IV. DICTIONARY LOOKUP AND UPDATING PROCEDURES

Mark Cane

1. Introduction

The present report describes the routines used to manipulate the thesaurus (or concept dictionary), including all thesaurus lookup programs, as well as the programs used to set up and update the hierarchy. The update programs are described first, including a specification of the input deck, as well as a detailed explanation of all setup and updating options. This is followed by a discussion of the lookup procedures. Examples are given and detailed flowcharts are shown to illustrate the programs.

The updating section sets up the thesaurus and suffix trees used by the thesaurus lookup programs. These programs are stored as the first file of a library tape. Inputs to these program blocks consist of a control card specifying what is to be done, followed by additional cards containing the data for the entries to the thesaurus and the suffix trees. A variety of options is provided. Either tree may be set up in its entirety; an existing tree may be copied onto a new library tape unaltered, or may be augmented by new data on cards; furthermore, the semantic and syntactic data associated with each entry in the trees may be altered.

The block of updating programs also handles the conversion of semantic and syntactic data from a format acceptable to the FORTRAN I-O package to the packed format used by later programs.

2. Input Deck, Control Card, and Data Card Formats

The input deck consists of a single control card, followed by the cards containing the thesaurus entries (if any), terminated by a card with the letter "Z" in columns 1 through 6; the thesaurus entry cards are in turn followed by the cards containing the suffix data (if any), which are again terminated by a card with six "Z"'s.

The control card is divided into four fields: columns 1-6, 7-12, 13-18, and column 24. All fields are left-justified. Table 1 lists the possible configurations of these fields and the options they control.

A thesaurus input card contains 15 fields. The first field (card columns 1 to 24) contains the English word to be entered. It must be left-justified, i.e., the first column must contain the first letter of the word. The next six fields (columns 25 to 48) store the semantic codes (category numbers) associated with the word. Each code is allotted four columns, so that the codes appear in columns 25-28, 29-32, 33-36, etc. The codes must be left-justified in the field, and must be stored consecutively. For example, if there are two codes, say 905 and 237, they must appear in the first two fields in columns 25-28 and 29-32, respectively. Furthermore, each code must be right-justified within each field. Thus, for the above example, columns 25-32 would appear as follows:

25	26	27	28	29	30	31	32
	9	0	5		2	3	7

.

The category numbers must be less than $2^{12} = 4096$ in magnitude.

	First Field Col. 1-6	Second Field Col. 7-12	Third Field Col. 13-18	Fourth Field Col. 24	Option
1	BOTH	START	START		Both the thesaurus and suffix trees are set up in their entirety. Nothing is done with the old tape (an explicit 0 may appear in column 24).
2	BOTH	START	START	1	Same as above except that the old tape is spaced past the file with the lookup data.
3	BOTH	START	UPDATE		The second field controls the action taken for the thesaurus tree: START means that it is setup in its entirety, UPDATE implies that the entries on the input cards are added to the tree on the old library tape. The third field controls the action taken for the suffix tree, with definitions as before.
4	BOTH	UPDATE	START		
5	BOTH	UPDATE	UPDATE		
6	THES	START			The action specified by the second field is taken on the tree specified in the first field. The tree not referred to in the first field is copied unaltered from the old library tape to the new.
7	THES	UPDATE			
8	SUFFIX	START			
9	SUFFIX	UPDATE			
10					If the first field is blank the lookup file is copied from the old tape to the new; nothing is altered.

Thesaurus and Suffix Setup and Update Control Card Formats

TABLE 1

The last eight fields (columns 49-72) represent the syntactic codes associated with the given word. Each code is allotted three columns (i.e., columns 49-51, 52-54, etc.). These codes must again be left-justified within the entire syntactic field and must be stored consecutively. Each code must be right-justified within its field. The syntactic codes must be less than $2^8 = 256$. The thesaurus input cards may therefore be read with a format of 4A6, 6I4, 8I3. There are no restrictions on the order of the input words.

The suffix input cards contain two fields. The first (columns 1-12) stores the English suffix; the second (columns 13-15) contains a number associated with each suffix in a one-one correspondence. This number must be right-justified within its field and less than $2^8 = 256$ in magnitude. The first suffix entered must begin with the letter "e" (see LØØK description).

3. Implementation of Setup and Updating

The thesaurus setup and update block consists of the following subroutines: CANE (with entry CANE3), ECRW, CANE1 (with entries CEAD and CRITE), CANE2, TIPUT, TREET (with entry TRADD), and SUFTR (with entry SUFAD). CANE1 copies the file from the old library tape to the new tape. CEAD reads the file into memory from the old tape, and CRITE writes the trees in memory onto the new tape. CANE is the only routine called from the main program. Its entry, CANE3, is an error return - in case of error during reading (i.e., redundancy, end of file encountered) or during writing

(i.e., redundancy), CANE1, et al. transfer control to ECRW, which prints out a message explaining the error and transfers to CANE3. The latter turns on sense light 1 to let the main program know that an error occurred, and then returns to the main program.

CANE2 controls the main flow of operations of this block. The flowchart for this routine (Flowchart 1) is generally self-explanatory; only a few remarks are needed.* The operations of spacing past the thesaurus file and of reading in the old trees to update them are performed in the same manner: CEAD is called to read in the whole file. If the thesaurus tree is to be set up solely from input cards, the old tree is simply ignored.

Messages are printed out stating the amount of tree storage remaining for both the thesaurus and suffix trees. This aids in determining whether future additions are likely to cause either tree to overflow their allotted storage.

In the event that either tree should exceed its allotted storage, a message stating the amount of the excess is printed and a printout indicates whether the thesaurus or suffix trees were involved; sense light 1 is also turned on. If the overflow occurs in the suffix tree, or if it occurs in the thesaurus and no suffix cards are present to be read, control is returned to the main program. If overflow is detected in the thesaurus, and suffix cards are to be read, it is desirable to space the input tape

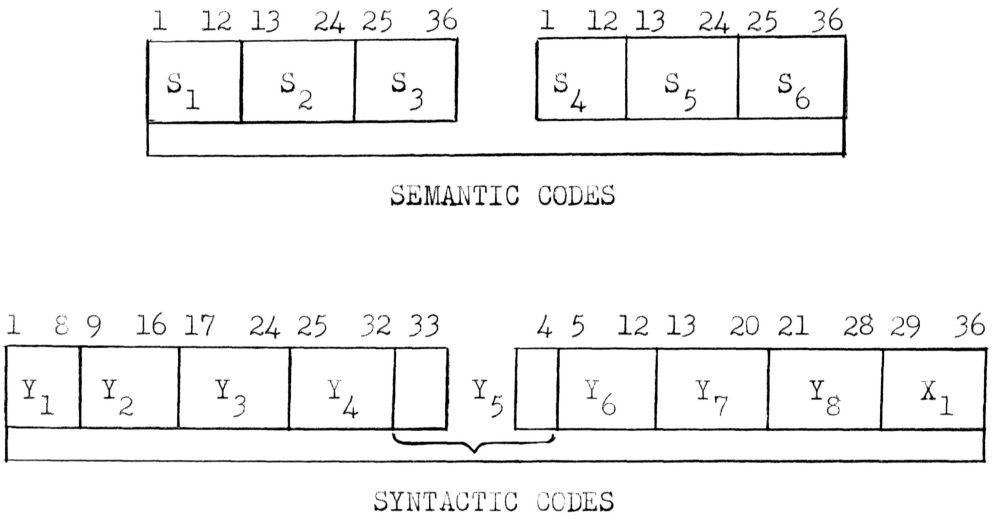
---

*The flowcharts appear in the Appendix to this section.

past the suffix cards, so that the next setup or update programs can again test for possible errors. Further, it is then also useful to process the suffix cards to determine whether the suffix tree will also overflow. In this case it is, however, undesirable to waste time writing a worthless file. The parameter KSPACE is then used to prevent such a tape from being written.

The buffer provided for the thesaurus input may not be large enough to contain all input simultaneously. TIPUT, which reads in the cards, is used to furnish a parameter IRØV, which is set to 1 if the reading operation is interrupted because of a filled buffer and is set to 0 if reading stops because a word of "Z"'s is encountered to indicate the end of the input cards. After returning from TREET or TRADD (depending on whether the tree is set up or updated, respectively), this parameter is tested. If it is 1, TIPUT is called again, and then TRADD is entered if the tree is being updated. In addition to reading in the thesaurus cards, TIPUT packs the semantic and syntactic codes into a format acceptable to the other routines in SMART.

