MEMORANDUM
RM-6265/2-PR
APRIL 1970

THE MIND SYSTEM:
THE MORPHOLOGICAL-ANALYSIS PROGRAM

Martin Kay and Gary R. Martins

PREPARED FOR:
UNITED STATES AIR FORCE PROJECT RAND

The RAND Corporation
SANTA MONICA - CALIFORNIA

THIS DOCUMENT HAS BEEN APPROVED FOR PUBLIC RELEASE AND SALE; ITS DISTRIBUTION IS UNLIMITED.
MEMORANDUM
RM-6265/2-PR
APRIL 1970

THE MIND SYSTEM:
THE MORPHOLOGICAL–ANALYSIS PROGRAM
Martin Kay and Gary R. Martins

This research is supported by the United States Air Force under Project RAND—Con-
tact No. F44620-67-C-0045—monitored by the Directorate of Operational Requirements
and Development Plans, Deputy Chief of Staff, Research and Development, Hq USAF.
Views or conclusions contained in this study should not be interpreted as representing
the official opinion or policy of Rand or of the United States Air Force.

DISTRIBUTION STATEMENT
This document has been approved for public release and sale; its distribution is unlimited.
PREFACE

This is one of a series of papers describing the design, implementation, and use of the MIND* system. The design goals for the MIND system are responsive to the increasingly urgent need for a means of fast and accurate information transactions between relatively senior command, control, and policymaking personnel, on the one hand, and very large, heterogeneous, loosely-formatted information banks on the other. The system is an unobtrusive servant; it understands, acts upon, and replies with, English sentences; its users will require no special training. It is thus a prototype for a class of systems that are well suited to the critical task of unifying, controlling, and exploiting the massive and often chaotic flow of information that centers upon the senior command levels of the Air Force and other services, especially in emergency situations.

The MIND system consists of nested and chained modules of high level programming language statements, and it is therefore relatively easy to modify, either for improvement or for adaptation to specialized applications.

*Management of Information through Natural Discourse.
SUMMARY

The ANALYZE module of the MIND information system is described in detail. This part of the system receives English sentences as input, segments these into words and reduces the words to their ultimate lexical components through morphological analysis. Dictionary entries providing morphological, syntactic, and semantic information about the components are retrieved and amalgamated into word-level grammatical representations. These latter are then concatenated to form a bottom-level grammatical representation of the input sentence. This representation is passed to the PARSE module for further processing.

Significant developments incorporated in the module include: (1) a sophisticated morphological analysis procedure, (2) a very efficient dictionary referencing organization, (3) powerful techniques for representing lexical generalizations, and (4) simple but effective means for morphological recombination of lexical components.
CONTENTS

PREFACE .................................................. iii

SUMMARY ................................................... v

Section
1. THE MORPHOLOGICAL ANALYZER ......................... 1
   1.1 What Is a Word? .................................. 1
   1.2 The Parts of the ANALYZE Module ................. 4
   1.3 Initial Segmentation of Input ..................... 5
   1.4 The Rule Table ................................... 8

2. THE DICTIONARY REFERENCE STRATEGY .................. 19

3. DICTIONARY CONTENTS AND MODULE OUTPUT ............ 45
   3.1 Introduction .................................... 45
   3.2 The Dictionary Entry ............................. 49
   3.3 Construction of the Initial Sentence Graph ...... 57
      3.3.1 Lexical Processing .......................... 57
      3.3.2 The Initial Syntactic Graph ............... 77
1. THE MORPHOLOGICAL ANALYZER

1.1. What Is a Word?

There are many different entities that we call "words" and, while the ambiguity does not disturb us in everyday situations, it is important in linguistic theory and when designing computer programs to distinguish one kind of entity from another. We may, for example, say that the first sentence in this paragraph is thirty-nine words long. This means simply that it contains thirty-nine strings of letters separated from one another by spaces and marks of punctuation. But it is equally true to say that a sentence contains thirty-seven words because "and" and "in" each occur twice. The definition of a word is the same as before, but we are interested now in how many different sequences of letters there are, and not in how many times each one occurs. In the first case we are counting the number of tokens in the sentence, and in the second, the number of types.

A word can be defined as a sequence of sounds instead of letters, and, while the number of tokens in a text should be the same according to both definitions, the numbers of types can reasonably be expected to be different because words with the same pronunciation are frequently spelled differently and two different spellings can represent the same sequence of sounds. If words are defined by their spelling, then "sea" and "see" will count
as two types whereas if words are defined by sounds they
will be a single type. On the other hand, the single
spelling type "read" corresponds to two phonetic types
according as it is interpreted as a present or past tense
verb.

Whereas the text may contain several tokens of the
same type, we normally think of a dictionary as containing
just one token of each type available in the language.
But, in reality, a relatively small proportion of the
types represented in a typical text can be found in a
standard dictionary because the dictionary maker defines
a word in yet another way. The tokens "walk", "walks",
"walked" and "walking" belong to four different types,
but only the first of them has an entry in the dictionary.
The entry contains an indication that this is a regular
weak verb and that other words can therefore be derived
from it by adding "s", "ed", and "ing". Since English
contains a large number of regular weak verbs, this device
makes it possible to reduce the over-all size of the dic-
tionary considerably. However, it considerably complicates
the process of identifying the dictionary entry correspond-
ing to a given token found in the text. Linguists often
refer to those types that appear in the dictionary in
exactly the same form as in the text as free stems, as
opposed to bound stems which are entered in the dictionary
in a form never encountered in text.
Looking once again at the first sentence of the first paragraph we see a number of words like "there", "different", and "ambiguity" that are free stems. On the other hand "words", "does" and "designing" are examples of words consisting of a stem followed by an affix.

The process of finding dictionary entries to correspond to the words of a text is further complicated by the fact that the form of a stem is sometimes changed slightly when affixes of a certain kind are added. The sentence we have been considering contains, for example, the word "entities" in which the final "y" of the stem has been changed to "ie". This does not make the word an exception because there is a morphographic rule which operates in English whenever certain affixes are added to a word ending in "y" and there is therefore no cause to make a separate entry in the dictionary for a word like "entities". Similarly there is a morphographic rule which doubles last consonants of certain verbs before the endings "ed" and "ing" are added.

The majority of dictionary look-up or word-recognition programs written in the past have paid no attention to morphophonemic rules but have relied on a dictionary in which each variant of every stem appears as a separate entry. Thus, for example, there would be an entry for "ripp" as well as "rip", the former being marked as occurring only before the endings "ed" and "ing". Where the
verb "try" requires only one entry in the standard printed dictionary, a dictionary of this special kind might also contain an entry for "tri" or "trie" according as the third person singular ending and the past tense ending are thought of as "es" and "ed" or "s" and "d". It is clear that a program that took account of the more productive morphophonemic rules in English, while it might work more slowly than its simpler predecessors, would require a dictionary both substantially smaller and a great deal easier to construct and modify. Accordingly, in the program described here, an attempt is made to provide a powerful and convenient formalism for stating morphophonemic rules.

1.2. The Parts of the ANALYZE Module

A morphological-analysis or dictionary look-up program falls naturally into three parts: (1) a routine that applies segmentation and morphophonemic rules to individual words and calls for certain letter sequences that it identifies to be looked up in the dictionary, (2) a routine that searches the dictionary for a given string of letters and either retrieves the information stored with that string or declares that it is not in the dictionary, and (3) a routine that surveys the record of rules applied to a word, and information obtained from the dictionary about parts of it, and constructs a coherent statement of its syntactic and semantic status to appear in the final output of the
program. The design of the ANALYZE module provides for a large measure of independence between these three principal functions, not only because they are indeed logically separate, but because theoretical or practical considerations that have not yet arisen may very well make it desirable to replace or severely modify one of these sections, and we want to be able to do this without rewriting the whole module.

The operation of the first routine is directed by a rule table, the details of which will be described below. Section 2 will discuss a dictionary reference structure and the more interesting parts of the algorithm used for consulting it. Section 3 will describe the contents of the dictionary and the procedures that the third major section of the program will use in constructing the initial surface syntax representation of an input sentence.

1.3. Initial Segmentation of Input

An input sentence appears, to the system, to consist of an unbroken string of characters bounded by machine-recognizable clues denoting the beginning and end of the string. Before applying the morphological analysis procedures, we must first break the input string up into manageable pieces. This initial segmentation procedure in fact embodies an operational definition of the notion "word" as it will be used within the system. While this definition (like all definitions) may be determined
arbitrarily, to some extent at least, there is much to be gained from keeping it as elementary as possible, putting the burden of refinement on the more powerful and sophisticated procedures that follow. Accordingly, the input string is segmented as follows:

(a) at every space character; the space character is discarded
(b) before "'X" (apostrophe X), where "X" stands for the class of characters which includes "s" and all non-alphabetic characters
(c) before and after all punctuation (including hyphen but not apostrophe).

By this procedure, the input sentence "THEY DON'T KNOW THE RULE—TABLE AUTHOR'S NAME, DO THEY?" is segmented as follows (words are separated by "/":

