various branches. The output routine scans the graph-like structure systematically and prints out all trees which satisfy the marked dependencies, have the special root node as their root, and in which each input item appears once and only once as a leaf.

The results are printed out in a tree format. As a compromise between readability and machine convenience, the tree is rotated counterclockwise through 90°, and the constituents of each node are justified upwards (after rotation) to the level of that node. Thus, the usual form of a tree, such as



would appear in the following output format:

```
A   D   P
        K   M   N
            L
    C   J
        I
    B   F   H
            G
        E
```

The problem of lining symbols up properly is simplified by our requirement that all symbols be five or less characters long.

As indicated above in the description of the types of conditions currently used, a distinction is made between optional and deletable constituents. The essential difference is that a deleted constituent should be filled in if the structural description is to be a reasonable approximation to deep structure. For example, the subject or object in a relative clause is deleted in the surface structure, but can be filled in in the deep structure by copying the noun in the noun phrase whose determiner contains the sentence which is relativized. Or, if the agent in a passive sentence is deleted in the surface structure, it can be filled in at least by an indefinite such as someone or something in the deep structure. At present, our output only indicates that a deleted constituent exists in the deep structure, but we have not yet formulated and programmed the rules for filling in such constituents. These rules are also needed during the analysis to permit condition testing where deleted items are involved. At present, if a condition test turns out to involve a deleted item, the test is ignored.

The Grammar

An overview of the grammatical categories (node types) and their relations in structural descriptions is given in Figure 1. Most of the underlined symbols are pre-lexical items, but some (those beginning with S) will be expanded later. The overline on some of the symbols is used to indicate that they are expanded elsewhere in the diagram. An example of a node type and its associated fields

is given in Figure 2 in the form in which we work with the grammar. Each line is on a separate punched card.

The collection of categories was taken, for the most part, from the phrase structure portion of a transformational grammar of English being developed at IBM. Some additional categories such as SREL, SMNL, SFT, SPT and SCOND were introduced to simplify condition statements using the current set of condition types. As more condition types are formulated and incorporated into the program, these additional and essentially redundant categories may be eliminated. Symbols beginning with X have no descriptive significance. They were also introduced as a temporary expedient to overcome certain defects in the present formalism, and will be eliminated when the formalism is appropriately modified. If these structurally superfluous nodes cause too much clutter in the output, they can be eliminated by simply erasing each one and connecting its immediate descendents to its ancestor as each tree is printed out.

The present grammar is very far from complete, in any sense of the word. The category types are reasonably extensive, but only a smattering of conditions are present, and these were selected mainly to test various portions of the program as they were completed. However, problems encountered thus far in using the formalism to express grammatical information have been solved without undue difficulty.

Some samples of output, and details regarding the programming will be presented during one of the informal afternoon group meetings.

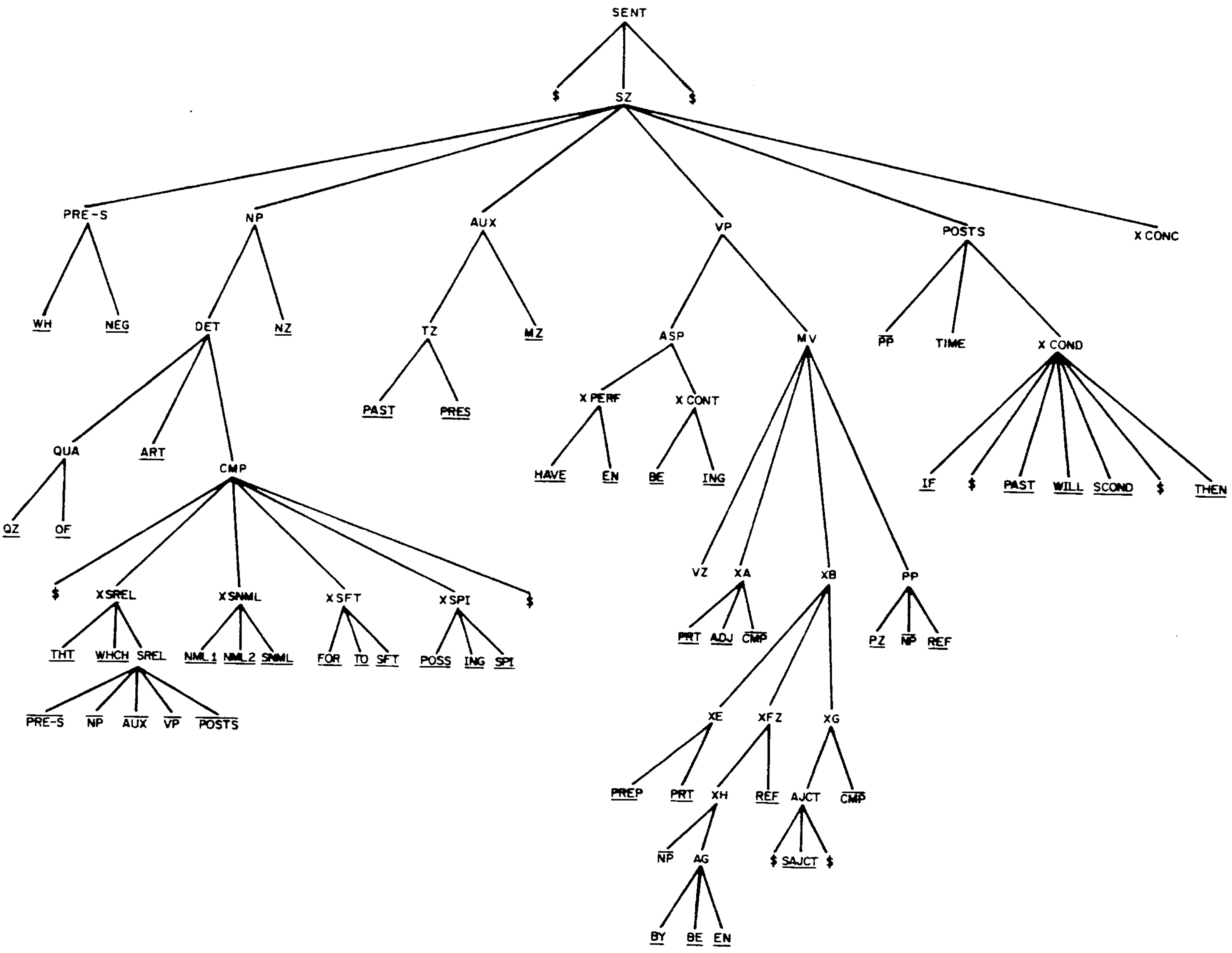


Figure 1

```
SREL                                                   SREL
XSREL                                                  SREL
PRE-S                                                  SREL
       B                                               SREL
       A      -(C1C1)                                  SREL
NP                                                     SREL
       DD     +(C4C2C3C2C1C1)                          SREL
       H      (C2C2)(C3C1)                             SREL
       H      (C2C2)(C4C2C1)                           SREL
       H      (B1B1B3B1C2)(C2)                         SREL
AUX                                                    SREL
       H      (C3C1)(C4C2)                             SREL
VP                                                     SREL
POSTS                                                  SREL
       B                                               SREL
SREL                                                   SREL
```

Figure 2