Internally, each entry in the thesaurus consists of four computer words of associated data as shown in Fig. 1. The semantic codes appear in the first two words in six 12-bit fields (S_1 to S_6 in the diagram). The codes are left-justified; that is, if for example a word has four semantic codes, they are stored in fields S_1, S_2, S_3, and S_4. The syntactic codes occupy the next two words in eight 8-bit fields (Y_1 to Y_8 in the diagram). Note that the fifth field is split between two adjacent words. The codes



Internal Formats of Semantic and Syntactic Codes

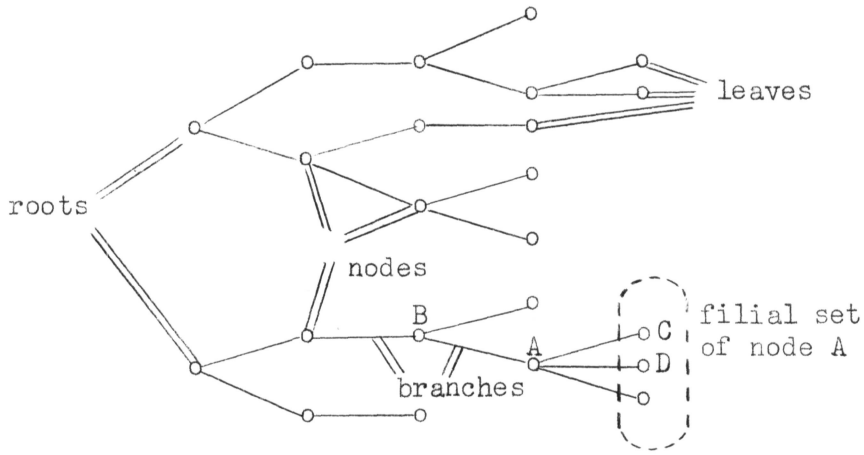
Figure 1

are again left-justified. The rightmost field of the syntactic words (X_1 in the diagram) is always set equal to zero. (It is reserved for a suffix code - see the description of L/ØK).

4. The Tree Programs - SUFTR, SUFAD, TREET, and TRADD

Before discussing the routines which set up the trees, it will be helpful to give a brief, nonrigorous explanation of some of the concepts involved.† The principal concepts are illustrated in Fig. 2. Roots are

†For a fuller, more precise discussion see Iverson, A Programming Language, Wiley (1962) and E. H. Sussenguth, "The Use of Tree Structures for Processing Files," CACM (May 1963).



A Pictorial Representation of a Tree

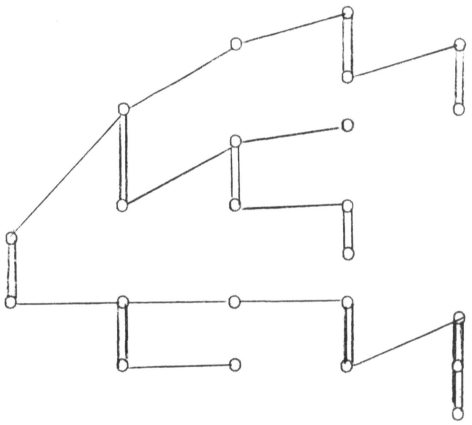
Figure 2

nodes with no branches entering them; leaves are nodes with no branches leaving them. Note that while many branches may leave one node, at most one branch enters each node. The filial set of a node A is the collection of nodes S such that there is a branch from A to the elements of S. A member of the filial set is called a son of A; two members of the same filial set are said to be "brothers." Thus, in the tree of Fig. 2, A is the son of B, and C and D are brothers.

In the trees to be described, each node consists of one machine word containing three fields ($\underline{w}_1$, $\underline{w}_2$, $\underline{w}_3$). The first field, $\underline{w}_1$, is called the key; the second, $\underline{w}_2$, points to a brother; the third ($\underline{w}_3$) to a son. The key is a BCD representation of an English letter. In the trees used in

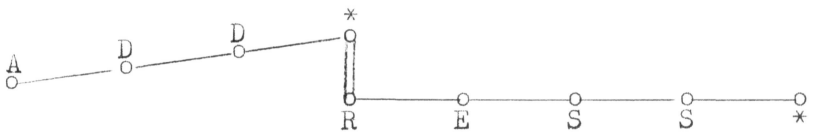
connection with the thesaurus, each node points to one son, which in turn points to its brother. Thus the tree of Fig. 2 can more accurately be represented by Fig. 3 (with double lines connecting brothers).

An English word is represented as a path from a root to a leaf. A word is always taken to be terminated by a blank (represented here by "*"). Thus the key ($\underline{w}_1$) of each leaf is the character "*" (represented in BCD code by the digits "60"). The $\underline{w}_3$ field of each leaf points to the data associated with the word. For example, the words "ADD" and "ADDRESS,"



Pictorial Representation of a Tree as Represented in Memory by SUFTR and TREET

Figure 3



A Pictorial Representation of "ADD" and "ADDRESS" in Tree Form

Figure 4

in tree form, are represented pictorially as shown in Fig. 4. The tree of Fig. 4 might be stored in memory as shown in Fig. 5.

Consider now the routines SUFTR and SUFAD (see Flowchart 3). SUFTR sets up the complete suffix tree initially. The routine starts by setting up the first word, and enters nodes consecutively from the beginning of the block reserved for the suffix tree. SUFAD adds suffixes to a tree already set up; it begins by adding from the next free location of the suffix

Location	Bits S, 1-5	Bits 6-20	Bits 21 - 35
	$\underline{w}_1$	$\underline{w}_2$	$\underline{w}_3$
100	A		101
101	D		105
⋮			
105	D		107
⋮			
107	*	115	XXX[†]
⋮			
115	R		120
⋮			
120	E		121
121	S		123
⋮			
123	S		124
124	*		XXX

A Possible Memory Configuration Corresponding to the Tree of Fig. 4

Figure 5

[†] The $\underline{w}_3$ fields marked "XXX" point to data.

block - as determined by an input parameter. Except for this difference, the two routines are identical and will henceforth be referred to as SUFTR.

SUFTR begins by entering a new suffix by comparing the first letter of the input word with the key of the first root of the tree. If a match is found, the next letter of the input is compared with the son of the matching key letter (pointed to by the $\underline{w}_3$ field). If the routine fails to find a match, it proceeds to the first brother (pointed to by $\underline{w}_2$) and compares the input letter with the new key (see Flowchart 3, especially the box labeled 3). The search is continued until a match is no longer possible (when no match is found and the $\underline{w}_2$ field is zero, indicating that there is no brother). The address of the next free location, K, is then inserted as the $\underline{w}_2$ field of this letter (box 4 of Flowchart 3) and control is transferred to box 1 of the flowchart where the letter of the input word is defined as the key ($\underline{w}_1$ field) of the word at K. The address of the next free location (K - 1) is then inserted as the $\underline{w}_3$ field of the word at K. The next letter (its son) will then be put in this new location. Sons are continually inserted in this way until the "letter" of the input word is a blank ("*"). At that point the data (suffix code) for this word is stored in the next free location.

If a match should be found for the entire suffix, this is taken to mean that an alteration of the data associated with that suffix is desired. Box 5 of Flowchart 3 handles this (of course, if the same suffix is inserted a second time by mistake no harm will be done; the suffix code will merely be stored over itself).

The algorithm of TREET is essentially the same as that of SUFTR (TRADD stands in the same relation to TREET as SUFAD does to SUFTR). The similarity of the algorithms may be seen by comparing the parts of Flowchart 2 above the dotted line with Flowchart 3. This similarity is obscured somewhat by the special requirements that LØØK places upon the order of the thesaurus tree (see the description of LØØK). Specifically, LØØK requires that blanks, final "e"'s, and final "y"'s be encountered before any other members of the same filial set, so that possible stems will be recognized. The TREET routine must therefore verify that a blank occurs always as the first element of a filial set (the one pointed to by the $\underline{w}_3$ field of the parent and not by the $\underline{w}_2$ field of a brother). An "e" is always inserted as the first element of a filial set, except if a blank is stored ahead of it. A "y" is inserted before everything except a blank or an "e". (Note that it is not strictly necessary, from the point of view of LØØK, to insert all "e"'s and "y"'s ahead of everything else; this is only necessary if these letters occur as final letters of a word. By doing it in every case, the setup routine becomes more efficient: no rearrangements or searches through a whole filial set are required if an "e" or "y" in the middle of a word should turn out to be a final "e" or "y" of another word.) An initial "e" or "y" is not inserted ahead of other roots.