/THEY/DON'T/KNOW/THE/RULE—/TABLE/AUTHOR/'S/
NAME/,/DO/THEY/?/

The individual words of the sentence, as thus determined, become the primary input units to the main morphological procedures.
<table>
<thead>
<tr>
<th>Name</th>
<th>Match</th>
<th>Cut</th>
<th>Add</th>
<th>Dictionary</th>
<th>Next Rule Match OK</th>
<th>Undo</th>
<th>Affix Entry</th>
<th>Next Rule, No Match</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>UN</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>XY</td>
<td>0</td>
<td>PNEG</td>
<td>0</td>
</tr>
<tr>
<td>XX</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>A2</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XY</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>A2</td>
<td>1</td>
<td>0</td>
<td>A2</td>
<td></td>
</tr>
<tr>
<td>A2</td>
<td>-#S</td>
<td>0</td>
<td>0</td>
<td>A3</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A3</td>
<td>-ES</td>
<td>0</td>
<td>0</td>
<td>A4</td>
<td>0</td>
<td>0</td>
<td>A7</td>
<td></td>
</tr>
<tr>
<td>A4</td>
<td>*SES</td>
<td>-1</td>
<td>1</td>
<td>A6</td>
<td>0</td>
<td>0</td>
<td>NPV3</td>
<td>A5</td>
</tr>
<tr>
<td>A5</td>
<td>*HHES</td>
<td>-1</td>
<td>1</td>
<td>A6</td>
<td>0</td>
<td>0</td>
<td>NPV3</td>
<td>B5</td>
</tr>
<tr>
<td>A6</td>
<td>-1</td>
<td>1</td>
<td>A7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A7</td>
<td>-ESS</td>
<td>0</td>
<td>0</td>
<td>A8</td>
<td>0</td>
<td>0</td>
<td>B7</td>
<td></td>
</tr>
<tr>
<td>A8</td>
<td>-RESS</td>
<td>-4</td>
<td>-OR</td>
<td>1</td>
<td>A9</td>
<td>0</td>
<td>NFEM</td>
<td>B0</td>
</tr>
<tr>
<td>A9</td>
<td>-2</td>
<td>-ER</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B0</td>
<td>-NESS</td>
<td>-4</td>
<td>0</td>
<td>B1</td>
<td>0</td>
<td>0</td>
<td>ADJN</td>
<td>B2</td>
</tr>
<tr>
<td>B1</td>
<td>B3</td>
<td>0</td>
<td>0</td>
<td>B2</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B2</td>
<td>E3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B3</td>
<td>-I</td>
<td>-1</td>
<td>-EY</td>
<td>1</td>
<td>B4</td>
<td>0</td>
<td></td>
<td>B8</td>
</tr>
<tr>
<td>B4</td>
<td>-2</td>
<td>-Y</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B5</td>
<td>-IES</td>
<td>-2</td>
<td>0</td>
<td>B3</td>
<td>0</td>
<td>0</td>
<td>NPV3</td>
<td>B7</td>
</tr>
<tr>
<td>B6</td>
<td>B3</td>
<td>0</td>
<td>0</td>
<td>C2</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B7</td>
<td>-1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>NPV3</td>
<td></td>
</tr>
<tr>
<td>B8</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B9</td>
<td>-#LY</td>
<td>-2</td>
<td>-L</td>
<td>1</td>
<td>C1</td>
<td>0</td>
<td>ADVB</td>
<td>C2</td>
</tr>
<tr>
<td>C1</td>
<td>-L</td>
<td>0</td>
<td>-L</td>
<td>1</td>
<td>D5</td>
<td>0</td>
<td></td>
<td>E4</td>
</tr>
<tr>
<td>C2</td>
<td>-ITY</td>
<td>-3</td>
<td>-E</td>
<td>0</td>
<td>C3</td>
<td>0</td>
<td>ADN2</td>
<td>0</td>
</tr>
<tr>
<td>C3</td>
<td>-ØS</td>
<td>0</td>
<td>-E</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C4</td>
<td>-#ING</td>
<td>-3</td>
<td>0</td>
<td>C5</td>
<td>0</td>
<td>0</td>
<td>VING</td>
<td>DO</td>
</tr>
<tr>
<td>C5</td>
<td>-*V</td>
<td>0</td>
<td>1</td>
<td>D4</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C6</td>
<td>-*K$</td>
<td>-1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C7</td>
<td>-<em>C</em>C</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C8</td>
<td>-<em>V</em>V*C</td>
<td>0</td>
<td>-E</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C9</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D0</td>
<td>-#ED</td>
<td>-1</td>
<td>1</td>
<td>D3</td>
<td>0</td>
<td>0</td>
<td>PASD</td>
<td>D7</td>
</tr>
<tr>
<td>D1</td>
<td>B3</td>
<td>0</td>
<td>0</td>
<td>C6</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D2</td>
<td>-#ESS</td>
<td>-3</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>NFEM</td>
<td>0</td>
</tr>
<tr>
<td>D3</td>
<td>-1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D4</td>
<td>-Y</td>
<td>-1</td>
<td>-IE</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D5</td>
<td>-1</td>
<td>-E</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D6</td>
<td>-1</td>
<td>-LE</td>
<td>1</td>
<td>D7</td>
<td>0</td>
<td>0</td>
<td></td>
<td>B2</td>
</tr>
<tr>
<td>D7</td>
<td>-ABLE</td>
<td>-4</td>
<td>1</td>
<td>D8</td>
<td>0</td>
<td>0</td>
<td>ABLE</td>
<td>E1</td>
</tr>
<tr>
<td>D8</td>
<td>B3</td>
<td>0</td>
<td>0</td>
<td>D9</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D9</td>
<td>-*K$</td>
<td>-1</td>
<td>1</td>
<td>D5</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E1</td>
<td>-IBLE</td>
<td>-4</td>
<td>1</td>
<td>E2</td>
<td>0</td>
<td>0</td>
<td>ABLE</td>
<td>0</td>
</tr>
<tr>
<td>E2</td>
<td>-S</td>
<td>-1</td>
<td>-D</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E3</td>
<td>-#LESS</td>
<td>-4</td>
<td>0</td>
<td>B3</td>
<td>0</td>
<td>0</td>
<td>LESS</td>
<td>D2</td>
</tr>
<tr>
<td>E4</td>
<td>E3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fig. 1
1.4. The Rule Table

A rule table (Fig. 1) contains one line for every rule, or instruction to the analysis program, and nine columns. The columns of the table fall naturally into four groups; column 1 contains a symbol identifying the rule, column 2 specifies the test to be made, columns 3 through 8 say what is to happen if the test is successful, and column 9 says what is to happen if it fails. The first (top) rule of the table is applied first.

Rule Al in the example* has no entry in column 2 and therefore no test is specified. Since the null test is always successful, the entry in column 9 for this rule has no significance. The only non-blank entries are in columns 5 and 6 and they are interpreted as follows: A "1" in column 5 calls for the word currently being analyzed to be looked up in dictionary number one. In this example we assume that there is only one dictionary, but, in general, there could be any number, and the appropriate one would be called into use by a rule with the corresponding number in column 5. If the word being analyzed is not found in the dictionary, the entry in column 6 specifies the rule

*The example in Fig. 1 is intended to be formally valid; little attention has been given to the content of the rules and no claims can be made for their exhaustiveness, nor even for their accuracy and consistency; the development of defensible rules is the object of research now in progress.
to be applied next, in this case, rule XY. Except in one special case, to be discussed shortly, the analysis terminates in failure whenever control is directed to rule zero by an entry in column 6 or 9.

Rule A2 contains an entry in column 2 consisting of a hyphen followed by a string of characters; in this case the string consists of "##s". The initial hyphen marks the string as a suffix (in rule XX the final hyphen indicates that "un" is a prefix). The special character "#" matches any non-blank character and is used to prevent the application of this rule to words like "as", "is", "was", "his", etc. If the word is four or more characters long, the last character will be examined and, if it is an "s", the test succeeds; otherwise it fails. The meaning of rule A2 can be summarized as follows: if the current word is four or more characters long and ends with an "s", then apply rule A3 next, otherwise apply rule B9.

Rule A3 is very similar to rule A2. If the current word ends in "es" then apply rule A4 next, otherwise apply rule A7.

Rule A4 determines whether the current word ends with any one of three different suffixes. Whenever an asterisk appears before a letter in column 2, that letter is interpreted as representing a class of one or more letters
which the author of the rule table must provide. Five such classes are referred to in Figure 1, namely in rules A4, A5, C5, C6, C7, C8, and D9. The definitions of the classes are as follows:

\[
\begin{align*}
*H &= \{c, s\} \\
*V &= \{a, e, i, o, u, y\} \\
*C &= \{b, c, d, f, g, h, j, k, l, m, n, p, q, r, s, t, v, x, z\} \\
*K &= \{b, d, f, g, k, m, n, p, r, t\} \\
*S &= \{s, x, z\}
\end{align*}
\]

Rule A4 therefore tests the current word for one of the three suffixes, "ses", "xes" or "zes". If one of these suffixes is found, then the entry for rule A4 in column 3 calls for the final letter to be removed from the word, and the entry in column 5 says that the remainder is to be looked up in dictionary number one. If a matching dictionary entry is found, then this will be the last rule applied to the current word; if no entry is found, then rule A6 will be applied next. If the word does not end with one of the three suffixes specified, then rule A5 will apply next. Rule A5 is similar to rule number A4; it looks for one of the two suffixes "ches" or "shes". If either one is found, the final "s" is removed and the remainder of the rule is looked up in the dictionary. If the dictionary lookup called for in rule A4 or A5 fails, then rule A6 is applied; one further letter, namely an "e", is removed from the string, which is then referred
to the dictionary again.

Rules A4, A5 and A6 are a statement of the following important facts about English morphophonemics. Nouns and verbs ending in a sibilant add "es", instead of simply "s", to give the corresponding plural and third-person singular present forms respectively. However, nouns and verbs ending in an "e" simply add an "s", so that it is, in general, not possible to tell, without referring to the dictionary, whether a noun or third-person singular, present verb form ending in a sibilant plus "es" contains a stem ending with a sibilant or "e". Thus, for example, the stem in "buses" ends with the sibilant "s" whereas the stem of "fuses" ends with "e". Similarly, the word "avalanches" contains a stem ending in "e" whereas nearly every other word in English whose final four letters are "ches" contains a stem in "ch". English probably contains no stems ending in "she" but, if there were any, their plurals or third person forms would certainly end in "shes" and not in "shs", so that the rules would identify them properly.

Rules A4 and A5 have non-blank entries in column 8, namely ".NPV3". This is the name of one of a special set of dictionary entries which specify the grammatical properties of the affixes recognized by the morphological procedures. The entries mentioned here are identical with ordinary entries in form and function. They are "special"
only in that replicas of them are stored as a group along with the hardware image of the morphological rule table, to save disc access time. The entry named "NPV3" is associated with the English suffix ("s" or "es") that forms the plural of nouns and the 3rd person singular of verbs—that is, with the suffix which is recognized by rules A4, A5, B5 and B7 (in conjunction with other rules). Whenever a morphological rule having a non-blank entry in column 8 applies to a word under analysis (i.e., when the specified matching test is successful), the name of the affix dictionary entry is added to a list of such names which is associated with the word. When the analysis of the word is complete, the listed affix dictionary entries are retrieved and stored, along with the dictionary entry for the stem, for use in the construction of word-level syntactic segments (see Section 3). At the conclusion of a successful analysis of the word "restlessness", for example, dictionary entries named "REST" (stem), "LESS" (rule E3), and "ADJN" (rule B0) should have been retrieved and stored, in the given order, for further processing.

Now let us consider how a plural noun or a third person singular verb would be recognized if the underlying stem did not end in a sibilant. In this case the test in rule A3 would fail and rule A7 would be applied next. Rule A7 applies only to words ending in "ess" and, as we have already seen, there are no regular plural nouns or third person forms that
have this ending. The rule therefore fails. The next one to be applied is rule B7, which removes the final "s", looks the remainder of the word up in the dictionary, and points to the affix dictionary entry "NPV3".

We have already remarked that the final "y" of an English stem is often replaced by "i" before certain affixes are added. Examples are: "entities", "prettiness", "merciless", "happily", "tried", "beautiful", "ionlier" and "multiplier". If the word ends in "ey", then the "e" is removed as well, as in "monies", "moniless", "burliness", etc. This variety of cases makes it appear that the rules that embody these facts will have to be repeated in many different places in the rule table. To avoid this, a mechanism is provided whereby a single entry can be made to stand for a group of other entries appearing explicitly elsewhere. The mechanism resembles what computer programmers refer to as a subroutine. The single rule that represents a group of other rules is said to call the subroutine.