This rearrangement is accomplished by the sections of TREET represented by the part of Flowchart 2 below the dotted line. If the next "letter" of the input word is a blank and there is no blank in the filial set of the last node for which a match was found, the blank is inserted as

the first son of this node and the address of the old first son is inserted in the $\underline{w}_2$ field of the new first son. This is done in box 7 of Flowchart 2. If a blank is already included in the tree at this point, an alteration of the data of the word is called for; this is handled as in the description of SUFTR. Similarly for an "e" if there is no blank in the tree at this point, or a "y" if there is no blank and/or "e" in the tree at this point. If the letter in question is already in the tree, TREET moves on to the next letter of the input word in the normal manner.

As an example of this process, Fig. 6 shows the tree of Fig. 4 with word "AD" inserted. Similarly, Fig. 5 is changed to the form shown in Fig. 7, assuming the next free location at 271.

If the letter of the input word to be searched is an "e" and a blank already occurs in the filial set, the "e" is inserted just under it (i.e., the $\underline{w}_2$ field of the blank is made to point to the "e"), unless the "e" is already present there, in which case TREET moves on to the next letter of the input word. This insertion is accomplished in box 8 of Flowchart 2. If the $\underline{w}_2$ field of the blank is zero, "e" is made a brother in the same manner as for any other letter (in box 6 of Flowchart 2). If "y" is the input and a blank and/or an "e" is already in the tree, the same procedure is followed.

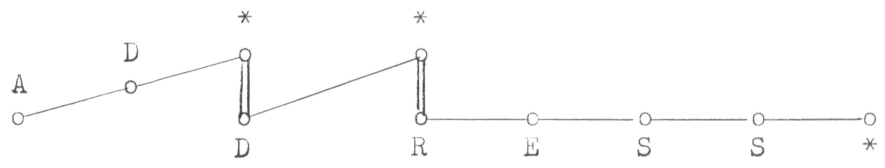


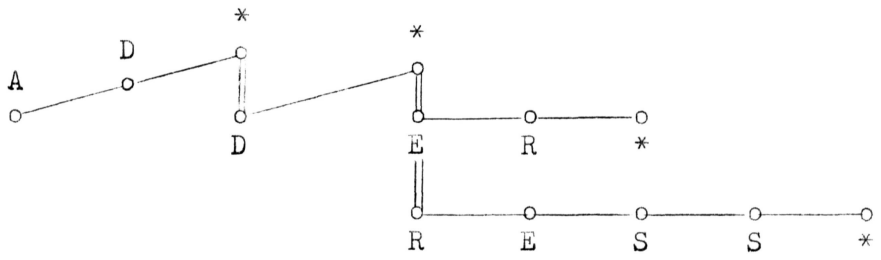
Figure 4 With the Word "AD" Added to the Tree

Figure 6

Location	$\underline{w}_1$	$\underline{w}_2$	$\underline{w}_3$
100	A		101
101	D		271
⋮			
105	D		107
⋮			
107	*	115	XXX
⋮			
115	R		120
⋮			
120	E		121
121	S		123
⋮			
123	S		124
124	*		XXX
⋮			
271	*	105	XXX

Figure 5 With "AD" Added to the Tree

Figure 7



Tree of Fig. 6 With "ADDER" Added

Figure 8

Location	$\underline{w}_1$	$\underline{w}_2$	$\underline{w}_3$
100	A		101
101	D		271
⋮			
105	D		107
⋮			
107	*	311	XXX
⋮			
115	R		120
⋮			
120	E		121
121	S		123
⋮			
123	S		124
124	*		XXX
⋮			
271	*	105	XXX
⋮			
311	E	115	312
312	R		313
313	*		XXX

Memory Map of Fig. 7 With "ADDER" Added to the Tree

Figure 9

For example, if the word "ADDER" were to be added to the trees depicted in Figs. 6 and 7, and the next free location were 311, Figs. 8 and 9 would be generated, respectively.

5. General Description of Thesaurus Lookup

This block of programs "looks up" an input text in a thesaurus dictionary and suffix lists which have previously been written on tape in tree form (see the description of the setup routines.)

Semantic codes (or a code indicating that the word was not found) are entered in a text output block, together with the sentence and word-in-sentence numbers which identify that word. If the word is not found, it is put in a "not found" block as is explained below. If requested (by leaving sense light 3 on), a block is assembled which can be used as input for syntactic analysis. Each item contains the English text word with its numbers, the semantic and syntactic codes which are entered in the thesaurus with the item of that word, and a code for the suffix of the word (if any). If the word is not found, a special code is entered (semantic and syntactic words are zeroed) and a search is made for a possible suffix to aid in syntactic analysis.

The search for a match with an input word proceeds from left to right one letter at a time. The longest possible match is looked for, but notice is taken of any possible shorter stems which may be found on the way. If the match is not complete (i.e., if the whole input word is not matched), a suffix is searched for. If no successful match is found for both stem and suffix, the routine backtracks to possible shorter stems and again searches for suffixes. Certain spelling rules to be described are built into the program.

The thesaurus and suffix tree formats were described as part of the setup programs. Other format specifications are as follows:

(1) Input Text Words. Each item consists of five machine words. The first stores the sentence number of the word in the decrement field and the word-in-sentence number in the address field. The next four words contain the English word in BCD form.

(2) Text Output. Each item consists of three words. The first contains the sentence and word numbers as above. The next two contain either the semantic codes associated with the word (or with its stem) in the thesaurus, or a "not found" code (all bits "on").

(3) Syntactic Output. Each item consists of nine words. The first four store the English word; the next the sentence and word numbers; the sixth and seventh contain the semantic codes; and the eighth and ninth represent the syntactic codes, with the last eight bits of the ninth word being a suffix code (if a suffix exists). If the word was not found, these last four words are set equal to zero. There may, however, still be a suffix code stored in the last word. Before the block of syntactic items is written onto tape it must be revised, since all blocks are stored in backward (FORTRAN II) order.

(4) Not Found Output. Each item consists of six words. The first five are identical with the input items. The sixth word contains a code in the decrement indicating whether

   (a) no stem was found, or

   (b) no suffix match could be made after a possible stem was detected.

The address of the sixth word contains the index of the first letter in the word for which no match could be found. This information is generated relative to the last pass of the lookup process. Since each individual letter appears as an entry in the thesaurus, a stem consisting of a single letter is always found: the first letter not found is then determined relative to this stem.

6. Implementation of Lookup

Three subroutines comprise this block: LØØK, RSSX, and GCRW. The sole function of the latter is to print messages in case of error while reading the input tape, or writing the syntactic output tape. RSSX (Flowchart 5) looks for a suffix if no stem is found. Since this is only wanted for the syntactic analysis, the routine is called only when syntactic output is desired. LØØK, the only routine called from the main program, performs all other functions (see Flowchart 4).

LØØK begins by reading in the thesaurus and suffix trees from tape. Sense light 3 is then examined and transfers are set throughout the program either to bypass or perform the operations that produce the syntactic output. If the sense light is on, syntactic output will be produced.

The data output section of LØØK is explicitly described in Flowchart 4. The boxes surrounding box 3 store data when a word is found in the thesaurus. The series of blocks starting with box 4 perform the operations needed when a word is not found. Since the buffer allotted for syntactic output is not large enough to hold the entire output for a text longer than

100 words, this output must be written onto tape in pieces. Before it is put on tape, the block must be reversed so as to appear in forward order. This operation is handled while the tape is being written so that no additional time is lost.

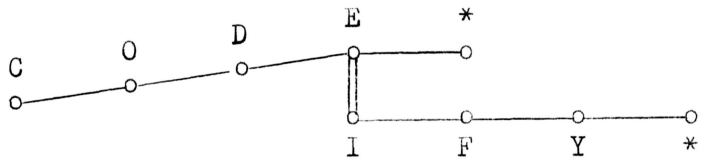
The actual search through the thesaurus tree is similar in outline to that of the search in SUFTR (see description of trees and of the SUFTR routine in Part 4 of this section). However, LØØK must also take note of possible stems which are not complete matches for the input word (e.g., HAND is a stem of HANDING but the two words obviously do not match completely). The following are considered indicative of possible stems:

(1) a blank indicating end of word;

(2) a final "e" (an "e" followed by a blank with a blank in its filial set);

(3) an "i" occurring as a letter of the input word with a final "y" in the tree (the parameter ITEST of Flowchart 4 ensures that no "y" will be noted unless the input letter is an "i").

The first possibility corresponds to a word taking a suffix as an ending, e.g., HAND + ING is HANDING. The second corresponds to a word dropping an "e" before taking a suffix, e.g., HOPE less the E + ING is HOPING. The third corresponds to a "y" changing to an "i" before taking a suffix, e.g., PRETTIER is PRETTY with the "Y" changed to an I + the suffix ER.



Pictorial Representation of a Part of a Thesaurus Tree

Figure 10

The necessity of becoming aware of these stems accounts for the fact that in TREET (see description of TREET in setup section) blanks, "e"'s, and "y"'s are inserted ahead of other members of a filial set. This restriction on the order implies that all possible stems must be recognized before any match is found.

The following example will serve to render this procedure clearer. Consider a (partial) thesaurus tree as shown in Fig. 10 and assume that the input word is CODING. LØØK moves from left to right along the tree matching C, O, and D. In trying to match the I, the final E of CODE is first encountered and is saved as a possible stem. The routine then moves on to find a match with the I of CODIFY. Eventually the routine fails to match the N of CODING with the F of CODIFY. So, remembering that CODE was a possible stem, a search is made for a suffix beginning with the I. When ING is found, CODING is recognized as the stem CODE plus the suffix ING.

7. Spelling Rules Incorporated into Lookup

In order to be able to recognize, for example, that HOPED is HOPE + ED while HOPPED is HOP + doubled letter + ED, certain spelling rules are built into LØØK. These are summarized in Fig. 11 ("*" represents blank).

Case Number	Case Description	Longer Ending	Shorter Ending
1	*		
(a)	B= C (a doubled letter)	*	*
(b)	B≠ C and B is an "e"[†]	*	
(c)	B≠ C and B is not an "e"	*	
2	e	e	
3	y (means C is an "i")		y
4	* and e		
(a)	B= C	*	*
(b)	B≠ C and C is a vowel	e	
(c)	B≠ C and C is not a vowel	*	
5	* and y	*	y
6	e and y	e	y

B refers to the last letter for which a match is found in the thesaurus;

C refers to the first letter for which no match is found in the thesaurus (the current letter being searched for).

Summary of Spelling Rules Built into LØØK

Figure 11

[†] "e" followed by the longer ending is also treated as a possible ending and is associated with the "e" stem.

The three possible types of stems - those terminated by a blank; those terminated by an "e" followed by a blank; and if C is an "i", those terminated by a "y" followed by a blank - are referred to in column 2 of Fig. 11 by "*", "e", and "y", respectively.

The "longer ending" includes C followed by the rest of the word; the "shorter ending" refers to the rest of the word not including C; "e" followed by the longer ending denotes the concatenation of the letter "e" with the letters of the longer ending.

For example, the input HANDIER would be decomposed into the possible stems HAND (denoted by "*") and HANDY (denoted by "y"). In this case, the longer ending would be "ier", the shorter "er". B would be the letter "d", and C the letter "i". HOPED would furnish HOPE as a possible stem. B = "e", C = "d"; the longer stem is just "d", and "e" concatenated with the longer stem is "ed".

The longer of two endings is always treated first; if no match is found in the suffix table, the shorter one is tried. Examples of cases of Fig. 11 are shown in the following list (L will be used as an abbreviation for the longer ending; S for the shorter):

(a) If the input word is HOPPED, the stem is HOP indicated by a terminal "*" (blank). B = p = C, so case 1(a) applies. First a match in the suffix list is attempted for L = "ped". None is found so S = "ed" is tried and matched. If, however, FINALLY were the input word and FINAL the stem, a match would be found for the longer ending, namely "ly".

(b) If the input word is HOPED, the stem is HOPE (a "*" stem); B is an "e", "e" + L = "ed" is searched first and a match is found. For HOPELESS, "e" + L would be "eless", and no match would be found; but the search for a suffix matching L = "less" would be successful.

(c) If the input word is ENDED, the stem is END. B is a "d", so that case 1(c) applies.

In all of the above cases the semantic and syntactic data are taken from the data in the thesaurus tree pointed to by the "*" of the recognized stem. This explains the significance of the "*" in the longer and shorter ending columns of Fig. 11.

Consider now the entries EASE and EASY, both included in the thesaurus, and the input words EASING and EASIER. In both cases, EASE (marked by the "e") and EASY (marked by the "y") will be found as possible stems. For EASING, L = "ing" will be found in the suffix list, so LØØK will take the data from the "e" stem EASE, and the suffix code from "ing". For EASIER, the program will fail to find a match for L = "ier" and so will try to match S = "er". Since this is successful, the data are taken from the "y" stem EASY and the suffix code from "er".

If the input word were HOPED, and both HOPE and HOP were in the thesaurus, LØØK would first attempt the longer match with HOPE – marked by a "*" since the match would be made through the "e" – and apply rule 1(b). If (as is not the case) no suffix were found to match, the routine would then backtrack one level to where it had found the possible stems HOP (marked by a "*") and HOPE (marked by an "e") and apply rule 4(b).

The suffix search uses the same basic search algorithm. It is capable of handling many types of multiple suffixes; e.g., for HOPELESSLY it will find the suffix "less". Since the son of the last "s" is an "*", and that no "ℓ" can be found in the filial set of this "s", the routine will re-enter the suffix tree from the beginning, trying to match the second "ℓ" of HOPELESSLY. It will proceed to find a match with "ly".

Only the code for the last suffix found is entered in the data of the syntactic output. For this reason it is desirable to enter certain compound suffixes explicitly in the tree to provide a maximum of syntactic information (e.g., if KINDNESSES is being looked up, and "ness" and "es" are in the suffix tree but "nesses" is not, LØØK will only pick up the data of "es"; if "nesses" were in the suffix list, enough information would be available to recognize a plural noun).

LØØK will not find multiple suffixes if there is an "e" telescoped between the parts; e.g., "ized" is "ize" + "ed", but the one "e" is "used" by the two parts. Such suffixes must also be entered separately (e.g., "ized" as well as "ize" and "ed").

8. Processing of Words Not Found by the Thesaurus Lookup

The general purpose of this program block is to supply information about the words that are not found by the thesaurus lookup. This information is useful for deciding upon later alterations (particularly additions) to the thesaurus.

If all the words are found by LØØK, a message to this effect is printed out. If not, distinct occurrences of the same English word are combined into one output item. This item contains:

(1) the English word in BCD format;

(2) a code indicating whether LØØK failed to find a stem in the thesaurus that matched the input word or whether, having found a possible stem, it was unable to find a suffix in the suffix list to match the rest of the word (see the description of the lookup block);

(3) the number in the word of the first letter of the input word for which no match was found in the thesaurus (see the description of the lookup block);

(4) the number of occurrences of the word in the text looked up;

(5) the sentence and word-in-sentence numbers of the occurrences of the word.

All of this information except the fourth item, the number of occurrences, is supplied explicitly by LØØK. For example, assume that the word PLANET is not included in the thesaurus, but that the word PLANE is, and is the only possible stem identified. Assume further that in a given text PLANET occurs as the seventh word of the third sentence, and as the tenth word of the eleventh sentence. The output for this word would then appear as in Fig. 12.

WORD	KIND	LOC	NUM	SENTENCE AND WORD NUMBERS	
PLANET	SUFFIX	6	2	3,7	11,10

Output Format for a Word not Found in the Thesaurus

Figure 12

LØØK would find a match up to the "e" of PLANE and thus think it had found a possible stem. It would then fail to find a suffix to match the "t". Thus, the stem-suffix code (KIND) indicates that a potential stem was found but no suffix was found. Since "t" is the sixth letter of the word PLANET, the location of the letter in the word (LOC) is 6. The number of occurrences of the word (NUM) is 2. The sentence and word-in-sentence numbers are as shown.