In Figure 1, rules B3 and B4 constitute a subroutine which is called in lines B1, B6, D1 and D8. Rule B3 is successful if the final letter of the current string is an "i" which, when replaced by "ey", causes the result to be found in the dictionary. If this fails, then rule B4 causes the final pair of letters, necessarily "ey", to be replaced by "y" and the result to be looked up in the dictionary. Rule
B4 contains a 0 in both column 6 and column 9, meaning that, if the lookup fails, then so does the subroutine as a whole. When a subroutine fails (i.e., is directed to proceed to rule zero), control passes back to the invoking rule. Control continues via column 6 of the invoking rule if at least one of the rules in the subroutine was successfully applied. Control passes on through column 9 of the invoking rule if none of the rules in the subroutine could be applied to the word under analysis.

Consider now how the rules shown in Figure 1 apply to the word "smallness". The word is presumably not in the dictionary, so that rule A2 is eventually applied, the final "s" is identified, and rule A3 is applied next. Rule A3 fails and rule A7 tentatively identifies "ess" as the suffix used to form feminine nouns such as "poetess". Rule A8 is tried next and fails, and then rule B0 would succeed. Rule B0 identifies "ness" as a suffix which forms nouns from adjectives as in "smallness". Since these four characters include the three tentatively identified as a suffix before, the new identification overrides the previous one. Rule B0 causes the last four letters to be removed so that the word under examination is now "small". The next rule is B1, which contains in column 2 a rule name (B3) instead of a hyphen and a string of characters. The name is therefore taken as identifying the first rule in a subroutine to be called at this point.
Rule B3 fails because the last character of "small" is not "i" and attention is therefore directed to rule B8. Rule B8 causes the word to be looked up in the dictionary where it is presumably correctly identified.

Consider now a more complicated example, namely "harmlessness". The last four letters of this word are recognized in rule B0 as before, and rule B1 calls the subroutine. Rule B3 fails to find an "i" at the end of "harmless" and the lookup called for in rule B8 fails to find "harmless" in the dictionary. Since there is a zero in column 6 for rule B8, the subroutine fails and we therefore return to the point from which it was called, namely rule B1, and take the direction given in column 6. Rule B2 calls another subroutine at E3 which directs us to look for a final "less"; this is identified and removed from the word, leaving "harm". Rule B3 is once again applied, and the stem of the word is finally identified in rule B8.

Notice that rules B3 and B8 are used twice in the analysis of this word, first as part of a subroutine, and second as part of the main sequence of rules.

We have noted that there are morphophonemic rules in English that double certain final consonants before the endings "ing" and "ed" as well as elsewhere. The consonants to which these rules apply are those defined earlier as belonging to the class "*k" (see page 12). To make it
easy to write rules that recognize these doubled consonants a special device is provided and this is exemplified in rule C6. The symbol "§" matches the second of a pair of identical letters. Notice that, while "*K*K" would match any pair of letters both of which belonged to the class "*K", "*K§" will only match a pair of identical letters from that class. "*K§§" would match a sequence of three identical letters, if this proved useful.

Column 7 of the rule table has so far escaped our attention, largely because it is effectively used in only one rule of the sample set of rules (i.e., in rule XY, near the top of the table). A nonzero entry in this column is interpreted as an instruction to "undo" all modifications that may have been made to the word under analysis or to the affix dictionary entry list during the application of the current rule (which may, of course, include subroutines called by the rule), before proceeding to the next rule via column 6. Continuation via column 9 implies that no changes were made to the word during the application of the rule. Thus, a rule which removes an affix and conducts an unsuccessful dictionary search would restore the affix in question before passing control to the next rule. Likewise, a rule invoking a subroutine which, in turn, makes numerous changes in the word under analysis, is required to restore the word to the form in which it was received before passing control, if the rule contains a nonzero entry in column 7.
Rule XY then has the following effect: (1) it calls for the application of a subroutine of rules, commencing with rule XX (which tests for the prefix "un"), (2) upon return from an unsuccessful analysis, it restores the word to its original form and passes control to the suffix analysis routines beginning with rule A2. The English word "uncles" would thus be first reduced to "cles", then to "cle" before the subroutine terminated in failure at rule B7. On return to rule XY the word is restored to "uncles" and the analysis continues, resulting in a presumably successful dictionary search during the second visit to rule B7. The main effect of this mechanism is to reduce the logical burden of rule ordering and to permit freer use of subroutines of rules.

In Figure 2 the analyses of a number of words are illustrated in summary form. The illustrations consist of strings of words and rule names, which, read from left to right, show the sequence of rules encountered and the changing shape of the word under analysis. A rule name preceded by "*" indicates that the rule could not be applied. A word preceded by "*" indicates an unsuccessful dictionary search. Subroutines of rules are parenthesized. Note the possibility that a successful subroutine may not "return". Names of dictionary entries retrieved for the word are given, enclosed in slashes, at the point of reference.
1. COUNT A1 /COUNT/
5. COUNTLESSNESS A1 *COUNTLESSNESS XY (*XX) A2 *A3 A7 *A8 B0 COUNTLESS /ADJN/ B1
    (*B3 B8 *COUNTLESS) B2 (E3 COUNT /LESS/ *B3 B8 /COUNT/)
7. RUNNING A1 *RUNNING XY (*XX) *A2*B9 C4 RUNN /VING/ *C5 C6 RUN /RUN/
8. UNTIED A1 *UNTIED XY (XX TIED *TIED /PNEG/ *A2 *B9 *C4 DO TIE /TIE/ /PASD/
9. UNCLE S A1 *UNCLES XY (XX CLES *CLES /PNEG/ A2 A3 *A4 *A5 *B5 B7 CLE *CLE/ /NPV3/)
    (see NOTE below) UNCLE S A2 A3 *A4 *A5 *B5 B7 UNCLE /UNCLE/ /NPV3/.
10. COUNTABLY A1 *COUNTABLY XY (*XX) *A2 B9 CO COUNTAB *COUNTAB /ADVB/ *C1 E4
    (*E3 *D2) D6 COUNTABLE D7 COUNT /COUNT/ /ABLE/
11. PITiable A1 *PITiable XY (*XX) *A2 *B9 *C4 *DO D7 PIDI *PITI /ABLE/
    D8 (B3 PITEY *PITY B4 PITY /PITY/}
12. DEFENSIBLE A1 *DEFENSIBLE XY (*XX) *A2 *B9 *C4 *DO *D7 E1 DEFENS *DEFENS /ABLE/
    E2 DEFEND /DEFEND/

Fig. 2

NOTE: At this point the nonzero entry in column 7 of rule XY causes restoration of the
word and deletion of the affix dictionary references made during the subroutine.
2. THE DICTIONARY REFERENCE STRATEGY

Suppose that the entries in the dictionary are stored in alphabetical order, and that somewhere the following sequence of words appears:

pleurisy
pliers
plum
plumage
ply
pneumatic

Suppose, furthermore, that the word being analyzed is "pliably". The first reference to the dictionary shows not only that there is no entry for this word, but also that if it were in the dictionary it would appear between "pleurisy" and "pliers". Of these two words the second shares a greater number of initial characters with the word being analyzed, namely three.

Suppose now that the rule table causes the final suffix to be replaced by "e" and calls for the resulting word "pliable" to be looked up in the dictionary. The first six letters of this word have not been changed since the last dictionary reference, when the longest matching initial string contained only three letters. Since six is greater than three, we know that the attempt to find "pliable" in the dictionary will have exactly the same result as before. The computer routine can there—
fore announce a failure without having to consult the dictionary itself at all.

The rule table will presumably go on to remove the suffix "able" and to replace "i" by "y". When the word "ply" is referred to the dictionary, the program will notice that only two letters of the longest initial string still remain intact, whereas the longest initial string previously matched contained three letters. Since two is less than three, the possibility that the word is in the dictionary cannot be ruled out as before; a search is instituted and the word is correctly identified.

As a second example, consider the word "plums". The first reference to the dictionary shows that there is no entry for this word but that its proper place would be between "plumage" and "ply". This time, however, notice that it is the earlier word, "plumage", which shares the longest string of initial letters with the word being analyzed. The program records the length of this initial match as four. The rule table removes the final "s" and the word "plum" is sent to the lookup routine. A string of four initial letters remains intact since the last reference, and this is exactly the number of letters in the previous longest match. The word may therefore be in the dictionary and a search must be conducted.
We have seen that at least some dictionary searches can be avoided if, when a search is unsuccessful, the length of the longest matching initial string is recorded. We shall now see that the work involved in the searches themselves as well as the amount of space occupied by the dictionary can be reduced by changing the way in which the words in the dictionary are represented.

An alphabetical list of words usually contains a great deal of redundant information. All the words that begin with the same letter are collected together in one place and it would therefore be possible to divide the list into sections, each of which contained all the words beginning with a given letter. If the section were labelled with that letter, then it would no longer be necessary to record the initial letter of each individual word explicitly. Thus, for example, the section labelled "p" might contain the following entries:

leūrisy
liers
lum
lumage
ly
neumatic

Each of these sections, however, is itself an alphabetical list of entries containing redundancies of exactly the
same kind as those found in the original dictionary. The space required to store the list can therefore be further reduced by dividing the sections into sub-sections so that the entry for the letter "p" would be rewritten as follows:

1. eurisy
   iers
   um
   umage
   y
   n
   eumatic

With such a small example as this, the benefits of making this subdivision are questionable; but a cursory inspection of the real dictionary makes it clear that a great deal is to be gained by dividing sections into subsections, subsections into sub-sub-sections, and so on.

These arguments suggest that it may be profitable to organize the words in a dictionary in the manner suggested by the following diagram:

```
A → A → R → D → V → A → R → K
   ↓        ↓        ↓
  B → A → C → K
      ↓      ↓
     E → T
     ↓      ↓
    L → E
     ↓
   C → A → D → E → M → I → Y
      ↓      ↓
     C
      ↓      ↓
    B → A → A → L
```

Fig. 3a
The letters with which words in the language can begin are listed down the left hand side of the diagram and are connected to one another by a series of vertical arrows. A horizontal arrow from each of these letters points to the section of the dictionary containing words with that initial letter. All the letters that can follow a given initial letter are listed in a similar vertical column, the first one in the top left-hand corner of the section. Each of these, in its turn, points to a subsection, and so on.

To identify a given word, say, "abet", begin in the top left hand corner of the diagram and compare the first letter of the word with the letter shown there. If the letters do not match, follow the vertical arrow and compare the first letter of the word with the letter it points to. If, as in the case of "abet", the letter does match, then follow the horizontal arrow and compare the next letter in the word with the letter that arrow points to. Since the second letter of "abet" is not "a", follow the vertical arrow and compare the second letter with "b". Since there is a match this time, follow the horizontal arrow and compare the third letter in the word with "a". Since there is now no match, follow the vertical arrow once again and compare the third letter with "e". Since this match is successful, follow the horizontal arrow and compare the last letter of the word with "t". At
this point the position of "abet" in the dictionary has been identified.

The diagram corresponding to our earlier example is as follows:

```
  P → L → E → U → R → I → S → Y
  ↓    ↓    ↓    ↓    ↓    ↓
  I → E → R → S
  ↓    ↓
  U → M* → A → G → E
  ↓
  *Y
```

```
  N → E → U → M → A → T → I → C
```

Fig. 3b

In this case we have used an asterisk to mark explicitly those places in the diagram corresponding to the recognition of a complete word. This is necessary to account for cases in which a short word constitutes an initial string in some longer word as "plum" does in "plumage". In recognition of the fact that "plum" is itself a complete word, the final "m" is marked with an asterisk in the diagram.

Let us now consider once again the steps leading to the recognition of the word "pliably". The initial "p" is compared with each of the letters in the alphabet, in turn, until it is eventually found to match the one shown in the top left hand corner of the above diagram. The horizontal arrow from here points to the letter "l" which matches the second letter in the word. The horizontal
arrow from here points to "e" which does not, however, match the third letter of "pliably". We therefore follow the vertical arrow and find a match for the third letter in the word. The horizontal arrow from here does not point to an "a" and, since there is now no vertical arrow to follow, the attempt to find the word in the dictionary has failed. However, we record the fact that the first three letters were successfully matched and the three positions in the diagram at which the matching letters were found.

As before, the rule table recognizes the final "ly" of "pliably" and orders a search of the dictionary for "pliable". But, since the first six letters of the word remain unchanged by the rule table and only three letters were matched on the last attempt, the second search need not in fact be carried out. The rule table now recognizes the final "able" of "pliable" and refers the stem "ply" to the dictionary. As before, the program notes that only two of the three originally matched letters are still intact, so that the possibility of finding the stem in the dictionary cannot be abandoned without a search. However, the search need not be conducted from the beginning as though nothing were known about the word. Records kept from the first attempt show where in the diagram the first two letters, which still remain unmodified, were matched. The search can therefore be taken up at that point. The
next step is to compare the final "y" of "ply" with "e". It will subsequently be compared with "i" and "u" before the word is correctly recognized.

The technique just described is particularly useful with a language like English in which words are modified more frequently by suffixes than by prefixes. Clearly, whenever the rule table changes the initial letters of the words being analyzed, it will be necessary to go through all the steps of the search process starting with the first letter. With a predominantly prefixing language the stems can be arranged in the dictionary in alphabetical order from right to left and the exact mirror image of the process just described can be used just as effectively.

In the dictionary-search diagrams just described, alternative characters for a given position within a word are chained together by vertical arrows, while contiguous sequences of characters are chained together by horizontal arrows. For simplicity of manipulation within the computer we reduce this two-dimensional arrangement to a simple linear form with the help of the following two conventions: (1) horizontal (sequential-chaining) arrows are replaced by contiguity within the address structure of the computer, and (2) vertical (alternative-chaining) arrows are replaced by relative displacement pointers to other parts of the list. In this way, Fig. 3b is transformed into Fig. 3c.
A new piece of information, here symbolized by "/", has been introduced to mark the endpoints of sequential chains; these endpoints will always occur at the end of a word, marked by an asterisk above, in the dictionary, but the reverse is not always true (cf. the end of the word "plum" above).

Generally speaking, the elements of alternative-character chains will not be systematically ordered, and thus no systematic "penalty" in terms of search time is associated with particular characters of the alphabet. This is a consequence of the dictionary building and maintenance programs whose functions are briefly outlined a bit later in this section. If true randomization of alternative-character chains were achieved, and if English text were perfectly homogeneous, that is, if all letter sequences constituted valid English words, we would expect (assuming an alphabet of 30 characters) that about 14.05 jump-and-compare operations would be needed in order to match each character of a given word. Of course, English text is far from homogeneous, a characteristic we will exploit in a manner to be described in a moment, but it is sufficiently dense in the initial character positions of
words to make the average character-search time for these positions relatively high. In other words, the number of actually occurring initial letter and digram sequences in English is a relatively high fraction of all typographically possible sequences. But the proportion of actual initial $n$-gram sequences falls off very sharply as $n$ increases.

To speed things up, the first two characters of each dictionary entry are removed and the remainder of the word is placed in a structure of the kind shown in Figure 3c. Each of the first two characters is mapped uniquely onto an integer, yielding a pair of subscripts by which to enter a 30 x 30 array called FIRST_PTR of relative pointers into the main dictionary structure. Thus, the word "plumage" is split into "pl" and "umage". This latter string is entered in the main dictionary structure in the way already described. The characters "p" and "l" are converted to subscripts (say, 18 and 12, respectively), and a relative pointer to the first entry in the main dictionary lists belonging to a word beginning with "pl" is stored at FIRST_PTR(18,12). Thus, where English text is relatively dense we provide a very fast access method whose price is a small amount of wasted space. The actual amount is in inverse proportion to the density of the language for these positions. Where English is sparse, the more compact scheme illustrated in Fig. 3c effects very great savings of space. The manner in which the two schemes are combined is depicted in Fig. 3d.
It is worth noticing here that the MAIN_DICT lists are divided, for the sake of the maintenance and updating functions, into 900 (= 30 x 30) sections, each of which begins at a point indicated by one of the 900 pointers in the FIRST_PTR array. We shall be in a better position to appreciate the benefits of this arrangement for such operations as dictionary updating later. It should be immediately obvious that this fragmentation of the dictionary permits a convenient limit for the maximum size of the ALT_PTR elements in the lists, namely 10 bits. Since pointers within the lists are relative, the incremental addition of new entries to the system's dictionary involves only (1) the updating of the end of the appropriate ALT_PTR chain, (2) finding (sometimes through physical displacement of subsequent lists) space for the new sequential entry chain, (3) updating by identical increments all entries in the two-dimensional FIRST_PTR array following that entry which refers to the start of the list in question (if physical displacement of sublists has been required in step (2) above).
Fig. 3d—Dictionary reference lists
The structure of a dictionary of this kind can be represented by a PL/I declaration* as follows:

```
DCL FIRST_PTR(30,30) BINARY FIXED(15) /* FOR THE INITIAL HIGH SPEED SECTION OF THE DICTIONARY */ ,
1 MAIN_DICT (30000), /* FOR THE COMPACT PART OF THE DICT. */
2 LETTER BIT(7),
2 INCR BIT(3),
2 ALT_PTR BIT(11),
2 MARK(3) BIT(1);
```

The array FIRST_PTR will contain subscripts with which to enter MAIN_DICT. Using BIN FIXED(15) numbers provides addresses from -32K to +32K. Since the size of MAIN_DICT is only about half as great as this range, we may use the absolute values of the entries in FIRST_PTR as pointers while using positive and negative values to distinguish digram sequences which may constitute entire words ("in", "at", etc.) from those which may not ("st", "ph", etc.).

---
*This and other examples of PL/I code are intended to be illustrative only.*
Arriving via an entry in FIRST_PTR at an appropriate spot in MAIN_DICT, say, MAIN_DICT(n), we are prepared to begin testing the 3rd character of the source word against the 7-bit character representation in LETTER. A successful match will send us to MAIN_DICT(n+1) to test the 4th character, unless MARK(1) = one. This corresponds to the slash ("/") used in Fig. 3c. A mismatch will send us instead to MAIN_DICT(n + ALT_PTR) to retest the 3rd character, unless ALT_PTR = 0, indicating that all alternatives have been exhausted. And so on. If we have successfully matched the final character in the source string, then MARK (2) is tested. This corresponds to the use of the asterisk ("*" ) in Fig. 3c. If MARK(2) = one, the search is considered successful, otherwise not. Before considering the functions of INCR and MARK(3), it will be useful to develop in somewhat greater detail the basic design concepts of the dictionary.

So far we have taken a dictionary as consisting simply of an alphabetical list of words. However, we must also consider the information about these words that the dictionary contains, and that the program described in this document is intended to retrieve. We shall consider the structure of this information in some detail in the next section. For the present, it is important only to observe that this kind of information constitutes by far the greater part of any dictionary entry. It is
probably unreasonable to imagine a complete dictionary being stored in the rapid access memory even of quite a large computer. More reasonable is to imagine its being stored on a disk or some other direct-access peripheral device and to design a scheme for retrieving it when needed. The economy of computer storage is such that we can afford to be very much less particular about the amount of space required for storage on a disk than in the main memory of a computer. These considerations in mind, it may well be worth employing a device, which we shall now describe, for further reducing core storage requirements at the expense of a slightly greater amount of storage on disk.

Hitherto we have considered strategies aimed at discovering simultaneously whether a given word is in the dictionary and, if it is, its location in the dictionary. These two functions can, in fact, be separated, and on occasion it is profitable to do so. The strategy we shall now describe has two parts. Given a particular word, the first part will, in certain instances, be able to determine that the word is not in the dictionary; otherwise, it will identify a particular entry in the dictionary. If the word is in the dictionary, then the entry identified will be the correct one; if it is not, then the entry identified is of no interest. It will be the purpose of the second part of the strategy to determine whether a dictionary
entry identified in the first part is in fact correct, and it will do this by simply matching the word sought against the word found in the dictionary entry. Simply stated, we are proposing an algorithm with a first part that can discriminate among the entries in the dictionary and a second part that determined whether a positive identification has been made.

Suppose, for the sake of an example, that we are concerned with a language that uses only six letters, namely, a, b, c, d, e and i. Suppose that the dictionary contains the following words:

acid
add
base
basic
bead
bed
bid
bide
cab
dead
deed
did
ebb
ice
To simplify the exposition, we will omit the high-speed initial-position tables and represent the whole of the vocabulary by means of an array of structures of the MAIN_DICT type; this representation appears in Figure 4.

There are 36 entries in the table (the original vocabulary itself has 50 characters) requiring 3 bytes each for storage; the entire table thus consumes 108 bytes of core storage. Can this figure be further reduced to any significant degree? Consider what happens when the word "acid" is sought in this dictionary. The first character successfully matches the LETTER element in the first entry of the table. Moving on to the second character of the source word, and the LETTER element of the next entry in the table, we find that they, too, match. Similarly for the third and fourth characters. At the fourth match we find that MARK(2) is set to one, indicating a potential end of word; since the source word contains no more characters, the lookup is considered successful.

Notice that if, on the first try, we failed to match the first character of the source word against the LETTER element of the first entry in the table, we could have gone on to try to find a match further down in the table, guided by the non-zero ALT_PTR element in the first entry.
Fig. 4—The initial MAIN_DICT structure
This particular ALT_PTR element tells us, among other things, that words in the vocabulary in question can begin with various initial letters; the chain of ALT_PTR elements which begins in the first entry in the table, leading in succession to LETTER elements "B", "C", "D", "E", and "I", spells out just what the actual alternative initial characters are. Similarly, if we had failed to match the second character of the source word against the LETTER element of the second entry in the table, another nonzero ALT_PTR element points the way to alternative possibilities. But this is significantly not true of the third entry in the table, nor of the fourth, nor of numerous others further down. The implication is clear: these entries do not serve to distinguish vocabulary items from one another.