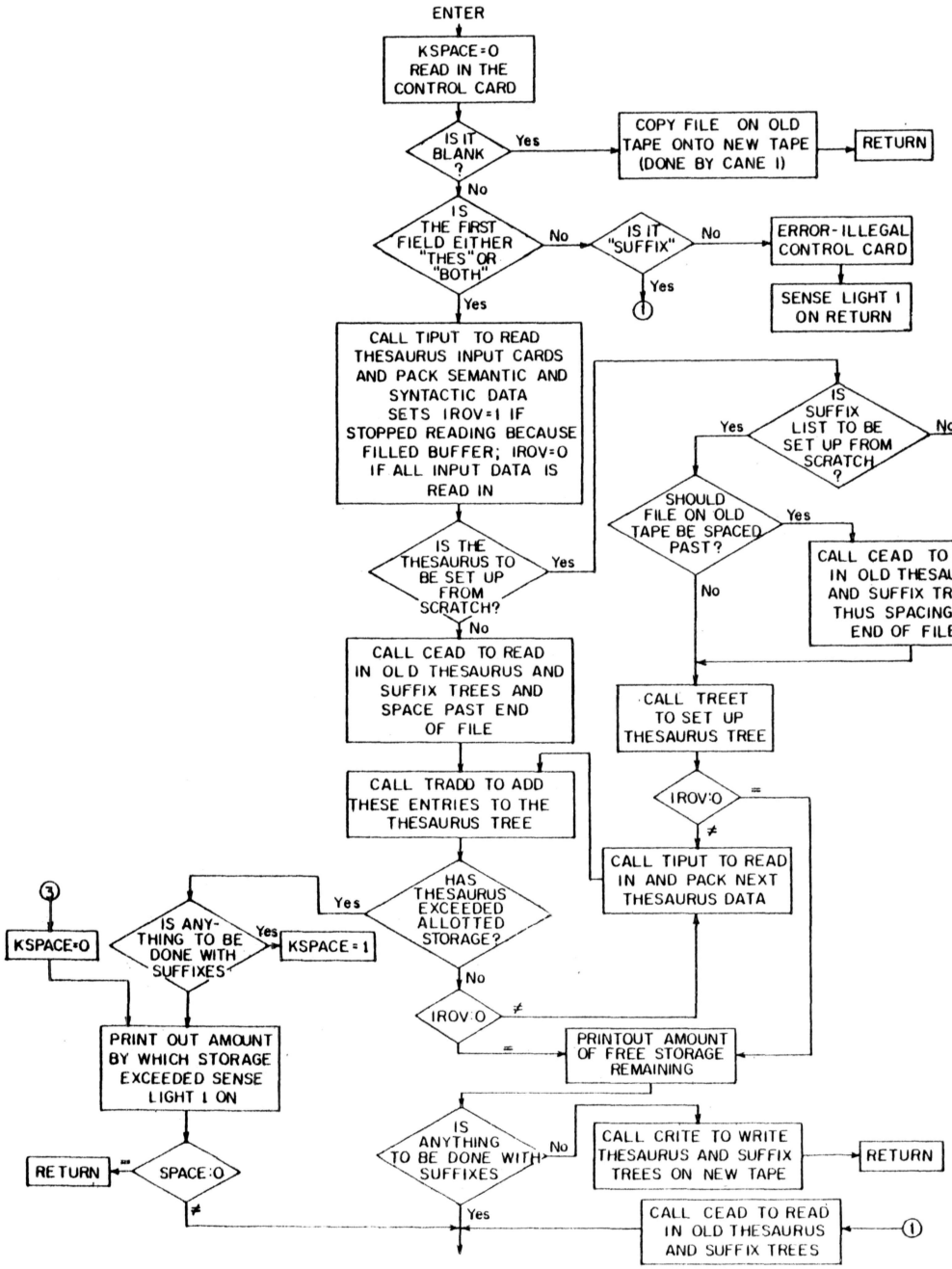
Subroutine LØØK provides two inputs for this block: an array of the words not found (for format see the description of the thesaurus lookup block), and a count of the number of computer words in this array. Currently space is provided for 250 items, each of which consists of six computer words. If more than 250 words are not found, only the first 250 are processed.

The program is composed of three subroutines: SNCC, BAG, and CRCC. Only SNCC is called from the main routine. No arguments are needed, since the inputs are left in common by LØØK. SNCC first checks to see if all the words were found. If so, a message to that effect is printed out and return is made to the main program. If not, a heading for the output is printed

and the first word of the output block is set up. CRCC is then called. If only one word is not found, CRCC immediately proceeds to process it for printing. If more than one word is not found, each additional word in the input block is compared successively with the words in the output block. If different from all of them, the five pieces of information mentioned above are stored as a new item in the output block with the occurrence count set to one. If, however, the input word is identical with one of the words in the output block, the occurrence count of that output item is increased by one and the sentence and word-in-sentence numbers of this latest occurrence are stored in the output block. Space is provided for 26 such occurrence numbers for each item. In the unlikely event that more than 26 occurrences are detected, the occurrence count is increased, but the numbers are not stored.

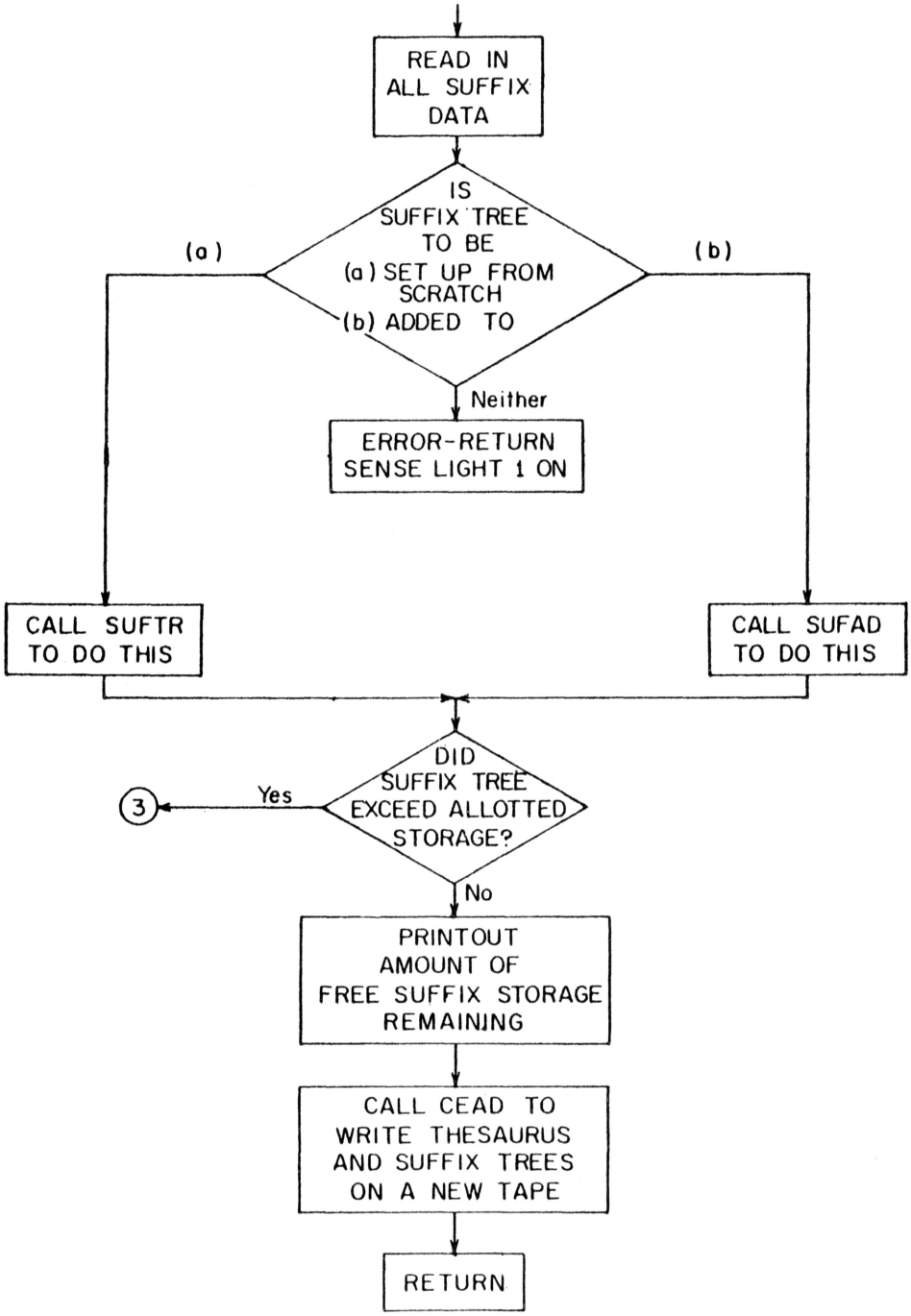
After all input words are treated in this manner, the output block is ready to be printed out. CRCC unpacks the sentence and word-in-sentence numbers of each item (see the writeup of LØØK for formats), and puts them in a form acceptable to the FORTRAN I-Ø routines. Subroutine BAG then prints out (off-line) the relevant information for each item in the format of Fig. 12. After all items are printed, control is returned to the main program. (Since subroutine CRCC uses multiple tagging, the computer is assumed to be in the multiple tag mode when this routine is called).

APPENDIX

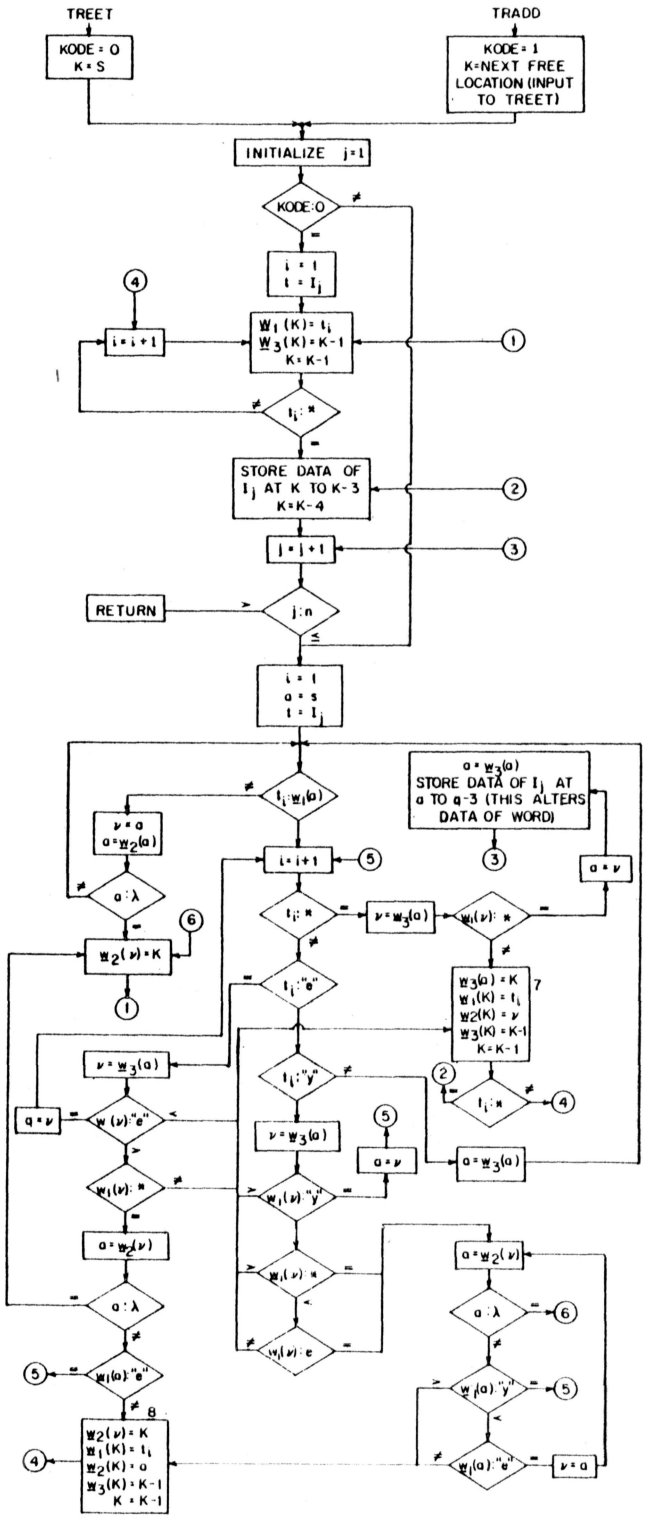


Main Logic of Thesaurus Setup Block
(Subroutine CANE2)

Flowchart 1

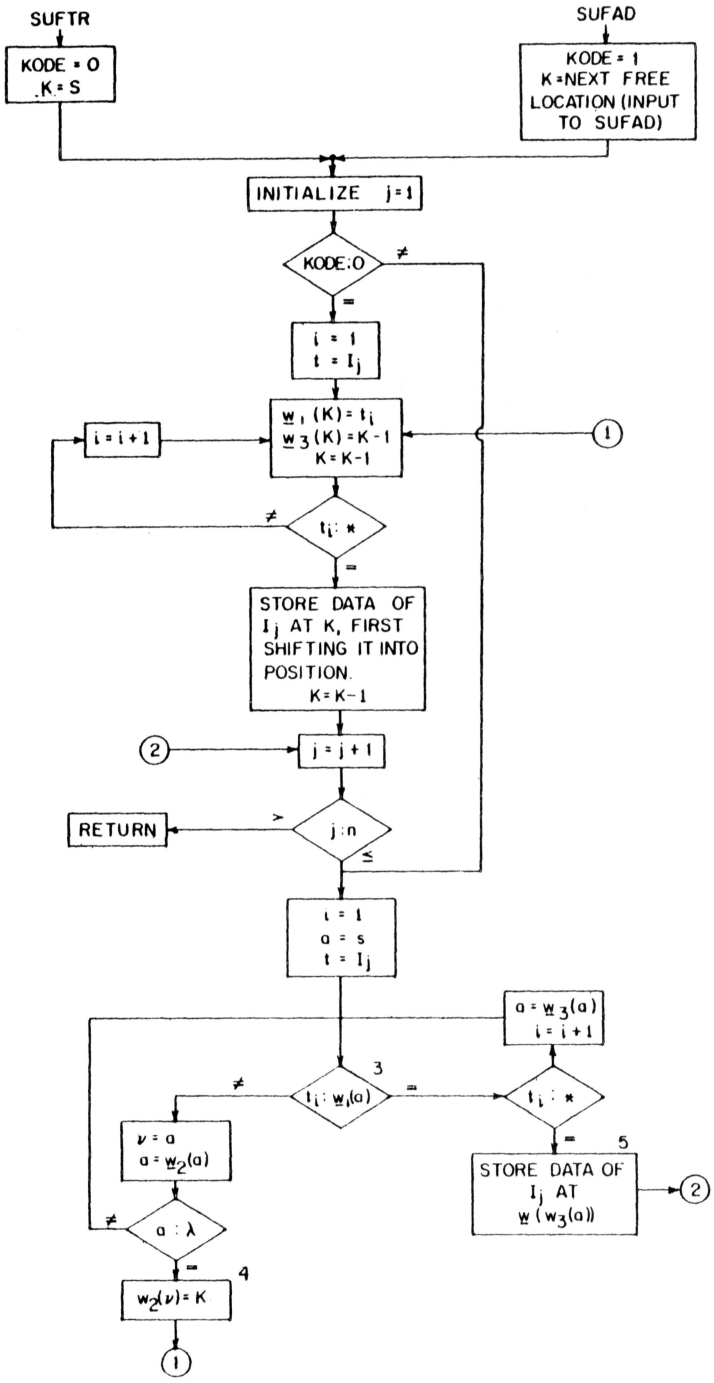


Flowchart 1 (continued)