The only function of the third and fourth entries in the table is to verify the correct identification of the source word—i.e., to distinguish this particular dictionary entry from all other English words not in the given vocabulary. But we have already decided to perform this function elsewhere; the entries are superfluous, and we may eliminate them from the table.

Can we eliminate all entries with zero ALT_PTR elements? The answer is no, because the final link in a
chain of alternative entries will itself have a zero ALT_PTR element, yet it is essential. The entry corresponding to the beginning of the word "ice", for example, cannot be eliminated, without tremendously complicating the scheme, despite its zero ALT_PTR element.

Can we then eliminate every entry which is not a member of an ALT_PTR chain? For simplicity, all members of ALT_PTR chains in Fig. 4 are marked with a dollar sign ("$"). The answer here is, not quite. Consider the entry corresponding to the final "e" in "bide", marked with a heavy arrow in Fig. 4. Within the narrow confines of our sample vocabulary, this letter contrasts contextually only with the blank, to distinguish "bide" from "bid"; but initial and final blanks are not considered to be part of words, in conventional dictionaries or here, and we do not store them. The function of the entry in question is simply to indicate that there is a vocabulary item distinct from "bid" by virtue of greater length. We shall see in a moment that the word "bide" could be changed to "bidet" or "bidentate", for example, without in the slightest affecting this portion of the dictionary. Notice that this special case, which could be expected to occur fairly frequently in a full-scale dictionary is characterized by the fact that the
immediately preceding entry (for "d") within the same word has \( \text{MARK}(1) = \text{zero} \) and \( \text{MARK}(2) = \text{one} \). We now have all the information necessary to reduce the size of the table to a bare minimum.

Assuming that we wish to construct a minimal table from a full table of the kind shown in Fig. 4, we do the following:

1. mark for retention all members of ALT_PTR chains
2. mark for retention all entries immediately following entries in which \( \text{MARK}(1) = \text{zero} \) and \( \text{MARK}(2) = \text{one} \)
3. move all MARK elements from entries to be deleted upward in the table to the nearest entry marked for retention
4. reduce each ALT_PTR element by the number of entries, located between it and the entry it points to, which are to be deleted
5. in the INCR element of each entry to be retained, write the number of entries immediately above it and within the same word which are to be deleted (this has been done in Figure 4)
6. delete all entries not now marked for retention
The results of these operations are displayed in Fig. 5. This highly compact table contains 20 entries and would consume 60 bytes of storage. The average number of character comparisons required to distinguish a word is just under 4.3. While these figures are respectable, the scale of the economies of space and search time cannot be effectively illustrated with a vocabulary as small as our sample.

The purpose of the INCR elements in the table should now be clear; they serve to synchronize the matching of LETTER elements with character positions of source words. A source-word character-position counter $k$ is always incremented by INCR + 1 when moving sequentially down the table, and by INCR when jumping along an ALT_PTR chain, where the INCR in question is taken from the entry whose LETTER element is next to be matched.* It is thought to be highly improbable that situations requiring a value of an INCR element greater than 7 will arise; if this should occur, the situation can be met by the simple expedient of including an "extra" entry in the table to bridge the extraordinary span of successive deletions.

It will be seen from Fig. 4 that we will occasionally delete entries from the table which correspond to the final

---

*This will almost always be zero in ALT_PTR chains.
Fig. 5—The reduced MAIN__DICT structure
character(s) of source words (as in the entries for "acid", "basic", etc.). This requires a modification of the manner in which successful table searches terminate, described earlier as a coincidence of (1) a successful match on the final character of the source word and (2) the presence of \texttt{MARK(2) = one} in the last table entry consulted. Clearly, this is no longer sufficient. To make it easier to state the rules by which success and failure are recognized, we will make use of the following simple convention: a "mismatch" will be said to occur if (1) a source character is different from the corresponding \texttt{LETTER} element, and from all \texttt{LETTER} elements further along in the \texttt{ALT\_PTR} chain (if any), or (2) an attempt is made to continue testing beyond the end of the source word, or (3) an attempt is made to continue testing beyond the end of a dictionary chain. A dictionary search is successful if a match is made at an entry in which \texttt{MARK(2) = one} and if the very next test results in a mismatch.

It remains to discuss the function of the \texttt{MARK(3)} elements of dictionary entries. A dictionary in an information processing system with the scope of the MIND system is not a static collection of data. Provision must be made for orderly and controlled growth, and this is
especially important during the early life of the system. It may be noticed that the dictionary lists considered so far in this discussion have been properly nested in the sense that ALT_PTR chains corresponding to higher character positions are neatly fitted between members of ALT_PTR chains corresponding to lower character positions. Dictionary lists having this property are more efficient than those without it. But a high price is paid (in computational terms) for the maintenance of this property if the dictionary is to be allowed to grow through the frequent addition of small numbers of arbitrarily selected entries; not only must these dictionary lists be revised, but the corresponding main dictionary materials (see next section) must be shifted about in a similar fashion on the disk. The scheme we have adopted to accommodate dictionary growth is a compromise; provision is made for arbitrary incremental growth, in which considerable savings of computing time are achieved through the sacrifice of proper nesting, which property is then recaptured from time to time by means of wholesale rearrangements of the dictionary.

Consider the dictionary sublist

\[ D I S A B L E / T A N T / \]

\[ * \]

\[ * \]
and suppose that we wish to add the word "dim" to the
dictionary. In this case, we simply add an entry for the
character "m" to the end of the list, and supply the approp-
riate value for the ALT_PTR element corresponding to the
character "s" in the word "disable". This gives

```
  D I S A B L E/T A N T/M/
       *  * *
```

To now add the word "dismal" upset the proper nesting
property of the list, but requires no special machinery;
the method is as just described, giving

```
  D I S A B L E/T A N T/M/M A L/
       *  * *  *
```

But suppose at this point that we wish to add the word
"diminutive" to the system's vocabulary. Ordinarily, we
would prefer to rearrange the list to create the required
space immediately after the character "m" of the word
"dim". But we have already mentioned the prohibitive cost
of frequently making such adjustments in a large dictionary.
And we cannot simply provide the "inutive" part of the word
at the end of the list on an ALT_PTR chain, as we have done
with the preceding additions to the vocabulary, precisely
because we need to recognize a correct match (which implies continued sequential testing) on the character "m".

The problem is solved with the use of the MARK(3) element of the entry corresponding to the character "m". This element is set to one, the ALT_PTR element of the same entry is set to point to the end of the list, and the remainder of the word ("inutive"), preceded by a copy of the character "m", is placed there. We then have

```
DISABLE/TANT/MAL/MINUTIVE/
```

the setting of MARK(3) being indicated by the underscoring. The condition that MARK(3)=one serves notice that an additional copy of the LETTER in question will be found further along the current ALT_PTR chain and that sequential testing may be resumed at that point. The condition that MARK(3)=one implies that MARK(1)=one and that MARK(2)=one.

It remains only to show how, when a word has been discriminated, the bulk of the corresponding dictionary entry can be retrieved from a disk or other storage device. Notice that no provision is made in the tables for the address of the main dictionary entry. Since relatively few table entries correspond to places at which a discrimi-
ination is made, it would probably be wasteful to provide space for a dictionary address with every entry, especially in view of the fact that these addresses will tend to be large. Since each table entry at which a discrimination can be made corresponds to a unique dictionary entry, it seems preferable to use the computer address of the entry itself as a key to the main dictionary. Various techniques are available for doing this, and we will describe a simple one by way of illustration.

Assume that the dictionary occupies a certain number of tracks on a disk, and is arranged so that those entries with higher addresses on the disk correspond to entries with higher addresses in the discrimination tables. In other words, the dictionary entries on the disk are arranged in ascending order according to the core addresses of the table entries that discriminate them. A relatively small table can now be constructed in the rapid-access store of the computer showing, for each track on the disk, the address in the discrimination tables corresponding to the first dictionary entry it contains. Using this table, the track containing the dictionary entry for a given word can readily be obtained. The tracks themselves can also contain tables entered via a discrimination-table address and giving the offset within the track at which syntactic and semantic information for the item may be found.
syntactic graph; it is bounded by vertices on the left and on the right, which are its points of contact with the edges representing neighboring segments. It is convenient to consider each edge of the graph as oriented from left to right. Thus it is strictly more correct to replace the term "edge" by the term "arc" etc.; the choice of terms here conforms to that used in other documents on the MIND system. Two vertices of the graph may be connected by a path consisting of (1) a single edge, or (2) a sequence of adjacent edges, or they may be disjoint; two vertices may be connected by several paths, and these may differ in length (i.e., number of edges). Thus, in the following diagram vertices 2 and 3 are disjoint; all others are connected. Vertex 3 is connected to vertex 5 by a single path consisting of edges I and J. Vertices 1 and 4 are joined by paths C,H and D,I and E,I and F. Path A,C,H,J,
connecting vertices 0 and 5 is of length 4. When a pair of vertices is connected by more than a single path, then either (1) the paths are alternatives (i.e., they represent a set of syntactic ambiguities), or (2) one (or more) of the paths syntactically dominates one or more of the others (i.e., the dominating path is compatible with those it dominates, but stands at a higher syntactic level). Thus, in the diagram, it is not clear whether the path consisting of the single edge B dominates path A,C,G (among others) or is simply an ambiguous alternative to it. For our present purposes we will assume that all such cases represent ambiguities, not dominance relations.

Some unrealistically simple examples may serve to illustrate the content and interpretation of this kind of syntactic graph. The textual item "read" might be represented as follows:

```
PS: Aux
T: Present
```

```
0
```

```
1
PS: Verb
```

```
2
T: Past
```

```
PS: Aux
```

Fig. 7
Here, the two paths connecting vertices 0 and 1 characterize the ambiguity of "read" with respect to tense. The single path joining vertices 1 and 2 gives the part-of-speech unambiguously as "verb". Note that the single word "read" is here treated as two adjacent syntactic segments, corresponding to the current linguistic practice of representing verbs as Aux + V.

The word "saw" might be represented as

![Diagram](image)

Fig. 8

This representation provides three syntactic readings for "saw". As the past tense of the verb "to see" it is characterized along the path connecting vertices 0, 2, and 3 by two syntactic segments marking it a past tense, animate verb. The single-edge path connecting vertex 0 directly to vertex 3 marks the word as an inanimate noun (the instrument for cutting wood etc.). The uppermost path in the diagram describes the word as a present tense verb, indifferent as to the attribute of animacy (the act of cutting wood etc.).
3.2 The Dictionary Entry

All entries in the MIND system's dictionary consist of a preface section and a main body. The preface is made up of an identification segment, and an edge map. The main body consists of attribute-value lists. In the following we briefly describe the internal composition of these elements.

The identification segment begins each entry. It consists simply of a literal character string (LCS) preceded by a single "literal length" (LL) byte giving its length. This character string will normally reflect the English spelling of the stem or affix to which the entry corresponds and it serves to identify the entry within the MIND system. The most common function of this literal string is to confirm the results of searches through the discrimination tables which make up the first part of the dictionary (see Section 2). The length of this literal string is variable and is given by the LL (the first byte of the dictionary entry).

The edge map begins immediately following the last character of the identification segment. The first byte of the edge map (EN, "edge number") is an integer giving the number of edges in the entry. Following this there appears, for each edge in turn, a 9-byte edge schema with the following appearance:
The individual parts of the edge schema are:

**Rem** (1 byte; read "Remote"). This fixed point 7-bit integer indicates that the edge in question contains (1) no remote references (=0), (2) a remote reference of Type 1 (=1), and (3) one or more remote references of Type 2 (=no. of references).*

**LMV** (half-word; read "Left Morphology Vector"). This half-word is treated as a 16-bit logical pattern, each bit of which indicates a specific kind of morphological compatibility with edges immediately to the left.

**RMV** (half-word; read "Right Morphology Vector"). Similar to LMV, but describes compatibility with edges immediately to the right.

**F** (1 byte; read "From Vertex"). The number of the vertex from which the edge is incident—out. Thus, in Fig. 8, the nominal edge schema would carry the number "0" in this byte.

An important convention governs the contents of the F and T fields: (a) the leftmost vertex of the entry is always designated by the identification integer zero ("0"), (b) all vertices have distinct identification integers,

*See part 3.3 for a fuller explanation of new terms and notions introduced here.*
(c) the rightmost vertex has identification integer n, if the entry has precisely n+1 vertices, (d) the largest identification integer of an entry is that of the rightmost vertex (hence, n).

$T$ (1 byte; read "To Vertex"). The number of the vertex to which the edge is incident-in. Thus, in Fig. 8, the nominal edge schema would carry the number "$3" in this byte. The edge in question connects the "From Vertex" to the "To Vertex".

$Ext$ (1 byte; read "Extent"). This is a number giving the length in bytes of the data pertaining to this edge in the main body of the entry.

$Dom$ (1 byte; read "Dominated"). If the edge in question is dominated by another edge, this byte gives the number of the edge schema of the dominating edge. It is otherwise zero.

The main body of the dictionary entry begins with the byte immediately following the last edge schema and consists of attribute-value lists (AVL), one corresponding to each edge in the entry, in the same order as the edge schemas in the preface. The length of each AVL is given in the Ext parameter of the corresponding edge schema. An AVL consists, with four kinds of exceptions, of a linear
sequence of pairs of bytes where, in each pair, the first member represents the name of a grammatical attribute, and the second member represents the name of a value for the attribute. Thus our earlier usage of "PS:Noun" (See Fig. 8), should be taken to represent a pair of bytes, the first of which refers to the attribute "Part of Speech", and the second of which refers to a specific part of speech, to wit, "Noun".

There are three kinds of embedding which are permitted to disturb the linearity of an AVL; all kinds of embedding complicate the value member of an attribute-value pair. In the first and simplest case, which we shall call disjunctive embedding, an attribute is provided with an exclusive disjunction of values. In such a case, the value of the attribute in question is unresolved until all but one of the alternative values have been eliminated by the parsing procedure. Such a situation might be symbolized "X:(A,B,C)", meaning "The value of attribute X is either A, or B, or C". The ambiguity implied by this type of representation is of a quite different kind from that which is represented by multiply-connected vertices. In the latter case, we have two or more legitimate syntactic readings for the span between the vertices. In the
former case we have only one reading which is incompletely specified. See /l/ for further discussion.

A more powerful device is that which we may call subordinate embedding, in which the value of the attribute in question is itself an AVL. This situation, whose motivation (discussed in /l/) is outside the scope of this paper, may be symbolized "X:(A:B C:D)", meaning "The value of attribute X is the AVL 'A:B C:D'". Both types of embedding, it is important to note, may be carried to any depth, and they may include one another. The following is therefore a legitimate, if confusing, AVL:

The third type of embedding permits the inclusion of literal strings of arbitrary length within the AVL as values for certain attributes which may refer to grammar-rule names, special words, etc. An example is the attribute which introduces remote references of Type 2. This situation may be written "X:('RULE15')", meaning "The value of attribute X is the literal string 'RULE15'."

For certain purposes (see part 3.3, "Segment Expansion") it is convenient to represent the negation of some value for an attribute. Thus, "PS:-Noun" will be interpreted as meaning "the 'PS' attribute may not take the value 'Noun'"
here". It is also useful to represent the negation of an attribute, as in "¬PS:X"; this would be interpreted as meaning "the 'PS' attribute may not occur here". In this latter case, the value specified for the negated attribute is of no interest and is never interpreted. Negation is fully compatible with both forms of embedding discussed above.

In the physical or hardware representation of an AVL the symbolic delimiters used above (colon, NOT, comma, space, and parentheses) are eliminated. Instead, it is assumed that every pair of bytes, starting at the "left" of every AVL (embedded AVL's included), represents a simple attribute-value pair, unless the pair of bytes is a negation indicator or a disjunctive or subordinate or literal embedding indicator (here symbolized "¬", "##", "**", "###" (double quotes), respectively). Each of these indicators consists of two consecutive identical bytes in order that, in subsequent operations within the ANALYZE module and in later modules, it will be possible to scan the AVL on an attribute-by-attribute or value-by-value basis, as well as byte-by-byte, with no danger of skipping over an indicator. In the case of negation, the assumed alternation of attributes and values is resumed immediately;
the negation affects only what follows it immediately. In the case of embedding, the two-byte indicator is followed by a one-byte integer giving the span, in bytes, of the embedded material (an AVL ("**"), a string of values ("##"), a literal (""') ); beyond the span the simple alternation of attributes and values is assumed. The following examples will help illustrate these notions (each printed character of underlined string of characters represents a single machine byte).

<table>
<thead>
<tr>
<th>Symbolic Representation</th>
<th>Hardware Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>A:B C:D E:¬F</td>
<td>ABCDE¬F</td>
</tr>
<tr>
<td>X:(A,B,C)</td>
<td>X##3ABC</td>
</tr>
<tr>
<td>X:(A,B,C,D) Y:E</td>
<td>X##4ABCDYE</td>
</tr>
<tr>
<td>W:¬X Y:(A:B C:D) Z:¬(E,F)</td>
<td>W¬XY**4ABCDZ¬##2EF</td>
</tr>
<tr>
<td>W:(X,(A:C B:D),Y) ¬U:Q</td>
<td>W##9X**4ACBDY¬UQ</td>
</tr>
<tr>
<td>E:(F,G) H:(M:N 0:(P,Q, (X:Y ¬T:U),R)) K:L</td>
<td>E##2FGH<strong>18MN0##12PQ</strong>6XY¬TURKL</td>
</tr>
<tr>
<td>X:(A,(&quot;LITERAL&quot;),B,C)</td>
<td>X##13A&quot;'&quot;7LITERALBC</td>
</tr>
</tbody>
</table>

A complete dictionary entry, corresponding to the representation of the word "saw" given in Fig. 8, would appear as in Fig. 9*

*In Fig. 9 the names of the data fields are given above the data themselves. LMV and RMV fields are left blank pending further discussion (pp. 60–66). In the AVL fields, slashes (/") are used to demarcate single byte boundaries; thus, for example, AVL₃ consists of 4 bytes of data.
<table>
<thead>
<tr>
<th></th>
<th>LL</th>
<th>LCS</th>
<th>EN</th>
<th>Rem₁</th>
<th>LMV₁</th>
<th>RMV₁</th>
<th>F₁</th>
<th>T₁</th>
<th>Ext₁</th>
<th>Dom₁</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3 saw</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rem₂</th>
<th>LMV₂</th>
<th>RMV₂</th>
<th>F₂</th>
<th>T₂</th>
<th>Ext₂</th>
<th>Dom₂</th>
<th>Rem₃</th>
<th>LMV₃</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td></td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RMV₃</th>
<th>F₃</th>
<th>T₃</th>
<th>Ext₃</th>
<th>Dom₃</th>
<th>Rem₄</th>
<th>LMV₄</th>
<th>RMV₄</th>
<th>F₄</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>T₄</th>
<th>Ext₄</th>
<th>Dom₄</th>
<th>Rem₅</th>
<th>LMV₅</th>
<th>RMV₅</th>
<th>F₅</th>
<th>T₅</th>
<th>Ext₅</th>
<th>Dom₅</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

AVL₁ AVL₂ AVL₃
PS/Aux/T/Pres/PS/Noun/Anim/-/PS/Aux/T/Past

AVL₄ AVL₅
PS/Verb/Anim/#/#/2/+/-/PS/Verb/Anim/+
3.3. Construction of the Initial Sentence Graph

In a certain sense the process of synthesizing the initial surface syntactic representation of the input sentence is the reverse of the analytic processes already described. The morphological analyzer decomposes the input sentence first into words and then into lexical constituents; the syntactic representations of these latter are then retrieved from the dictionary, yielding forms which suggest further analysis (i.e., in general, the syntactic representation of a lexical constituent has its own arbitrarily complex internal structure). In constructing the initial sentence graph, dictionary entries will first be combined to form word-level syntactic segments; these will then be combined to form the initial sentence representation.

3.3.1. Lexical Processing. As dictionary entries are retrieved they are retained in a temporary string in the order of the appearance of their referents in the input sentence. Word boundaries in the input sentence are indicated by special word boundary markers inserted in the string at the left and right ends of the sentence and at other points corresponding to the initial decomposition of the sentence into words. The following procedures, which produce word-level syntactic segments
by concatenation of entries, operate only upon material between consecutive word boundary markers. The procedures are applied, in turn, between each pair of word boundary markers, beginning at the start of the sentence.

Concatenation is implemented through the modification of the T and F fields of the edge schemas of the entries. To concatenate entry A to the right of entry B, we add to every T field and F field in A the highest vertex identification integer to be found in any edge schema of B (by convention, this is the identification integer of the rightmost vertex in B). Since (by convention) the leftmost vertex of A initially has the identification integer "0", it will subsequently carry the same identification integer as does the rightmost vertex in B; the rightmost vertex of A will have for its identification integer a number \( m = n - 1 \) where \( n \) is the total number of vertices in the combined graph.

**Morphological Compatibility.** The lexical constituents which are to be combined to form a word-level syntactic segment must be morphologically compatible with one another. This relationship is tested in order to detect possible errors arising from the preceding analytic operations or from erroneous input, and to eliminate from
further consideration alternative syntactic paths which are inconsistent with neighboring constituents. The English lexical item "saw" has at least two common verbal and one common nominal interpretations (see Fig. 8). In the word "sawing", only the verbal interpretation associated with the meaning "to cut wood" is compatible with the suffix "-ing". Significant savings in time and space can be achieved by eliminating the superfluous data from the final surface sentence representation.