TREET, TRADD

Flowchart 2

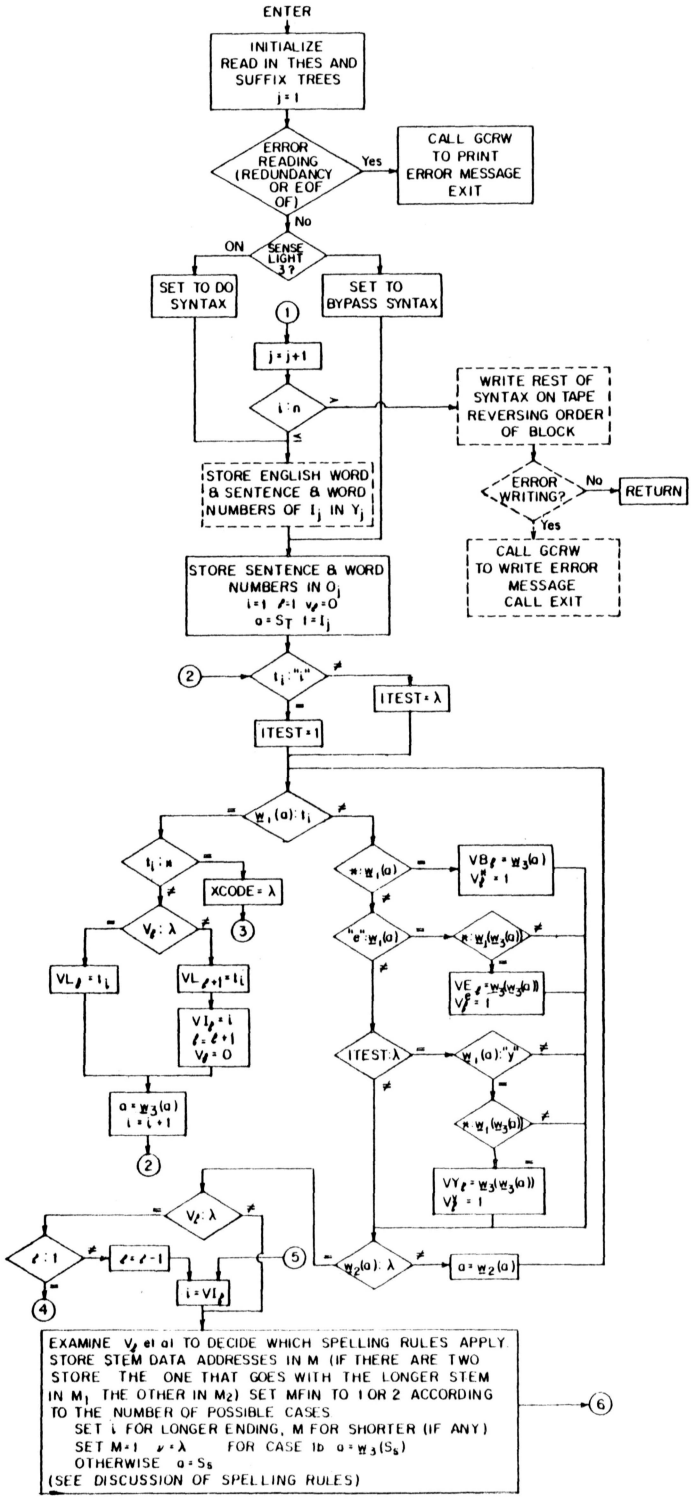


SUFTR, SUFAD

Flowchart 3

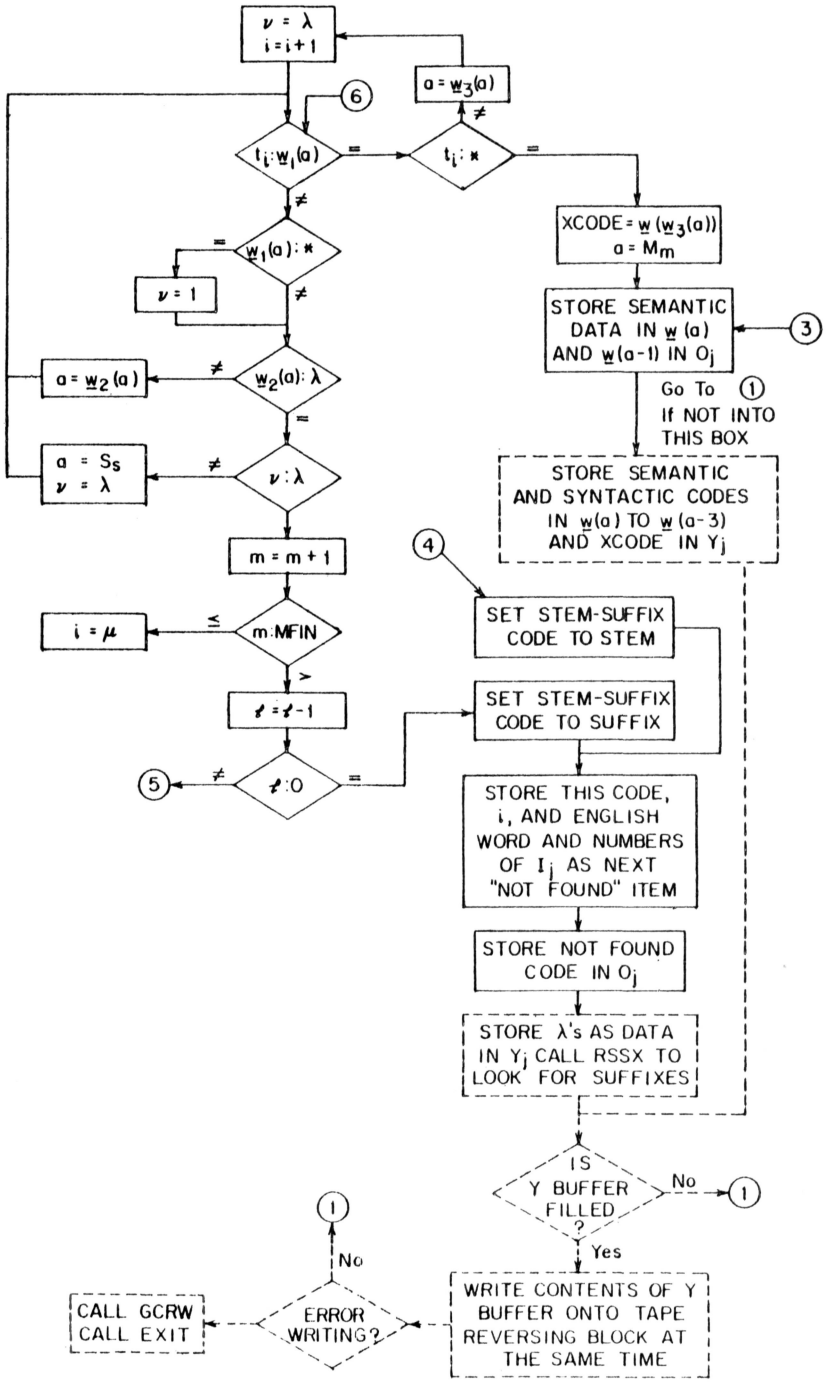
LEGEND FOR FLOWCHARTS 2 AND 3 – SUFTR, SUFAD, TREET, TRADD

I	Input data.
j	Index of I; I_j is the jth input item.
t	Input word (≥ 2 characters, always including the terminating character blank).
$\underline{w}$	Machine word.
	$\underline{w}_1$ – is the key.
	$\underline{w}_2$ – is the address of next $\underline{w}$ in the same filial set (the next brother of $\underline{w}$).
	$\underline{w}_3$ – is the address of first $\underline{w}$ in the filial set (the first son of $\underline{w}$) if $\underline{w}_1$ = * then $\underline{w}_3$ is the address of the data for the word.
$\underline{w}(a)$	Machine word in address a.
i	Indexes t; t_i is the ith letter of word t.
*	Indicates the character blank.
n	Number of input items.
λ	A blank machine word, field or character.
s	Starting address of the tree.
K	Indexes free space: K is the address of the next free location in the block. K starts at s and is decreased.
ν	Temporary storage.
"e"	The character "e" (in general "x" means the character x).