This procedure makes use of the Left and Right Morphology Vectors associated with each edge of a dictionary entry. These 16-bit patterns are coded to reflect the gross morphosyntactic properties of the edge to which they are attached, the significance of each of the 16 bits being independent of the others. More correctly, they reflect the morphosyntactic properties of the path(s) of which the particular edge is a part, as well as the peculiar properties of the edge itself. In the diagram below, edges A and B are said to be compatible if and only if the logical intersection of the RMV of A and the LMV of B is non-null. Similarly, B is compatible with C if and only

\[ \text{Fig. 10} \]
if the RMV of B gives a non-null intersection with the LMV of C. Compatibility is not defined between edges which are not adjacent; thus, to speak of the compatibility of edges A and C in Fig. 10 is meaningless.

Before two dictionary entries are concatenated in the construction of a word-level syntactic segment, compatibility tests are carried out for all pairs of edges which will meet at the junction of the two constituent graphs. Any edge which is found to be incompatible with all adjacent edges at the junction is simply eliminated from further consideration. Specifically, if edge i is to be dropped, then the F and T fields are set to zero in the i-th edge schema. Furthermore, if the elimination of an edge leaves other edges "stranded", then the stranded edges are eliminated, etc. Finally, whenever an edge is eliminated, compatibility tests are carried out for all pairs of remaining edges which meet at either of the vertices formerly connected by the eliminated edge (this may set off further eliminations, of course). These notions are illustrated in Fig. 11a below. Let us assume that the lexical graph composed of edges A, B, C, D, E, and F represents an English stem, while the graph made up of edges X and Y represents an appropriate suffix. Conca-
tenation of the two graphs to form a word-level syntactic segment would involve the amalgamation of vertex 4 of the left-hand graph and vertex 0 of the right-hand graph (dotted line), making edges E and F adjacent to edges X and Y. These edges must therefore be tested for compatibility. In the diagram the LMV's and RMV's have been truncated to three bits each; these are given in parentheses to the left and right, respectively, of the edge label. We use the raised dot ("·") to represent the operation of vector intersection. For edges F and X we have \((100) \cdot (101) = (100)\); the intersection is non-null and therefore the edges are compatible. Notice that it will not be necessary to further test either of these two edges unless the compatibility of some third edge cannot be established without such further tests. Edges E and Y give \((010) \cdot (001) = (000)\). It is thus necessary to test E and X. We have \((010) \cdot (101) = (000)\). Edge E is incompatible
with the morphological situation represented by the example and will be eliminated. Edge Y must likewise be dropped, since for Y and F we have \((001) \cdot (100) = (000)\). Deleting edges E and Y we obtain the result depicted in Fig. 11b.

![Diagram](image)

**Fig. 11b**

This is unacceptable as it stands; vertex 2, and edge C leading into it, have been left stranded by the deletion of edge E. (A stranded vertex is a vertex---other than the leftmost or rightmost vertex of an entry which have distinctive identification numbers---which no edges enter or from which no edges depart; i.e., its identification number appears in one or more of the F fields or one or more of the T fields of the entry's edge schemas, but not both. A vertex whose identification number appears in neither the T nor the F fields simply does not exist.) A stranded edge is an edge leading into or out of a stranded vertex; i.e., the identification number of a
stranded vertex appears in the F field or the T field of the corresponding edge schema. In such a case, the stranded vertex and edge(s) are eliminated. We see the results in Fig. 11c.

Since a stranded edge has been removed we are now obliged to perform compatibility testing at the vertices it joined. One of these--vertex 2--has also been removed; it remains to carry out compatibility tests at vertex 1 (of the left segment). Edges A and D yield (100)·(100) = (100), and are compatible. Edges B and D give (011)·(100) = (000), and are not compatible. Edge B, being incompatible with all (surviving) adjacent edges, must also be eliminated. This deletion leaves no stranded vertices, and so the morphological compatibility testing is complete. The final results, including concatenation of the entries, are shown in Fig. 11d.
Similar morphological tests are carried out at the leftmost and rightmost vertices of a potential word-level segment, just as though the word boundary markers (which surround the segment) themselves had Left and Right Morphology Vectors. These tests serve as a check on the initial decomposition of the input sentence into words and, of course, on the correctness of input data.

If this process of morphological testing should result in the deletion of an entire entry, normal system operations are suspended and diagnostic data are produced.

**Segment Expansion.** The explicit recognition and expression of certain important kinds of lexical generalizations about the English language have great practical as well as linguistic value for the MIND system. In practical terms, significant savings in storage space and in the costs of system maintenance can be achieved through the elimination of lexical redundancies. For an obvious example, we may consider the case of "strong" English verbs.
In Fig. 12 we show somewhat contrived representations of dictionary entries for the words "fight" and "fought".

Fig. 12

Here the symbol "(Fight)" is intended as a shorthand device to represent an arbitrarily complex set of features (i.e., attribute-value pairs), distributed over an appropriate network of edges and vertices connecting vertices 1 and 2 (in both segments). The diagram asserts, simply, that the two dictionary entries have much in common (despite the fact that the morphological analysis procedure could not discover this) and that the similarities and differences between the two are neatly circumscribed by the topology of the segments. That is, they differ in what we may call the "Auxiliary" part of the entry and they are identical elsewhere.
One way of expressing this observation would be to establish three distinct dictionary entries, one each for "fight", "fought", and "(Fight)", and to introduce into the first two of these a special kind of pointer to the third, as in Fig. 13.

![Diagram](image)

Fig. 13

We interpret a schematic attribute-value list of the form "%X" as an instruction to replace the edge bearing it with the complete dictionary entry with the name "(X)". This is a remote reference of Type 1.

An edge containing a remote reference of Type 1 has an AVL consisting solely of a character string exactly 6 bytes in length. This character string, when extended to 8 bytes by surrounding parentheses, becomes the name of the dictionary entry which is to be fetched and inserted in the syntactic structure in place of the edge where the reference occurs. Any edge of a segment may consist of
a Type 1 remote reference, **including edges of entries fetched via such a reference.** Clearly the dictionary maker must take pains to guarantee that the depth of such reference chains is reasonably finite.

The mechanics of structure insertion, like those of concatenation, involve the manipulation of the T and F fields of the edge schemas involved, though insertion is somewhat more complicated. First, in the entry to be inserted, these fields are incremented by the integer \( m = n - 1 \), where \( n \) is the identification integer of the rightmost vertex of the receiving structure, except where a field contains zero initially (it is then left alone). Next, the incremented identification integer of the rightmost vertex in the inserted structure is substituted for that of the receiving structure (everywhere in the T and F fields of the receiving structure). Finally, the vertex identification integers of the leftmost and rightmost vertices of the inserted structure are replaced (everywhere in the edge schemas of the inserted edges) by the identification integers of the vertices spanned by the referencing edge in the receiving structure.

There are several schemes for accomplishing this task. This method, whose details are unimportant, is based upon
the assumption that inserted structures will generally be smaller than receiving structures; its motivation is computational economy.

Since the execution of a Type I remote reference involves the deletion of an edge and its replacement by an arbitrarily complex structure, morphological compatibility testing must be carried out at the two vertices formerly spanned by the deleted edge. This testing may, of course, lead to the deletion of edges (in or adjacent to the inserted structure) and consequent further testing, etc. It is possible, one should bear in mind, that an edge consisting of a Type I remote reference may be deleted as a result of morphological testing; in such a case the reference "disappears" and is not carried out.

A second kind of lexical redundancy, which has given rise to the notion of prototypical attribute-value lists (PAVL), more closely resembles the kind of lexical redundancy familiar to students of generative grammar. It will be found that certain sets of attribute-value pairs occur in a relatively high proportion of AVL's throughout the dictionary. For instance, there are subcategorical word classes, such as "personal, transitive verbs". In some cases these sets reflect some sort of deeply rooted
redundancies in the system of features* itself; i.e., certain features, or groups of features, "imply" others (as in the familiar lexical redundancy rules). Thus, for example, all "human" nouns are (necessarily) "animate", "concrete", "countable", etc. Other cases have a more empirical character; we may observe that a given collection of features occurs often enough to warrant recognition on either linguistic or computational grounds. In either case, the feature set is established as a PAVL. It is given a 4-byte name, the first byte of which is the character "$", and entered in the dictionary. The form of the PAVL dictionary entry is simpler than that of ordinary entries; it consists of the 4-byte name, a single byte integer (EL) giving the length, in bytes, of the attribute-value list, and the prototype attribute-value list itself.

\[
\begin{array}{ccc}
\text{NAME} & \text{EL} & \text{PAVL} \\
\$XYZ & 21 & A:B C:(D,E) M:(N:O P:Q) S:-T \\
\end{array}
\]

Fig. 14
A Prototype Attribute-Value List Dictionary Entry

Whereas a remote reference of Type 1 has the effect of replacing a single edge by an artibrarily complex network,

*We use the simpler term "feature" here as equivalent to "attribute-value pair".
a remote reference of Type 2 replaces a single attribute-value pair with an arbitrarily long attribute-value list. The attribute-value pair to be replaced consists of a unique attribute name (here symbolized by "!")) and a three-byte literal name of a PAVL list. The three-byte name, when prefixed with "$", becomes the name of the PAVL dictionary entry whose AVL is to replace the special attribute-value pair. In principle, there is no limit to the number of remote references of Type 2 which an edge may contain; the actual number of such references on a given edge is given by the positive-integer-valued Rem field of the corresponding edge schema. As with remote references of Type 1, morphologically motivated deletion of an edge causes any Type 2 remote references it contains to vanish. Note that the execution of a remote reference of Type 1 may bring further remote references, of both types, into play. The execution of a remote reference of Type 2 may occasion additional references of Type 2, but obviously cannot cause any deeper Type 1 references.

When the Rem field indicates that one or more remote references of Type 2 are contained on a given edge, the AVL of the edge is scanned from left to right. Each attribute-value pair is examined in turn for the occurrence of
any of four conditions:

- (1) negation of the attribute; the scan is suspended, the attribute-value pair containing the negation is deleted, and all attribute-value pairs to the left of it having the same attribute are deleted, whereupon the scan is resumed.

- (2) negation of the value; the scan is suspended, the attribute-value pair is deleted (only the value if other unnegated values are given), and all attribute-value pairs to the left of it having the same attribute and value are deleted (again, the attributes are left intact if there remain one or more unnegated values), and the scan is resumed.

- (3) a remote reference of Type 2 is encountered; the scan is suspended, the referenced PAVL is retrieved from the dictionary, the referencing attribute-value pair is deleted and the retrieved PAVL is inserted in its place in the AVL string, and the scan is resumed (beginning with the first attribute-value pair of the inserted PAVL).
(4) the end of the AVL is reached; expansion of the edge is complete.

The motivation for the negation of attributes and values can now be clarified; it is the device whereby exceptions to certain classes of linguistic (and computational) generalizations are expressed. Consider the following purely formal example of an AVL containing a remote reference of Type 2:

\[ A:B \ C:D \ !:\$XYZ \ -E:Y \ F:\neg G \ H:(A,J,\neg K) \]

Fig. 14a

In scanning this string from left to right we encounter the first action condition at the pair symbolized by "!:\$XYZ". We halt the scan and retrieve from the dictionary the indicated PAVL entry. Suppose this latter PAVL is:

\[ E:L \ \ M:N \ O:P \ F:(S,G) \ !:\$UVW \ T:\neg S \]

Fig. 14b

The referencing attribute-value pair is deleted from the original AVL and is replaced by the referenced PAVL, yielding

\[ A:B \ C:D \ E:L \ M:N \ O:P \ F:(S,G) \ !:\$UVW \ T:\neg S \ -E:Y \ F:\neg G \ H:(A,J,\neg K) \]

Fig. 14c
The scan is now resumed at the point of interruption (i.e., beginning with the attribute-value pair "E:L") and proceeds without incident up to the second Type 2 remote reference "!:#$UVW". Again, the scan is suspended, the named PAVL is fetched (Fig. 14d), and is inserted in the AVL in place of the reference (Fig. 14e).

H:(A,B,K) T:(R,S)

Fig. 14d


Fig. 14e

The scan continues (beginning with "H:(A,B,K)"") and is next halted at "T:=S" because of the indicated negation of the value. While the primary scan is suspended at this point, a secondary scan is begun (working from this point back from right to left) whose goal is to detect occurrences of the attribute "T" with value "S". This condition is satisfied by "T:(R,S)"; the first item encountered in the secondary scan. The value "S" is deleted in this case,
but, since another unnegated value ("R") remains, the attribute is not deleted; in short, "T:(R,S)" becomes "T:R". The secondary scan continues all the way back to the start of the original AVL, but no further deletable items are found. The pair "Tː¬S" is now deleted.

The original scan again resumes, and is halted at once by "¬E:Y". The negation of an attribute is simpler and more sweeping in its effects than the negation of a value; all attribute-value pairs (to the left) whose attribute is negated are eliminated without regard to their values. The value member of the pair where the negation is specified (in this case, "Y") is meaningless and simply fills space. Here, the pair "E:L" is deleted. Then the pair "¬E:Y" is eliminated.

And so on. "F:¬G" causes "F:(S,G)" to reduce to "F:S" while "H:¬K" (from "H:(A,J,¬K)") reduces "H:(A,B,K)" to "H:(A,B)". The results are shown in Fig. 14f.


Fig. 14f

Finally, the AVL is consolidated. That is, attribute-value pairs whose attribute members are identical are merged into a single attribute-value pair whose value member is the disjunctively embedded (where necessary) logical union of the original value members. In Fig. 14f
the attribute "H" appears in two attribute-value pairs, with the values "(A,B)" and "(A,J)". These two pairs are consolidated, yielding "H:(A,B,J)". The final results of these procedures are shown below:

\[
\begin{array}{cccccccc}
\end{array}
\]
Fig. 14g

It may be useful to emphasize at this point that all negations are eliminated during the expansion of remote references of Type 2. Negative attributes and negative values are meaningless for the syntactic component of the MIND system; they serve only to mark exceptions to lexical generalizations of the kind we have just described. It follows that the use of this type of negation, except within PAVL's and to the right of Type 2 remote references within ordinary AVL's, is an error. The computational consequences of the misuse of this type of negation range from the trivial (e.g., negation to the left of the first Type 2 remote reference within an AVL) to the potentially disastrous (as in the use of negation in an AVL containing no Type 2 remote references, thus baffling the syntax processor).

Summary. Lexical processing builds word-level syntactic segments from dictionary entries by concatenation of the graph structures of the entries. Before this can be completed, the entries themselves must be expanded (i.e.,
their remote references of Type 1 and Type 2 must be processed) and all sets of edges made adjacent during these operations must be tested for morphological compatibility. The ordering among the three principal lexical processes may be summarized in Fig. 15.
The morphological compatibility testing procedure is accorded a kind of priority simply because both of the expansion procedures depend upon it in the important sense that they need not be carried out on a particular edge unless the edge is compatible with its neighbors. Conversely, the expansion of Type 2 remote references is carried out last because the results of such expansions cannot affect either of the other operations.

3.3.2 The Initial Syntactic Graph. All that remains at this point is to reformat the materials produced by the main portions of the ANALYZE module to facilitate their use by the subsequent PARSE module of the MIND system.Momentarily ignoring the detailed changes of form required by the conventions of the system, it is possible to characterize this reformatting of data as a process of concatenation of the word-level segments derived through the operations of the ANALYZE module. The concatenation procedures described in the foregoing sections are, however, no longer appropriate.

The word-level segments to be united in an initial bottom-level syntactic representation of the received sentence comprise a set of individual word-level graphs, ordered in terms of their position of occurrence in the
input sentence, and separated from one another by distinctive word-boundary indicators. Each of these graphs has a leftmost vertex whose identification integer is zero, and a rightmost vertex identified by an integer \( n = m - 1 \), where \( m \geq \) the number of vertices in the graph. It is now necessary to combine these in a single bottom-level graph for the sentence as a whole, and to do so in a form acceptable to the PARSE module. The explanation of the basically simple procedures by which this is accomplished will be accompanied by a brief review* of the form of the PARSE module's sentential representations. It involves no difficulties to imagine that all vertices in the word-level segments have been uniquely numbered with consecutive integers as though in accordance with the intra-word concatenation scheme presented earlier.

Within the PARSE module, all syntactic vertices are catalogued in an array of half-word pointers which we shall here refer to by the name VERT_LIST. Within this array, vertex identification numbers may be used as subscripts by which the relative addresses of individual vertex-data-blocks within the effective PARSE region may be obtained.

*For a fuller explanation of the PARSE module and its components, see /3/.
The relative half-word pointer VERT_LIST(i) provides the address of the vertex-data-block (VERT_BLOK) corresponding to the i-th vertex of the input sentence representation. The VERT_BLOK in question consists of four bytes of data, depicted by half-words as follows:

Bytes 1 and 2   Bytes 3 and 4
INCL (=0)*      NEXTVERT

Whereas, in the dictionary, data are generally arranged according to the edges of syntactic subgraphs, the initial parsing structure is arranged, in accordance with the requirements of the PARSE module, by vertices, then by edges. The dualism suggested by this account of the transformation of data is, for our immediate purposes, of greater practical (i.e., programmatic) than descriptive utility. In the place of a fuller treatment (/3/), we shall here only sketch the functions of the relevant elements of the VERT_BLOK structures. The algorithms for deriving these from the dictionary structures discussed above are generally too obvious to require extended discussion.

*INCL plays no part in the linguistic processes to be outlined here. Its function is fully described in /3/.
NEXTVERT is an integer half-word index into the VERT_LIST array, by which the next vertex to the left, in a single spanning chain over the set of vertices representing a sentence, is indicated. In the VERT_BLOK for the leftmost vertex of a sentence, the value of NEXTVERT is zero, of course. What is required here is the selection of a chain which spans the vertices of a sentence from right to left. For the purposes of the PARSE module of the MIND system, such a chain may be chosen with local arbitrariness, except that no chosen subchain may be allowed to parallel any subchain in the originally given structure. It is the chain of NEXTVERT arcs which guide the major processes of the PARSE module.

Each VERT_BLOK, excepting the rightmost, is immediately followed by one or more EDGE_BLOK's—one for each edge incident-out from the vertex in question. The EDGE_BLOK format in fullwords is as follows:

<table>
<thead>
<tr>
<th>Word</th>
<th>Bytes 1,2</th>
<th>Bytes 3,4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ETYPE/ERULE</td>
<td>SUCCESSOR</td>
</tr>
<tr>
<td>2</td>
<td>DAUGHTER</td>
<td>DESCENDANT</td>
</tr>
<tr>
<td>3</td>
<td>DEPENDANT</td>
<td>EXCL</td>
</tr>
<tr>
<td>4</td>
<td>NCODES</td>
<td>/<em>DATA</em>/</td>
</tr>
</tbody>
</table>
The most significant bit of the ETYPE byte is set to "1" just in case more than one edge is incident-in to the vertex at the left-hand end of the edge in question (i.e., the vertex with whose VERT_BLOK the current EDGE_BLOK is associated in storage). The least significant bit of ETYPE is set to "1" just in case this edge serves as a dominance-pointer (see below). The rest of ETYPE, as well as all of the ERULE byte and the EXCL and DEPENDANT half-words, are not involved in the functions of the morphological processor and are set to zeros by convention.

The SUCCESSOR half-word points to the vertex at the right-hand end of the edge, via the VERT_LIST array. The DESCENDANT half-word points to the first byte (i.e., the ETYPE byte) of the next EDGE_BLOK incident-out from--and associated in storage with--the same vertex. Where the low-order bit of ETYPE is zero (the common case), the NCODES half-word gives the length in bytes of the AVL for the edge. The AVL itself follows immediately.

The relation of syntactic dominance between edges has only been mentioned very briefly heretofore (see p. 49 and p. 53) because it plays an infrequent role in the operations of the morphological processor. Its treatment is quite straightforward, however. Whenever a relation of syntactic dominance is indicated in a dictionary entry (through
the **Dom** field of one or more of the edge schemata) a simple linear graph, identical in length to the sub-graph of dominated edges, is constructed in exactly the same representational format as given just above; but this dominance graph shares no vertices with the main graph, even though it may occupy storage space contiguous with parts of the main graph. The **DAUGHTER** half-word of the dominating edge (in the main graph) points to the left-most vertex of the dominance graph via the **VERT_LIST** array. Each edge in the dominance graph will point to one edge of the dominated sub-chain of edges in the main graph. To indicate this, the low-order bit of the **ETYPE** byte for each edge in the dominance graph is set to **one**, and the content of **NCODES** is made a relative pointer to the first byte of the **EDGE_BLOK** for the corresponding dominated edge. The **SUCCESSOR** half-word is set to point, via the **VERT_LIST** array, to the next vertex on the right within the dominance graph. All other entries in the **EDGE_BLOK** of an edge in a dominance graph are set to **zeros**. There are, of course, no AVL's associated with edges of a dominance graph.

As vertices—whether of the main graph or of dominance graphs—are constructed, they are indexed in consecutive locations of the **VERT_LIST** array. A zero entry is made in
this array immediately following the last meaningful vertex pointer, indicating the end of the list. This VERT_LIST array and the structures representing the bottom-level syntactic graph (and any auxiliary dominance graphs) are maintained in a storage area known to the ANALYZE and PARSE modules. As the PARSE module overlays the ANALYZE module and prepares to carry out a syntactic analysis on these structures, it is provided with an integer which serves as a pointer, through the VERT_LIST array, to the right-most vertex of the main syntactic graph, where parsing is begun.
REFERENCES