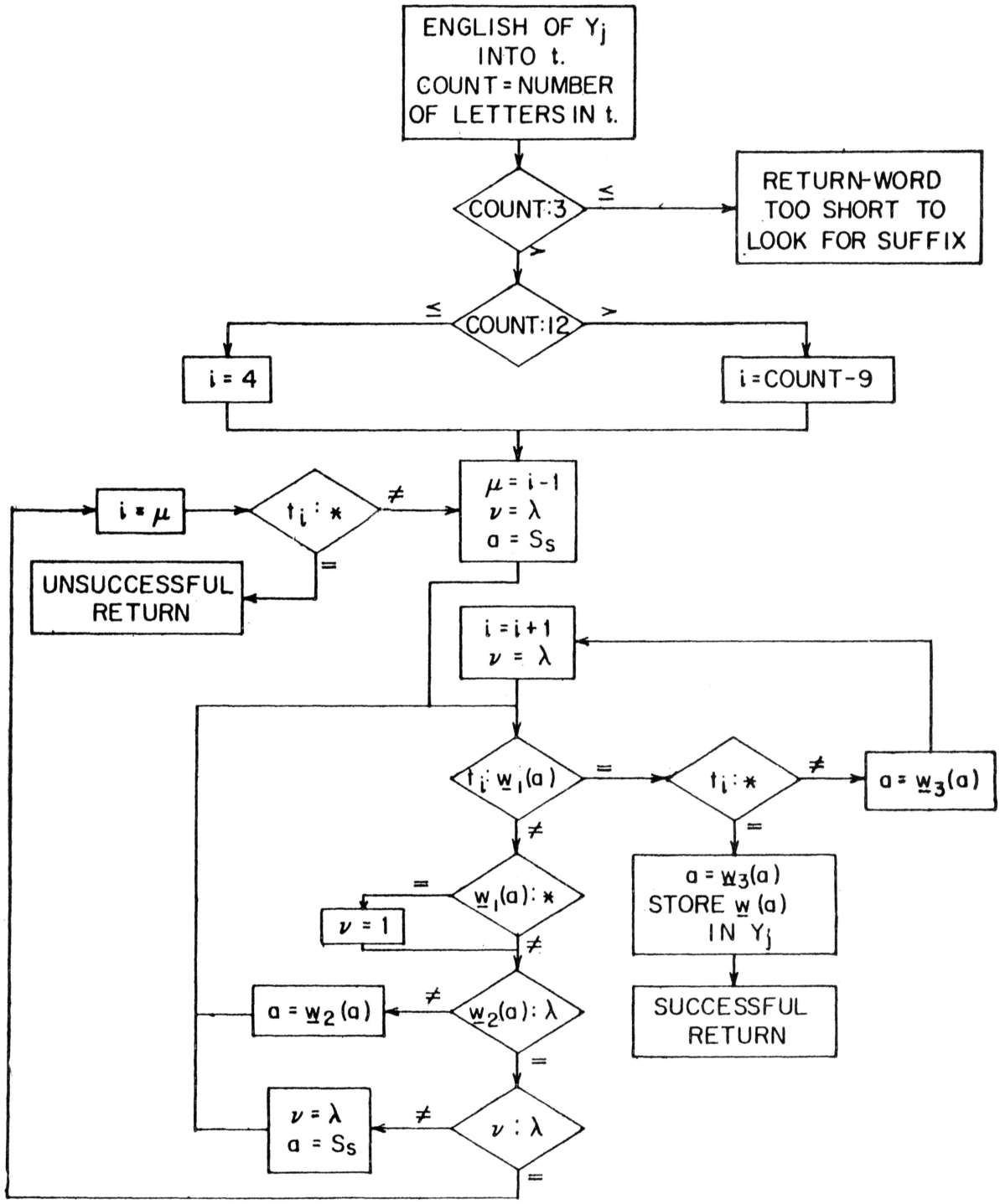


LØØK

Flowchart 4



Flowchart 4 (continued)



RSSX

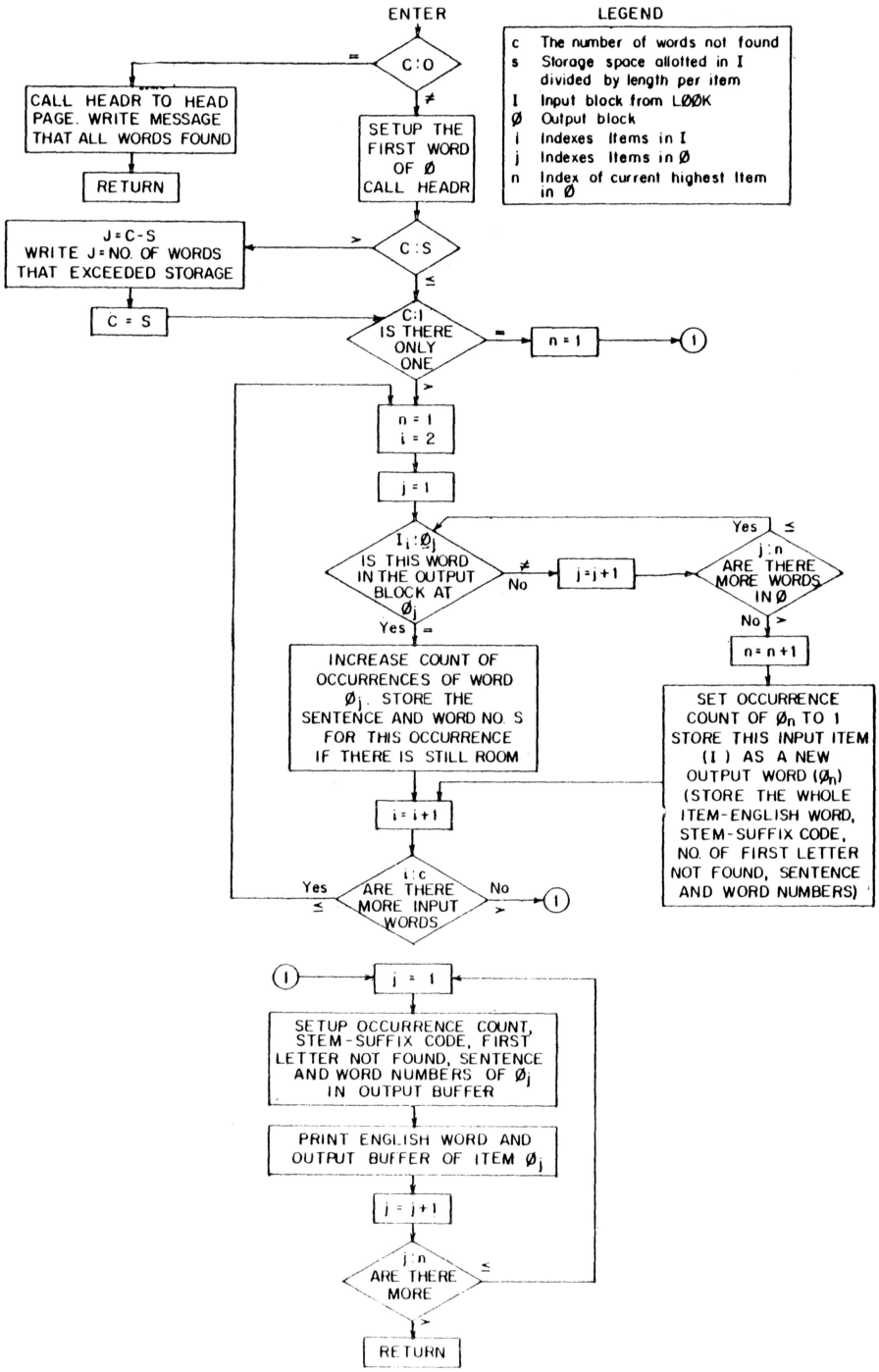
Flowchart 5

LEGEND FOR FLOWCHARTS 4 AND 5 – LØØK, RSSX

The definitions, unless otherwise noted, are those given in the Legend for Flowcharts 3 and 4 (SUFTR, TREET, SUFAD, TRADD). I now refers to input text items. Since in the content of LØØK s is ambiguous it will be subscripted with a t when referring to the start of the thesaurus, and an s when referring to the start of the suffix tree.

- y — The block of output used as input to the syntactic programs, indexed by j (though the buffer for this block is too small to contain all syntax output).
- ℓ — Indexes all of the following information connected with possible stems.
  - VI — the value of i (the index of the input word).
  - V — has three fields V^*, V^e, V^y indicating whether possible stems were distinguished by a blank, a final "e", or a final "y" respectively.
  - VB,VE,VY — contain the address of the data for the stems distinguished by blanks, "e", and "y" respectively.
  - VL — contains the previous letter (i.e. if $VI_\ell = i$, $VL_\ell = t_{i-1}$).
- O — Text output – indexed by j.
- XCODE — Suffix code.
- M — Holds addresses of stem data.
- m — Indexes m.
- MFIN — Total number of addresses in M.
- μ — Temporary storage.

Note: The dotted boxes are entered only if syntactic outputs are desired.



NOT FOUND BLOCK

Flowchart 6